



南開大學

Nankai University

计算机学院  
并行程序设计实验报告

MPI 优化高斯消元算法

姓名：张天歌

学号：2211123

专业：计算机科学与技术

2024 年 6 月 4 日

# 目录

<b>1 概述</b>	<b>2</b>
1.1 实验平台 . . . . .	2
1.2 正确性验证说明 . . . . .	2
1.3 实验用时测量说明 . . . . .	2
1.4 普通高斯消元串行算法 . . . . .	3
<b>2 普通高斯消元 MPI 优化</b>	<b>3</b>
2.1 MPI 优化思路 . . . . .	3
2.1.1 MPI 介绍 . . . . .	3
2.1.2 算法设计 . . . . .	4
2.2 不同进程数的探究 . . . . .	4
2.3 数据划分方式的探究 . . . . .	6
<b>3 ARM 平台与 x86 平台对比</b>	<b>7</b>
3.1 不同进程数的探究 . . . . .	8
3.2 数据划分方式的探究 . . . . .	9
<b>4 进阶要求</b>	<b>10</b>
4.1 非阻塞通信 MPI 的探索 . . . . .	10
4.2 MPI 流水线形式的改进 . . . . .	10
4.3 MPI 和多线程和 SIMD 的结合 . . . . .	11
<b>5 基于 Gröbner 基计算的特殊高斯消元</b>	<b>12</b>
5.1 问题简介 . . . . .	12
5.2 算法设计 . . . . .	12
5.3 实验结果 . . . . .	15
<b>6 总结与反思</b>	<b>16</b>

## Abstract

本次实验主要完成了普通高斯消元和特殊高斯消元的 MPI 并行化，分析了不同进程数、不同数据划分方式对并行化的影响，同时结合 OpenMP、SIMD 进行优化，进行了非阻塞通信方式、流水线形式的优化；此外，实验还在 Windows x86 环境下进行了实验，并与 ARM 环境下的性能进行了对比。

关键字：并行；高斯消元优化；MPI；数据划分；非阻塞；流水线

## 1 概述

基于高斯消元并行优化的期末选题报告，本次实验采用 MPI 优化方法进行多进程并行编程，根据 MPI 特性设计实验思路，并结合 SIMD 和 pthread/OpenMP，分别对普通高斯消元和特殊高斯消元进行不同程度的优化尝试。

### 1.1 实验平台

本次实验中，ARM 基于华为鲲鹏云平台进行，x86 平台中的实验基于 Code::Block 进行。平台本机处理器信息如下图1.1所示：


Processor						
Name	Intel Core i5 12500H					
Code Name	Alder Lake	Max TDP	45.0 W			
Package	Socket 1744 FCBGA					
Technology	10 nm	Core VID	1.273 V			
Specification	12th Gen Intel®Core™i5-12500H					
Family	6	Model	A	Stepping	3	
Ext. Family	6	Ext. Model	9A	Revision	L0	
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3, SHA					
Clocks (P-Core #0)			Cache			
Core Speed	4489.02 MHz		L1 Data	4 x 48 KB + 8 x 32 KB		
Multiplier	x 45.0 (4.0 - 31.0)		L1 Inst.	4 x 32 KB + 8 x 64 KB		
Bus Speed	99.76 MHz		Level 2	4 x 1.25 MB + 2 x 2 MB		
Rated FSB			Level 3	18 MBytes		
Selection	Socket #1		Cores	4P + 8E	Threads	16

图 1.1: x86 处理器信息

### 1.2 正确性验证说明

在进行高斯消元算法的并行化程序开发时，确保结果的准确性是至关重要的。由于高斯消元过程中的数据具有强烈的时序依赖性，验证最终输出矩阵的正确性可以为我们提供算法正确执行的强有力证据。在实际操作中，用具有确定性特征的矩阵，如对角矩阵、上三角矩阵或下三角矩阵，验证算法的正确性；如果测试程序输出正确，此后重复测试过程中，仅比对输出矩阵各元素均值，若无误，则认为并行化程序正确性得到检验，数据可以采信。

### 1.3 实验用时测量说明

在时间性能的测量上，x86 平台与 ARM 平台分别选用高精度计时函数 QueryPerformanceCounter() 与 time.h 来计时，并通过重复多次取平均值的策略计算单次程序/函数运行时间。且由于精确计时函数开销巨大，实验中统一在循环体外进行计时，忽略循环条件判断等微小开销。

## 1.4 普通高斯消元串行算法

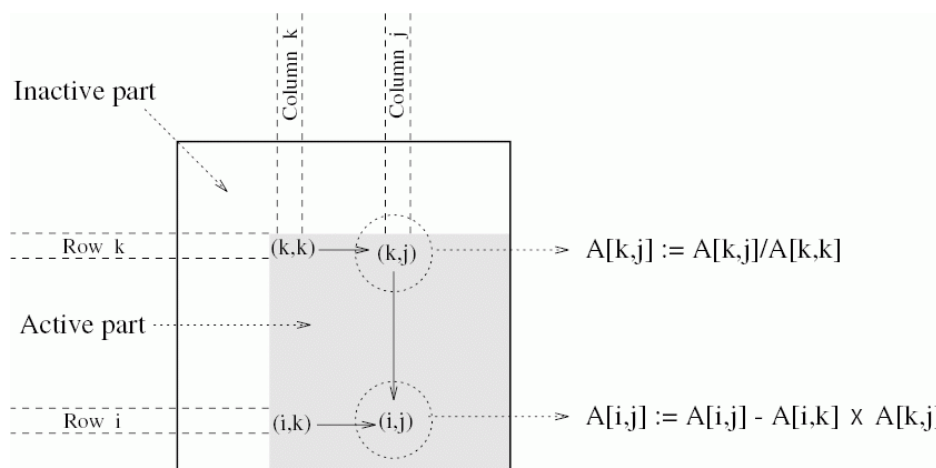


图 1.2: 高斯消去法示意图

高斯消去的计算模式如图1.2所示，主要分为消去过程和回代过程。在消去过程中进行第  $k$  步时，对第  $k$  行从  $(k, k)$  开始进行除法操作，并且将后续的  $k + 1$  至  $N$  行进行减去第  $k$  行的操作。串行代码的算法如下面伪代码（算法 1）所示：

---

### Algorithm 1 Gaussian Elimination 串行算法

---

```

1: procedure GAUSSIAN_ELIMINATION( $A, B$ )
2:    $n \leftarrow \text{size}(A)$  ▷ 消元过程
3:   for  $k$  from 1 to  $n$  do
4:     for  $i$  from  $k + 1$  to  $n$  do
5:        $\text{factor} \leftarrow \frac{A[i, k]}{A[k, k]}$ 
6:       for  $j$  from  $k + 1$  to  $n$  do
7:          $A[i, j] \leftarrow A[i, j] - \text{factor} \times A[k, j]$ 
8:       end for
9:        $B[i] \leftarrow B[i] - \text{factor} \times B[k]$ 
10:    end for
11:  end for
12: end procedure

```

---

对于本问题，在初始化数据时，我们设计了一个 `reset` 函数，令主对角线上的元素  $m1[i][i] = 1$ ，令上三角区域的数据  $m1[i][j] = \text{rand}() \% 1000 + 1$ ，下三角区域的数据  $m1[i][j] += m1[k][j]$ 。这种初始化可以重复生成样例、避免产生无穷数、检查程序正确性等。

## 2 普通高斯消元 MPI 优化

### 2.1 MPI 优化思路

#### 2.1.1 MPI 介绍

MPI 是一种用于编写并行程序的通信库接口，全称为 Message Passing Interface（消息传递接口），可以理解为是一种独立于语言的信息传递标准，其有着多种具体实现，本次则探究了 ARM 平台上的 MPICH 和 x86 平台上的 MS-MPI。MPI 库包含了一组函数和语法规则，使得计算机集群上运行的多

个进程可以相互通讯、协同工作。MPI 库的优点在于其具有可移植性和扩展性，可以支持不同操作系统和硬件平台，并且可以轻松地增加处理器数量来提高计算性能。

进程 (process) 是操作系统进行资源分配的最小单元，线程 (thread) 是操作系统进行运算调度的最小单元，进程可以包含多个线程，但每个线程只能属于一个进程。如图1.1所示，每个进程有独立的地址空间（外部框表示），同一进程内不同线程（曲线）共享该进程的内存。但是不同进程不共享地址空间，如果一个进程要访问另一个进程的数据就只能通过特定的函数进行通信，所以其任务分配和多进程的进行方式要谨慎构思。

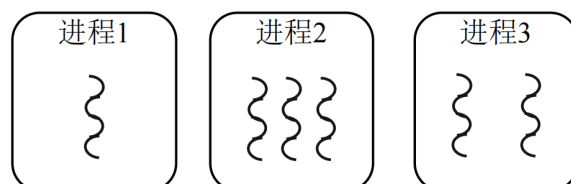


图 2.3: 进程与线程概念

### 2.1.2 算法设计

算法的主要思想包括以下几个步骤：

- 标准化对角线元素：在每一步中，首先将当前行（第  $k$  行）的对角线元素 ( $A_{kk}$ ) 通过除法操作标准化为 1，这样每步消元后，对角线上的元素都是 1。
- 消元过程：接着，使用当前行去消去其下方所有行的对应列元素。这是通过将第  $k$  行乘以一个系数，然后从下方的行中减去这个结果来实现的。
- 进程间通信：在 MPI 环境中，不同的进程可以并行地执行消元步骤。每个进程负责矩阵的一部分。当一个进程完成了其部分的消元后，它会将结果传递给其他需要这些数据来继续消元的进程。
- 数据传递：算法中使用了 `MPI_Send` 和 `MPI_Recv` 来在进程之间传递数据。`MPI_Send` 用于发送数据，而 `MPI_Recv` 用于接收数据。在消元过程中，如果一个进程需要其他进程的数据来完成消元，它会使用 `MPI_Recv` 来接收数据。
- 并行处理：算法通过在多个进程之间分配任务来实现并行处理，这样可以显著减少总体的计算时间，尤其是在处理大型矩阵时。

在这个算法中，每当一个进程将某行向量消元并且对角线元素置 1 之后就会将其数据广播到整个进程域；这样的方法有一个好处，就是在最终消元之后不必再次将结果传回给 0 号线程，在整个算法执行完后即可在 0 号线程中得到正确结果。

## 2.2 不同进程数的探究

在 arm 平台上进行 100, 500, 1000, 1500, 2000, 3000 规模的 MPI 基础算法不同线程数探索，进程数分别为 4 进程、8 进程、12 进程。以串行算法的耗时与并行算法的耗时之比作为优化力度评估优化算法的影响，对于规模小的程序，运行时以 1 秒为限进行循环，循环结束后除以循环次数来获取平均值；对于规模大的程序，则采用重复运行 5 次来取平均值。得出实验结果如下（单位：s）

从图像2.4和数据中，可以得出以下结论：

**Algorithm 2** MPI 优化的普通高斯消元算法

---

```

1: function GAUSSIAN_ELIMINATION(Matrix A, begin, end, total)
2:   for  $k \leftarrow 1$  to  $n$  do
3:     if  $begin \leq k < end$  then
4:       for  $j \leftarrow k + 1$  to  $n$  do
5:          $A_{nj} \leftarrow \frac{A_{nj}}{A_{rk}}$  ▷ 标准化, 使对角线元素为 1
6:       end for
7:        $A_{nk} \leftarrow 1.0$ 
8:       for  $j \leftarrow 0$  to  $total$  do
9:         MPI_Send(&Ak0, N, MPI, j, ...) ▷ 将除法部分完成的行向量数据传递给各进程
10:      end for
11:     else
12:       MPI_Recv(&Ak0, N, MPI, src, ...) ▷ 注意, 数据源进程 Src 需根据 k 进行判断
13:     end if
14:     for  $i \leftarrow \max(k + 1, begin)$  to  $end$  do
15:       for  $j \leftarrow k + 1$  to  $n$  do
16:          $A_{ij} \leftarrow A_{ij} - A_{ik} \cdot A_{kj}$ 
17:       end for
18:        $A_{ik} \leftarrow 0$ 
19:     end for
20:   end for
21: end function

```

---

进程数	100	500	1000	1500	2000	3000
平凡算法	0.025	0.31	2.58	8.74	22.1	74.57
4 进程	0.084	0.74	1.91	5.56	13.33	44.74
8 进程	0.2	0.96	2.41	4.6	8.59	30.32
12 进程	0.35	1.59	3.59	6.28	9.61	25.55

表 1: 不同进程数下 MPI 基础优化用时

- 多进程 MPI 的初始负优化效果:** 在数据规模较小 (例如 100 和 500) 时, 多进程 MPI 可能会因为进程间通信和数据同步的开销而导致性能下降, 即所谓的负优化效果。这可能是因为在小数据规模下, 进程间通信和数据同步的时间占总执行时间的比例较大, 从而抵消了并行计算带来的速度提升。
- 数据规模对优化效果的影响:** 随着数据规模的增加 (从 1000 开始), 多进程 MPI 开始展现出优化效果。这是因为随着数据量的增加, 每个进程可以处理更多的数据, 从而减少了进程间通信和同步的相对成本。
- 进程数对优化效果的影响:** 在数据规模较大时, 更多的进程数可以带来更好的优化效果。例如, 8 进程 MPI 在 1500 规模后开始超过 4 进程 MPI, 而 12 进程 MPI 在 2000 规模时超过 4 进程, 并且在 3000 规模时成为最优。这说明随着数据规模的增加, 更多的进程可以更有效地并行处理数据, 从而提高整体性能。
- 进程数和数据规模的匹配:** 在不同的数据规模下, 最优的进程数是不同的。在实际应用中, 需要根据数据规模和计算需求来选择合适的进程数, 以达到最佳的并行优化效果。

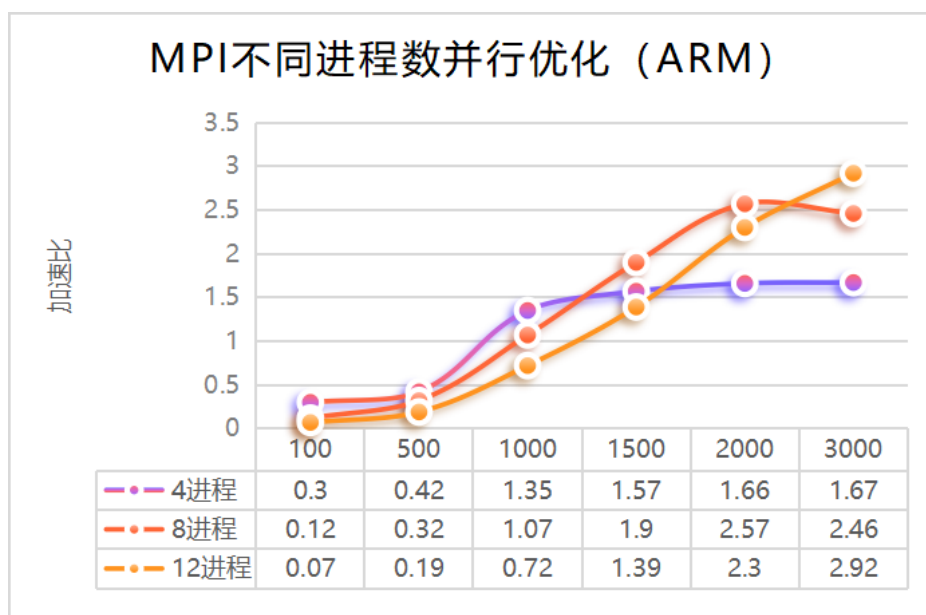


图 2.4: MPI 不同进程数并行优化加速比

### 2.3 数据划分方式的探究

在上一节的测试中用的是块划分(**Block Decomposition**)方式,每个线程给予  $N/MPI\_Comm\_size$  个行向量的任务量,而当  $N \% MPI\_Comm\_size \neq 0$  时,最末尾的一个进程会被赋予额外的任务量,最多可以是  $MPI\_Comm\_size - 1$  个行向量,因此可以尝试其他的任务划分方式。

一种是利用循环划分(**Cyclic Decomposition**)的方法,进行等步长的划分,以  $MPI\_Comm\_size$  为间距为进程划分任务,可以做到即使末尾有多出来的任务,也能分配给前  $N \% MPI\_Comm\_size$  个进程,从而达到更均衡的任务分配。但是这样的方式在相互传递信息和进行接收信息、进行消去部分的代码需要重新设计,略微复杂。

#### 循环划分代码设计

```

1  for (j = 1; j < total; j++) { //等步长的信息传递
2      for (i = j; i < N; i += total) {
3          MPI_Send(&A[i][0], N, MPI_FLOAT, j, 1, MPI_COMM_WORLD);
4          //1 是初始矩阵信息, 向每个进程发送数据
5      }
6  }
7  ...
8  int begin = k;
9  while (begin % total != rank) //等步长的消去操作
10     begin++;
11  for (i = begin; i < N; i += total) {
12      for (j = k + 1; j < N; j++) {
13          A[i][j] = A[i][j] - A[i][k] * A[k][j];
14      }
15      A[i][k] = 0;
16  }
17 }
```

还有一种方式是在块划分的基础上，将末尾多余的行向量均分给前  $N \% MPI\_Comm\_size$  个进程，这样的方式在进程数较多（即末尾余下数据越多）时能使各进程间负载更均衡，防止出现最末尾进程的任务量明显高于其余其他进程的情况。

实验数据和图2.5如下（耗时（s））：

数据划分方式	1000	1500	2000	3000
平凡算法	2.58	8.74	22.1	74.57
普通块划分	2.04	5.89	14.19	49.16
任务更均衡的块划分	1.95	5.81	14.15	45.3
循环划分	1.76	5.82	12.97	43.71

表 2: MPI 不同数据划分方式耗时

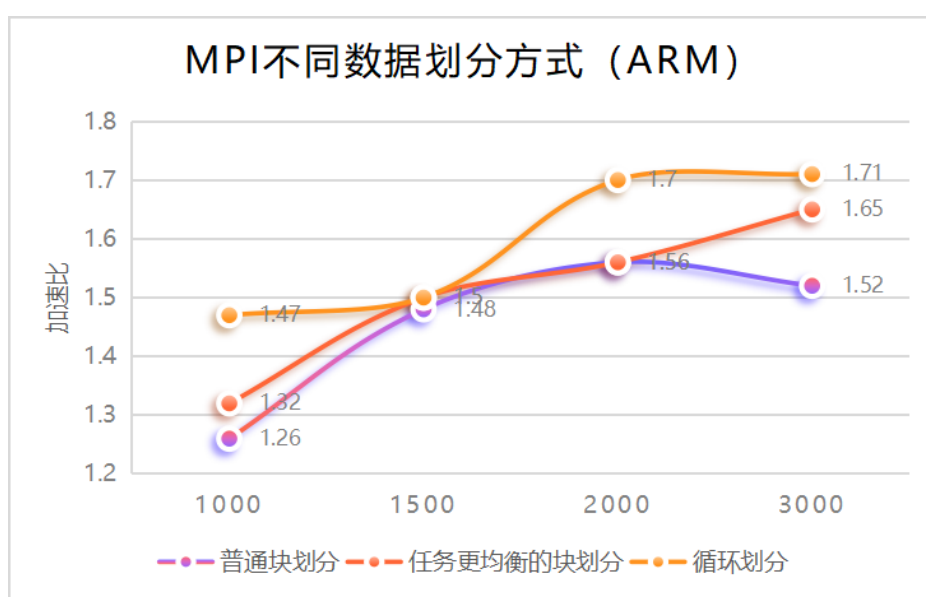


图 2.5: MPI 不同数据划分方式加速比

实验结果表明：普通的块划分耗时最长，这可能是因为这种划分方式没有考虑到数据的局部性和任务的均衡性，导致某些进程可能在等待其他进程完成，或者某些进程的工作负载比其他进程重。负载均衡后的块划分与原始块划分的性能表现相近，这是因为它们的总体结构相同，而且当数据规模较小时，二者差别不大，除非是单一一行行向量数据巨大（3000 规模），或进程数目较多时，二者差别才会明显。循环划分的性能相对较好，它是一种更为平衡的任务划分方式，循环划分通过循环地将数据分配给各个进程，可以更好地平衡各个进程的工作负载，从而提高整体性能。

### 3 ARM 平台与 x86 平台对比

本次在 x86 平台上的实验选用了 MS-MPI 这一种 MPI 接口的实现，MS-MPI 是 Microsoft Message Passing Interface 的简称，是微软公司开发的一种用于编写并行分布式计算程序的通信库。MS-MPI 主要用于在 Windows 平台上开发高性能计算应用程序，因此此次 MS-MPI 将搭配 Visual Studio 2019 进行 MPI 编程实验。

实验环境和编译选项：



- Visual Studio 2019 集成开发环境
- MS-MPI 分布式计算程序通信库
- ISO C++14 标准
- 预处理器定义 MPICH\_SKIP\_MPICXX

### 3.1 不同进程数的探究

x86 平台的 MPI 优化的思路与 ARM 平台的相同，代码上也大同小异，测试规模 100, 500, 1000, 1500, 2000, 3000，具体测试结果如下 (ms)：

进程数	100	500	1000	1500	2000	3000
平凡算法	0.5	140.5	1047.75	3518.75	8467.74	28751.34
4 进程	0.59	50.07	385.57	1309.24	2880.15	10906.46
6 进程	1.06	41.46	313.23	1061.31	2215.07	7858.7
8 进程	1.81	37.41	289.42	934.04	2006.88	6997.9

表 3: 不同进程数下 MPI 基础优化用时 (x86)

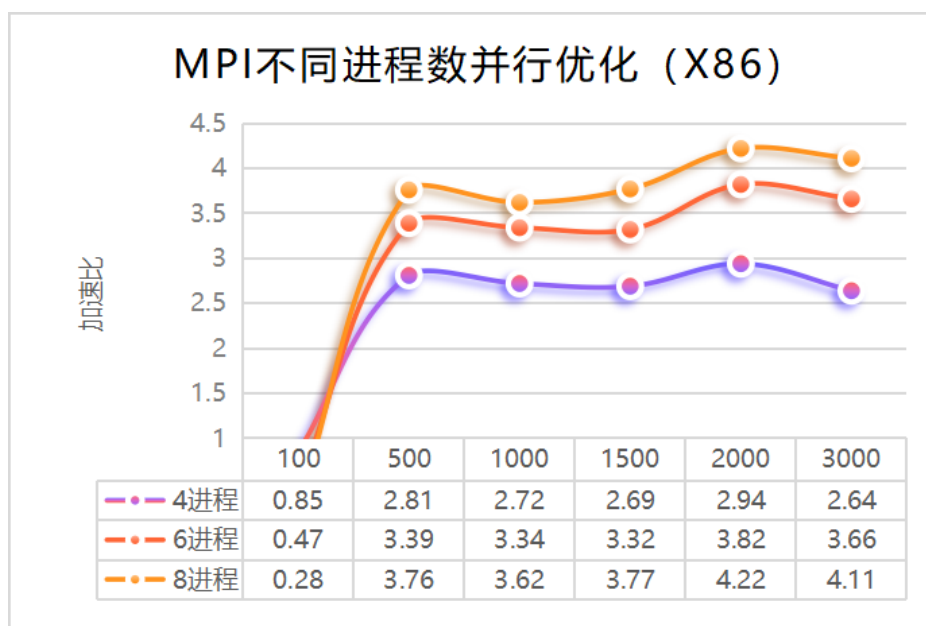


图 3.6: MPI 不同进程数并行优化 (x86)

和 arm 平台 (图2.4) 表现类似，数据量较小时，MPI 多进程反而会有负优化的作用，大概是因为进程越多耗时越多，管理进程的耗时越多。MPI 有很好的加速效果，特别是在 x86 平台上，在数据规模较大时四进程时就已经有很明显的优化效果，加速比稳定在 2.5 以上 (如图3.6)，而当增加进程数至 6 和 8 时，加速比还有进一步的提升，在数据规模较大 (3000) 时，八进程的加速比超过了 4，有着优异的表现。

### 3.2 数据划分方式的探究

这部分主要对循环划分和块划分以及均衡负载后的块划分这三种方式进行探究，将在 x86 平台上，四线程的设置下检测不同划分方式和编程策略的差别，经过多次实验后取得数据如下（单位：ms）：

数据划分方式	500	1000	1500	2000	3000
块划分	2.81	2.71	2.69	2.89	2.67
任务更均衡的块划分	2.84	2.77	2.69	2.87	2.69
循环划分	2.77	3.43	3.63	3.73	3.6

表 4: MPI 不同数据划分方式耗时 (x86)

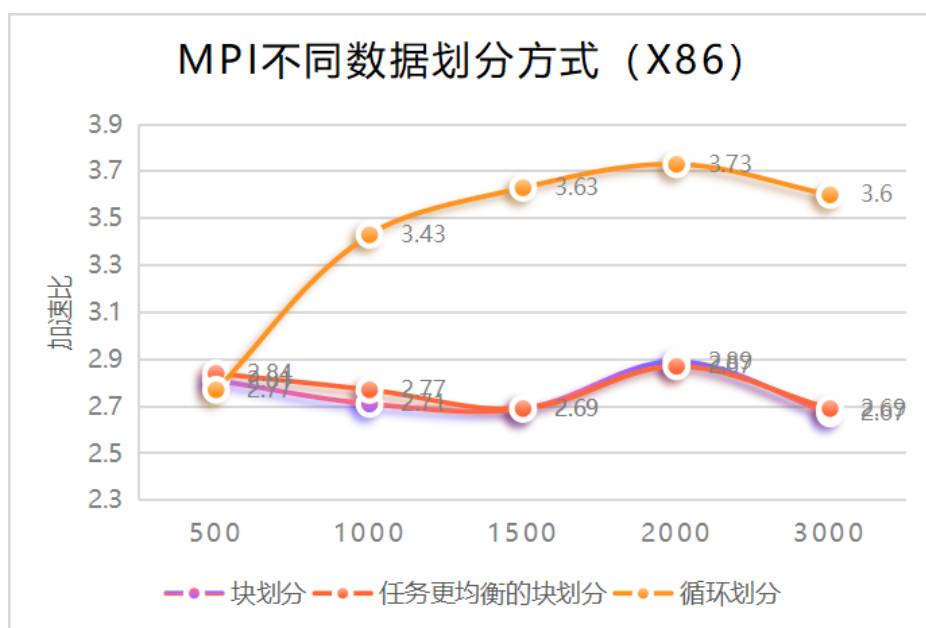


图 3.7: MPI 不同数据划分方式 (x86)

对比图3.7和图2.5：

- 负载均衡后的块划分的方式和普通块划分差别不大，这是因为 4 进程时，末尾的多余任务为小于 4 个行向量，除非是每个行向量的计算量巨大，否则这种方法和普通块划分差别微小。
- 循环划分是所有里面耗时最少的，因为这种方式任务划分是十分均衡的，加速比相对于块划分的提升是较为明显，在 2000 规模时，两种块划分的加速比在 2.89 左右，而循环划分加速比在 3.73，是块划分的 1.3 倍左右，这一点比较与 arm 平台实验结果类似。
- x86 平台对于任务划分的优化性能更敏锐，相比 arm 可以更高效地实现 MPI 的多进程并行化，因为 arm 数据划分的最优性能加速比为 1.7，而 x86 加速比可达 3.73，是 arm 平台加速比的两倍多。

## 4 进阶要求

### 4.1 非阻塞通信 MPI 的探索

非阻塞通信策略，即发送进程将在消息被接收进程接收之前一直等待的通信方式，直到接收进程成功接收消息后，发送进程才能继续执行；阻塞通信简单易用，由于发送进程要一直等待接收进程的响应，因此可以避免出现竞争条件。但是，这会导致整个程序的效率降低，因为发送进程必须等待接收进程处理完消息之后才能继续执行。

在高斯消元算法中，每个进程发送的数据相互独立，不会相互干扰，因此适合采取非阻塞通信，用 `MPI_Isend(void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request)` 进行数据的非阻塞传递。但要注意，在这次实验的算法的数据接收一定要等到接收完毕后才能执行下一步，所以要用 `MPI_Wait(&request, MPI_STATUS_IGNORE)` 进行阻塞。

在 x86 平台，四进程，块划分上运用非阻塞策略得到的结果如下 (单位: ms):

通信方式	500	1000	1500	2000	3000
平凡算法	140.5	1047.75	3518.75	8467.74	28751.34
阻塞版本 MPI	50.41	364.11	1283.72	3075.4	10745.62
非阻塞版本 MPI	45.86	298.95	1023.52	2872.43	10167.29

表 5: 改变通信方式 MPI 优化

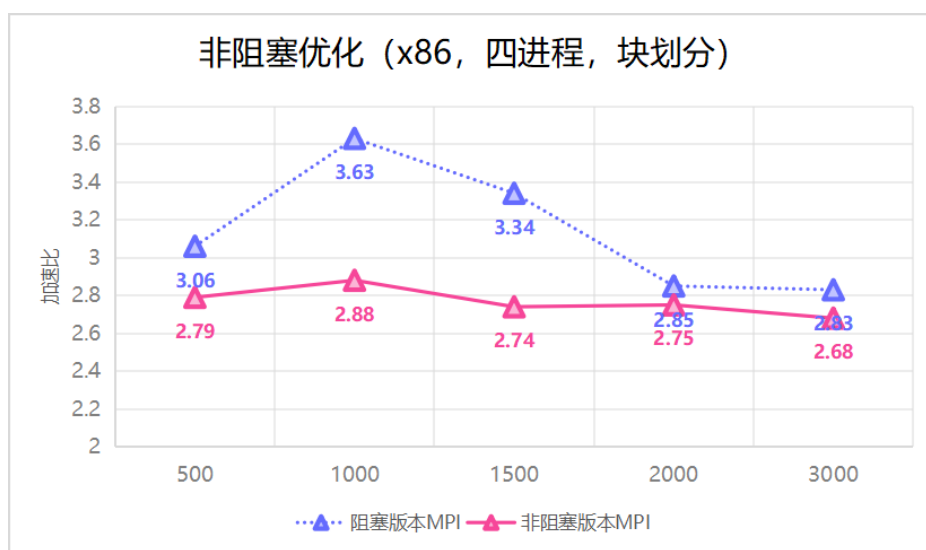


图 4.8: 非阻塞优化 (x86, 四进程, 块划分)

从实验结果来看，运用非阻塞技术是有效且合理的，最高可达 **3.63 加速比** (1000 数据规模)，能切实提升 MPI 程序的速度和效率，虽然在更大数据规模中提升有限，但是仍然效果明显。后续实验设计如果在更复杂的环境条件下，需要更谨慎判断非阻塞是否会造成数据传送的混乱。

### 4.2 MPI 流水线形式的改进

在先前的算法中，每个进程会将消元完毕的行向量传递给到整个进程域，然而在块划分中，实际上只有该进程之后的进程需要使用该行向量，因此只用将数据传递给相邻之后的进程。流水线一般操作，这样可以减少约一半的数据传递操作，最终在最后一个进程中得到最终结果。

在设计代码中，根据上述探究的非阻塞通信，只需要当前进程非阻塞地传递数据给之后的进程即可，这样可以避免每个进程接收数据后又传递的“搬运”过程。

流水线改进	1000	1500	2000	3000
ARM 普通 MPI (s)	1.94	5.85	13.42	46.52
ARM 流水线版 MPI (s)	1.52	4.13	9.2	30.63
x86 普通 MPI (ms)	389.63	1312.24	2873.25	10825.35
x86 流水线版 MPI (ms)	324.12	1101.77	2321.26	8208.36

表 6: MPI 流水线改进

有实验结果可知，减少信号传递次数后的流水线版本 MPI，在速度上确实有提升，可以使得算法执行更快；但是提升幅度不算很高，推测可能是因为原先的运行中，数据传递占据的时间就不长，因此提升不明显，但是流水线改进依然有用。

### 4.3 MPI 和多线程和 SIMD 的结合

在上一次的实验中，将 OpenMP 和 SIMD 结合后获得了更高的加速比，此次可以尝试将 MPI、OpenMP、SIMD 三者结合起来，根据多进程的特点可以判断三者结合后能更好的处理大规模数据。

从 MPI 算法中可以发现，在执行行消去的部分时，每个进程要负责  $N/\text{total}$  个行向量的消去，此时可以结合 OpenMP 进行多线程优化，每个线程负责  $N/\text{total}/\text{threads}$  个行向量，并且采用了任务分配更灵活均衡的 guided 方式，能够有效加快每个进程的执行速度。

而在除法部分，或是对单一行进行消去时，可以结合 SIMD 的技术进行优化，一次性对一行中多个元素同时除去对角线元素或同时进行行间的消去，也能达到提升执行速度的效果。此次选择 x86 平台的 AVX 指令集进行 SIMD 的并行优化，能一次处理 8 个 float 类型的元素。得到测试结果如下（单位 ms）：

结合方式	500	1000	1500	2000	3000
普通 MPI	50.41	364.11	1283.72	3075.4	10745.62
非阻塞 MPI	45.86	298.95	1023.52	2872.43	10167.29
非阻塞 MPI+AVX	25.28	145.63	782.16	1920.57	5825.33
非阻塞 MPI+OpenMP	20.35	90.82	301.24	650.25	3524.42
非阻塞 MPI+OpenMP+AVX	18.88	53.54	160.19	361.9	1748.55

表 7: MPI 结合 OpenMP & SIMD (x86, 块划分)

三种优化手段结合，得到结论：

- 在本次结合了 MPI 的多进程优化后，高斯消元算法有多个进行分配任务，每个进程里又有多个线程进行并行计算，每个线程又会利用 AVX 指令集进行 SIMD 加速，最终达到的加速比最高可达 23.4。
- 当数据量较小（500）时加速比的提升无论是和 AVX、OpenMP 单独结合还是同时结合二者，加速效果都不明显，三种方式差距也不大，推测认为是因为数据量较小，无法完全体现加速优势。
- 当数据量增加时，加速比一度升高，而在 3000 规模时，可能由于进程间通信增多或计算量增大等，同时结合 SIMD 和 OpenMP 的加速比降低至 16.44。

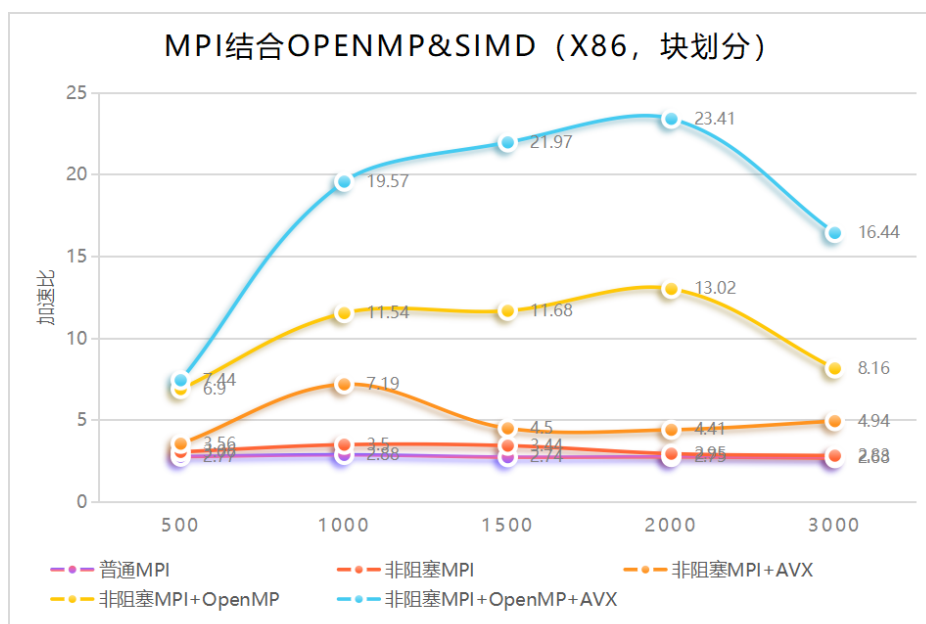


图 4.9: MPI 结合 OpenMP &amp; SIMD 加速比

- 依据上面非阻塞通信带来的优势，结合 AVX、OpenMP 后又带来了新的飞跃，这三者的有机结合使得算法大大加速。

## 5 基于 Gröbner 基计算的特殊高斯消元

### 5.1 问题简介

特殊高斯消元算法在有限域  $GF(2)$  上进行，主要涉及两个操作：消元子和被消元行的操作。该算法的流程是：

- 将文件中的消元子和被消元行读入内存。
- 对每个被消元行，检查其首项，如有对应消元子，则将其减去（异或）对应消元子，重复此过程直至其变为空行（全 0 向量）或首项在范围内但无对应消元子或该行。
- 如果某个被消元行变为空行，则将其丢弃，不再参与后续消去计算；如其首项被覆盖，但没有对应消元子，则将它“升格”为消元子，在后续消去计算中将以消元子身份而不再以被消元行的身份参与。
- 重复第二步和第三步的过程，直至所有被消元行都处理完毕。

本次实验将对特殊高斯消元进行 MPI 优化，延续之前的设计思路，探讨多进程、结合 OpenMP 和 SIMD 的优化效果，在 x86 平台进行代码编写并实验验证。

### 5.2 算法设计

其 MPI 优化算法的代码如下所示，其中  $R[i]$  代表首项为  $i$  的消元子， $E[i]$  代表第  $i$  个被消元行， $lp(E[i])$  代表被消元行第  $i$  行的首项。

探究特殊高斯算法设计中的思路：

**Algorithm 3** MPI 优化的特殊高斯消元算法

---

```

1: function GAUSSIAN_ELIMINATION(MatriA, begin, end, total)
2:   for i = begin to end do
3:     while Rip(E,)  $\neq$  NULL do                                ▷ 先按照原本的消元子对行向量进行消元
4:       进行异或消元
5:     end while
6:     消元完毕先不升格为消元子
7:   end for
8:   for i = 0 to rank do
9:     MPI_Recv from i                                ▷ 接收来自此前进程的结果, 可能可以对自己进程的任务进行消元
10:    将接收的结果作为消元子, 再次消元
11:   end for
12:   将自己进程的行向量逐个升格为消元子, 如果顺序执行时有冲突, 则再次消元
13:   for i = rank + 1 to total do
14:     MPI_Send to i                                ▷ 将自身消元结果发送至此后的进程
15:   end for
16: end function

```

---

- **数据划分:** 该算法中, 行之间的消元存在强烈的顺序依赖性, 即前一行的消元结果可以作为后续行的消元子。因此, 不适合使用循环划分, 因为这会增加算法的复杂性并破坏行之间的依赖关系, 最终选择了普通的块划分方式, 这有助于保持行的前后依赖关系不变。
- **通信方式:** 在算法中, 是否使用阻塞通信对程序运行的影响不大。接收消元行时, 需要记录消元行首项的信息, 因此接收部分必须是阻塞通信以避免错误。在发送消元结果时, 由于进程即将完成其任务, 即使使用非阻塞发送, 目标进程也需要阻塞接收, 因此使用非阻塞通信在这里并不必要。
- **结合多方法并行优化:** 消元部分和升格消元子部分可以通过结合 OpenMP 和 AVX 指令集进一步优化, 以提高加速比。具体方法是, 在循环消元时使用 `#pragma omp for ordered schedule(guided)` 语句, 采用 `guided` 方式进行多线程优化。在升格消元子时, 需要使用 `#pragma omp critical` 来确保消元的顺序依赖性得到保持。

关键代码如下:

## 特殊高斯消元结合 MPI、OpenMP、SIMD

```

void GE_MPI_AVX_omp(int argc, char* argv[]) {
    for (i = begin; i < end; i++) {
        int first = findfirst(i);
        while (first != -1) { // 未消元完毕, 存在首项
            if (ifBasis[first] == 1) { // 存在首项为 first 消元子
                for (j = 0; j + 8 < maxsize; j += 8) {
                    __m256i vij = _mm256_loadu_si256((__m256i*) & gRows[i][j]);
                    __m256i vj = _mm256_loadu_si256((__m256i*) & gBasis[first][j]);
                    __m256i vx = _mm256_xor_si256(vij, vj);
                    _mm256_storeu_si256((__m256i*) & gRows[i][j], vx);
                }
                for (; j < maxsize; j++) {
                    gRows[i][j] = gRows[i][j] ^ gBasis[first][j];
                }
            }
        }
    }
}

```

```

        first = findfirst(i);
    }
    else {    //升级为消元子
        .....
    }
}
}
for (i = 0; i < rank; i++) {
    int b = i * (num / total);
    int e = b + num / total;
    for (j = b; j < e; j++) {
        MPI_Recv(&answers[j][0], maxsize, MPI_INT, i, 2, MPI_COMM_WORLD,
            &status); //接收来自进程i的消元结果，可能作为之后的消元子
        int first = _findfirst(j);
        firstToRow.insert(pair<int, int>(first, j)); //记录下首项信息
    }
}
#pragma omp for schedule(guided)
for (j = begin; j < end; j++) {
    //非0进程要进行二次消元，以此前进程的结果作为消元子
    int first = findfirst(j);
    while ((firstToRow.find(first) != firstToRow.end() || ifBasis[first] == 1) &&
        first != -1) { //存在可消元项
        if (firstToRow.find(first) != firstToRow.end()) { //消元结果有消元子
            int row = firstToRow.find(first)->second;
            int k = 0;
            for (k = 0; k + 8 < maxsize; k += 8) {
                __m256i vij = _mm256_loadu_si256((__m256i*) & gRows[j][k]);
                __m256i vj = _mm256_loadu_si256((__m256i*) & answers[row][k]);
                __m256i vx = _mm256_xor_si256(vij, vj);
                _mm256_storeu_si256((__m256i*) & gRows[j][k], vx);
            }
            for (; k < maxsize; k++) {
                gRows[j][k] = gRows[j][k] ^ answers[row][k];
            }
            first = findfirst(i);
        }
        if (ifBasis[first] == 1) {
            int k = 0;
            for (k = 0; k + 8 < maxsize; k += 8) {
                .....
            }
            for (; k < maxsize; k++) {
                gRows[j][k] = gRows[j][k] ^ gBasis[first][k];
            }
            first = findfirst(i);
        }
    }
}
}
}
}

```



```

MPI_Finalize();
}

```

### 5.3 实验结果

实验环境与 x86 的 MPI 优化普通高斯消元相同 (单位:ms):

结合方式	Example Number									
	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10
平凡算法	0	10	11	298	929	10259	58730	401253	758432	2358545
4 进程 MPI	0	4	4	115	308	4362	19646	140619	276549	920784
4 进程 +8 线程 OpenMP+AVX	0	7	7	102	272	2764	10532	71056	98524	320412
8 进程 MPI	0	6	4	98	203	2374	12293	84908	131237	381230
8 进程 +8 线程 OpenMP+AVX	1	9	10	108	238	1825	9918	45251	80160	240411

表 8: 特殊高斯消元 MPI 优化耗时

由于例 1 规模太小, 无明显加速比, 则从例 2 开始进行讨论加速比:

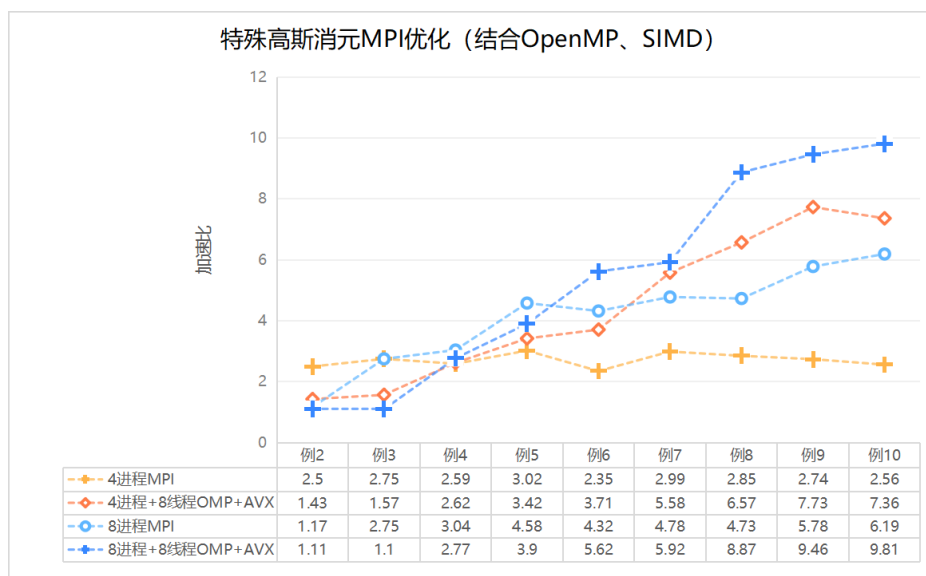


图 5.10: 特殊高斯消元 MPI 优化 (结合 OpenMP、SIMD)

根据实验结果, 以下是对特殊高斯消元算法在 MPI 多进程优化、OpenMP 多线程和 SIMD 指令集 (AVX) 结合使用时的性能表现的总结:

- MPI 多进程优化效果显著: 在进行特殊高斯消元算法的并行化时, 使用 MPI 进行多进程优化能够获得明显的加速效果。具体来说, 4 进程 MPI 普遍能够达到大约 2.5 到 3 倍的加速比, 而 8 进程 MPI 在处理大规模数据时能够达到 5 到 6 倍的加速比, 显示出良好的扩展性。
- 小规模数据的优化策略: 对于小规模数据, 单纯增加进程数量或结合多线程优化可能并不是最佳选择。这是因为进程和线程的管理开销以及进程间通信的开销可能会随着进程数量的增加而增长, 从而抵消了并行计算带来的性能提升。



- 大规模数据下的最优配置：在处理极大规模的数据（如例 10 所示）时，结合 8 个进程、8 个线程以及 AVX 指令集的优化策略表现出最佳性能，实现了接近 10 倍的加速比。这是本学期在特殊高斯消元算法中达到的最高加速比。

综上所述，特殊高斯消元算法在并行化时需要根据数据规模的不同选择合适的并行策略，MPI 多进程优化结合 OpenMP 多线程和 AVX（SIMD）指令集可以在大规模数据处理中实现显著的加速效果。

## 6 总结与反思

本次实验针对 MPI 优化问题做了一系列探究，探索多进程并行化随着数据规模变化的效果，又根据数据运算特征进行任务划分，对比块划分、循环划分和负载均衡后的块划分，实验结果证明循环划分可以显著提高性能，考虑了负载均衡的优化方法在并行计算中起到了关键作用。

在进阶部分探究了非阻塞通信和流水线改进，实验结果表明，非阻塞通信一定程度上可以提升加速比；流水线方式可以降低等待时延，从而提高性能。结合前两次实验的优化方法，逐一比较结合效果，这种综合优化方法可以充分利用多个层次的并行性，加速高斯消元问题的求解过程，得出 MPI、OpenMP、SIMD 三者结合可达到最优效果加速比 23 倍。

本次实验是高斯优化过程的关键环节，这次实验的代码设计相比于前两次顺利一些，这是因为已经熟悉高斯消元过程并且熟悉并行优化环节了，有一些困难的地方是 mpi 优化步骤较多，稍微繁杂一些，但是好在优化结构清晰，结果大致符合预期。

代码详见：[MPI 并行优化实验](#)