南開大學
Nankai University

计算机学院

并行程序设计期末研究报告

特殊高斯消去算法的并行加速

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 7 月 10 日

目录

1 问题背景 **2**
1.1 问题介绍 . . . 2
1.2 前人工作总结 . . . 2

2 实验介绍 **2**

3 程序设计思路 **2**
3.1 算法分析 . . . 2
3.2 平凡算法设计 . . . 3
3.3 算法优化方向 . . . 4
3.3.1 SIMD 优化 . . . 4
3.3.2 Pthread 与 OpenMp 优化 . . . 5
3.3.3 CUDA 优化 . . . 5
3.3.4 MPI 优化 . . . 5

4 实验环境说明 **5**

5 实验方案设计 **5**
5.1 SIMD 并行化方案设计 . . . 5
5.2 Pthread 并行化方案设计 . . . 5
5.3 OpenMp 并行化方案设计 . . . 5
5.4 CUDA 并行化方案设计 . . . 6
5.5 MPI 并行化方案设计 . . . 6

6 实验结果呈现及分析 **6**
6.1 SIMD 并行化结果 . . . 6
6.2 Pthread 并行化结果 . . . 7
6.3 OpenMp 并行化结果 . . . 8
6.4 CUDA 并行化结果 . . . 9
6.5 MPI 并行化结果 . . . 10

7 实验源码项目 GitHub 链接 **11**

1 问题背景

1.1 问题介绍

Gröbner(Gröbner Bases) 理论是计算机代数的一个基石。对于与多项式相关的或者是能够将问题转化为与多项式相关的问题，Gröbner 基的理论和技巧就能发挥重要做用。在有限域上，结合 Gröbner 基的一些良好的性质，可以在信息安全领域中的很多方面进行有效工作，如：密钥分发、加密、解密、零知识证明、数字签名、不可否认消息认证等。而且随着相关算法的进步与完善、计算机速度的提高和计算机网络的发展 Gröbner 基的实用价值进一步得到体现，其应用范围也进一步扩展，由于其良好的性质，Gröbner 基已经成为了一种进行密码分析的有力工具。然而基本的计算 Gröbner 基的生成算法的计算效率是很低的，这在一定程度上影响了 Gröbner 基的实用价值。因此，本实验我们将探究对 Gröbner 基的一种计算算法–特殊高斯消去算法进行并行优化，以提高计算的效率。[1]

1.2 前人工作总结

如下为 Gröbner 基相关高斯消去的发展历史与算法演进概述。

最初，求解代数方程组有两种相异的方法：Gröbner 基法和标准基法 [2]，这些算法都与高斯消去有关。对于 Gröbner 基计算来说，主要的代价产生于由输入方程生成理想多项式矩阵的高斯消去 [3]，这些算法生成的矩阵有不寻常特性：稀疏、几乎全是块三角形。已有代数库 M4R1、代数包 ATLAS、LinBox 等对于稠密矩阵计算有效，但并不适配于单位大小为素数的 F4/F5 矩阵。

基于此，并行的 Gröbner 基高斯消去也被研究 [3]：依据枢轴量 pivot 将原矩阵划分为四个子矩阵，在充分从 cache 中获益的情形下，每个矩阵被切分成块进行基本行变换；研究人员还针对稀疏三角矩阵、稀疏矩阵块和混合矩形块，分别进行算法实验与分析。

2 实验介绍

本实验将尝试使用不同的并行加速技术对特殊高斯消去算法进行加速，研究不同并行化方案对算法性能带来的影响。使用的并行化技术分别为：SIMD 向量化、Pthread 多线程、OpenMp 多线程、CUDA 并行化、MPI 多进程加速这几种程序并行化加速方法。

3 程序设计思路

3.1 算法分析

特殊高斯消去算法与普通高斯消去计算的区别为：

- 1 运算均为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1。其加法运算实际上为异或运算：0+0=0、0+1=1、1+0=1、1+1=0。由于异或运算的逆运算为自身，因此减法也是异或运算。乘法运算 0*0=0、0*1=0、1*0=0、1*1=0。因此，高斯消去过程中实际上只有异或运算——从一行中消去另一行的运算退化为减法。
- 2 矩阵行分为两类，“消元子” 和 “被消元行”，在输入时即给定。消元子是在消去过程中充当 “减数” 的行，不会充当 “被减数”。所有消元子的首个非零元素（即首个 1，称为首项）的位置（可通过将消元子放置在特定行来令该元素位于矩阵对角线上）都不同，但不会涵盖所有对角线元素。

被消元行在消去过程中充当“被减数”，但有可能恰好包含消元子中缺失的对角线 1 元素，此时它“升格”为消元子，补上此缺失的对角线 1 元素。

首先考虑算法的执行流程，我们需要循环遍历每一个消元行对其进行消元。对于每一个消元行的消元，消元过程首先寻找是否有消元子与该行具有相同的首项，若存在满足条件的消元子，则用相应的消元子对该消元行进行消去，即将消元子的每一位和消元行进行异或操作，之后判断消元行是否消元为 0，若仍未被消元为 0，则重复寻找消元子继续进行消元；若没有满足条件的消元子，则该消元行“升格”为消元子，参与后续消元行的消元过程，并结束该消元行的消元过程。

伪代码如下：

```
//R: 所有消元子构成的集合
//R[i]: 首项为 i 的消元子
//E: 所有被消元行构成的数组
//E[i]: 第 i 个被消元行
//lp(E[i]): 被消元行第 i 行的首项
for i := 0 to m - 1 do
    while E[i]! = 0 do
        if R[lp(E[i])]! = NULL then
            E[i] := E[i] - R[lp(E[i])]
        else
            R[lp(E[i])] := E[i]
            break
        end if
    end while
end for
return E
```

3.2 平凡算法设计

考虑实现算法使用的数据结构。首先是数据的存储，为了节约内存，我们使用位图的形式存储矩阵，并将消元行和消元子存储在同一个二维矩阵 A 之中，矩阵的前 NUME 行为消元行，后 NUMR 行为消元子行。同时，我们还需要记录每一行的首项值，定义一位数组 lp 进行值的存储。最后，为了快速地查找与某一消元行具有相同首项的消元子，我们使用 *unordered_map* 数据结构，该数据结构维护了一个哈希表，可以实现 key 到 value 的快速查找，将变量名定义为 lpE2R。具体平凡算法代码如下所示：

```
//平凡算法
for (int i = 0; i < NUME; i++) {
    while (lp[i] > -1) { //lp[i] 为-1 则表示该行被消为零
        if (!(lpE2R.find(lp[i]) == lpE2R.end())) { //寻找符合的消元子
            int rowR = lpE2R[lp[i]];
            bool lpHasChanged = false;  //该 bool 变量用于判断本消元行是否消元为零
```

```
7               int p = totalCols - 1;
8               for (p; p >= 0; p--) {
9                   A[i][p] ^= A[rowR][p]; //消元过程
10              }
11              for (p = totalCols - 1; p >= 3; p -= 4) {
12              //此循环及接下来的循环为更新该消元行的首项值
13              //此处为了加快速度使用循环展开
14                  int x = ((A[i][p] | A[i][p - 1]) | (A[i][p - 2] | A[i][p - 3]));
15                  if (x == 0)
16                      continue;
17                  break;
18              }
19              for (p; p >= 0; p--) {
20                  if (A[i][p] != 0) { //消元行的首项值一定在第一个不为零的 32 个 bit 中
21                      lpHasChanged = true;
22                      for (int k = 31; k >= 0; k--) {
23                          if ((A[i][p] & (1 << k)) != 0) {
24                              lp[i] = p * 32 + k;
25                              break;
26                          }
27                      }
28                      break;
29                  }
30              }
31              if (lpHasChanged == false) //判断该消元行是否被消元为零
32                  lp[i] = -1;
33              }
34          else {
35              lpE2R[lp[i]] = i;   //找不到合适的消元子，该消元行成为消元子
36              break;
37          }
38      }
39  }
```

3.3 算法优化方向

对特殊高斯消去平凡算法进行分析不难看出，我们可以对上述特殊高斯消去算法中第 8 到 10 行的循环进行并行化处理，利用不同的并行化技术对其进行优化。

3.3.1 SIMD 优化

我们可以对上述所提到的循环进行 SIMD 优化，可以结合不同的 SIMD 优化方法如 NEON、SSE 等对算法进行优化。

3.3.2 Pthread 与 OpenMp 优化

多线程方面，我们可以使用 Pthread 多线程及 OpenMp 多线程对上述循环进行并行化优化，每个线程负责循环中一部分元素的消去。

3.3.3 CUDA 优化

我们可以将上述循环编写为 GPU 的核函数，使用 GPU 进行被消元行的并行消去任务。

3.3.4 MPI 优化

类似的，我们对特殊高斯消去串行算法中执行消去的循环进行 MPI 优化。
具体优化算法请见文末 Github 链接。

4 实验环境说明

我们主要使用三个不同的实验平台，分别为：ARM 架构的鲲鹏服务器、Intel DevCloud 服务器、x86 架构的个人 PC。个人 PC 主要参数如表1所示：

CPU Name	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
CPU NumberOfCores	4
CPU NumberOfLogicalProcessors	8
OS Version	Microsoft Windows 11

表 1: 个人 PC 主要参数

5 实验方案设计

5.1 SIMD 并行化方案设计

分别对特殊高斯消去算法进行 NEON 和 SSE 优化，并在课程提供的测试数据的前七组数据集上进行测试，采集不同平台不同优化算法程序运行的时间，进行对比分析。其中 ARM 平台使用鲲鹏服务器，并使用 g++ 编译器编译程序；x86 平台使用 Intel DevCloud 环境，并使用 icpc 编译器编译程序。

5.2 Pthread 并行化方案设计

对特殊高斯消去算法进行 Pthread 多线程并行化，并在课程提供的测试数据的前七组数据集上进行平凡算法和 Pthread 多线程算法性能测试，对比分析多线程算法性能。实验只在鲲鹏服务器中进行，使用 g++ 编译器编译程序。

5.3 OpenMp 并行化方案设计

实验将分别在 ARM 平台和 x86 平台进行。在两平台中分别测试 OpenMp 算法以及 OpenMp 结合 SIMD 算法的性能，并将其与平凡算法及 Pthread 算法进行对比。实验将分别使程序在 2、4、8 线程数目下运行给定的前 7 组测试数据，并对比不同算法的运行时间。其中，ARM 平台为鲲鹏服务器，使用 g++ 编译器。x86 平台为 DevCloud 服务器，使用 icpx 编译器。

5.4 CUDA 并行化方案设计

CUDA 并行化优化方面，我将编写特殊高斯消去的 CUDA 优化程序，并在个人 PC 上对课程提供的测试数据的前七组数据集进行测试，分析算法的性能。

5.5 MPI 并行化方案设计

对特殊高斯消去算法进行 MPI 并行化，并尝试结合 OpenMp 和 SIMD 对程序加速。在个人 PC 上进行性能测试，在课程提供的测试数据的前七组数据集上进行测试，分析算法的性能，探究不同进程数下 MPI 算法性能特点以及结合 OpenMp 和 SIMD 对 MPI 算法的影响。

6 实验结果呈现及分析

6.1 SIMD 并行化结果

根据实验方案进行实验，得到不同平台平凡算法与优化算法的运行时间如表2所示。表3及图6.1为两平台优化算法的加速比。

平台\数据集编号		1	2	3	4	5	6	7
鲲鹏	平凡算法	0.0191553	0.745921	0.908	26.811	152.731	2534.5	32054.9
	NEON 优化算法	0.0194575	0.761949	0.921733	26.3242	137.913	2021.87	21608.3
DevCloud	平凡算法	0.0035091	0.183244	0.200015	5.33096	18.837	193.661	2136.21
	SSE 优化算法	0.0015959	0.162156	0.192901	2.04701	8.85248	131.617	1381.48

表 2: 特殊高斯消去算法 SIMD 优化不同平台下平凡算法与优化算法运行时间（ms）

平台\数据集编号	1	2	3	4	5	6	7
鲲鹏	0.984469	0.978964	0.985101	1.01849	1.10744	1.25354	1.48345
DevCloud	2.19882	1.13005	1.03688	2.60427	2.12788	1.4714	1.54632

表 3: 特殊高斯消去算法 SIMD 优化不同平台下优化算法加速比

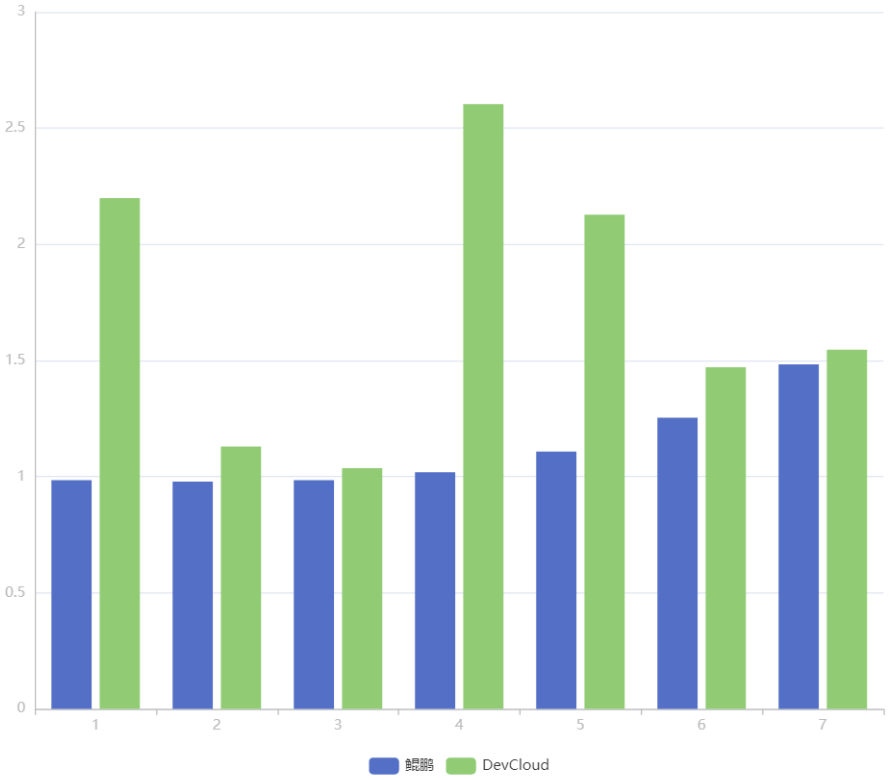

图 6.1: 特殊高斯消去算法 SIMD 优化不同平台下优化算法加速比

对结果进行分析，首先我们比较每个平台内部程序运行结果的特点。可以看到，在 ARM 平台中，当数据集规模较小时，优化算法并未取得较好的加速比，当数据规模较大时优化算法才显现出较好的加速比。这可能是由于规模较小时 NEON 的 SIMD 指令减少程序指令数带来的优化效果未能弥补具有更长时钟周期的 SIMD 指令对程序的影响。在 DevCloud 平台中，优化算法加速比变化趋势并不是十分明显，但是加速比均大于 1，且在某些数据规模下可以达到 2 以上的加速比，具有较好的加速效果。

接下来我们进行横向对比，比较两个平台之间程序运行结果的差异。首先，一个最明显的差异是 DevCloud 平台下程序的运行时间远小于 ARM 平台下程序的运行时间。在某些数据规模下，ARM 平台程序运行时间甚至达到了 DevCloud 平台程序运行时间的 10 倍以上。由此可见平台差异对程序运行时间的影响之大。此外，可以看到 SSE 优化算法相比于 NEON 优化算法具有更好的加速比。

6.2 Pthread 并行化结果

实验结果数据如表4所示。表5和图6.2为 Pthread 优化算法不同线程数下加速比结果。

算法\数据集编号		1	2	3	4	5	6	7
ordinary		0.0183503	0.70837	0.877486	25.6882	148.768	2475.54	27832.6
pthread	thread=2	0.388963	11.3767	10.1758	243.428	1203.14	11894.9	101095
	thread=4	0.634828	14.8964	15.0408	423.547	1649.8	20868.8	138333
	thread=8	1.2467	34.3823	32.7167	889.394	3594.53	42285.6	262860

表 4: 特殊高斯消去 Pthread 优化不同算法运行时间（单位：ms）

线程数\数据集编号	1	2	3	4	5	6	7
2	0.047177	0.062265	0.086233	0.105527	0.12365	0.208118	0.275311
4	0.028906	0.047553	0.05834	0.06065	0.090173	0.118624	0.2012
8	0.014719	0.020603	0.026821	0.028883	0.041387	0.058543	0.105884

表 5: 特殊高斯消去 Pthread 优化算法不同线程数下程序加速比

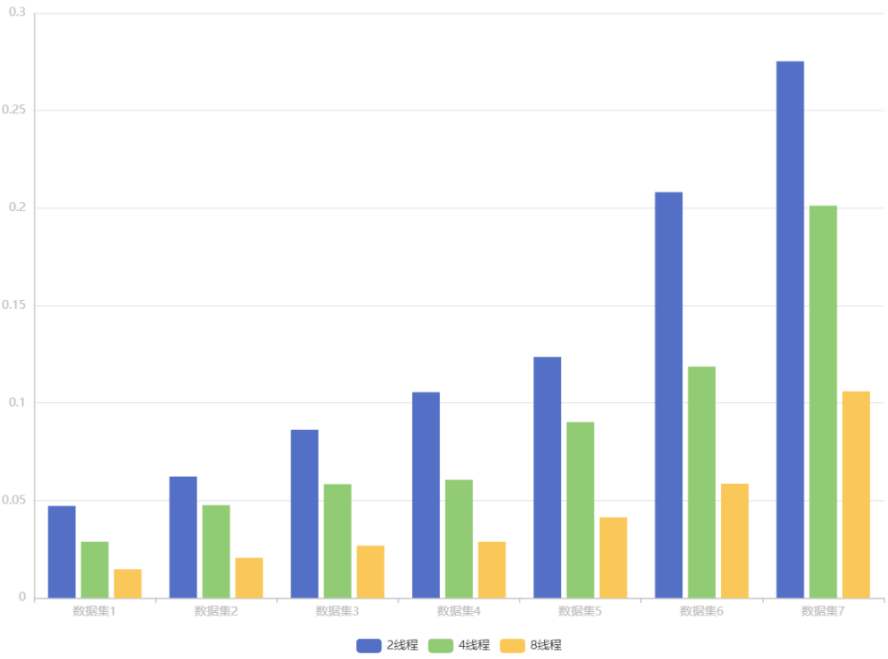

图 6.2: 特殊高斯消去 Pthread 优化算法不同线程数下程序加速比

对实验结果进行分析，可以看到，在各个数据集下，Pthread 多线程算法的运行时间均远大于平凡算法的运行时间，在某些数据集下运行时间甚至为平凡算法的 10 倍左右，这应该是由于线程间同步开销过大造成的。此外，观察多线程算法不同线程数目下程序运行时间，我们发现，随着线程数目的增多，程序运行时间并没有像我们期望的那样减少，而是随着线程数目的增加而增长，原因可能为线程数目越多同步开销越大而带来的增益却并不显著。我们可以很直观的从图6.2中看到，随着同一数据集下，线程数越多，加速比越小；在各个线程数下，算法的加速比随着数据集的增大而增大，但都远小于 1。可见，对于特殊高斯消去这样一个控制流较为复杂的算法，Pthread 多线程优化并没有取得较好的加速性能。

6.3 OpenMp 并行化结果

实验结果数据如表6和表7所示。

算法\数据集编号		1	2	3	4	5	6	7
oridinary		0.00168126	0.0855458	0.11732	4.04767	24.465	521.816	6272.9
thread=2	pthread	0.357605	9.72246	8.21281	222.604	815.224	13045.1	82296.5
	OpenMp	0.0366877	1.38973	1.38923	38.5723	154.745	2085.42	14685.2
	OpenMp+simd	0.0365758	1.36024	1.36894	37.4309	150.936	1890.74	12622.1
thread=4	pthread	0.624288	16.0543	14.6927	385.573	1501.82	20018.7	133814
	OpenMp	0.0851278	3.29226	3.20804	86.733	332.274	3795.97	50298.5
	OpenMp+simd	0.0843615	3.30396	3.21848	87.484	334.6	3820.35	52077.2
thread=8	pthread	1.55593	50.111	48.8632	1327.55	5019.44	43941.2	251895
	OpenMp	0.163059	6.73288	6.50736	176.824	686.815	11857	69863.9
	OpenMp+simd	0.165388	6.70789	6.50338	177.092	749.663	11654.1	41401.1

表 6: ARM 平台下特殊高斯消去 OpenMp 优化不同算法运行时间（单位：ms）

算法\数据集编号		1	2	3	4	5	6	7
oridinary		0.0016195	0.063741	0.086081	3.39692	24.019	253.158	2860.83
thread=2	pthread	0.823395	14.0153	16.247	277.182	1067.67	12769.5	80741
	OpenMp	0.0650811	1.13435	2.55243	72.9886	283.828	3436.17	21677.4
	OpenMp+simd	0.0299685	2.17192	2.3207	69.098	276.604	3211.51	21695.3
thread=4	pthread	1.39741	27.2938	32.426	511.07	1873.81	21466.9	134131
	OpenMp	0.0908785	3.07769	4.12242	110.175	425.243	4754.4	28155.7
	OpenMp+simd	0.0477417	4.03641	3.80731	104.446	422.511	4828.54	30641.7
thread=8	pthread	2.80016	51.9665	56.8114	2237.67	8497.78	91478.5	576468
	OpenMp	0.151287	3.47391	5.41607	153.64	505.347	5697.93	38692.6
	OpenMp+simd	0.0788958	3.21907	4.71191	120.32	485.075	5824.51	40395.9

表 7: x86 平台下特殊高斯消去 OpenMp 优化不同算法运行时间（单位：ms）

对结果进行分析，我们看到，两平台中算法所体现出来的性能特征是相似的。与 Pthread 多线程优化算法类似的，OpenMp 多线程优化算法也没有取得好的性能提升，但是 OpenMp 算法相比于 Pthread 算法明显有着更短的运行时间，可见 OpenMp 算法线程间的同步开销要更小一些。此外，相同测试数据集下，随着线程数的增加，OpenMp 算法运行时间也在增加，这应该也是因为线程数增加带来的同步开销过大导致的。由此可见，特殊高斯消去算法可能不适合使用多线程进行优化。此外，我发现，OpenMp 结合 SIMD 优化算法的性能相比于只使用 OpenMp 优化算法的性能并没有较为明显的性能提升，在某些数据集及线程数目下结合 SIMD 的 OpenMp 优化算法的运行时间甚至长于只使用 OpenMp 优化的算法，这可能是因为由于线程间同步开销过大，SIMD 优化带来的性能提升相对于程序总体运行时间占比较小造成的。

6.4 CUDA 并行化结果

实验结果如表8和表9所示，可以看到，当数据规模较小时，程序没有取得好的性能提升，显然这是由于 Host 端和 Device 端之间数据移动开销过大造成的。随着数据规模地增大，我们可以看到，程序的加速比也在增大，在第七个数据集下，加速比超过了 4，可见 CUDA 并行化具有较好的性能提升。

算法\数据集编号	1	2	3	4	5	6	7
ordinary	0.0076	0.2602	0.3054	10.648	63.443	1084.855	12825.062
CUDA	0.9959	0.4103	0.3867	11.729	54.715	281.124	2943.554

表 8: 特殊高斯消去 CUDA 优化运行时间与平凡算法运行时间（单位：ms）

算法\数据集编号	1	2	3	4	5	6	7
CUDA	0.007631	0.63417	0.78976	0.907835	1.159517	3.858991	4.356999

表 9: 特殊高斯消去 CUDA 优化算法加速比

6.5 MPI 并行化结果

按照实验方案进行实验，得到实验结果如表10和表11所示。表10为不同进程数下特殊高斯消去 MPI 优化算法与平凡算法运行时间对比。表11为特殊高斯消去 MPI 优化算法结合 OpenMp 优化、SSE 优化后的程序运行时间表，其中 MPI 进程数设置为 8、OpenMp 线程数设置为 4。对数据进行处理，得到相应的加速比柱状图图6.3和图6.4。

算法\数据集编号	1	2	3	4	5	6	7
ordinary	4.90E-06	0.000411	0.000479	0.032821	0.352464	6.68365	104.437
MPI_2processes	0.0011	0.0014	0.0032	0.0279	0.2565	4.8763	56.8748
MPI_4processes	0.004	0.0046	0.0091	0.0409	0.2973	4.6983	40.6397
MPI_8processes	0.1447	0.137	0.3469	0.5276	1.4227	4.4584	36.5529

表 10: 不同进程数下特殊高斯消去 MPI 优化算法与平凡算法运行时间（单位：s）

算法\数据集编号	1	2	3	4	5	6	7
MPI	0.1447	0.137	0.3469	0.5276	1.4227	4.4584	36.5529
MPI+omp	0.2408	0.217	0.3236	0.458	1.1209	3.6368	31.8118
MPI+omp+sse	0.079	0.0118	0.0574	0.2009	0.3413	2.1507	24.9662

表 11: 特殊高斯消去 MPI 优化算法结合 OpenMp 优化、SSE 优化后的程序运行时间（单位：s）

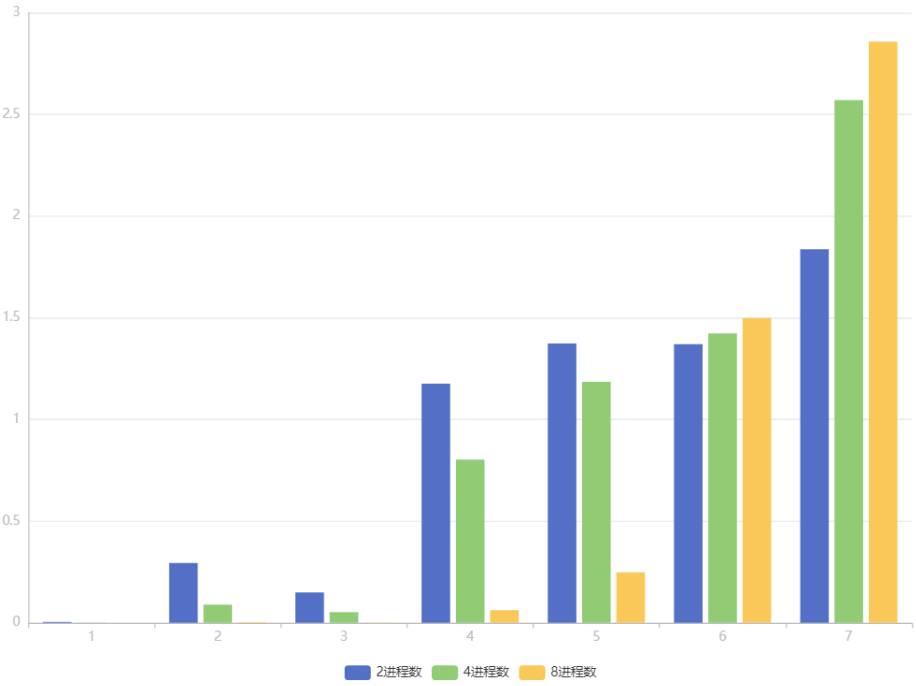

图 6.3: 特殊高斯消去 MPI 优化算法不同进程数下程序加速比

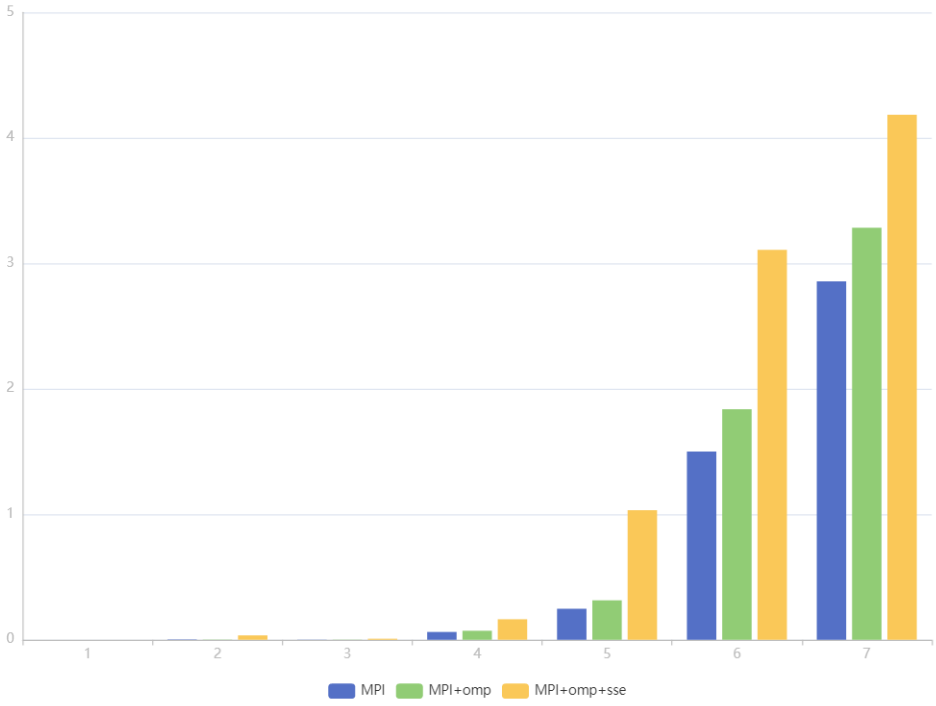

图 6.4: 特殊高斯消去 MPI 优化算法结合 OpenMp 优化、SSE 优化后的程序加速比

首先，分析进程数对 MPI 程序性能的影响，观察图6.3我们可以看到，随着数据规模的增加，各个进程数下程序的加速比均逐渐增大。规模较小时，程序加速比远小于 1，这应该是由于进程间通信开销等占比过大造成的。此外，我们发现，当数据规模较小时，2 进程数程序加速比大于 4 进程数程序加速比，4 进程数程序加速比大于 8 进程数程序加速比，而当数据规模达到一定程度后，这三者的顺序发生了反转。我认为这是由于数据规模较小时，进程数越多，进程间的额外开销越大造成的；而当数据规模较大时，进程数增加所减少的程序运行时间要大于进程数增多带来的更多的额外开销，因此此时进程数越多程序加速比越大。可以看到，数据规模较大时，各进程数下程序均得到了不错的加速比。

接下来，我们探究结合 OpenMp 优化和 SSE 优化对 MPI 算法性能的影响。观察图6.4，类似的由于并行化算法额外开销较大，当数据规模较小时，各个算法的加速比均远小于 1。随着数据规模的增大，程序的加速比也随之增大。此外，我们不难发现，在 4、5、6、7 数据集下，MPI 结合 omp 优化算法的加速比要大于单纯使用 MPI 优化的算法的加速比，MPI 结合 omp 和 sse 优化算法的加速比要大于 MPI 只结合 omp 优化算法的加速比。这与我们的预期相符，也告诉我们可以通过结合多种技术来加速程序的运行性能。

7 实验源码项目 GitHub 链接

实验源码项目 GitHub 链接

参考文献

[1] 狄鹏.(2008).Gröbner 基生成算法的并行 (硕士学位论文, 西安电子科技大学).https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD2011&filename=2010083495.nh

[2] Lazard, D. (1983). Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations. In: van Hulzen, J.A. (eds) Computer Algebra. EUROCAL 1983. Lecture Notes in Computer Science, vol 162. Springer, Berlin, Heidelberg.https://doi.org/10.1007/3-540-12868-9_99

[3] Jean-Charles Faugère and Sylvain Lachartre. 2010. Parallel Gaussian elimination for Gröbner bases computations in finite fields. In Proceedings of the 4th International Workshop on Parallel and Symbolic Computation (PASCO ' 10). Association for Computing Machinery, New York, NY, USA, 89–97. DOI:https://doi.org/10.1145/1837210.1837225