
南开大学

Nankai University

# 南　　开　　大　　学

## 计　算　机　学　院

**并行程序设计实验报告**

---

# MPI 编程实验

---

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 6 月 17 日

摘要

本实验基于两种高斯消去算法，进行 **MPI** 并行实验。分别对**普通高斯消去**和**特殊高斯消去**两种算法，实现了串行算法和 **MPI** 等的并行优化。

**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

### 1. 总体思路

与之前编写的 pthread、OpenMP 并行程序不同，MPI 编程采用消息传递式并行程序设计，基于分布式内存体系结构，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。而在通信方面，进程间由于地址空间独立，通过传递消息进行通信，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，需要基于进程间的同步和数据移动两方面实现。

普通高斯消去的伪代码如下：

**Algorithm 1** procedure LU (A)

```
1:  for k = 1 → n do
2:      for j = k + 1 → n do
3:          A[k, j] ← A[k, j]/A[k, k]
4:      end for
5:      A[k, k] ← 1.0
6:      for i = k + 1 → n do
7:          for j = k + 1 → n do
8:              A[i, j] ← A[i, j]−A[i, k]×A[k, j]
9:          end for
10:         A[i, k] ← 0
11:     end for
12: end for
```

分析算法的依赖关系可知，对于每一个消去步来说，要先对该步在矩阵中对应的行进行除法使得对角线元素为 1，然后才能进行下面的消去，即利用刚做完除法的这一行对此行之下的所有行进行消去使得这些行的指定列全为 0。

在对高斯消去进行 MPI 程序设计时，我们考虑让每个进程分管指定几行的除法和消去。对于特定进程来讲，分为除法和消去两部分。我们在进程函数中仍然不改变原程序框架进行循环遍历，对于除法循环来讲，当最外层循环步遍历到自己进程的任务时，进行除法运算，并将运算结果发送给其他进程；若当前循环步不是自己进程的任务，则接收其他进程传来的消息。对于消去部分来讲，在每个循环步内，进行完除法部分（除法运算 + 发送消息或接收从其他进程传来的消息）后，即进行该循环步内属于自己进程所分配的行的消去运算。

上面所述是每个进程内的具体运算过程，而对于进程之间消息传递而言，可以采用主从式的编程模型，首先让 0 号进程为其他进程分配任务，把其他进程所需要的矩阵数据发送过去，然后 0 号进程执行自己的任务，此后再接收其他进程完成任务后的结果，整个程序结束。其他进程则需要先接收任务，然后开始运算，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，下面我们将基于该框架进行多种算法和实现方式的探究。

**Algorithm 2** procedure LU (A) for MPI

**Input:** 矩阵 $A[n,n]$，问题规模 $n$

**Output:** 上三角矩阵 $A[n,n]$

```
1: function LU(rank, num_proc)
2:     根据 rank 计算自己进程的任务范围
3:     for k = 1 → n do
4:         if 当前行是自己进程的任务 then
5:             A[k,j] ← A[k,j]/A[k,k]
6:             向其他进程发送消息
7:         else
8:             接收其他进程传来的消息
9:         end if
10:        for j = k + 1 → n do
11:            if 当前行是自己进程的任务 then
12:                for j = k + 1 → n do
13:                    A[i,j] ← A[i,j]−A[i,k]×A[k,j]
14:                end for
15:                A[i,k] ← 0
16:            end if
17:        end for
18:    end for
19: end function
20:
21: function MAIN()
22:    MPI_Comm_size (MPI_COMM_WORLD, &num_proc)
23:    MPI_Comm_rank (MPI_COMM_WORLD, &rank)
24:    if rank == 0 then
25:        任务划分
26:        LU(rank, num_proc)
27:        接收其他进程的运算结果
28:    else
29:        接收从 0 号进程分配的任务
30:        LU(rank, num_proc)
31:        将运算结果传回 0 号进程
32:    end if
33: end function
```

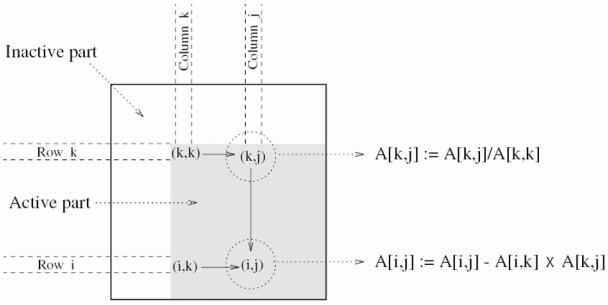


图 1: 普通高斯消去

**2. 设计思路**

本小节给出对于实验设计的概览，具体实现思路见下面的实验内容和算法实现一节。

- 与 SIMD 和多线程 OpenMP 的结合

由于 MPI 采用基于分布式内存体系结构，每个系统有自己独立的地址空间，而每个进程都可以再启动多个线程，在 MPI 的基础上使用 OpenMP 和 SIMD 可以达到更好的效果。因此，在本实验中，我们设计将 MPI 与 OpenMP 和 SIMD 结合，并进行对比分析。

- 块划分、循环划分

关于将矩阵划分给各进程的方式，本实验采用了一维块划分和循环划分的方法。对于块划分来讲，将矩阵按照问题规模/进程数划分成不同的块，将每个块分给每个进程。对于循环划分来讲，将矩阵从上到下依次一行给一个进程，一轮划分完之后再从头循环划分一行给一个进程，直到所有行都分完。

- 流水线算法

将进程形成逻辑链进行转发，在 $P_k$ 广播给其他进程时，不是一下子传给所有其他进程，而是先只转发给它的下一个进程 $P_{k+1}$，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都收到数据，即可进行消去计算，从而实现流水线并行。

- 不同平台

MPI 支持多种操作系统和架构，实验设计在鲲鹏服务器、金山云服务器、笔记本电脑等不同平台进行 MPI 实验并对比分析。

- 其他策略

在块划分时，我们发现在数据传递时，由于进程所分数据的次序关系，我们不必将除法结果传给其他所有进程，而可以只传给部分需要的进程，这样可以大大减少通信所用开销，加速现有程序。（具体实现见下面实验内容）

## (三) 理论分析

- **块划分和循环划分**

相比于块划分，循环划分具有两方面优势：

一方面，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。并且

另一方面，将任务分给不同进程时，由于进程数未必能整除，因此余数部分会被分给某个或某几个进程，导致负载不均，再加上高斯运算的特点，导致越靠近矩阵右下角的进程负载越大，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 $(n-k-1)\frac{n}{p}$ 次乘法和除法，并行时间大致为 $2(\frac{n}{p})\sum_{k=0}^{n-1}(n-k-1) \approx \frac{n^3}{p}$，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，所以总的时间为 $n^3$，算法的时间复杂度是 $O(n^3)$。

**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 $n^3$，而循环划分中，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，由于共有 n 个循环步，每个循环步中，p 个处理器又各有 $O(n)$ 时间的空闲等待，因此进程空闲的时间最多为 $O(n^2p)$。

- **流水线算法**

回顾高斯消去的 MPI 算法，我们在第 k 个循环步内完成除法后，需要将第 k 行广播给其他进程。但是，算法在此存在缺陷——进程必须等其他所有进程都接收到了消息才能继续向下运行。这会导致该进程的开始时间产生了无谓的延迟。因此，我们采用流水线算法。流水线算法的本质是，尽可能让互不影响的 A 进程的除法部分与 B 进程的消去部分并发执行。通信与通信、通信与计算达到重叠，提高并行度。

在流水线中，进程形成逻辑链结构 $P_0$、$P_1$、...。在第 k 个循环步完成除法后，$P_k$ 不再将数据传给全部进程，而是只转发给下一个进程 $P_{k+1}$，$P_{k+1}$ 首先收到，然后转发给 $P_{k+2}$，同样，$P_{k+2}$ 将数据转发给 $P_{k+3}$ 后即可计算，而 $P_{k+1}$ 完成消去计算后，即可启动第 k+1 个步骤。这样使得进程无需等待所有其他进程都收到数据，即可进行消去计算，从而尽早地启动下一个步骤。

**流水线的时间复杂度：**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 $O(n)$，又因为有 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 $O(n^3)$

在流水线的设计中，需要先判断进程保存的数据是否有其他进程需要，若进程保存的数据有其他进程需要，应先将数据发送出去，再之后如果现有的数据可用来进行计算，则进行计算，以免耽误其他进程的运行进度，提高并行度。

- **其他策略——块划分的改进**

由于在普通高斯消去中，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，而块划分可以很好地利用这一点来减少通信开销。在块划分中，将矩阵从上到下依次分给从编号从小到大的进程，使得 rank=k 的进程所负责的行号一定小于 rank=k+1 的进程所负责的行号。因此，我们可以在第 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。而在消去之前，只接收该进程前面的进程传来的消息。

采用这样的策略，相比于之前的块划分，可以减少一半左右的通信开销。预测在越细的粒度越能体现优势。因为在细粒度划分时，通信开销占大部分。

## （四） 实验内容及算法实现

首先，我们在不同节点数、不同问题规模下采用不同的方法进行了 MPI 实验，并且尝试了其他策略和平台。

限于篇幅,报告中只展示部分代码(可以在https://github.com/hanmaxmax/parallel_homework5中详见代码)。

- **块划分**

基于块划分的 MPI 算法如下:

普通高斯消去 MPI 块划分

```
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, 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()
{
    timeval t_start, t_end;
    int num_proc;//进程数
```

```
38      int rank;//识别调用进程的rank, 值从0~size-1
39
40      MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
41      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
42
43      int block = N / num_proc, remain = N % num_proc;
44
45      //0号进程--任务划分
46      if (rank == 0){
47          reset_A(A, arr);
48          gettimeofday(&t_start, NULL);
49          //任务划分
50          for (int i = 1; i < num_proc; i++){
51              if (i != num_proc - 1){
52                  for (int j = 0; j < block; j++)
53                      MPI_Send(&A[i * block + j], N, MPI_FLOAT, i, 0,
                            MPI_COMM_WORLD);
54              }
55              else{
56                  for (int j = 0; j < block + remain; j++)
57                      MPI_Send(&A[i * block + j], N, MPI_FLOAT, i, 0,
                            MPI_COMM_WORLD);
58              }
59          }
60          LU(A, rank, num_proc);
61          //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
62          for (int i = 1; i < num_proc; i++){
63              if (i != num_proc - 1){
64                  for (int j = 0; j < block; j++)
65                      MPI_Recv(&A[i * block + j], N, MPI_FLOAT, i, 1,
66                          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
67              }
68              else{
69                  for (int j = 0; j < block + remain; j++)
70                      MPI_Recv(&A[i * block + j], N, MPI_FLOAT, i, 1,
71                          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
72              }
73          }
74          gettimeofday(&t_end, NULL);
75          cout << "Block MPI LU time cost: "
76              << 1000 * (t_end.tv_sec - t_start.tv_sec) +
77              0.001 * (t_end.tv_usec - t_start.tv_usec) << "ms" << endl;
78      }
79
80      //其他进程
81      else{
82          //非0号进程先接收任务
83          if (rank != num_proc - 1){
```

```
            for (int j = 0; j < block; j++)
                MPI_Recv(&A[rank * block + j], N, MPI_FLOAT, 0, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else{
            for (int j = 0; j < block + remain; j++)
                MPI_Recv(&A[rank * block + j], 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
        if (rank != num_proc - 1){
            for (int j = 0; j < block; j++)
                MPI_Send(&A[rank * block + j], N, MPI_FLOAT, 0, 1,
                    MPI_COMM_WORLD);
        }
        else{
            for (int j = 0; j < block + remain; j++)
                MPI_Send(&A[rank * block + j], N, MPI_FLOAT, 0, 1,
                    MPI_COMM_WORLD);
}}}
```

- **循环划分**

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）

普通高斯消去 MPI 循环划分

```
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 (int(k % num_proc) == rank){
            ...
        }
        //当前行不是自己进程的任务——接收消息
        else{//接收来自进程号为int(k % num_proc)的消息
            MPI_Recv(&A[k], N, MPI_FLOAT, int(k % num_proc), 2,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        for (int i = k + 1; i < N; i++){
            if (int(i % num_proc) == rank){
                ...
}}}}
void f_mpi()
```

```
{
    ...
    //0号进程——任务划分
    if (rank == 0){
        ...
        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);
        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);
}}}
```

- **流水线算法**

而在流水线算法中，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（不同于循环划分的部分）

```
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++){
        //如果当前行是是自己的任务，则进行除法
        if (int(k % num_proc) == rank){
            ...
            //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 (int(k % num_proc) != next_proc)
                MPI_Send(&A[k], N, MPI_FLOAT, next_proc, 2, MPI_COMM_WORLD);
        }
        //消去部分（同循环划分的代码）
        ...
    }
}
```

- **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：

MPI 块划分更改部分

```
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
        ...
}}}}
```

- **与 SIMD 和多线程的结合**

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;
}}}}
```

## （五） 实验结果及分析

### 1. 不同进程数的对比

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。

| 问题规模 | 1000 | 2000 | 3000 |
|---|---|---|---|
| 串行程序 /ms | 978.35 | 7901.05 | 26747.5 |
| 进程数 =2 /ms | 882.04 | 7758.84 | 23858.5 |
| 进程数 =4 /ms | 484.82 | 4058.77 | 15001.6 |
| 进程数 =8 /ms | 290.83 | 2223.3 | 7483.22 |

表 1: 普通 MPI 程序

| 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

可以看到，随着进程数的增加，加速效果越来越好，在相同问题规模和方法下，不同进程数之间的时间之比近似等于进程数之比。我们将不同进程数的时间比与其相对于串行算法加速比做成对比图，如图2。

可以看到，进程数 =8 的用时/进程数 =4 的用时近似为 2，进程数 =8 的用时/进程数 =2 的用时在 3 以上，未达到 4。这说明，进程数越大，加速比越接近进程数之比。

进一步分析，进程数为 2 的加速比远未达到 2，而进程数为 4、8 的加速比也只达到了其进程数的一半（2、4 附近）。从程序运行的角度来看，这种情况的原因主要是在 MPI 程序中，进

程存在空闲等待，而块循环中的空闲等待时间又较长，因此进程未能完全利用。这将在下面的不同划分方式和不同算法中得到改善。

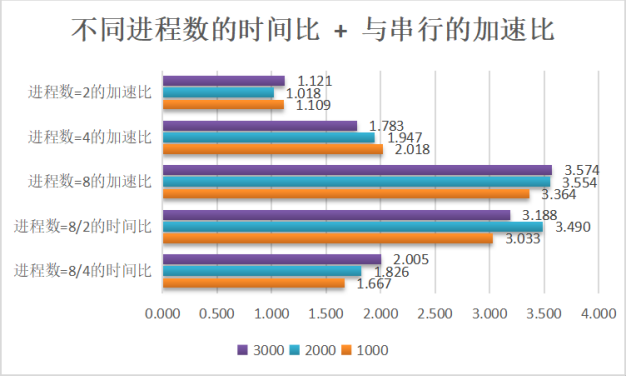


图 2: 加速比及时间比

### 2. 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）

我们计算两种划分方式的加速比，得到如图3所示的结果。从图中看出，在两种进程数下，循环划分的加速比普遍高于块划分，且随着问题规模的增大，循环划分的加速比越大。在理论分析中，我们曾提到，块划分中总的进程空闲时间达到 $O(n^3)$，而循环划分的空闲时间最多为 $O(n^2p)$。在进程数相同的情况下，两者的加速比相差一个量级，因此，n 越大时，循环划分的效果越好，这与我们得出的结果也是一致的。

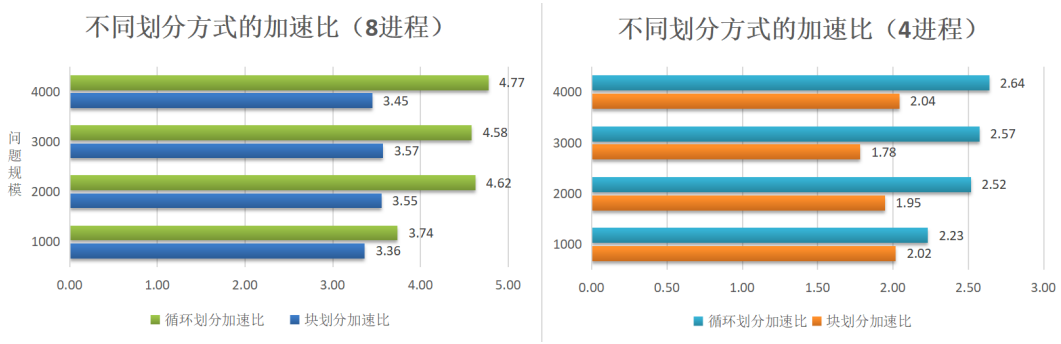


图 3: 加速比及时间比

### 3. 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

### 4. 与 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。

| 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

从最终结果可以看出，MPI 结合 OpenMP 和 SIMD 之后的用时是普通的 MPI 程序用时的一半左右。如果单从 SSE 向量大小和 OpenMP 线程数大小来说，并未达到应有的效果（比如4）。从 MPI 程序来看，每个进程有空闲等待的时间，因此实际的运算时间并不占总时间的大部

分。这就导致在空闲时间内，SIMD 和 OpenMP 发挥不了作用。因此，提高程序性能的重点还是在于减少空闲时间。

### 5. 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 | 1000 | 2000 | 3000 | 4000 |
|---|---|---|---|---|
| 块划分 /ms | 290.83 | 2223.3 | 7483.22 | 17826.4 |
| 改进后的块划分/ms | 275.21 | 1989.47 | 6682.63 | 15975 |
| 循环划分 /ms | 261.67 | 1708.6 | 5836.91 | 12884.5 |

表 7: 改进块划分

可以看出，改进后的块划分要比未改进的更快，但其仍未超过循环划分。在循环划分的代码中，发送消息是向其他所有进程发送，在改进后的块循环是向后面的进程发送；但是循环划分的负载要比块划分均衡，进程空闲等待时间更少，能够并发执行的进程更多，因此效果更好。

### 6. 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。鲲鹏服务器的架构如图4，笔记本电脑配置如图5。

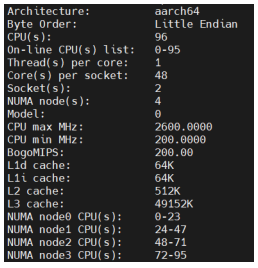


图 4: 鲲鹏服务器

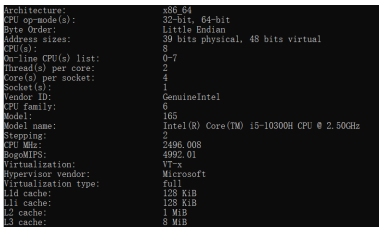


图 5: PC

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

金山云上程序的运行时间普遍高于笔记本电脑和鲲鹏上的时间。且个人电脑的用时最少（这与之前的《SIMD》、《Pthread》等实验结果一致）。开 8 个进程时，利用 vtune 进行 profiling 如图6所示

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

表 8: 不同平台结果

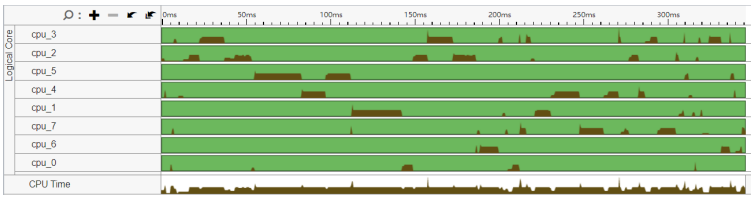


图 6: vtune

# 二、 特殊高斯消去

## (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以及结合 SIMD 的并行的算法，方法如下：

1) 逐批次读取消元子 Act[ ]。

2) 对当前批次中每个被消元行 Pas[ ]，检查其首项（Pas[row][last]）是否有对应消元子；若有，则将与对应消元子做异或并更新首项（Pas[row][last]），重复此过程直至 Pas[row][last] 不在范围内。

3) 运算中，若某行的首项被当前批次覆盖，但没有对应消元子，则将它“升格”为消元子，copy 到数组 Act 的对应行，并设标志位为 1 表示非空，然后结束对该被消元行的操作。

运算后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，直至所有批次都处理完毕。

在算法的设计中，我们分批对消元子进行读取，然后使用当前这批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，作为消元行进入到后续的消元中。然而，由于这种方法会在程序运行中间对某些被消元行进行升格，而升格成的这些消元子会影响到后续的消元环节，即“升格”的存在会造成程序前的后依赖关系。

在 MPI 程序的设计中，由于这些依赖关系的存在，不同进程需要及时知道在它负责的任务之前的进程是否有“升格”行为，若 $P_k$ 进程的任务需要升格，则其应在 $P_{k+1}$ 进程的任务之前进行，但前提是我们并不知道 $P_k$ 进程什么时候需要升格，并且大部分时间内，$P_k$ 进程是不需要升格的。因此，若每个进程都要等待它前面的进程升格与否的结果，一是通信开销巨大，二是根本没那么多“升格”操作，大部分进程都变成了无用的等待，等待开销也巨大，且浪费资源，因此，我们采用另一种更适合 MPI 编程的特殊高斯消去算法。

我们发现，如果不考虑升格，则消元子对不同的被消元行进行处理互不冲突，可以将被消元行划分给不同的进程，每个进程负责对一部分的被消元行进行处理。因此，我们考虑将升格操作单独拿出来，不参与到并行运算。

1) 每一轮将被消元行划分给不同进程，各进程不升格地处理自己的被消元行；

2) 所有进程处理完之后，传回数据给一个进程，使其对可能升格的被消元行升格，并设置是否还需要下一轮消去的标志 sign=true。若该轮中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，若 sign 为 true，说明还需要下一轮消去。传消息并划分任务给所有进程，使用更新后的消元子进入下一轮的消去；

4) 直到所有被消元行都不用再升格，说明消去完成，退出程序。

对上述算法进行 MPI 程序的设计时，总共分为消去和升格两个部分，这两个部分的算法如下：

- **消去部分**

1) 0 号进程将被消元行划分给不同的进程，每个进程负责自己那部分的被消元行；

2) 其他进程接收任务之后直接开始运算，所有循环步遍历结束，没有自己的任务之后，传回该进程的运算结果给 0 号进程；

3) 0 号进程划分完任务后，执行自己的运算任务，此后接收来自其他进程的运算结果；

4) 在每个进程内部，依次遍历各循环步，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（与普通高斯消去不同，在消去这部分的运算中，被消元行之间完全没有冲突，可以不用等待其他进程的结果）。

- **升格部分**

在上述的一轮消去过后，应对所有被消元行进行的遍历并判断哪些行需要升格。由于这个升格过程中，后面判断到的被消元行是否升格是依赖于它前面的被消元行在该位置的升格结果，因此，升格需要串行，且只能在一个进程内完成。由于最后数据会传回 0 号进程，所以我们让 0 号进程完成这个串行操作。但是这就设计到 MPI 程序特有的**数据运输与同步**的两个关键问题：在 0 号进程完成升格后如何向其他进程传达是否需要升格（是结束循环 or 继续划分任务）的信息以及如何统一各进程数据：

1) 关于升格的选择信息，我们只需要在 0 号进程完成判断后，将标志 sign 传给其他各进程，其他进程根据标志来判断是结束循环、退出程序还是继续接收任务、执行任务。

2) 关于数据统一，我们需要在各进程执行完任务后，将其结果传回 0 号进程，而此时，该进程进入等待，待 0 号进程传回 sign 标志，再由此决定下一步的走向。这样，0 号进程就有了所有被消元行被不同进程运算后的数据，其才可以遍历所有行进行升格操作。

## (三) 理论分析

该方法对应的串行程序与之前设计的串行程序相比，由于升格的单独化，消去的循环次数会增多，对被消元行的遍历升格次数也会增多，性能会下降。但是其对应的并行算法相对于较好的串行算法可以达到优化的效果。即性能：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。

## (四) 实验内容及算法实现

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）

特殊高斯消去

```
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 - 8 >= -1; i -= 8)
    {
#pragma omp for schedule(dynamic,20)
        for (int j = 0; j < pasNum; j++)
        {
            //当前行是自己进程的任务——进行消去
            if (int(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){
                            //Pas[j][k] = Pas[j][k] ^ Act[index][k];
                            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_store_ss((float*)&(Pas[j][k]), va_Pas);
                        }
                        for (; k < Num; k++){
                            Pas[j][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 = temp >> 1;
                                    S_num++;
                                }
                                S_num += num * 32;
                                break;
                            }
                        }
                        Pas[j][Num] = S_num - 1;
                    }
                    else//消元子为空
                        break;
    }}}}
```

```
48 #pragma omp parallel num_threads(thread_count)
49     for (int i = lieNum % 8 - 1; i >= 0; i--){
50         //每轮处理1个消元子，范围：首项等于i
51         ......//代码与前面相似，故省略
52         }
53     }
54     //不升格地处理被消元行end————————
55 }
56 
57 void f_mpi()
58 {
59     int num_proc;//进程数
60     int rank;//识别调用进程的rank，值从0~size-1
61     MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
62     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
63 
64     //0号进程——任务划分
65     if (rank == 0){
66         int sign;
67         do{
68             //任务划分
69             for (int i = 0; i < pasNum; i++){
70                 int flag = i % num_proc;
71                 if (flag == rank)
72                     continue;
73                 else
74                     MPI_Send(&Pas[i], Num + 1, MPI_FLOAT, flag, 0,
                        MPI_COMM_WORLD);
75             }
76             super(rank, num_proc);
77             //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
78             for (int i = 0; i < pasNum; i++){
79                 int flag = i % num_proc;
80                 if (flag == rank)
81                     continue;
82                 else
83                     MPI_Recv(&Pas[i], Num + 1, MPI_FLOAT, flag, 1,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
84             }
85             //升格消元子，然后判断是否结束
86             sign = 0;
87             for (int i = 0; i < pasNum; i++){
88                 //找到第i个被消元行的首项
89                 int temp = Pas[i][Num];
90                 if (temp == -1){
91                     //说明他已经被升格为消元子了
92                     continue;
93                 }
```

```
94                  //看这个首项对应的消元子是不是为空，若为空，则补齐
95                  if (Act[temp][Num] == 0){
96                      //补齐消元子
97                      for (int k = 0; k < Num; k++)
98                          Act[temp][k] = Pas[i][k];
99                      //将被消元行升格
100                     Pas[i][Num] = -1;
101                     //标志设为true，说明此轮还需继续
102                     sign = 1;
103                 }
104             }
105             for (int r = 1; r < num_proc; r++){
106                 MPI_Send(&sign, 1, MPI_INT, r, 2, MPI_COMM_WORLD);
107             }
108         } while (sign == 1);
109     }
110
111     //其他进程
112     else{
113         int sign;
114         do{
115             //非0号进程先接收任务
116             for (int i = rank; i < pasNum; i += num_proc){
117                 MPI_Recv(&Pas[i], Num + 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
118             }
119             //执行任务
120             super(rank, num_proc);
121             //非0号进程完成任务之后，将结果传回到0号进程
122             for (int i = rank; i < pasNum; i += num_proc){
123                 MPI_Send(&Pas[i], Num + 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
124             }
125             MPI_Recv(&sign, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
126
127         } while (sign == 1);
128     }
129 }
```

## （五） 实验结果及分析

在进程数分别为 4、测试环境为金山云并且结合 SIMD 和 OpenMP 编程的条件下，我们对以下样例进行实验，结果如表9。

测试样例 8：矩阵列数 23045，非零消元子 18748，被消元行 14325

测试样例 9：矩阵列数 37960，非零消元子 29304，被消元行 14921

测试样例 10：矩阵列数 43577，非零消元子 39477，被消元行 54274

从表??中可看出，预期结果与我们理论分析的结果大致相同，单独处理升格的串行算法要比

| 测试样例 | 8 | 9 | 10 |
|---|---|---|---|
| 不单独处理升格的串行算法/ms | 257.48 | 485.28 | 2084.17 |
| 单独处理升格的串行算法/ms | 419.47 | 1006.49 | 4220.47 |
| MPI 算法/ms | 204.86 | 418.64 | 1788.68 |

表 9: 特殊高斯实验结果

不单独处理升格的串行算法慢，但是这种“较差”的串行算法要比“较好”的串行算法更适合应用于并行编程，且其并行处理过的程序相比于“较好”的串行算法要更快。

# 三、 源代码

GitHub 仓库地址：https://github.com/hanmaxmax/parallel_homework5