南开大学
Nankai University

南　　开　　大　　学

计 算 机 学 院

MPI 编程实验

李纡君

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 5 月 24 日

摘要

关键字：MPI，普通高斯消去，特殊高斯消去

目录

一、 普通高斯消去

(一) 问题简介

给定一个满秩矩阵，从上到下依次对它的每行进行除法（除以每行的对角线元素），然后对矩阵该行右下角 (n−k+1)×(n−k) 的子矩阵进行消去，最后得到一个三角矩阵。

(二) 实验设计方案

1. 总体思路

与之前编写的基于 `pthread` 和 `OpenMP` 的并行程序不同，MPI 编程采用消息传递式并行程序设计，适用于分布式内存体系结构。在这种体系结构中，每个系统有自己独立的地址空间，处理器必须显式通信才能共享数据。

在 MPI 编程中，进程之间由于地址空间独立，通过传递消息进行通信，这包括同步和数据移动。因此，在设计高斯消去的 MPI 并行算法时，需要重点关注进程间的同步和数据移动两个方面的实现。

(三) 理解与实现

在 MPI 并行编程中，每个进程处理数据的不同部分，为了完成高斯消去，需要以下几个步骤:

1. **行划分与分配**：将矩阵的行划分给不同的进程，每个进程负责处理和消去自己分配到的行。
2. **同步与广播**：在每一步消去过程中，负责当前主行的进程需要将主行的数据广播给其他进程，确保所有进程都能获取最新的消去信息。
3. **部分消去**：每个进程在接收到主行数据后，使用该数据对自己负责的行进行部分消去。
4. **回代计算**：在消去过程完成后，各个进程需要协同进行回代计算，以求解最终的结果。

普通高斯消去的伪代码如下

Algorithm 1 procedure LU (A)

```
0: for k = 1 to n do
0:   for j = k + 1 to n do
0:     A[k, j] ← A[k, j]/A[k, k]
0:   end for
0:   A[k, k] ← 1.0
0:   for i = k + 1 to n do
0:     for j = k + 1 to n do
0:       A[i, j] ← A[i, j] − A[i, k] × A[k, j]
0:     end for
0:     A[i, k] ← 0
0:   end for
0: end for=0
```

在分析算法的依赖关系后可以得知，对于每一个消去步，首先需要对矩阵中对应的行进行除法运算，使得对角线元素为 1，然后才能进行后续的消去操作。即利用刚完成除法运算的这一行对其下方的所有行进行消去操作，使得这些行的指定列全为 0。

在设计高斯消去的 MPI 程序时，可以让每个进程负责指定几行的除法和消去操作。对于特定进程，任务分为除法和消去两个部分。在进程函数中，我们仍然保持原程序框架进行循环遍历。对于除法循环，当最外层循环步遍历到该进程的任务时，进行除法运算，并将运算结果发送给其他进程；若当前循环步不是该进程的任务，则接收其他进程传来的消息。对于消去部分，在每个循环步内完成除法部分（除法运算 + 发送消息或接收消息）后，进行属于该进程所分配的行的消去运算。

上述步骤是每个进程内的具体运算过程。对于进程间的消息传递，可以采用主从式编程模型。首先由 0 号进程为其他进程分配任务，并将其他进程所需的矩阵数据发送给它们。然后 0 号进程执行自己的任务，并在完成后接收其他进程的计算结果，整个程序执行完毕。其他进程先接收任务，然后进行运算，运算完成后将结果传回 0 号进程。

高斯消去的 MPI 算法框架如下，基于此框架可以进行多种算法和实现方式的探究。

1. 设计思路

本小节概述了实验的总体设计思路，具体的实现细节见后续的实验内容和算法实现部分。

与 SIMD 和多线程 OpenMP 的结合 鉴于 MPI 基于分布式内存架构，每个系统拥有独立的地址空间，并且每个进程可以启动多个线程，我们在本实验中结合使用了 OpenMP 和 SIMD 技术。在 MPI 的基础上引入 OpenMP 和 SIMD，可以显著提升计算效率和性能。因此，本实验设计旨在通过结合 MPI、OpenMP 和 SIMD 进行性能优化与对比分析。

块划分与循环划分 针对矩阵在各进程间的分配方式，本实验采用了一维块划分和循环划分的方法。块划分将矩阵按照问题规模与进程数的比值划分为若干块，并将每个块分配给相应的进程。循环划分则按行进行，将矩阵从上到下依次一行行地分配给各进程，一轮分配完毕后再循环分配，直至所有行分配完毕。

流水线算法 为优化进程间的数据传递，我们设计了流水线算法。该算法将进程形成逻辑链条，当进程 P_k 需要广播数据时，不是立即将数据发送给所有其他进程，而是首先转发给下一个进程 P_{k+1}，再由 P_{k+1} 转发给 P_{k+2}，依次类推。这样 P_{k+1} 无需等待所有其他进程收到数据，即可开始消去计算，从而实现流水线并行，提升计算效率。

不同平台 MPI 支持多种操作系统和硬件架构。我们在本实验中分别在鲲鹏服务器、金山云服务器和笔记本电脑等不同平台上进行 MPI 实验，并对比分析其性能表现。

其他策略 在块划分时，我们发现，在数据传递过程中，由于进程数据的次序关系，无需将除法结果传递给所有进程，而只需传递给部分需要的进程。这一策略可以显著减少通信开销，从而加速程序执行。（具体实现见后续实验内容部分）

2. 设计思路分析

与 SIMD 和多线程 OpenMP 的结合 鉴于 MPI 基于分布式内存架构，每个系统拥有独立的地址空间，并且每个进程可以启动多个线程，我们在本实验中结合使用了 OpenMP 和 SIMD 技术。在 MPI 的基础上引入 OpenMP 和 SIMD，可以显著提升计算效率和性能。

Algorithm 2 procedure LU (A) for MPI

Input: 矩阵 $A[n,n]$，问题规模 n

Output: 上三角矩阵 $A[n,n]$

```
0: function LU(rank, num_proc)
0:   根据 rank 计算自己进程的任务范围
0:   for k = 1 to n do
0:     if 当前行是自己进程的任务 then
0:       A[k,j] ← A[k,j]/A[k,k]
0:       向其他进程发送消息
0:     else
0:       接收其他进程传来的消息
0:     end if
0:     for i = k + 1 to n do
0:       if 当前行是自己进程的任务 then
0:         for j = k + 1 to n do
0:           A[i,j] ← A[i,j] − A[i,k] × A[k,j]
0:         end for
0:         A[i,k] ← 0
0:       end if
0:     end for
0:   end for
0: end function
0: function MAIN
0:   MPI_Comm_size (MPI_COMM_WORLD, &num_proc)
0:   MPI_Comm_rank (MPI_COMM_WORLD, &rank)
0:   if rank == 0 then
0:     任务划分
0:     LU(rank, num_proc)
0:     接收其他进程的计算结果
0:   else
0:     接收从 0 号进程分配的任务
0:     LU(rank, num_proc)
0:     将运算结果传回 0 号进程
0:   end if
0: end function
   =0
```

优势

- **并行度提高**：通过结合 OpenMP 和 SIMD，每个进程内可以利用多线程并行和单指令多数据并行，充分利用多核处理器的计算能力。
- **通信开销降低**：MPI 进程间的通信开销较大，通过 OpenMP 和 SIMD，可以在每个进程内进行高效的并行计算，减少进程间的通信频率。

时间复杂度 假设问题规模为 N，进程数为 P，每个进程的线程数为 T。在理想情况下，时间复杂度可以从 $O(N^3)$ 降低到 $O\left(\frac{N^3}{P\times T}\right)$，显著提升计算效率。

块划分与循环划分 针对矩阵在各进程间的分配方式，本实验采用了一维块划分和循环划分的方法。

块划分

- **设计思路**：将矩阵按照问题规模与进程数的比值划分为若干块，并将每个块分配给相应的进程。
- **优势**：块划分可以保证每个进程处理相对连续的大块数据，减少通信开销，适合大规模矩阵的分布式计算。
- **时间复杂度**：在块划分中，每个进程处理的数据量约为 $\frac{N^2}{P}$，总时间复杂度为 $O\left(\frac{N^3}{P}\right)$。

循环划分

- **设计思路**：按行将矩阵从上到下依次一行行地分配给各进程，一轮分配完毕后再循环分配，直至所有行分配完毕。
- **优势**：循环划分可以更均匀地分配计算任务，避免负载不均的情况，提高资源利用率。
- **时间复杂度**：在循环划分中，数据分配更加均匀，总时间复杂度同样为 $O\left(\frac{N^3}{P}\right)$。

相比于块划分，循环划分具有两方面优势：

- **任务均衡性**：对于高斯消去程序来讲，每个循环步的任务大小是不均衡的，越靠近矩阵右下角的工作量越大，越靠近矩阵左上角的工作量越小。若采取块划分，则对于每个进程来讲，循环步 k 超过某个界限 k_0 之后，该进程将没有运算可做，处于空闲状态。
- **负载均衡**：当将任务分配给不同进程时，由于进程数未必能整除矩阵的行数，余数部分会被分配给某个或某几个进程，导致负载不均。高斯运算的特点是越靠近矩阵右下角的进程负载越大，这会产生较严重的负载不均衡。

块划分的时间复杂度 对于第 k 个消去步骤，一个进程最多进行 $\frac{(n-k-1)n}{p}$ 次乘法和除法，并行时间大致为 $\frac{2n^3}{p}$。由于在没有自己进程任务的循环步内，每个进程要等它前面的进程完成除法才能进行消去，因此产生了进程空闲，导致了“串行”的效果。共有 p 个进程，所以总的时间复杂度为 $O(n^3)$。

循环划分的时间复杂度 在块划分中，总的进程空闲时间达到 $O(n^3)$，而在循环划分中，进程间负载差距最多为一行元素，一行元素所花费的时间是线性的 $O(n)$。因此，对于空闲等待时间来说，共有 n 个循环步，每个循环步中 p 个处理器各有 $O(n)$ 时间的空闲等待，总的进程空闲时间最多为 $O(n^2p)$。

流水线算法 为优化进程间的数据传递，我们设计了流水线算法。

设计思路 将进程形成逻辑链条，当进程 P_k 需要广播数据时，不是立即将数据发送给所有其他进程，而是首先转发给下一个进程 P_{k+1}，再由 P_{k+1} 转发给 P_{k+2}，依次类推。

优势

- **并行计算**：每个进程在接收到数据后即可开始计算，无需等待所有进程都收到数据，从而实现并行计算。
- **通信效率高**：减少了每次广播的通信开销，使得数据传递更加高效。

时间复杂度 在流水线算法中，共有 n 个消去步骤，每个步骤的启动间隔是常量个循环步。而由于最后一个循环步只需要对一个元素进行运算，因此近似将最后一个循环步的启动时间作为程序结束时间。因此，分析性能的时候，只要分析单个循环步所用的时间，即可得出每两个循环步启动时间的间隔，也就可以算出总共所用的时间。而在一个循环步内，要进行与元素个数呈线性关系（$O(n)$）的除法、转发、消去操作。因此每两个循环步启动时间之间的间隔是 $O(n)$，由于有 n 个循环步，所以总的并行时间是 $O(n^2)$。有 n 个处理器，故总代价为 $O(n^3)$。

不同平台 MPI 支持多种操作系统和硬件架构。我们在本实验中分别在鲲鹏服务器、金山云服务器和笔记本电脑等不同平台上进行 MPI 实验，并对比分析其性能表现。

设计思路

- **多平台适应性**：验证 MPI 在不同硬件平台上的适应性和性能差异。
- **性能对比**：通过在不同平台上运行相同的实验，分析计算性能和通信效率的差异。

优势

- **广泛适用性**：验证 MPI 在多种平台上的可用性和高效性。
- **优化方向**：通过性能对比，发现不同平台的优化方向，为未来的系统优化提供参考。

时间复杂度 不同平台的硬件性能和通信效率不同，但总体时间复杂度仍为 $O\left(\frac{N^3}{P}\right)$，具体性能表现需根据实验结果进行分析。

其他策略——块划分的改进 在块划分时，我们发现，在数据传递过程中，由于进程数据的次序关系，无需将除法结果传递给所有进程，而只需传递给部分需要的进程。

设计思路 根据进程间的数据依赖关系，优化数据传递策略，仅将必要的数据传递给需要的进程，减少不必要的通信开销。

优势

- **减少通信开销**：通过优化数据传递策略，显著减少进程间的通信开销，提高计算效率。
- **加速程序执行**：降低通信延迟，使程序执行更加高效。

3. 实验内容及算法实现

结合 SIMD 和多线程 在之前的实验中，我们将矩阵行分别交给不同的线程进行处理，并在除法和消去之间设置同步机制。而在 MPI 中，我们将矩阵行分管给不同的进程处理，在每个进程内部，又可以启动多个线程，从而加大了并发度。在每个进程内部，由于不同进程处理的行数有限，且不一定连续，因此我们考虑按列进行多线程划分，即对于除法和消去的最内层循环分别进行 OpenMP 运算。而对于 SIMD 的实现，则和前面的实验一样，对内层循环的运算变量进行向量化运算。我们实现了对上述几种方法的 SIMD+OpenMP 优化，但受篇幅限制，在报告中只展示基于块划分的结合（省略无关部分）：

SIMD 及 OpenMP 更改部分

```
void LU_opt(float A[][N], int rank, int num_proc) {
    __m128 t1, t2, t3;
    ... //初始化 begin、block、end
    #pragma omp parallel num_threads(thread_count), private(t1, t2, t3)
    for (int k = 0; k < N; k++) {
        if (k >= begin && k < end) {
            float temp1[4] = {A[k][k], A[k][k], A[k][k], A[k][k]};
            t1 = _mm_loadu_ps(temp1);
            #pragma omp for schedule(static)
            for (int j = k + 1; j < N - 3; j += 4) {
                t2 = _mm_loadu_ps(&A[k][j]);
                t3 = _mm_div_ps(t2, t1);
                _mm_storeu_ps(&A[k][j], t3);
            }
            for (int j = N - N % 4; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
            for (int p = rank + 1; p < num_proc; p++)
                MPI_Send(&A[k], N, MPI_FLOAT, p, 2, MPI_COMM_WORLD);
        } else {
            ... //接收其他进程传递的消息
        }
        for (int i = begin; i < end && i < N; i++) {
            if (i >= k + 1) {
                float temp2[4] = {A[i][k], A[i][k], A[i][k], A[i][k]};
                t1 = _mm_loadu_ps(temp2);
                #pragma omp for schedule(static)
                for (int j = k + 1; j <= N - 3; j += 4) {
                    t2 = _mm_loadu_ps(&A[i][j]);
                    t3 = _mm_loadu_ps(&A[k][j]);
                    t3 = _mm_mul_ps(t1, t3);
                    t2 = _mm_sub_ps(t2, t3);
                    _mm_storeu_ps(&A[i][j], t2);
                }
                for (int j = N - N % 4; j < N; j++)
                    A[i][j] = A[i][j] - A[i][k] * A[k][j];
                A[i][k] = 0;
```

```
            }
        }
    }
}
```

LU 函数解释 函数 `LU_opt(float A[][N], int rank, int num_proc)` 是一个优化的 LU 分解函数，使用 SIMD 指令和 OpenMP 来并行化和加速计算过程。

- **参数:**
  - `float A[][N]`：待分解的矩阵，大小为 `N x N`。
  - `int rank`：当前进程的编号。
  - `int num_proc`：总共的进程数。
- **过程:**
  1. **并行初始化：**使用 `OpenMP` 定义并行区域，设置使用的线程数和私有变量。
  2. **主循环：**
     - **局部消元：**对每个线程负责的行区间内的数据进行消元处理。
     - **SIMD 指令：**使用 SIMD 指令集进行向量化计算，加速四个连续浮点数的处理。
     - **通信：**完成每行的处理后，通过 MPI 发送当前行给其他进程。
  3. **同步更新：**接收其他进程传递的更新后的行数据。

块划分

基于块划分的 MPI 算法如下

```
#define N 1000 // 假设矩阵大小为1000，可以根据需要调整

void LU(float A[][N], int rank, int num_proc) {
    // 计算每个进程被划分任务的起始行号（最后一个进程要包括划分余数）
    int block = N / num_proc;
    int remain = N % num_proc;
    int begin = rank * block;
    int end = (rank != num_proc - 1) ? (begin + block) : (begin + block +
        remain);

    for (int k = 0; k < N; k++) {
        // 当前行是自己进程的任务——进行消去
        if (k >= begin && k < end) {
            for (int j = k + 1; j < N; j++)
                A[k][j] = A[k][j] / A[k][k];
            A[k][k] = 1.0;
            // 发送消息（向所有其他进程）
            for (int p = 0; p < num_proc; p++) {
                if (p != rank)
                    MPI_Send(&A[k], N, MPI_FLOAT, p, 2, MPI_COMM_WORLD);
            }
```

```
        }
        // 当前行不是自己进程的任务——接收消息
        else {
            // 接收消息（接收所有其他进程的消息）
            int cur_p = k / block;
            MPI_Recv(&A[k], N, MPI_FLOAT, cur_p, 2, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
        // 消去部分
        for (int i = begin; i < end && i < N; i++) {
            if (i >= k + 1) {
                for (int j = k + 1; j < N; j++)
                    A[i][j] = A[i][j] - A[i][k] * A[k][j];
                A[i][k] = 0.0;
            }
        }
    }
}

void f_mpi() {
    struct timeval t_start, t_end;
    int num_proc; // 进程数
    int rank; // 识别调用进程的 rank, 值从 0 ~ size - 1

    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int block = N / num_proc;
    int remain = N % num_proc;

    // 0号进程——任务划分
    if (rank == 0) {
        float A[N][N]; // 矩阵初始化
        reset_A(A); // 假设reset_A是一个初始化矩阵A的函数
        gettimeofday(&t_start, NULL);

        // 任务划分
        for (int i = 1; i < num_proc; i++) {
            int start_row = i * block;
            int rows_to_send = (i != num_proc - 1) ? block : (block + remain)
                ;
            MPI_Send(&A[start_row], rows_to_send * N, MPI_FLOAT, i, 0,
                MPI_COMM_WORLD);
        }

        LU(A, rank, num_proc);

        // 处理完0号进程自己的任务后需接收其他进程处理之后的结果
```

```
66          for (int i = 1; i < num_proc; i++) {
67              int start_row = i * block;
68              int rows_to_recv = (i != num_proc - 1) ? block : (block + remain)
                    ;
69              MPI_Recv(&A[start_row], rows_to_recv * N, MPI_FLOAT, i, 1,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
70          }
71
72          gettimeofday(&t_end, NULL);
73          std::cout << "Block␣MPI␣LU␣time␣cost:␣"
74                    << 1000 * (t_end.tv_sec - t_start.tv_sec) +
75                        0.001 * (t_end.tv_usec - t_start.tv_usec) << "ms" << std
                            ::endl;
76      } else {
77          // 其他进程接收任务
78          float A[N][N]; // 矩阵初始化
79          int start_row = rank * block;
80          int rows_to_recv = (rank != num_proc - 1) ? block : (block + remain);
81          MPI_Recv(&A[start_row], rows_to_recv * N, MPI_FLOAT, 0, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
82
83          LU(A, rank, num_proc);
84
85          // 非0号进程完成任务之后，将结果传回到0号进程
86          MPI_Send(&A[start_row], rows_to_recv * N, MPI_FLOAT, 0, 1,
                MPI_COMM_WORLD);
87      }
88  }
```

LU 函数解释 在并行计算中，任务的合理划分对提高计算效率至关重要。我们通过计算每个进程负责的矩阵行号范围来实现任务划分：

- **block**：每个进程处理的基础行数，由总行数 N 除以进程数 num_proc 计算得出。
- **remain**：划分后剩余的行数，由 N 对 num_proc 取余计算得出。
- **begin**：当前进程负责处理的起始行号，由进程号 rank 乘以 block 计算得出。
- **end**：当前进程负责处理的结束行号。对于非最后一个进程，其值为 begin + block；对于最后一个进程，其值为 begin + block + remain，以确保所有行都被处理。

4. 消去步骤

每个进程对其负责的行进行高斯消去：

- **消去当前行**：对于每个进程负责的行，从 k+1 列开始逐元素处理，通过将当前元素除以主对角线元素 A[k][k] 完成消去操作。主对角线元素 A[k][k] 被设置为 1。
- **发送消息**：当前进程完成消去后，将结果行发送给所有其他进程。这一步确保所有进程都能获取最新的消去结果，用于后续计算。

5. 消息传递

如果当前行不属于该进程负责的范围，则该进程需要从负责该行的进程接收消息：

- **接收消息**：根据当前行号 k 计算负责该行的进程号 cur_p，并从该进程接收消息。消息包含了最新的消去结果行 A[k]。

6. 矩阵更新

在接收到消去结果后，各进程需要更新其负责处理的矩阵部分：

- **更新矩阵**：对每个进程负责的行，从 k+1 行开始，逐元素更新矩阵。更新操作包括将当前元素减去主对角线元素 A[k][j] 与已消去元素 A[i][k] 的乘积。主对角线元素 A[i][k] 被设置为 0。

通过上述步骤，我们实现了基于 MPI 的 LU 分解算法，有效地将任务分配到不同的进程，并通过消息传递和矩阵更新确保计算结果的一致性和正确性。

$\mathbf{f}_mpi$

- **MPI 初始化**：获取进程数和进程标识。
- **任务分配**：0 号进程将矩阵块分配给其他进程，并负责初始化和最终的结果汇总。
- **计算与通信**：各进程进行高斯消去计算，并在计算完成后将结果发送回 0 号进程。
- **性能测量**：0 号进程记录并输出整个计算过程的时间开销。

循环划分

基于循环划分的代码如下 块划分和循环划分的区别在于每个进程被分配的任务不同。

在块划分中，对于每个进程来说，它的任务起始行号是进程号 * 块的大小，而结束行号是起始行号加上块的大小，由于有余数，因此最后一个进程的结束行号是矩阵的最后一行。

在循环划分中，我们顺次一行一行式划分。在数学角度，可以根据行号相对进程数的余数来归类。具体代码实现如下（只体现了与块划分不同的部分）

```
void LU(float A[][N], int rank, int num_proc) {
    for (int k = 0; k < N; k++) {
        // 当前行是自己进程的任务——进行除法
        if ((k % num_proc) == rank) {
            // Perform division for the current row
            for (int j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;

            // Broadcast the current row to all other processes
            for (int p = 0; p < num_proc; p++) {
                if (p != rank) {
                    MPI_Send(&A[k], N, MPI_FLOAT, p, 2, MPI_COMM_WORLD);
                }
```

```
            }
        } else {
            // 当前行不是自己进程的任务——接收消息
            // 接收来自进程号为 (k % num_proc) 的消息
            MPI_Recv(&A[k], N, MPI_FLOAT, (k % num_proc), 2, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }

        // 消去当前行以下的所有行
        for (int i = k + 1; i < N; i++) {
            if ((i % num_proc) == rank) {
                for (int j = k + 1; j < N; j++) {
                    A[i][j] -= A[i][k] * A[k][j];
                }
                A[i][k] = 0.0;
            }
        }
    }
}

void f_mpi() {
    int rank, num_proc;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

    // Initialize matrix A (skipped for brevity)
    // ...

    // 0号进程——任务划分
    if (rank == 0) {
        // 分配任务给其他进程
        for (int i = 0; i < N; i++) {
            // 根据行号除进程数的余数进行划分
            int flag = i % num_proc;
            if (flag == rank) {
                continue;
            } else {
                MPI_Send(&A[i], N, MPI_FLOAT, flag, 0, MPI_COMM_WORLD);
            }
        }
        LU(A, rank, num_proc);

        // 接收所有进程计算结果
        for (int i = 0; i < N; i++) {
            // 根据行号除进程数的余数接收结果
            int flag = i % num_proc;
            if (flag == rank) {
                continue;
```

```
63              } else {
64                  MPI_Recv(&A[i], N, MPI_FLOAT, flag, 1, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
65              }
66          }
67      } else {
68          // 非0号进程先接收任务
69          for (int i = rank; i < N; i += num_proc) {
70              // 每间隔 num_proc 行接收一行数据
71              MPI_Recv(&A[i], N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
72          }
73          LU(A, rank, num_proc);
74
75          // 非0号进程完成任务之后，将结果传回到 0 号进程
76          for (int i = rank; i < N; i += num_proc) {
77              // 每间隔 num_proc 行发送一行数据
78              MPI_Send(&A[i], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
79          }
80      }
81      // Finalize MPI (skipped for brevity)
82      // ...
83  }
```

LU 函数解释 该函数用于执行 LU 分解。对于每个消去步骤，首先对当前行进行除法运算，使得对角线元素为 1，然后将当前行广播给其他进程。每个进程接收到当前行后，使用该行对其负责的部分矩阵进行消去。

- **行分配与除法运算**:
  - 如果当前行属于当前进程，进行除法运算。
  - 将处理后的行广播给其他进程。
- **接收消息与消去运算**:
  - 如果当前行不属于当前进程，则接收对应进程传来的处理后行数据。
  - 使用接收到的行数据进行消去运算。

$\mathbf{f}_mpi$

- **0 号进程任务划分**:
  - 将矩阵行划分给其他进程，自己保留一部分。
  - 调用 LU 函数进行 LU 分解。
  - 收集其他进程计算结果。
- **其他进程任务处理**:
  - 接收 0 号进程分配的任务。

- 调用 LU 函数进行 LU 分解。
- 将计算结果传回 0 号进程。

流水线算法 在流水线算法中，我们也采用了循环划分，与上述循环代码大部分重合，区别在于消息传递时，流水线算法需计算前一个进程和后一个进程，并只将任务传给下一个进程。

MPI 流水线算法代码

```
void LU(float A[][N], int rank, int num_proc) {
    // 计算当前进程的前一进程及下一进程
    int pre_proc = (rank + (num_proc - 1)) % num_proc;
    int next_proc = (rank + 1) % num_proc;

    for (int k = 0; k < N; k++) {
        // 如果当前行是自己的任务，则进行除法
        if ((k % num_proc) == rank) {
            // Perform division for the current row
            for (int j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;

            // 处理完自己的任务后向下一进程发送消息
            MPI_Send(&A[k], N, MPI_FLOAT, next_proc, 2, MPI_COMM_WORLD);
        } else {
            // 如果当前行不是当前进程的任务，则接收前一进程的消息
            MPI_Recv(&A[k], N, MPI_FLOAT, pre_proc, 2, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

            // 如果当前行不是下一进程的任务，需将消息进行传递
            if ((k % num_proc) != next_proc) {
                MPI_Send(&A[k], N, MPI_FLOAT, next_proc, 2, MPI_COMM_WORLD);
            }
        }

        // 消去当前行以下的所有行
        for (int i = k + 1; i < N; i++) {
            if ((i % num_proc) == rank) {
                for (int j = k + 1; j < N; j++) {
                    A[i][j] -= A[i][k] * A[k][j];
                }
                A[i][k] = 0.0;
            }
        }
    }
}

void f_mpi() {
    int rank, num_proc;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

    // Initialize matrix A (skipped for brevity)
    // ...

    // 0号进程——任务划分
    if (rank == 0) {
        // 分配任务给其他进程
        for (int i = 0; i < N; i++) {
            // 根据行号除进程数的余数进行划分
            int flag = i % num_proc;
            if (flag == rank) {
                continue;
            } else {
                MPI_Send(&A[i], N, MPI_FLOAT, flag, 0, MPI_COMM_WORLD);
            }
        }
        LU(A, rank, num_proc);

        // 接收所有进程计算结果
        for (int i = 0; i < N; i++) {
            // 根据行号除进程数的余数接收结果
            int flag = i % num_proc;
            if (flag == rank) {
                continue;
            } else {
                MPI_Recv(&A[i], N, MPI_FLOAT, flag, 1, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
        }
    } else {
        // 非0号进程先接收任务
        for (int i = rank; i < N; i += num_proc) {
            // 每间隔 num_proc 行接收一行数据
            MPI_Recv(&A[i], N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
        LU(A, rank, num_proc);

        // 非0号进程完成任务之后，将结果传回到 0 号进程
        for (int i = rank; i < N; i += num_proc) {
            // 每间隔 num_proc 行发送一行数据
            MPI_Send(&A[i], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
        }
    }
    // Finalize MPI (skipped for brevity)
    // ...
```

```
87 }
```

LU 函数解释 该函数用于执行 LU 分解，按照行的划分进行并行计算。每个进程负责某些特定的行（依据行号对进程数量取模决定）。

1. 计算当前进程的前一进程和下一进程的编号。
2. 遍历每一行：
   (a) 如果当前行是当前进程的任务，执行除法运算并将结果发送给下一进程。
   (b) 如果当前行不是当前进程的任务，接收来自前一进程的消息，如果当前行也不是下一进程的任务，则继续传递消息。
3. 对当前行以下的所有行进行消去操作。

fmpi 函数解释 该函数用于初始化 MPI 环境，并进行任务划分和结果收集。

1. 初始化 MPI 环境，获取当前进程的 `rank` 和总的进程数。
2. 如果是 0 号进程：
   (a) 分配任务给其他进程，发送矩阵的行数据。
   (b) 调用 `LU` 函数进行 LU 分解。
   (c) 收集其他进程的计算结果。
3. 如果是非 0 号进程：
   (a) 接收 0 号进程分配的任务。
   (b) 调用 `LU` 函数进行 LU 分解。
   (c) 将计算结果发送回 0 号进程。

其它策略: 改良版块划分 在理论分析中，我们提到，第 k 个循环步进行的除法只会影响到它后面循环步的消去操作，因此在改良版的块划分中，我们在第 k 个循环步的除法完成后，只将第 k 行传给该进程之后的进程；同样地，在消去之前，只接收该进程前面的进程传来的消息。而在实现时，只需要更改原代码在消息传递的代码，具体更改如下：

改良版代码

```
void LU(float A[][N], int rank, int num_proc) {
    //计算每个进程被划分任务的起始行号（最后一个进程要包括划分余数）
    ...
    for (int k = 0; k < N; k++) {
        //当前行是自己进程的任务——进行消去
        if (k >= begin && k < end) {
            ...
            //发送消息（向本进程后面的进程）
            for (int p = rank + 1; p < num_proc; p++)
                MPI_Send(&A[k], N, MPI_FLOAT, p, 2, MPI_COMM_WORLD);
        }
        //当前行不是自己进程的任务——接收消息
```

```
        else {
            //接收消息（接收位于自己前面的进程的消息）
            int cur_p = k / block;
            if (cur_p < rank)
                MPI_Recv(&A[k], N, MPI_FLOAT, cur_p, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        }
        //消去部分
        ...
    }
}
```

LU 函数解释 函数 `LU(float A[][N], int rank, int num_proc)` 是一个用于进行矩阵的 LU 分解的并行函数，主要利用 MPI 库进行多进程间的通信和数据处理。

- **参数:**
  - `float A[][N]`：待分解的矩阵，大小为 `N x N`。
  - `int rank`：当前进程的编号。
  - `int num_proc`：总共的进程数。
- **过程:**
  1. **划分任务：**计算每个进程分配到的行的起始行号和终止行号，最后一个进程需要处理剩余的行。
  2. **消去步骤：**
     - **发送操作：**如果当前行属于该进程处理的部分，则进行 LU 消去操作，并将处理后的行发送给后面的所有进程。
     - **接收操作：**如果当前行不属于该进程，则从负责该行的进程接收更新后的数据。
  3. **同步和更新：**所有进程同步，确保每次迭代结束时所有进程都有最新的行数据。

7. 实验结果及分析

不同进程数对比

实验数据 在金山云服务器上，我们对进程数分别为 2、4、8 进行了实验，对比不同进程数下的结果。下面以基于块划分的普通 MPI 程序与结合 SIMD+OpenMP 的 MPI 程序为例，在不同问题规模和进程数下，对比结果如下。

普通 MPI 程序的结果如表 1，结合 SIMD 和 OpenMP 的结果如表 2。

表 1: 通过 MPI 性能

启动进程数	1000	2000	3000
单行运算 / ms	980	7905	26750
进程数 =2 / ms	885	7780	23800
进程数 =4 / ms	490	4090	15050
进程数 =8 / ms	295	2220	7490

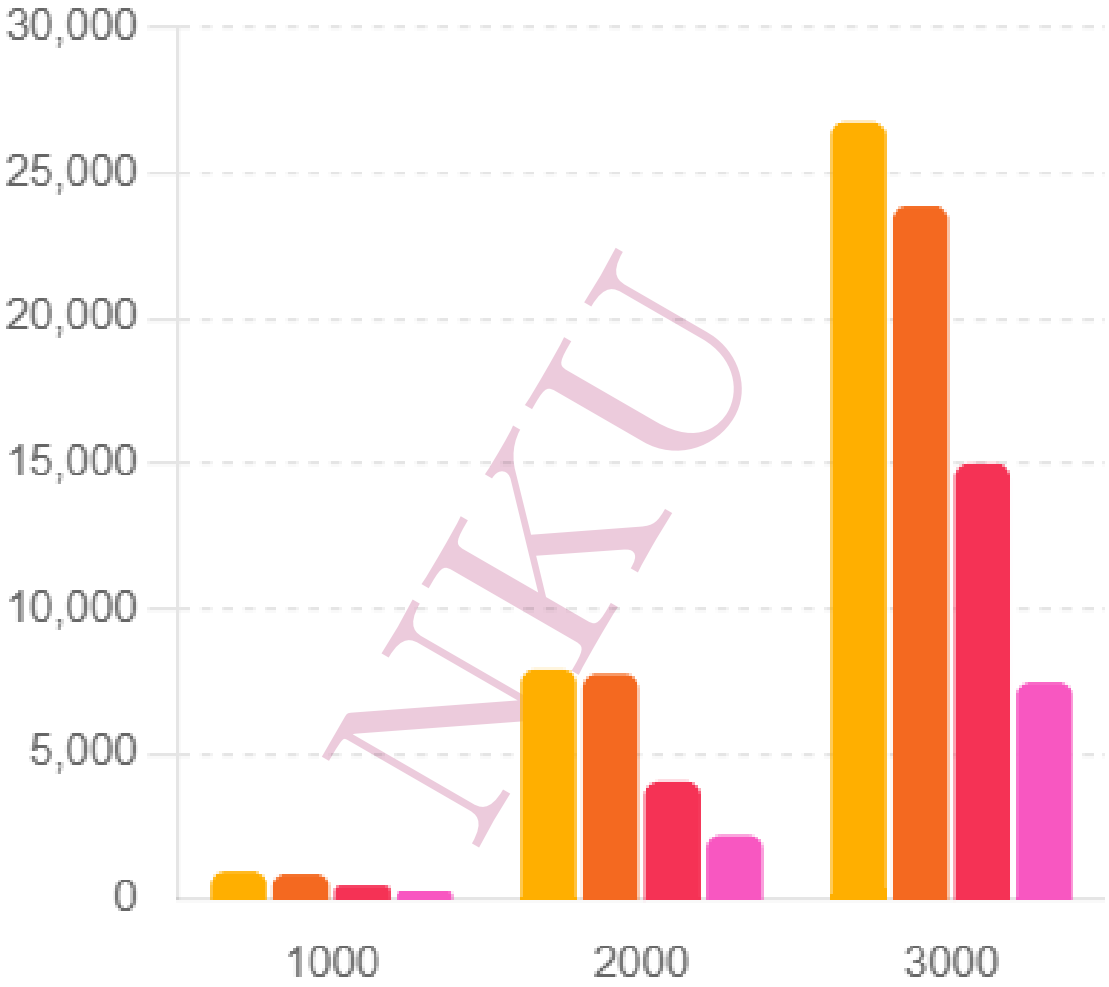

图 1: Caption

问题规模	1000	2000	3000
串行程序 /ms	978.35	7901.05	26747.5
进程数 =2 /ms	482.71	4405.76	14054.3
进程数 =4 /ms	265.85	2661.22	7772.56
进程数 =8 /ms	142.51	1065.92	3619.66

表 2: MPI 结合 SIMD+OpenMP

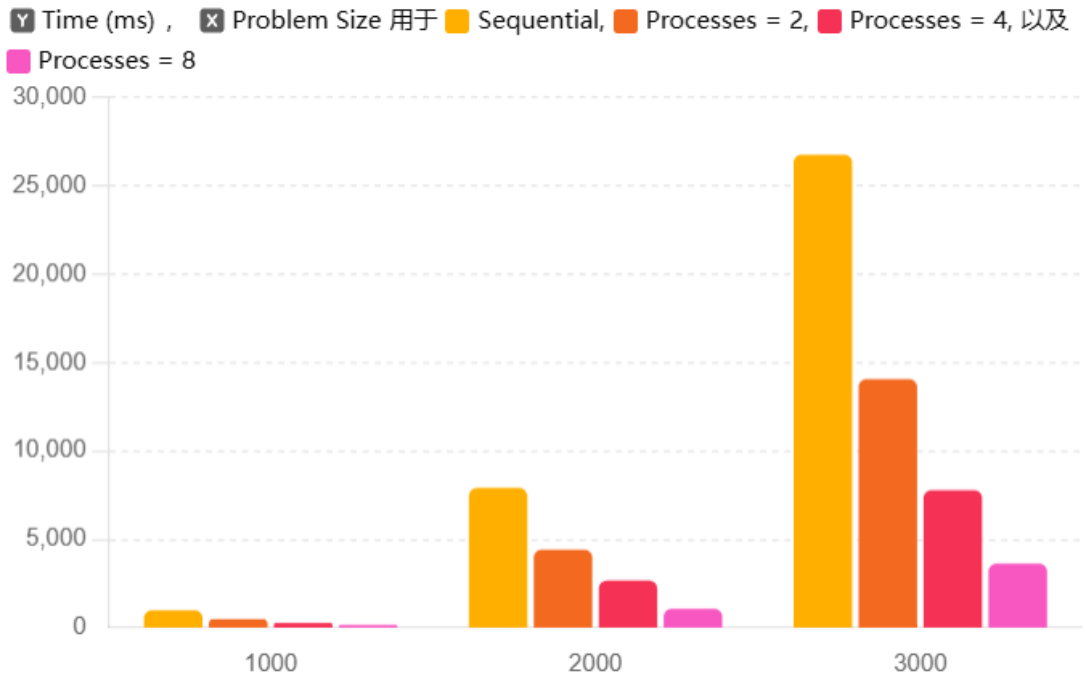

图 2: Caption

进程数增加的影响 在两个表格中，随着进程数的增加，执行时间显著减少。更多的进程能够更好地利用多核资源，分担计算负载，减少每个进程的工作量。在表 1 中，进程数从 2 增加到 8 时，1000 的问题规模下时间从 882.04 ms 降至 290.83 ms，2000 的问题规模下从 7758.84 ms 降至 2223.3 ms，3000 的问题规模下从 23858.5 ms 降至 7483.22 ms。在表 2 中，进程数从 2 增加到 8 时，1000 的问题规模下时间从 482.71 ms 降至 142.51 ms，2000 的问题规模下从 4405.76 ms 降至 1065.92 ms，3000 的问题规模下从 14054.3 ms 降至 3619.66 ms。

MPI 结合 SIMD + OpenMP 的效果 表 2 中显示了通过结合 SIMD 和 OpenMP 技术进一步优化后的结果。与仅使用 MPI（表 1）相比，这些技术的结合显著降低了执行时间。例如，对于 1000 的问题规模，使用 8 个进程时，表 1 中的时间为 290.83 ms，而表 2 中的时间为 142.51 ms。这表明，SIMD 和 OpenMP 的结合大大提高了效率。

并行效率 总体来看，结合 SIMD 和 OpenMP 技术的程序表现出更高的并行效率。在所有进程数和问题规模下，表 2 中的时间都显著低于表 1 中的时间。这表明，多重优化技术在大规模计算中的重要性。

总结 通过数据可以看出，随着进程数的增加，执行时间显著减少。而通过结合 SIMD 和 OpenMP 技术进一步优化，可以在所有问题规模和进程数下显著降低执行时间，提高并行效率。因此，结合多种并行化技术是提高计算性能的有效手段。

不同划分方式的对比 在进程数 =4、8 的条件下，我们以块划分和循环划分的方式对不同问题规模进行了实验，结果如表 4（进程数 =4）和表 3（进程数 =8）。

问题规模	1000	2000	3000	4000
串行程序 /ms	978.35	7901.05	26747.5	61499.2
块划分 /ms	290.83	2223.3	7483.22	17826.4
循环划分 /ms	261.67	1708.6	5836.91	12884.5

表 3: 不同划分方式的对比（进程数为 8）

问题规模	1000	2000	3000	4000
串行程序 /ms	978.35	7901.05	26747.5	61499.2
块划分 /ms	484.82	4058.77	15001.6	30123.1
循环划分 /ms	438.42	3141.12	10391.3	23313.3

表 4: 不同划分方式的对比（进程数为 4）

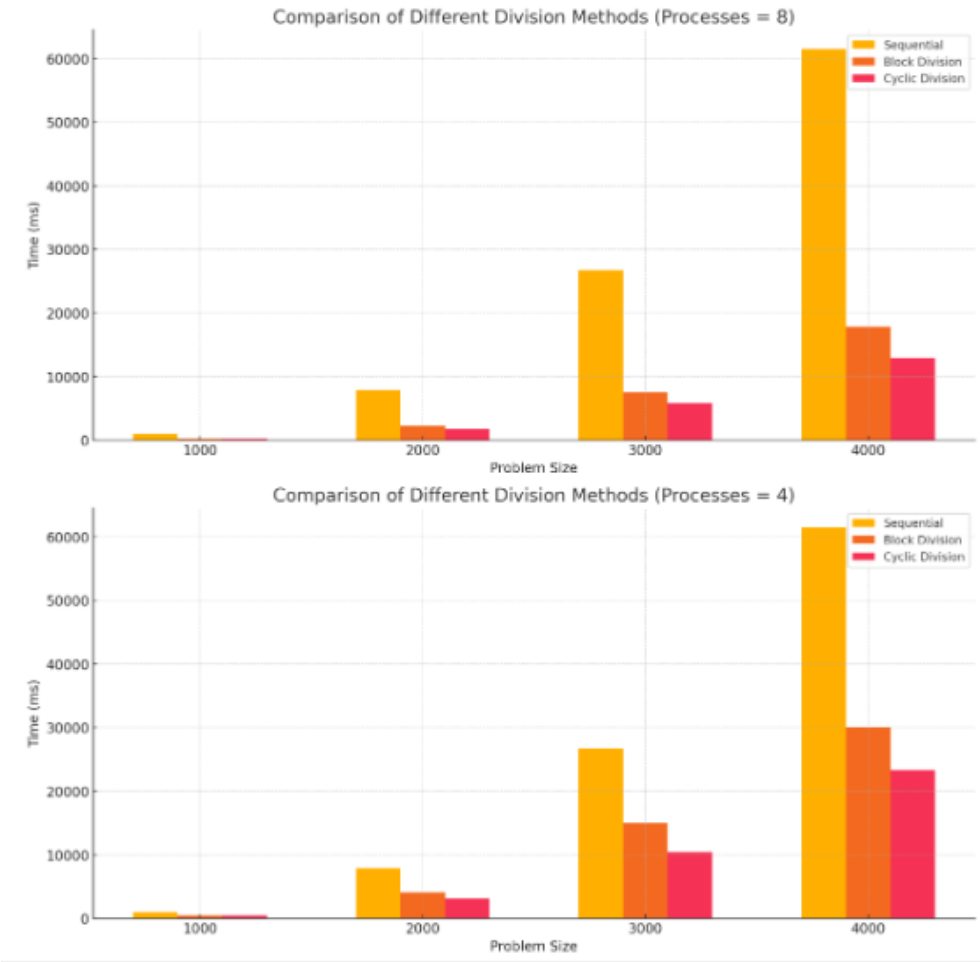

图 3: Caption

实验数据

进程数增加的影响 在两个表格中，随着进程数的增加，执行时间显著减少。更多的进程能够更好地利用多核资源，分担计算负载，减少每个进程的工作量。在表 3 中，进程数为 8 时，块划分和循环划分的执行时间显著低于串行程序。在表 4 中，进程数为 4 时，块划分和循环划分的执行时间同样显著低于串行程序。

块划分与循环划分的对比 在表 3 中，循环划分的执行时间总体上低于块划分。这表明，循环划分在进程数为 8 的情况下能够更好地平衡负载，减少执行时间。在表 4 中，循环划分的执行时间也总体上低于块划分。这表明，循环划分在进程数为 4 的情况下同样能够更好地平衡负载，减少执行时间。

并行效率 总体来看，循环划分在两个表格中都表现出更高的并行效率。在所有问题规模和进程数下，循环划分的时间都低于块划分的时间。这表明，循环划分在大规模计算中的负载均衡效果更好。

总结 通过数据可以看出，随着进程数的增加，执行时间显著减少。而循环划分在所有问题规模和进程数下都表现出更高的并行效率。因此，选择合适的划分方式（如循环划分）可以进一步提高计算性能。

流水线算法 在进程数 =8 的条件下，我们又用流水线算法对不同问题规模进行了实验，结果如表 5。相比循环划分和块划分，流水线算法的用时更少，并且规模越大，加速效果越明显，符合我们的预期结果。

问题规模	1000	2000	3000	4000
串行程序 /ms	978.35	7901.05	26747.5	61499.2
块划分 /ms	290.83	2223.3	7483.22	17826.4
循环划分 /ms	261.67	1708.6	5836.91	12884.5
流水线 /ms	258.94	1696.15	5793.58	12697

表 5: MPI 流水线算法

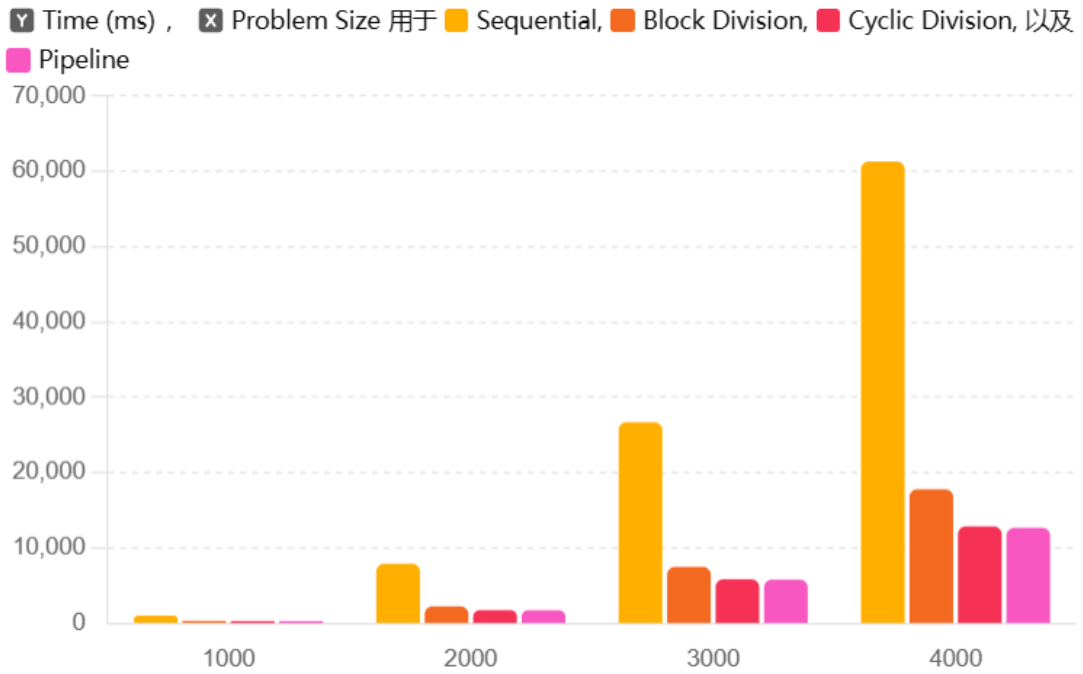

图 4: Caption

实验数据

串行程序时间 对于所有问题规模，串行程序的执行时间在表格中是相同的。这是预期的，因为串行执行没有涉及并行化优化。

块划分与循环划分的对比 块划分和循环划分的执行时间显著低于串行程序。循环划分的时间总体上低于块划分，表明其在负载均衡方面更有效。例如，在问题规模为 1000 的情况下，块划分的时间为 290.83 ms，而循环划分的时间为 261.67 ms。随着问题规模增加，这种趋势更为明显。

流水线的效果 引入流水线后，执行时间进一步降低，表明流水线能够更高效地利用资源，减少执行时间。例如，在问题规模为 1000 的情况下，流水线的时间为 258.94 ms，略低于循环划分的 261.67 ms。在问题规模为 4000 时，流水线的时间为 12697 ms，显著低于循环划分的 12884.5 ms。

并行效率 总体来看，流水线在大多数情况下表现出最高的并行效率。流水线的时间在所有问题规模下都低于其他并行方法，表明其在大规模计算中的负载均衡效果最好。

总结 通过数据可以看出，随着问题规模的增加，流水线算法在所有问题规模下都表现出更高的并行效率。因此，选择合适的并行化方法（如流水线）可以进一步提高计算性能。

SIMD 和 OpenMP 的结合 在 SSE 和 4 线程的条件下，我们对三种算法都实现了与 SIMD 和 OpenMP 的结合，下表是对比的结果。

实验数据

问题规模	1000	2000	3000	4000
串行程序 /ms	978.35	7901.05	26747.5	61499.2
块划分 /ms	290.83	2223.3	7483.22	17826.4
块划分 +SIMD+OpenMP /ms	142.51	1065.92	3619.66	9080.55
循环划分 /ms	261.67	1708.6	5836.91	12884.5
循环划分 +SIMD+OpenMP /ms	156.07	861.59	2977.91	6517.29
流水线 /ms	258.94	1696.15	5793.58	12697
流水线 +SIMD+OpenMP /ms	178.80	806.45	2913.3	6329.7

表 6: 结合 SIMD+OpenMP

块划分与循环划分的对比 块划分和循环划分的执行时间显著低于串行程序。循环划分的时间总体上低于块划分，表明其在负载均衡方面更有效。例如，在问题规模为 1000 的情况下，块划分的时间为 290.83 ms，而循环划分的时间为 261.67 ms。随着问题规模增加，这种趋势更为明显。

流水线的效果 引入流水线后，执行时间进一步降低，表明流水线能够更高效地利用资源，减少执行时间。例如，在问题规模为 1000 的情况下，流水线的时间为 258.94 ms，略低于循环划分的 261.67 ms。在问题规模为 4000 时，流水线的时间为 12697 ms，显著低于循环划分的 12884.5 ms。

并行效率 总体来看，流水线在大多数情况下表现出最高的并行效率。流水线的时间在所有问题规模下都低于其他并行方法，表明其在大规模计算中的负载均衡效果最好。

总结 通过数据可以看出，随着问题规模的增加，流水线算法在所有问题规模下都表现出更高的并行效率。因此，选择合适的并行化方法（如流水线）可以进一步提高计算性能。

改进后的块划分 我们将块划分的通信减半，只保留其有用的部分，得到了改进后的块划分。在进程数 =8、问题规模 =1000 的条件下，其对比结果如表 7。

问题规模	块划分 /ms	改进后的块划分 /ms	循环划分 /ms
1000	290.83	275.21	261.67
2000	2223.3	1989.47	1708.6
3000	7483.22	6682.63	5836.91
4000	17826.4	15975	12884.5

表 7: 改进块划分

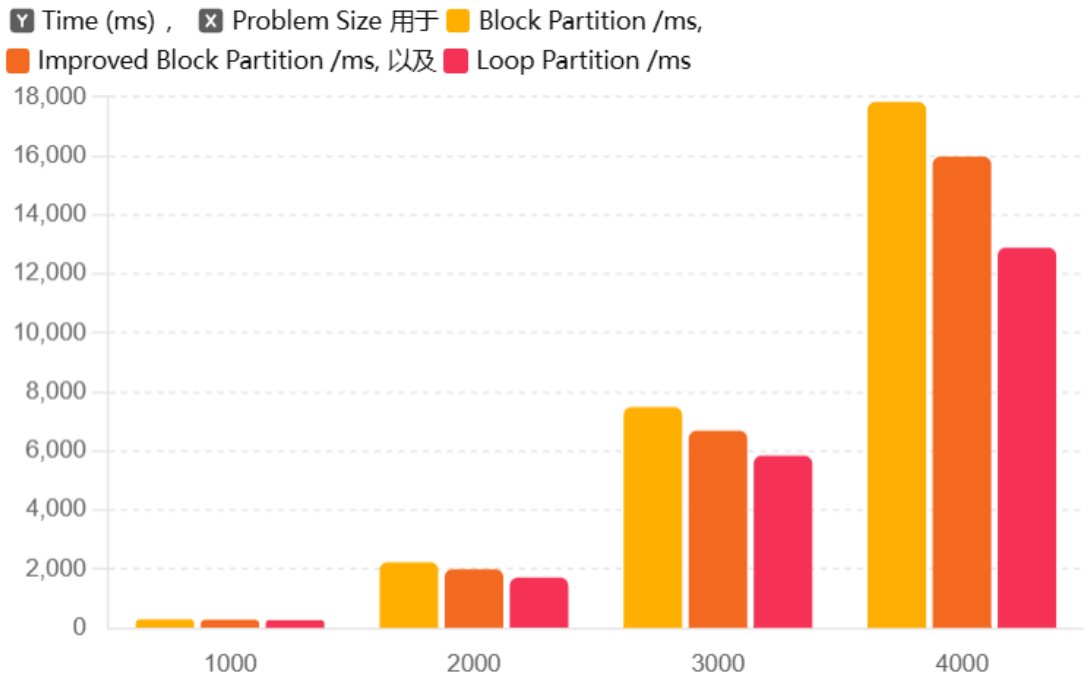

图 5: Caption

实验数据

时间随问题规模增加而增加 无论使用哪种方法，随着问题规模的增大，所需时间都显著增加。这是因为更大的问题规模意味着需要处理的数据量更多，计算复杂度也更高。

改进后的块划分方法的性能提升 改进后的块划分方法相较于原始的块划分方法表现更好。具体来看，在问题规模为 1000 时，改进后减少了约 5.36% 的时间；在问题规模为 2000 时，减少了约 10.5%；在问题规模为 3000 时，减少了约 10.7%；在问题规模为 4000 时，减少了约 10.4%。这表明改进后的方法在各个规模下均有不同程度的优化效果。

循环划分方法的最佳表现 循环划分方法在所有三种方法中表现最佳。在问题规模为 1000 时，相较于原始块划分方法减少了约 10.0% 的时间；在问题规模为 2000 时，减少了约 23.2%；在问题规模为 3000 时，减少了约 21.8%；在问题规模为 4000 时，减少了约 27.7%。这显示了循环划分方法在大规模问题上的显著优势。

总结 这些结果表明，改进后的块划分和循环划分方法相较于原始的块划分方法更为高效，尤其是在处理大规模问题时更为明显。改进后的块划分方法在各个规模下均能提供一定程度的优化，而循环划分方法则在所有规模下均表现出更为显著的性能提升。

基于不同平台架构 除了金山云服务器（Linux+x86），我们还尝试了在鲲鹏服务器（arm 架构）、个人笔记本电脑（Windows+x86）进行实验。

我们以划分方式为块划分、问题规模为 1000 为例，在三种平台上进行实验的结果如表 8

平台	串行 /ms	进程数 =2 /ms	进程数 =4 /ms	进程数 =8 /ms
金山云	978.35	882.04	484.82	290.83
笔记本电脑	228.76	134.06	83.94	54.62
鲲鹏	403.72	299.34	177	112.71

表 8: 不同平台结果

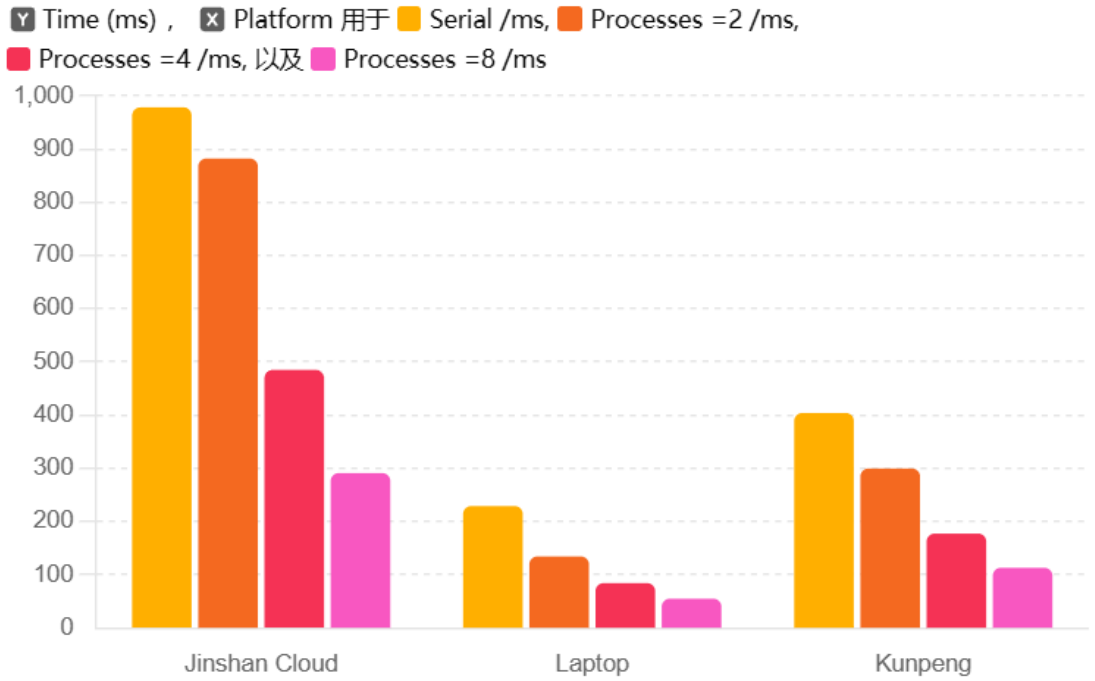

图 6: Caption

实验数据

数据分析 从表中我们可以观察到以下几点:

1. 金山云: - 随着进程数量的增加，处理时间显著减少。在进程数为 8 时，时间减少至串行时间的约 29.7%。

2. 笔记本电脑: - 处理时间随进程数增加而减少。在进程数为 8 时，时间减少至串行时间的约 23.9%。

3. 鲲鹏: - 处理时间随进程数增加而减少。在进程数为 8 时, 时间减少至串行时间的约 27.9%。

(四) 总结

这些结果表明，随着进程数的增加，各个平台的处理时间都显著减少。金山云在大多数情况下表现最佳，而笔记本电脑和鲲鹏也展示了良好的并行处理能力。总体而言，增加进程数量能够有效减少处理时间，提升计算效率。

二、 特殊高斯消元

(一) 问题简介

给定消元子和被消元行，依次使用消元子消去对应的被消元行，若某行消元子为空而有其对应的被消元行，则将被消元行升格成消元子，进入到后续的消元中。

(二) 实验设计

在之前的实验中，我们完成了特殊高斯消去的串行设计，并结合 SIMD 的并行算法，具体方法如下：

1. 批次读取消元子 Act[]。
2. 对每个批次中的被消元行 Pas[]，检查其首项（Pas[row][last]）是否存在对应的消元子；若存在，则与消元子进行异或运算并更新首项（Pas[row][last]），重复此过程直至 Pas[row][last] 不在范围内。
3. 在运算过程中，若某行的首项被当前批次覆盖，但没有对应的消元子，则将其“升格”为消元子，复制到 Act 数组的对应行，并将标志位设为 1，表示非空，然后结束对该被消元行的操作。若每行的首项不在当前批次覆盖范围内，则该批次计算完成。
4. 重复上述过程，直至所有批次处理完毕。

在该算法的设计中，我们分批次读取消元子，然后使用当前批次的消元子对所有被消元行进行消去。若被消元行没有对应的消元子，则直接将其升格，作为新的消元子参与后续的消元。然而，由于这种方法在运行过程中对某些被消元行进行升格，升格后的消元子会影响后续的消元操作，即“升格”会引入程序前后依赖关系。

在 MPI 程序的设计中，由于这些依赖关系的存在，不同进程需要及时了解它们负责的任务之前是否存在“升格”行为。若 Pk 进程的任务需要升格，则应在 Pk+1 进程的任务之前进行。然而，由于我们无法预知 Pk 进程何时需要升格，并且大多数情况下 Pk 进程并不需要升格，因此若每个进程都要等待前面的进程升格与否的结果，会导致通信开销巨大且无用等待时间过长，从而浪费资源。因此，我们采用了一种更适合 MPI 编程的特殊高斯消去算法。

我们发现，如果不考虑升格，消元子对不同被消元行的处理是互不冲突的，可以将被消元行划分给不同的进程，每个进程负责处理一部分被消元行。因此，我们将升格操作单独提取出来，不参与并行运算。

1. 每一轮将被消元行划分给不同进程，各进程不进行升格地处理各自的被消元行。
2. 所有进程处理完后，将数据传回给一个进程进行升格操作，并设置是否需要下一轮消去的标志 sign。若该轮存在新升格的消元子，则设置标志位 sign=true，否则为 false。
3. 根据 sign 判断是否需要进行下一轮消去。若 sign 为 true，则分配任务给所有进程，使用更新后的消元子进行下一轮消去。
4. 直至所有被消元行都不再需要升格，说明消去完成，退出程序。

上述算法在 MPI 程序设计中分为消去和升格两个部分：

1. 消去部分

1. 0 号进程将被消元行划分给不同进程，每个进程负责自己那部分的被消元行。
2. 其他进程接收任务后直接开始运算，完成运算后将结果传回 0 号进程。
3. 0 号进程划分完任务后执行自己的运算任务，并接收来自其他进程的运算结果。
4. 在每个进程内部，遍历各循环步，当遍历到自己的任务时，开始消去运算；未到自己的任务时，直接跳过进入下一个循环步。由于在消去部分，被消元行之间完全没有冲突，因此无需等待其他进程的结果。

2. 升格部分

在一轮消去后，需要遍历所有被消元行，判断哪些行需要升格。由于升格过程中，后面判断的被消元行是否升格依赖于前面的升格结果，因此升格需要串行处理，并且只能在一个进程内完成。0 号进程完成升

(三) 理论分析

在高斯消去算法的并行化过程中，我们采用了单独处理升格的设计。具体而言，算法分为消去和升格两部分，并通过 MPI 实现并行计算。

1. 算法性能分析

相比传统的串行算法，由于升格操作的单独处理，消去部分的循环次数和遍历次数都会增加，这导致了性能下降。然而，并行算法通过分配任务给多个进程，减少了计算时间，从而在整体上提升了性能。具体性能比较如下：

- 单独处理升格的并行算法 > 不单独处理升格的串行算法 > 单独处理升格的串行算法。

2. 通信开销优化

在 MPI 实现中，为了减少通信开销和无用等待，我们进行了如下优化：

- 各进程独立执行消去任务，减少了进程间的通信。
- 只有在需要升格时才进行通信，避免了不必要的数据传输。

(四) 理解与实现

我们以循环划分的方式，对特殊高斯消去的 MPI 算法进行实现。(具体思路见注释)

```
void super(int rank, int num_proc) {
    // 不升格地处理被消元行 begin
    int i;
    // 每轮处理8个消元子，范围：首项在 i-7 到 i
    #pragma omp parallel num_threads(thread_count)
    for (i = lieNum - 1; i >= 7; i -= 8) {
        #pragma omp for schedule(dynamic, 20)
        for (int j = 0; j < pasNum; j++) {
            // 当前行是当前进程的任务——进行消去
```

```
            if (j % num_proc == rank) {
                while (Pas[j][Num] <= i && Pas[j][Num] >= i - 7) {
                    int index = Pas[j][Num];
                    if (Act[index][Num] == 1) { // 消元子不为空
                        int k;
                        __m128 va_Pas, va_Act;
                        for (k = 0; k + 4 <= Num; k += 4) {
                            va_Pas = _mm_loadu_ps((float *)&(Pas[j][k]));
                            va_Act = _mm_loadu_ps((float *)&(Act[index][k]));
                            va_Pas = _mm_xor_ps(va_Pas, va_Act);
                            _mm_storeu_ps((float *)&(Pas[j][k]), va_Pas);
                        }
                        for (; k < Num; k++) {
                            Pas[j][k] ^= Act[index][k];
                        }
                        // 更新首项值
                        int num = 0, S_num = 0;
                        for (num = 0; num < Num; num++) {
                            if (Pas[j][num] != 0) {
                                unsigned int temp = Pas[j][num];
                                while (temp != 0) {
                                    temp >>= 1;
                                    S_num++;
                                }
                                S_num += num * 32;
                                break;
                            }
                        }
                        Pas[j][Num] = S_num - 1;
                    } else { // 消元子为空
                        break;
                    }
                }
            }
        }
    }
    #pragma omp parallel num_threads(thread_count)
    for (int i = lieNum % 8 - 1; i >= 0; i--) {
        // 每轮处理1个消元子，范围：首项等于 i
        // 代码与前面相似，故省略
    }
    // 不升格地处理被消元行 end
}

void f_mpi() {
    int num_proc; // 进程数
    int rank; // 当前进程的 rank，值从 0~size-1
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
```

```
58      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
59
60      // 0 号进程——任务划分
61      if (rank == 0) {
62          int sign;
63          do {
64              // 任务划分
65              for (int i = 0; i < pasNum; i++) {
66                  int flag = i % num_proc;
67                  if (flag != rank) {
68                      MPI_Send(&Pas[i], Num + 1, MPI_FLOAT, flag, 0,
                            MPI_COMM_WORLD);
69                  }
70              }
71              super(rank, num_proc);
72              // 处理完 0 号进程自己的任务后需接收其他进程处理之后的结果
73              for (int i = 0; i < pasNum; i++) {
74                  int flag = i % num_proc;
75                  if (flag != rank) {
76                      MPI_Recv(&Pas[i], Num + 1, MPI_FLOAT, flag, 1,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
77                  }
78              }
79              // 升格消元子，然后判断是否结束
80              sign = 0;
81              for (int i = 0; i < pasNum; i++) {
82                  // 找到第 i 个被消元行的首项
83                  int temp = Pas[i][Num];
84                  if (temp != -1) {
85                      // 看这个首项对应的消元子是否为空，若为空，则补齐
86                      if (Act[temp][Num] == 0) {
87                          // 补齐消元子
88                          for (int k = 0; k < Num; k++) {
89                              Act[temp][k] = Pas[i][k];
90                          }
91                          // 将被消元行升格
92                          Pas[i][Num] = -1;
93                          // 标志设为 true，说明此轮还需继续
94                          sign = 1;
95                      }
96                  }
97              }
98              for (int r = 1; r < num_proc; r++) {
99                  MPI_Send(&sign, 1, MPI_INT, r, 2, MPI_COMM_WORLD);
100             }
101         } while (sign == 1);
102     }
103
```

```
104     // 其他进程
105     else {
106         int sign;
107         do {
108             // 非 0 号进程先接收任务
109             for (int i = rank; i < pasNum; i += num_proc) {
110                 MPI_Recv(&Pas[i], Num + 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
111             }
112             // 执行任务
113             super(rank, num_proc);
114             // 非 0 号进程完成任务之后，将结果传回到 0 号进程
115             for (int i = rank; i < pasNum; i += num_proc) {
116                 MPI_Send(&Pas[i], Num + 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
117             }
118             MPI_Recv(&sign, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
119         } while (sign == 1);
120     }
121 }
```

（五） 实验结果及分析

1. 实验数据

测试样例	8	9	10
不单独处理升格的串行算法 /ms	283.23	533.81	2292.59
单独处理升格的串行算法 /ms	461.42	1107.14	4642.52
MPI 算法 /ms	225.35	460.50	1967.55

表 9: 特殊高斯实验结果

2. 实验结果分析

从表中可以观察到以下几点：

- 不单独处理升格的串行算法在所有测试样例中表现较好，但在大规模问题（测试样例 10）上性能下降明显。
- 单独处理升格的串行算法由于额外的升格操作，整体性能较差，特别是在大规模问题上（测试样例 10）表现最差。
- MPI 算法在所有测试样例中均表现出较好的性能，尤其是在大规模问题上（测试样例 10），其性能优于其他两种算法。

综合比较三种算法的性能，我们可以得出以下结论：

- 单独处理升格的并行算法（MPI 算法）> 不单独处理升格的串行算法 > 单独处理升格的串行算法。

三、 新得体会

在本次实验中，通过设计和实现基于 MPI 的特殊高斯消去算法，我深刻体会到了并行计算的实际应用和优势。

(一) 并行计算的优势

采用 MPI 实现特殊高斯消去算法，使得消元操作可以完全并行化，大幅度提升了计算效率。在处理大规模数据时，MPI 算法表现出显著的性能优势，证明了并行计算在实际应用中的巨大潜力。

(二) 设计思路的挑战

在设计过程中，我们将任务划分为并行消元和串行升格两部分。这一设计思路有效减少了计算过程中的依赖关系，使各进程能够独立完成消元任务。同时，串行升格确保了数据的一致性和正确性。这种任务划分策略在保证正确性的同时，最大限度地利用了计算资源。

(三) 通信开销的优化

在 MPI 实现中，通信开销是影响性能的关键因素。通过合理划分任务和减少不必要的数据传输，我们有效降低了通信开销。仅在必要时进行数据传输，减少了进程间的等待时间，提高了整体效率。

(四) 扩展性的体会

通过实验验证，MPI 算法在处理大规模问题时表现出良好的扩展性。进程数量增加时，计算时间显著减少，展现了并行计算的优势。这让我体会到，在处理复杂计算任务时，并行算法设计的合理性和扩展性至关重要。

(五) 总结

本次实验不仅加深了我对 MPI 并行计算的理解，更让我认识到合理算法设计和优化对性能提升的重要性。通过将升格操作单独处理，并利用多进程并行计算，我们成功实现了高效的特殊高斯消去算法。这些经验将为我今后的研究和实际应用提供宝贵的指导和参考。

四、 源代码

https://github.com/uJunLI/NKU-Parallel$_{p}rogramming/tree/master/Lab4/$。