中山大學
SUN YAT-SEN UNIVERSITY

# 并行程序设计与算法实验

Lab1-基于 MPI 的并行矩阵乘法


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 3 月 26 日

# 1 实验目的

- 掌握 MPI 程序的编译和运行方法。
- 理解 MPI 点对点通信的基本原理。
- 了解 MPI 程序的 GDB 调试流程。

# 2 实验内容

- 使用 MPI 点对点通信实现并行矩阵乘法。
- 设置进程数量（1~16）及矩阵规模（128~2048）。
- 根据运行时间，分析程序的并行性能。

# 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进程数和矩阵规模下的运行时间：

| 进程数 | 矩阵规模 | | | | |
|---|---|---|---|---|---|
| | 128 | 256 | 512 | 1024 | 2048 |
| 1 | 0.010457 s | 0.076594 s | 0.959686 s | 23.784264 s | 260.503915 s |
| 2 | 0.006418 s | 0.043293 s | 0.629409 s | 9.020635 s | 223.294958 s |
| 4 | 0.002601 s | 0.038895 s | 0.252226 s | 2.990915 s | 162.275957 s |
| 9 | 0.063701 s | 0.021131 s | 0.376537 s | 5.072317 s | 67.776249 s |
| 16 | 0.061480 s | 0.097709 s | 0.748899 s | 3.929359 s | 37.507805 s |

## 3.2 结果分析

- 在本实验中，我的代码设计思路如下：主进程首先初始化 MPI 环境，并根据输入的矩阵尺寸 $m$、$n$、$k$ 随机生成矩阵 A 和 B，然后利用点对点通信(MPI_Send/MPI_Recv)依次将矩阵尺寸广播给所有非主进程，确保每个进程都获得整个问题的规模信息；接着，主进程使用点对点通信将完整的矩阵 B 发送给各个进程，使得每个进程在进行局部矩阵乘法时都拥有所需的 B 矩阵数据；随后，主进程将矩阵 A 按行分片分发给各进程，在分片时考虑到 $m$ 不能被进程数整除，对前 $remain$ 个进程多分配一行以实现负载均衡，每个进程通过 MPI_Recv 接收自己对应的 A 数据块；各进程在获得对应的局部矩阵 A 和全局矩阵 B 后，进行局部矩阵乘法计算，得到自

已的局部结果矩阵；当所有进程完成局部计算后，主进程通过点对点通信依次从各进程收集局部结果矩阵，并拼接形成全局结果矩阵 C；最后，主进程输出整个计算耗时，并可选择性地显示部分结果矩阵以验证计算正确性。

- 这里着重解释一下对 A 的分片：在对矩阵 A 进行分片时，考虑到 A 的行数 $m$ 可能无法被进程数 $size$ 整除，因此我采用了一种负载均衡的分配策略。具体来说，首先计算每个进程至少需要处理的行数 $rows = m/size$，然后计算剩余的行数 $remain = m \mod size$。为了保证计算负载尽可能均匀地分布，前 $remain$ 个进程会多分配一行，而后续的进程则按照 $rows$ 进行分配。例如，如果 $m = 10$ 且 $size = 3$，则 $rows = 3$ 且 $remain = 1$，此时 rank 0 处理 4 行，rank 1 和 rank 2 各处理 3 行。代码中通过计算每个进程的接收数据量（即行数乘以列数 $n$）以及在 A 中的起始偏移量，利用 MPI_Send 将相应数据块发送给各进程，非主进程使用 MPI_Recv 准确接收，从而实现对 A 的正确分片，这种分片方式可以在点对点通信模式下保证负载均衡，避免因数据分配不均导致部分进程成为瓶颈。

```
// 计算每个进程应处理的行数
int rows = m / size;   // 每个进程至少处理的行数
int remain = m % size; // 剩余行数
int local_rows = (rank < remain) ? (rows + 1) : rows;
printf("Process %d computing %d rows.\n", rank, local_rows);

// 主进程对A按行分片分发，使用点对点通信
// 计算各进程数据的起始行
int start_row;
if (rank < remain)
    start_row = rank * (rows + 1);
else
    start_row = rank * rows + remain;
```

- 在实验结果中，可以看到以下特点：**小规模矩阵时的并行效率**：对于 128×128 或更小的矩阵，串行计算本身耗时极短，而使用点对点通信时，由于需要逐个发送矩阵尺寸、B 和 A 的分片，以及逐个收集计算结果，进程间的启动、通信、同步开销会明显占据主要比例，导致总时间甚至超过串行计算时间，从而并行加速不明显甚至退化；**大规模矩阵时的加速效果**：当矩阵规模增大到 1024×1024 或 2048×2048 时，计算量大幅增加，通信开销相对计算量而言被摊薄，多进程计算能显著降低总运行时间，但由于点对点通信模式下每次发送和接收操作都存在较高的延迟和开销，实际加速比仍然小于理想的线性加速；**负载均衡对性能的影响**：通过对 A 进行负载均衡的分片，使得前 remain 个进程多分配一行，从而尽可能均衡各进程的计算负载，但在进程数很多或矩阵规模较小时，这种分配策略仍可能因为数据

块太小、通信次数过多而导致通信开销显著；**进程数过多时的通信瓶颈**：点对点通信方式需要逐个发送和接收数据，随着进程数增加，数据传输的次数和同步延迟也会增加，从而在进程数较多（如 16、32）时，即使矩阵规模很大，也会出现“通信同步时间 + 数据传输时间”迅速增长的现象，导致总运行时间不再显著下降甚至反而上升。

- 因此，对于小规模矩阵，串行或使用较少进程的计算更高效，而对于大规模矩阵，多进程并行计算能显著降低计算时间，但在点对点通信模式下，由于每个数据块的发送和接收操作都带来较大的通信开销和同步延迟，再加上负载均衡策略并非完美，进程数过多时反而会因通信瓶颈而降低总体加速效果。实验中所观察到的现象表明，点对点通信实现的并行矩阵乘法在大规模矩阵上能够获得较好的加速效果，但加速比受到通信、同步以及数据分片不均衡等因素的限制，远达不到理想的线性加速，因此在设计并行算法时需要综合考虑计算与通信的平衡问题。

```
Process 1 received matrix dimensions: m=256, n=256, k=256.
Process 1 received matrix B.
Process 1 computing 29 rows.
Process 2 received matrix dimensions: m=256, n=256, k=256.
Process 2 received matrix B.
Process 2 computing 29 rows.
Process 3 received matrix dimensions: m=256, n=256, k=256.
Process 3 received matrix B.
Process 3 computing 29 rows.
Process 4 received matrix dimensions: m=256, n=256, k=256.
Process 5 received matrix dimensions: m=256, n=256, k=256.
Process 6 received matrix dimensions: m=256, n=256, k=256.
Process 7 received matrix dimensions: m=256, n=256, k=256.
Process 8 received matrix dimensions: m=256, n=256, k=256.
Process 0 received matrix dimensions: m=256, n=256, k=256.
Process 0 sent matrix B to process 1.
Process 0 sent matrix B to process 2.
Process 0 sent matrix B to process 3.
Process 4 received matrix B.
Process 4 computing 28 rows.
Process 0 sent matrix B to process 4.
Process 0 sent matrix B to process 5.
Process 0 sent matrix B to process 6.
Process 5 received matrix B.
```

图 1: 9 线程并行日志

- 在实验中，我通过日志打印了各个进程的执行过程，可以看到输出顺序并不严格按照代码逻辑，因为在点对点通信模式下，各进程是并行执行、独立调度的，打印信息会根据各进程的执行进度和网络延迟交织在一起，这种输出杂乱是 MPI 并行程序中常见的现象，并不代表程序错误，而是反映了进程间并行执行的不确定性。

# 4 讨论题

- 在内存受限情况下，如何进行大规模矩阵乘法计算?

  回答: 当矩阵规模非常大，单个节点可能无法存储完整的矩阵，甚至单个进程的内存也可能不足。我们可以考虑使用**分块**，将矩阵划分为多个子块，按块进行矩阵乘法，减少单次计算时的内存占用。把矩阵 A 按行块划分，矩阵 B 按列块划分。让每个进程只处理 A 的一部分行和 B 的一部分列，计算局部块的结果后，再进行归约。采用 **Cannon' s Algorithm 或 SUMMA (Scalable Universal Matrix Multiplication Algorithm)** 等优化通信的并行算法。使用**异步通信与流水线**减少进程等待时间，提高计算效率。在一个进程计算当前块时，另一个进程预加载下一块数据（'MPI_Isend' / 'MPI_Irecv'），计算与数据传输重叠，避免同步等待。采用**混合并行**，结合 MPI + OpenMP，利用多核 CPU 并行计算，同时减少 MPI 进程数量，节省内存开销。让每个 MPI 进程处理一部分矩阵，并使用 OpenMP 在线程内部进行并行加速。

- 如何提高大规模稀疏矩阵乘法性能?

  回答: 对于稀疏矩阵，直接使用普通的矩阵存储和计算方式会造成大量无效计算（计算 0 元素）和存储浪费。采用稀疏矩阵存储格式，使用 **CSR (Compressed Sparse Row) 或 CSC (Compressed Sparse Column)** 形式存储稀疏矩阵，只存储非零元素，减少存储开销。在每个进程只存储非零元素，并在计算时使用索引访问而非完整矩阵遍历。'local_A[i]' 只包含实际的非零元素及其列索引，计算时按索引进行乘法累加。采用稀疏矩阵专用乘法算法: 使用 **SpMV (Sparse Matrix-Vector Multiplication)** 加速计算。采用 **SpGEMM (Sparse General Matrix-Matrix Multiplication)** 优化矩阵乘法。采用**分块存储 (Block Compressed Sparse Row, BCSR)**，使局部计算时更高效地访问数据。使用 Intel MKL 或 cuSPARSE（GPU 加速）提高计算速度。采用**非均匀负载均衡**: 由于稀疏矩阵的非零元素分布不均匀，直接均匀划分计算任务会导致部分进程计算负载远超其他进程。根据非零元素数量进行动态任务分配，而不是简单按行划分。使用任务池技术，让空闲进程从其他进程接管部分计算任务。