中山大學

SUN YAT-SEN UNIVERSITY

并行程序设计与算法

实验 1-MPI 矩阵乘法

姓 名 王志杰

学 号 22331095

学 院 计算机学院

专 业 计算机科学与技术

2025 年 3 月 26 日

目录

1 实验目的

实现基于 MPI 的并行矩阵乘法算法 $C = A \times B$，其中：

- A 为 $m \times n$ 矩阵
- B 为 $n \times k$ 矩阵
- C 为 $m \times k$ 结果矩阵

通过任务划分和进程间通信，验证并行计算的加速效果。

2 实验方法

2.1 并行策略

1. **数据划分**: 将矩阵 A 按行划分为大小相等的块（最后一块可能略大）
2. **通信模式**:
   - 主进程（Rank 0）使用 MPI 点对点通信，将矩阵 B 发送至所有从进程
   - 主进程发送 A 的行块至各从进程
   - 从进程返回计算结果块至主进程，主进程拼接结果
3. **负载均衡**: 要求 m 能被进程数整除，保证均匀分配

2.2 关键实现

核心代码结构如下（完整代码见附件）：

```
// 任务分发（主进程）
for (int p = 1; p < size; p++) {
    MPI_Send(A + p*rows_per_proc*n,
             rows_per_proc*n, MPI_FLOAT, p, 1);
}

// 本地计算（所有进程）
for (int i = 0; i < rows_per_proc; i++) {
    for (int j = 0; j < k; j++) {
        float sum = 0.0;
        for (int l = 0; l < n; l++) {
            sum += local_A[i*n + l] * B[l*k + j];
```

```
            }
            local_C[i*k + j] = sum;
        }
    }

    // 结果收集（主进程）
    memcpy(C, local_C, rows_per_proc*k*sizeof(float));
    for (int p = 1; p < size; p++) {
        MPI_Recv(C + p*rows_per_proc*k,
                 rows_per_proc*k, MPI_FLOAT, p, 2);
    }
```

如何运行程序：

```
mpicc -o matmul matmul_mpi.c -O3
mpirun -np 8 ./matmul 128 128 128 // 只作为示例，参数可改
```

3 实验结果

3.1 示例输出

如图 1所示，改变了进程数，计算结果可以保持准确（控制了相同的随机种子去生成矩阵）

3.2 结果总结

??展示了所有的进程数和矩阵规模对应实验得到的结果。其中每项单位为秒。可以注意到，矩阵规模扩大运行时间显著提高，而进程数增加则运行时间减少，这都是十分显然的，说明并行计算能够有效缩短任务执行时间。但是可以注意到，进程增加过多的时候，运行时间反倒提高了，这是因为进程间的数据交换成本随进程数呈非线性增长，另外频繁的上下文切换导致 L3 缓存命中率下降可能也有影响，这种进程间切换的成本开销已经足够大以至于使用更多进程已经不能再为计算提速了，反而会减速。

4 讨论

4.1 内存受限优化策略

4.1.1 分块计算

将大矩阵分割为内存容纳的子块，通过滑动窗口逐块加载计算，减少单次内存占用。

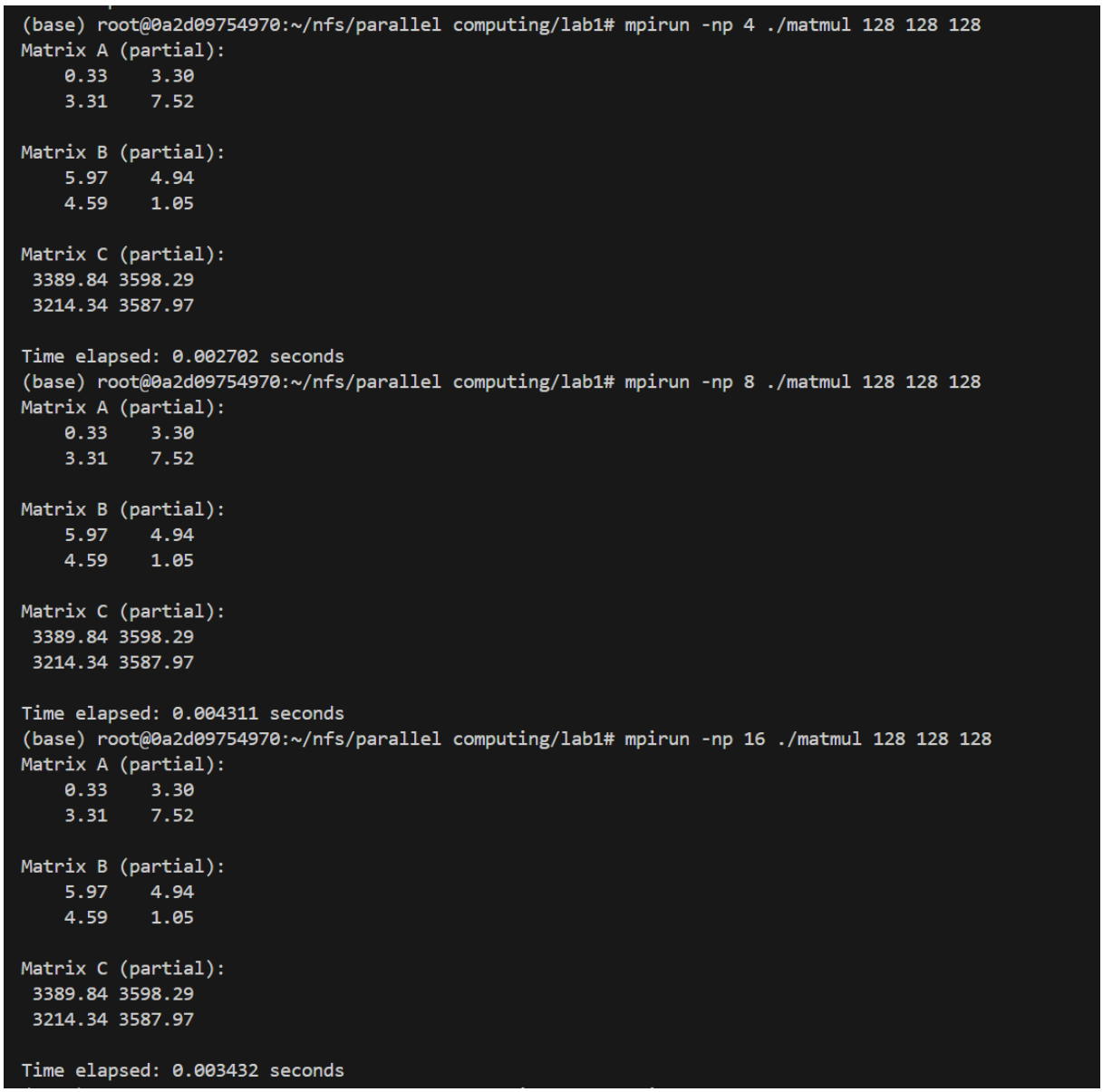

图 1: 示例输出，控制了相同的随机种子，可以看到改变进程数计算结果保持准确

进程数	矩阵规模 (n×n)				
	128	256	512	1024	2048
1	0.0028	0.0300	0.3204	2.3799	67.0500
2	0.0031	0.0251	0.1641	1.2309	35.2091
4	0.0027	0.0167	0.0859	0.6647	18.6490
8	0.0028	0.0140	0.0486	0.4409	9.7006
16	0.0034	0.0135	0.0404	0.4163	5.3492

表 1: 不同进程数与矩阵规模的执行时间（单位：秒）

4.1.2 外存交换

采用 LRU 缓存策略，将不活跃数据暂存磁盘，使用内存映射文件加速数据交换。

4.1.3 精度压缩

对矩阵元素进行半精度 (FP16) 或混合精度量化，结合误差补偿算法保持计算精度。

4.2 稀疏矩阵优化策略

4.2.1 存储格式优化

转换为 CSR/CSC 格式，仅存储非零元及其坐标，典型压缩率可达 90

4.2.2 计算路径优化

在循环中跳过零元素，采用间接寻址和非规则访存模式，减少无效计算。

4.2.3 负载重平衡

基于超图划分算法分配非零元，消除进程间的计算密度差异。