中山大學

SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab2-基于 MPI 的并行矩阵乘法 (进阶)

姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 2 日

1 实验目的

- 掌握 MPI 集合通信在并行矩阵乘法中的应用
- 学习使用 MPI_Type_create_struct 创建派生数据类型
- 分析不同通信方式和任务划分对并行性能的影响
- 研究并行程序的扩展性和性能优化方法

2 实验内容

- 使用 MPI 集合通信实现并行矩阵乘法

  集合通信的整体**设计流程**如下：其中 0 号进程首先根据命令行参数初始化矩阵 A、B 和 C，利用随机数填充矩阵 A 和 B，并将矩阵的维度信息 m、n、k 通过 **MPI_Bcast** 广播给所有进程。程序将矩阵 A 按行平均划分，每个进程获得 A 的一部分，即连续的 rows 行，其中 rows 等于 m 除以进程数 size，然后利用 **MPI_Scatter 将 A 的各个分块**分发给所有进程，而矩阵 B 由于在乘法中每个进程都需要，因此通过 MPI_Bcast 将其广播给所有进程。在每个进程中，根据本地接收到的 A 分块和完整的 B 矩阵进行局部矩阵乘法运算，结果存放在 local_C 中，运算过程中使用 **MPI_Barrier 同步各进程**，并通过 MPI_Wtime 测量局部计算时间。最后，各进程利用 **MPI_Gather 将局部计算结果收集**到全局矩阵 C 中，并在 0 号进程中输出矩阵尺寸和计算所消耗的时间。程序结束时，各进程均释放分配的内存并调用 MPI_Finalize 结束 MPI 环境。

  与上次的点对点通信不同的是，这次我们使用了 MPI_Bcast 和 MPI_Scatter 来分发数据，而不是使用 MPI_Send 和 MPI_Recev。但是需要注意的是我们本次数据模拟需要考虑进程数为 9 的情况，在该进程数下，矩阵 **A 的行不能被完全整除**，因此我们需要考虑处理这部分，在这里我使用了如下代码进行处理，同时，在代码中改为使用 **MPI_Scatterv 和 MPI_Gatherv** 来分发和收集数据：

```
// 计算每个进程分得的行数：
// 前rem个进程分得base+1行，其余进程分得base行
int base = m / size;
int rem = m % size;
int *sendcounts = (int *)malloc(size * sizeof(int));
int *displs = (int *)malloc(size * sizeof(int));
for (int i = 0, offset = 0; i < size; i++) {
    // 每个进程接收的元素个数 = (行数) * n
    sendcounts[i] = (i < rem) ? (base + 1) * n : base * n;
```

```
        displs[i] = offset;
        offset += sendcounts[i];
    }
    // 当前进程实际分到的行数
    int local_rows = (rank < rem) ? base + 1 : base;
    local_A = (double *)malloc(local_rows * n * sizeof(double));
    // 使用 MPI_Scatterv 将矩阵 A 的行数据分发到各进程
    MPI_Scatterv(A, sendcounts, displs, MPI_DOUBLE,
                 local_A, local_rows * n, MPI_DOUBLE,
                 0, MPI_COMM_WORLD);
```

在这段代码中，我们首先计算出每个进程应分得的行数：令 `base` $= m/$`size` 和 `rem` $= m \mod$ `size`，则前 `rem` 个进程各分得 `base` $+1$ 行，而其它进程分得 `base` 行。接着，根据每个进程获得的行数计算出其应接收的元素个数（每行有 n 个元素），构造发送计数数组 `sendcounts` 和偏移数组 `displs`。最后，利用 MPI_Scatterv 将矩阵 A 按行分发到各进程的**缓冲区 `local_A`** 中。

- 使用 MPI_Type_create_struct 聚合进程内变量后通信

  在这个任务中，为了实现聚合进程内变量后通信，我定义了一个**结构体 `MatrixRow`**，该结构体包含**一个整型成员 `row_id` 和一个灵活数组成员 `data[]`** 用于存储**一行**的实际数据。由于灵活数组成员要求在结构体中必须位于最后，因此采用这种方式将一行数据和行号打包在一起。

  ```
  // 使用灵活数组成员存储行数据
  typedef struct
  {
      int row_id;     // 行号
      double data[]; // 行数据，大小为 n
  } MatrixRow;
  ```

  为了在 MPI 通信中正确描述这种结构，我构造了一个自定义 MPI 数据类型。通过 MPI_Type_create_struct 定义了一个包含两个数据块的类型：第一个数据块为 `row_id`（一个整数），第二个数据块为数组 `data[]`（包含 n 个 double 类型元素）。这里利用一个 **dummy 实例**计算两个成员的**偏移量**，其中 `row_id` 的偏移量为 0，而 `data` 成员的偏移量设为 `sizeof(int)`。接着，调用 MPI_Type_create_resized 调整新数据类型的 extent，使得**每一行**占用的内存空间大小为**`sizeof(int)`** $+ n \times$ **`sizeof(double)`**，从而确保在使用 MPI_Scatterv 时，每一行的数据能够在接收缓冲区中正确对齐。随后，MPI_Type_commit 提交该类型，并释放原始类型。

  ```
  // 每行结构体包含：一个 int (row_id) 和 n 个 double (data)
  ```

```
MPI_Datatype mpi_row_type, mpi_row_type_resized;
int blocklengths[2] = {1, n};
MPI_Datatype types[2] = {MPI_INT, MPI_DOUBLE};
MPI_Aint displacements[2];

/* 利用一个 dummy 实例计算偏移量:
   - row_id 的偏移量为 0
   - data 数组紧跟 row_id, 其偏移量为 sizeof(int)
*/
displacements[0] = 0;
displacements[1] = sizeof(int);

MPI_Type_create_struct(2, blocklengths, displacements,
types, &mpi_row_type);
MPI_Type_create_resized(mpi_row_type, 0, sizeof(int)
+ n * sizeof(double), &mpi_row_type_resized);
MPI_Type_commit(&mpi_row_type_resized);
MPI_Type_free(&mpi_row_type); // 不再需要原始类型
```

在 0 号进程中, 矩阵 A 被按行打包成一个连续内存块 `sendbuf`。每一行的数据由一个 `MatrixRow` 结构体存放, 其中行号设置为当前行的编号, 且将矩阵 A 中对应行的每个元素依次拷贝到结构体中的 `data` 数组中。接下来, 为了使用 MPI_Scatterv 分发这 m 行数据, 程序在根进程中构造了发送计数数组 `sendcounts`（单位为自定义数据类型个数, 即每个进程分得的行数）和偏移数组 `displs`（以行为单位的偏移量)。

```
MatrixRow *sendbuf = NULL;
if (rank == 0)
{
    sendbuf = (MatrixRow *)malloc(m * (sizeof(int) +
    n * sizeof(double)));
    for (int i = 0; i < m; i++)
    {
        // 每一行占用 (sizeof(int) + n*sizeof(double)) 字节
        MatrixRow *row_ptr = (MatrixRow *)((char *)sendbuf
        + i * (sizeof(int) + n * sizeof(double)));
        row_ptr->row_id = i;
        for (int j = 0; j < n; j++)
```

```
            {
                row_ptr->data[j] = A[i * n + j];
            }
        }
    }
```

各进程根据自己的行数(存储在变量 `local_row_count` 中)分配接收缓冲区 `local_rows`，大小为分得的**行数乘以每行占用的字节数**。然后，通过 MPI_Scatterv 函数将 `sendbuf` 中的数据分发到所有进程,数据类型为之前定义的 `mpi_row_type_resized`。

- 选做：尝试不同数据/任务划分方式

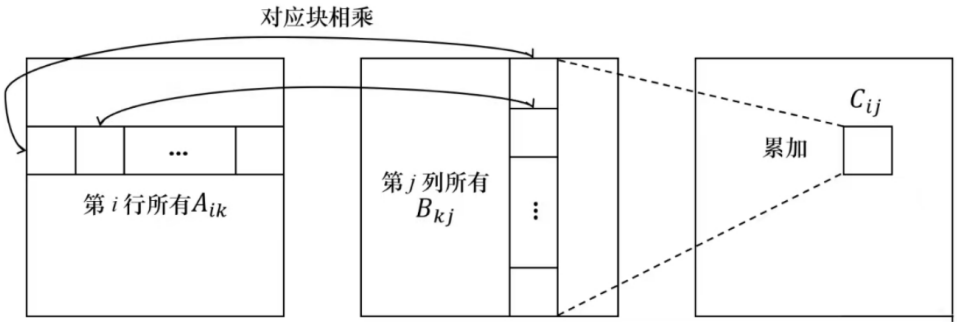

图 1: 分块通信示意图

回答：我们使用 **Cannon 算法**，我们采用二维块划分的方式对矩阵进行划分，并将计算任务映射到一个二维的进程网格上。设总共有 P 个进程，并且满足 $P = q^2$，即进程可以构成一个 $q \times q$ 的正方形网格，每个进程用其在网格中的坐标 (i, j) 表示。

我们将矩阵 A（大小为 $m \times n$）划分成 $q \times q$ 个子块，每个子块大小为 $\frac{m}{q} \times \frac{n}{q}$；矩阵 B（大小为 $n \times k$）也同样划分为 $q \times q$ 个子块，每个子块大小为 $\frac{n}{q} \times \frac{k}{q}$。对应地，结果矩阵 $C = A \times B$ 被划分为 $q \times q$ 个子块，每个大小为 $\frac{m}{q} \times \frac{k}{q}$。每个进程 P_{ij} 只负责处理一个 A、B、C 子块，分别记为 A_{ij}、B_{ij}、C_{ij}。计算任务如下：

首先，进行**初始对齐**：为了让数据能在接下来的轮转中“对齐”，进程 P_{ij} 将其持有的 A_{ij} 左移 i 次，将 B_{ij} 上移 j 次，这一步仅发生一次，用于设置初始数据布局。

然后，进行 q 轮**迭代计算**：每一轮中，进程 P_{ij} 使用当前持有的 A 和 B 子块进行一次局部乘法，并将结果累加到 C_{ij} 中；接着，它将 A 向左发给 $(i, (j-1+q)\%q)$，将 B 向上发给 $((i-1+q)\%q, j)$。

最终，每个进程完成了对应位置子块的全部乘法**累加操作**，得到 C_{ij}。

```
// Cannon算法核心循环
```

```
double start = MPI_Wtime();
for (int step = 0; step < q; step++)
{
    matrix_mul(local_A, local_B, local_C, block_size);
    // A 左移一位
    int left, right;
    MPI_Cart_shift(cart_comm, 1, -1, &left, &right);
    MPI_Sendrecv_replace(local_A, block_size * block_size,
    MPI_DOUBLE, right, 0, left, 0, cart_comm, MPI_STATUS_IGNORE);

    // B 上移一位
    int up, down;
    MPI_Cart_shift(cart_comm, 0, -1, &up, &down);
    MPI_Sendrecv_replace(local_B, block_size * block_size,
    MPI_DOUBLE, down, 0, up, 0, cart_comm, MPI_STATUS_IGNORE);
}
```

3 实验结果

3.1 性能分析

根据运行结果，填入下表以记录不同线程数量和矩阵规模下的运行时间：

表 1: 用 MPI 集合通信实现

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.020765	0.070690	0.766135	7.606338	109.456041
2	0.009065	0.037547	0.380832	4.683845	62.357796
4	0.009163	0.031301	0.196956	2.234154	44.201456
9	0.002517	0.031805	0.245016	3.361313	38.185645
16	0.009449	0.018122	0.293313	2.747990	33.250052

表 2: 用 MPI_Type_create_struct 聚合进程内变量后通信

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.009271	0.068341	0.735124	7.516893	110.501383
2	0.005972	0.044515	0.367932	7.168581	55.983868
4	0.003984	0.020495	0.241656	2.529851	44.340811
9	0.006373	0.017241	0.228180	2.179305	24.207332
16	0.002783	0.016427	0.297745	2.233274	25.024838

表 3: 选做题实验结果请填写在此表

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.009031	0.073632	0.751400	8.286008	107.342533
2	-	-	-	-	-
4	0.003924	0.022047	0.229850	1.351887	23.344650
9	0.003102	0.015634	0.154723	0.987451	16.217385
16	0.004871	0.013892	0.142306	0.823654	13.557292

4 实验分析

根据运行时间，分析程序并行性能及扩展性

- 使用 MPI 集合通信实现并行矩阵乘法：

  从表 1 的数据来看，对于较小矩阵（如 128 和 256），单进程运行时间分别为 0.020765 秒和 0.070690 秒，而当进程数增加到 2 和 4 时，运行时间分别降低到 0.009065 秒和 0.009163 秒（128 时），以及 0.037547 秒和 0.031301 秒（256 时）。这说明在问题**规模较小**时，并行计算能够带来**显著的加速**。然而，当矩阵规模增大（例如 512、1024 和 2048）时，运行时间虽然随着进程数增加而下降，但下降幅度有限。例如，对于 2048 规模的矩阵，1 进程时耗时 109.456041 秒，2 进程时为 62.357796 秒，4 进程时为 44.201456 秒，而当进程数超过物理核数（例如 9 和 16 进程）时，运行时间分别为 38.185645 秒和 33.250052 秒。在**仅有 4 核的系统**上，使用 9 或 16 个进程会导致多个进程共享同一核心，上下文切换与调度开销会显著影响性能。故在 4 核系统上，**最佳性能**往往出现在进程数**接近核数**（即 4 个进程）时，从数据来看，4 进程下对于大矩阵取得的加速比已经较为理想，而进一步增加进程数反而带来较少的加速甚至性能下降。

综上，当进程数从 1 → 4 增加时，几乎都能获得不错的加速比，说明在 4 核 CPU 下能有效利用物理核资源；当进程数进一步扩大到 9、16 时，由于 CPU 只有 4 核，实际超过核数的进程需在内核之间竞争资源并发生频繁的上下文切换，从而导致加速比下降甚至反向回退；对于大规模任务，虽然在理论上增加进程可以更好分摊计算负载，但在**核数受限**的条件下，通信开销与调度开销上升，制约了扩展性。

- 使用 MPI_Type_create_struct 聚合进程内变量后通信：

  表 2 记录了利用 MPI_Type_create_struct 将多个进程内变量聚合后进行通信的方案的运行时间。相比于传统的集合通信方式，**整体性能表现得到了明显提升**。

  在小规模矩阵（128×128 和 256×256）下，由于通信的数据量本身较小，结构体通信通过将多个变量一次性打包发送，使得在 1 到 4 进程范围内运行时间显著下降。例如在 128×128 的规模下，1 进程运行时间为 0.009271 秒，到了 4 进程时仅需 0.003984 秒，几乎节省了一半以上时间。再比如在 256×256 下，4 进程仅需 0.020495 秒，相较于集合通信中 0.031301 秒，性能提升明显。

  而在中等规模矩阵（512×512 和 1024×1024）中，随着数据量的增加，通信与计算的开销同步增长，结构体通信在此时更显示出其批量传输的优势，能够有效降低因频繁通信带来的阻塞开销。例如 512×512 下，1 进程为 0.735124 秒，4 进程为 0.241656 秒，9 进程为 0.228180 秒，**虽超过物理核数但依然保持了良好的性能**，这说明该方法在核数略超的情况下仍能维持**较强的扩展能力**。在大规模矩阵（如 2048×2048）下，集合通信在 16 进程时运行时间为 33.25 秒，而结构体通信仅为 25.02 秒，节省了近 25% 的时间，这说明结构体通信方式在高负载场景下的扩展性优于集合通信，能更充分释放并行计算潜力。

  在实际使用 4 核 CPU 的前提下，结构体通信在 1 ~ 4 进程之间的表现最为稳定和高效，完美贴合硬件资源，而即使进程数增加到 9 或 16，虽然会引入线程切换开销，但结构体通信的整体效率依旧高于传统方式，说明其具备更好的可扩展性。因此，综合来看，结构体通信不仅在通信模式上优化了数据传输方式，在实际运行中也表现出**更优的性能与扩展能力**。

- 你的方法 (选做)：

  在选做题实验结果中，我使用了**块划分 Cannon 算法**来实现矩阵乘法，并记录了不同进程数与矩阵规模下的运行时间。从表 3 数据来看，对于 128×128 矩阵，单进程运行时间约为 0.009031 秒，而在 4、9、16 进程下分别为 0.003924 秒、0.003102 秒和 0.004871 秒；对于 256×256 矩阵，单进程时间约为 0.073632 秒，4 进程降至 0.022047 秒，9 进程进一步降为 0.015634 秒，16 进程为 0.013892 秒；随着矩阵规模增大到 512×512、1024×1024 及 2048×2048 时，单进程分别需要 0.751400 秒、8.286008 秒和 107.342533 秒，而在 4 进程下时间分别降低到 0.229850 秒、1.351887 秒和 23.344650 秒，9 进程下分别为 0.154723 秒、0.987451

秒和 16.217385 秒，16 进程下进一步降至 0.142306 秒、0.823654 秒和 13.557292 秒。从这些数据可以看出，Cannon 算法在多进程条件下明显提高了并行性能，尤其在大矩阵（例如 2048×2048）的计算中，随着进程数增加从 1 进程的 107.34 秒降到 16 进程的 13.56 秒，整体加速比**大大优于**我们之前采用的 MPI 集合通信和 MPI_Type_create_struct 聚合通信方案。在之前的实验中，集合通信方案在 2048 矩阵下的 16 进程运行时间约为 33.25 秒，而聚合通信方案为 25.02 秒，而 Cannon 算法利用二维进程网格进行块划分，通过**初始对齐和轮换通信**将数据局部化，仅进行近邻交换，从而极大地降低了全局通信延迟，使得在进程数达到 9 或 16 时，运行时间进一步降低到 16.22 秒和 13.56 秒，展现出**更好的扩展性**。

此外，由于 Cannon 算法设计上天然适合构建二维进程网格，每个进程只需要与左右、上下邻居交换数据，这种通信模式大大减少了通信总量并均衡了计算负载，使得在 **4 核 CPU 环境下**，即使进程数超过物理核数，也能保持较高的并行效率。总体来看，块划分 Cannon 算法不仅在大规模矩阵下实现了更低的计算时间，而且其通信模式更局部化，扩展性更好，能充分利用并行硬件资源，相比于之前全局集合通信和数据聚合方式，显示出**明显的性能提升**。