南開大學
Nankai University

计算机学院

并行程序设计期末报告

并行加速的 Gröbner 基高斯消去算法

2022 年 7 月

目录

1 问题描述 **2**

2 非共享内存模型下 Gröbner 基高斯消去消去算法的优化 **2**
2.1 Xsimd 库介绍 . . . 2
2.2 优化思路 . . . 3
2.2.1 降低通讯开销 . . . 3
2.2.2 手动 SIMD 并行化 . . . 4
2.3 核心代码 . . . 5
2.3.1 主进程 . . . 5
2.3.2 从进程 . . . 7

3 使用稀疏矩阵的 Gröbner 基高斯消去消去算法及其并行化 **7**
3.1 Boost.MPI 介绍 . . . 7
3.2 实现思路 . . . 8
3.2.1 稀疏矩阵 . . . 8
3.2.2 异或算法 . . . 8
3.2.3 MPI 并行化 . . . 10
3.3 核心代码 . . . 10

4 稀疏矩阵和稠密矩阵算法的并行化性能分析 **10**
4.1 实验环境 . . . 10
4.2 性能测试 . . . 10
4.3 结果分析 . . . 10
4.3.1 Xsimd 库与 O2 优化 . . . 10
4.3.2 位图矩阵算法 . . . 12
4.3.3 稀疏矩阵算法 . . . 13

5 总结 **14**

Abstract

高斯消去，是线性代数的一个算法，用于将矩阵转化为行阶梯形矩阵。高斯消去算法可被广泛地应用于多种具体问题的求解。例如，在 Gröbner 基生成算法中，多项式所产生的矩阵多为大型稀疏矩阵，而约化这些大型矩阵的过程，则可以通过 $GF(2)$ 上的特殊高斯消去算法来完成。

在本学期并行程序设计的学习中，笔者通过体系结构优化、SIMD、Pthread、OpenMP 和 MPI 等方法和工具，完成了多种高斯消去算法的并行化加速。但是由于时间有限，这些工作的完成多有遗憾——比如，在 MPI 作业中，笔者设计的 Gröbner 基消去算法性能就比较差，笔者也没能实现磁盘读写异步、稀疏矩阵等算法，且由于学业较重、笔者也没能在服务器关闭前，在真正的集群上运行 MPI 的程序。此外，在课内外学习中，笔者也发现了诸如 Highway、Xsimd、Boost.MPI 等课程中没有深入介绍过的并行化工具和方法等。

基于以上的现状，笔者决定在期末报告中对此前 MPI 作业进行优化，并尝试通过 Xsimd 库编写跨平台的 SIMD 加速代码。笔者还通过 Boost.MPI 以及 C++ 的标准模板库实现一个 C++ 风格的、使用稀疏矩阵的 Gröbner 基消去算法。在本报告中，笔者将会介绍以上算法及其并行化的实现思路，并且对其性能进行比较，尝试找出性能差异的原因。

关键字：高斯消去、Gröbner 基生成算法、Xsimd 库、Boost.MPI 库、稀疏矩阵

1 问题描述

消元子模式的高斯消去消去算法，是一可被应用在 Gröbner 基生成算法中的子问题解法，后者已经被广泛应用于公钥密码、数字签名和零知识证明等领域。

在消元子模式的高斯消去消去算法中，输入的矩阵均为 GF(2) 上的矩阵，这也就是说矩阵中只有 0 和 1 两种元素。此外，输入的矩阵被分为了消元子矩阵和被消元行矩阵，消元子矩阵基本符合上三角矩阵的形态，但是其中包含了一些元素全为 0 的空行。算法的基本步骤如下：

1. 从上而下取出一定数量的消元子和被消元行，尝试使用被消元行减去（异或）消元子，消除被消元行的首非零元。
2. 消去过程中可能遇到被消元行的首非零元对应的消元子为空行的情况，当这种情况发生，将被消元行升格为消元子。被升格后的被消元行不再作为被消元行，而是作为消元子参与到消元过程中。
3. 当所有被消元行要么全部为 0 或者已经被升格时，消元完成。

2 非共享内存模型下 Gröbner 基高斯消去消去算法的优化

2.1 Xsimd 库介绍

在完成 SIMD 编程作业时，笔者编写了多个不同的函数，并运用了条件编译指令，来实现跨平台的、多指令集的 SIMD 并行化性能测量。虽然 ARM 和 X86 平台的 Intrinsic 文档都比较完善，但是为每个指令集编写不同的代码实在有些麻烦。因此，笔者找到了 Intel 的用于将 ARM NEON 的 SIMD 代码移植到 SSE 指令集上运行的工具，但是这一工具也只是实现了同为 128 位宽度的 SIMD 指令集之间的转换，而对于 AVX、AVX512 等寄存器长度不为 128 的指令集，我们依然需要编写不同的代码。

那么，存不存在一种方案，可以实现忽略寄存器长度，进行 SIMD 并行化呢？答案是肯定的。Xsimd 库便是一个对 SIMD 操作进行了封装，并给出模板风格接口的 C++ 库。通过使用模板和自动推导类型等特性，我们可以编写能够在多个平台、多种 SIMD 指令集上通用的代码。[2]

2.2 优化思路

2.2.1 降低通讯开销

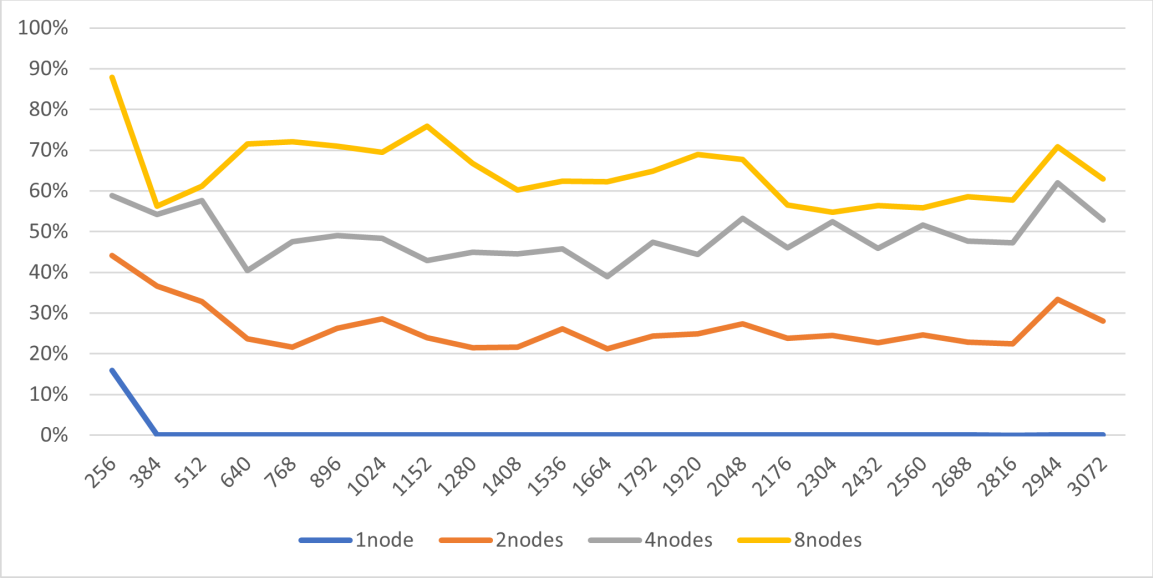

图 2.1: 普通高斯消去算法并行化中通讯时间占比-输入矩阵行数

如图2.1所示，在下 Gröbner 基高斯消去消去算法 MPI 并行化的实验中，我们通过运行计时发现，通讯所消耗的时间关于输入数据成正比，而在通讯时间在算法总体的运行时间中的占比和输入数据规模之间的关系为常数阶。通过对于算法的复杂性分析，我们发现导致这一原因的是通信复杂度和算法的整体复杂度同为 $O(n^3)$，而通信复杂度还会随着线程数的增加而线性增长，因此该并行实现的可拓展性是极差的。

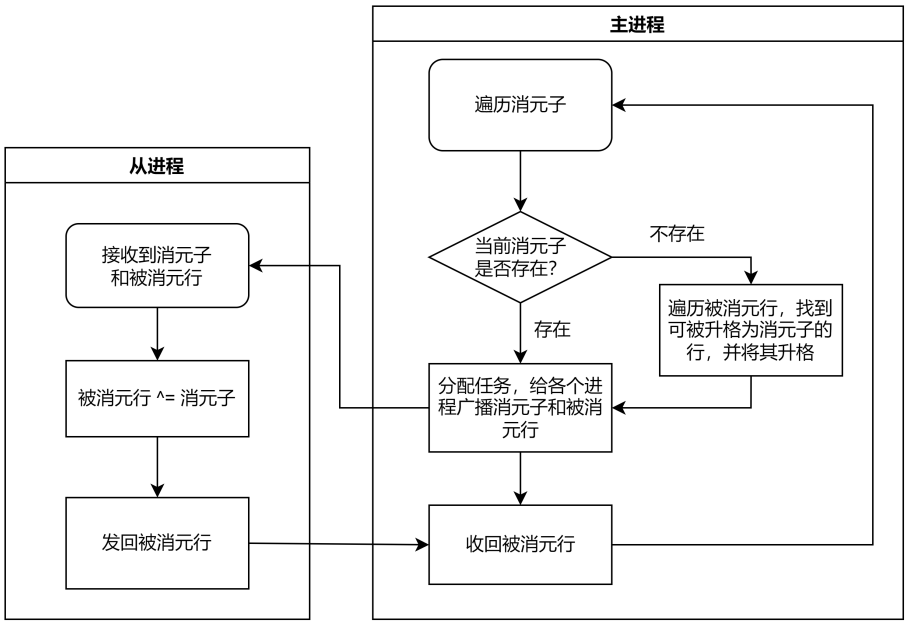

图 2.2: 通信复杂度为 $O(n^3)$ 的 Gröbner 基高斯消去算法框图

要改善这一问题，我们需要重新审视算法的工作流程。如图2.2所示，在原本的并行实现中，我们使用了主从的通讯策略：当主进程遍历到一个存在的消元子后，便将被消元行矩阵进行划分和下发，并对当前消元子进行广播。而从进程完成主进程发来的任务后，则进行计算，并将结果发回主进程，主进程再将结果恢复到原被消元行矩阵中。

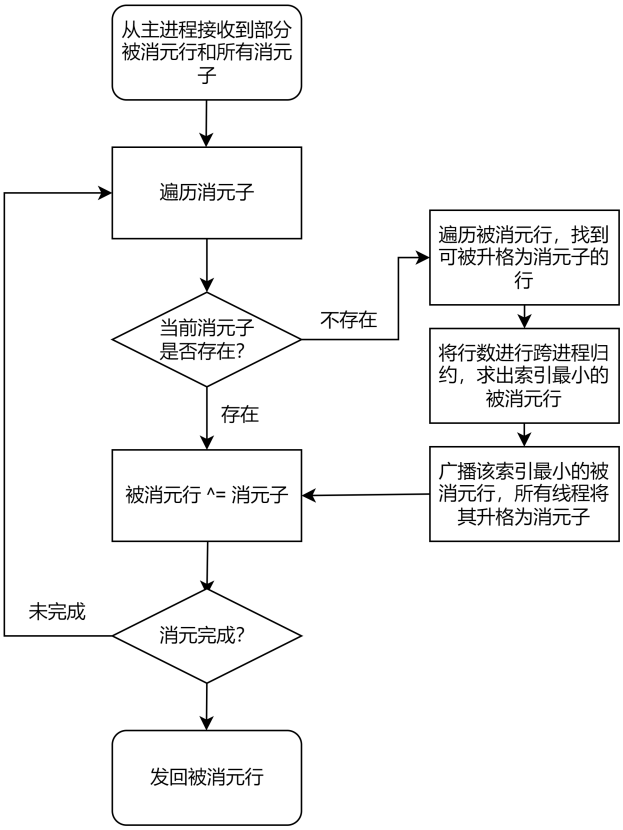

图 2.3: 最佳通信复杂度为 $O(n)$ 的 Gröbner 基高斯消去算法框图

然而，从主进程收到从进程运算结果，到下一次主进程发出任务和数据之间，被消元行矩阵并不会有任何变化，各个线程其实可以自主的对消元子进行遍历，除非遇到需要升格被消元行的情况，线程之间其实完全不需要沟通。根据这一情况，笔者设计了如图2.3所示的新并行实现。在这一实现中，输入的矩阵数据只要在进程开始和结束时分别收发一次即可，在不遇到升格的最好情况下复杂度为 $O(n)$。此外，这一新并行实现也能实现对升格被消元行中搜索的并行化，而这一点是之前作业所没有实现的。

2.2.2 手动 SIMD 并行化

此前，笔者在 MPI 作业中，使用 OpenMP 进行了被消元行和消元子间异或操作的 SIMD 并行化，然而在调试中，笔者发现通过 OpenMP 进行 SIMD 并行化，无法确保使用笔者所希望使用的、效率较高的 AVX 指令集，也无法保证 Load、Store 等操作的对齐。因此，笔者才找到了 Xsimd 这个 C++ 库，并通过模板实现了不定寄存器长度的 SIMD 并行化。

2.3 核心代码

2.3.1 主进程

```
#define mat_t unsigned int
#define mat_L 32
#define LEN_LINE ((COL / mat_L + 1) / 16 * 16 + 16)
#define SIMD_INST_SET xsimd::avx2
#define SIMD_ALIGN xsimd::aligned_mode()

void run_master(mat_t (*ele)[LEN_LINE], mat_t (*row)[LEN_LINE])
{
    bool upgraded[ROW] = {0};

    using mat_simd_t = xsimd::batch<unsigned int, SIMD_INST_SET>;
    std::size_t simd_inc = mat_simd_t::size;
    mat_simd_t ele_vec, row_vec;

    int n_workload = ROW / world_size + 1; // 单个进程分配到的被消元行行数
    for (int th = 1; th < world_size; th++)
    { // 发送被消元行
        int i_offset = n_workload * (th - 1);
        MPI_Send(row + i_offset, n_workload * (LEN_LINE) * sizeof(mat_t),
                 MPI_BYTE, th, 0, MPI_COMM_WORLD);
    }
    MPI_Bcast(ele, COL * (LEN_LINE) * sizeof(mat_t),
              MPI_BYTE, 0, MPI_COMM_WORLD); // 发送消元子

    for (int j = COL - 1; j >= 0; j--)
    { // 遍历消元子
        if (!(ele[j][j / mat_L] & ((mat_t)1 << (j % mat_L))))
        { // 如果不存在对应消元子则进行升格
            MPI_Barrier(MPI_COMM_WORLD);
            int tobe_upgraded = (1 << 31) - 1, real_upgraded = 0;
            for (int i = n_workload * (world_size - 1); i < ROW; i++)
            { // 遍历被消元行
                if (upgraded[i])
                    continue;
                if (row[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
                {
                    tobe_upgraded = i;
                    break;
```

```
39                  }
40              }
41              MPI_Reduce(&tobe_upgraded, &real_upgraded, 1, MPI_INT,
42                         MPI_MIN, 0, MPI_COMM_WORLD);
43              MPI_Bcast(&real_upgraded, 1, MPI_INT, 0, MPI_COMM_WORLD);
44              if (real_upgraded == (1 << 31) - 1)
45                  continue;
46              int th_with_min = (real_upgraded / n_workload + 1) % world_size;
47              if (tobe_upgraded != (1 << 31) - 1)
48                  memcpy(ele[j], row[tobe_upgraded], (LEN_LINE) * sizeof(mat_t));
49              MPI_Bcast(ele[j], (LEN_LINE) * sizeof(mat_t),
50                        MPI_BYTE, th_with_min, MPI_COMM_WORLD);
51              upgraded[real_upgraded] = true;
52          }
53  #pragma omp parallel for num_threads(4)
54          for (int i = n_workload * (world_size - 1); i < ROW; i++)
55          { // 遍历被消元行
56              if (upgraded[i])
57                  continue;
58              if (row[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
59              { // 如果当前行需要消元
60                  int p = 0;
61                  for (; p <= COL / mat_L; p += simd_inc)
62                  {
63                      ele_vec = mat_simd_t::load(&ele[j][p], SIMD_ALIGN);
64                      row_vec = mat_simd_t::load(&row[i][p], SIMD_ALIGN);
65                      row_vec ^= ele_vec;
66                      xsimd::store(&row[i][p], row_vec, SIMD_ALIGN);
67                  }
68                  for (; p <= COL / mat_L; p++)
69                      row[i][p] ^= ele[j][p];
70              }
71          }
72      }
73
74      for (int th = 1; th < world_size; th++)
75      { // 回收被消元行
76          int i_offset = n_workload * (th - 1);
77          MPI_Recv(row + i_offset, n_workload * (LEN_LINE) * sizeof(mat_t),
78                   MPI_BYTE, th, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
79      }
80  }
```

2.3.2 从进程

从进程与主进程代码高度相似，具体逻辑在图2.3中也有给出，因此这里的具体代码不再在报告中给出，但是可以在GitHub 仓库中查看。

3 使用稀疏矩阵的 Gröbner 基高斯消去消去算法及其并行化

3.1 Boost.MPI 介绍

Boost 库是一个开放源代码的、具有同行评审机制的、功能强大的跨平台 C++ 库。Boost 库中包含了许多实用的组件，其中就包括了 Boost.SIMD、Boost::thread 等可用于项链和线程并行化的工具。Boost 还包含了 Boost.MPI。和 MPICH、OpenMPI 不同，Boost.MPI 并非一个 MPI 实现，而是对 MPI 的 C++ 封装，旨在提供友好易用、支持现代 C++ 开发风格的 MPI 接口。

在本报告的稀疏矩阵 Gröbner 基高斯消去消去算法的实现中，笔者使用到了大量的 C++ 标准模板库容器，而如果要使用 MPI 的 C 标准接口，来发送这些容器中的数据，无疑是非常麻烦的。而 Boost.MPI 作为 C++ 风格的 MPI 接口，可以通过隐式调用 Boost.Serialization 这一序列化库，来轻松地发送 STL 容器。

```
1  //Boost.MPI 的三个 send 函数
2  template<typename T>
3  void communicator::send(int dest, int tag, const T& value);
4
5  template<typename T, typename A>
6  void communicator::send(int dest, int tag, const std::vector<T,A>& value);
7
8  template<typename T>
9  void communicator::send(int dest, int tag, const T* values, int n);
10
11 //MPI 底层 C 接口的 Send 函数
12 int MPI_Send(const void *buf,
13              int count,
14              MPI_Datatype datatype,
15              int dest, int tag,
16              MPI_Comm comm);
```

笔者在此给出 MPI 编程中最为常用的 send 函数在 MPI 底层 C 接口定义，和 Boost.MPI 的 C++ 接口定义。不难看出，Boost.MPI 使用了模板、函数重载、面向对象等 C++ 风格的编程方法，相应的，函数中的参数数量也得到了大幅的降低。这样定义的接口可以有效降低编程的复杂性，并提升程序的可读性，帮助我们更轻松高效地实现程序的并行化。[1]

3.2 实现思路

3.2.1 稀疏矩阵

Gröbner 基高斯消去消去算法在实际运用中，常常需要处理稀疏矩阵，如果对含 0 元素比例很高的矩阵采用稠密矩阵的存储方式，会对内存和性能造成很大的浪费。因此，笔者实现了一个使用稀疏矩阵的 Gröbner 基高斯消去消去算法。

在这一算法的串行实现中，消元子和被消元行矩阵中非零元素的数量是不确定的，因此笔者将按行以稀疏向量格式存储到 STL 容器 vector 中，具体定义消元子和被消元行矩阵和读入数据的相关代码如下：

```
ifstream data_ele((string)DATA + (string) "1.txt", ios::in);
mat_t temp, header;
string line;
array<vector<mat_t>, COL> ele;
array<vector<mat_t>, ROW> row;

for (mat_t i = 0; i < ELE; i++)
{ // 遍历消元子文件
    getline(data_ele, line);
    istringstream line_iss(line);
    line_iss >> header; // 消元子位置
    ele[header].push_back(header);
    while (line_iss >> temp)
        ele[header].push_back(temp);
}
data_ele.close();
// 省略：读入被消元行
```

3.2.2 异或算法

高斯消去消去算法的关键在于行变换操作的实现，在 $GF(2)$ 上，两行相减的行变换操作表现为按位异或操作。如果采用“稠密矩阵 + 位图”方式对矩阵进行存储，那么异或的操作可以直接被实现，而对于稀疏矩阵，按位异或的实现就要复杂一些了：我们借用合并有序链表的算法，设置两个指针指向待异或的数组的大端，并对两者指向的数据进行比较，如果相等，则两指针同进一步，否则将数据大者放入结果向量中，并将其指针指向下一个元素。当两个指针中的一个已经指向末尾，则遍历剩下的一个数组，将剩余的元素全部推入结果向量中。

在 C++ 下该算法的实现如下：

```
mat_t pRow = 0, pEle = 0;
vector<mat_t> result;
while (pRow < row[i].size() && pEle < ele[j].size())
{
```

```
        if (row[i][pRow] == ele[j][pEle]) {
            pRow++;
            pEle++;
        }
        else if (row[i][pRow] > ele[j][pEle]) {
            result.push_back(row[i][pRow]);
            pRow++;
        }
        else {
            result.push_back(ele[j][pEle]);
            pEle++;
        }
    }
    for (; pRow < row[i].size(); pRow++)
        result.push_back(row[i][pRow]);
    for (; pEle < ele[j].size(); pEle++)
        result.push_back(ele[j][pEle]);
```

初步进行运行计时后发现该异或算法性能较差，因此从体系结构角度出发，对该算法进行了一些优化。比如，将 while 循环中的 if 语句改写为了?: 表达式的形式，来更好地利用分支优化的特性、将一些常用的矩阵值预存到变量中，防止多次读取、使用固定长度的 array 来存储运算结果等，修改后的代码如下：

```
mat_t pRow = 0, pEle = 0;
int rowMax = row[i].size();
int eleMax = ele[j].size();
mat_t row_i_p, ele_j_p, last = 0;
array<mat_t, COL> buffer;

while (pRow < rowMax && pEle < eleMax)
{
    row_i_p = row[i][pRow];
    ele_j_p = ele[j][pEle];
    pRow += row_i_p >= ele_j_p ? 1 : 0;
    pEle += ele_j_p >= row_i_p ? 1 : 0;
    buffer[last] = max(row_i_p, ele_j_p);
    last += row_i_p == ele_j_p ? 0 : 1;
}
// 省略：处理矩阵结尾的代码，与上文几乎一致
```

3.2.3 MPI 并行化

使用稀疏矩阵的 Gröbner 基高斯消去消去算法工作流程几乎和位图矩阵的算法一致，都是按照图2.3中的流程进行升格、消元等操作，除了稀疏向量异或的算法几乎很难被 SIMD 并行化之外，使用稀疏矩阵的 Gröbner 基高斯消去消去算法的并行化思路基本和位图矩阵算法的并行化思路一致。

3.3 核心代码

由于稀疏矩阵算法的流程与图2.3中的位图矩阵算法的工作流程几乎完全相同，并且在上文中，稀疏矩阵数据结构相关代码、稀疏向量异或算法和 Boost.MPI 的接口实例都已经给出，因此这里不再在论文中给出完整代码。完整代码可以在Github 仓库查看。

4 稀疏矩阵和稠密矩阵算法的并行化性能分析

4.1 实验环境

本次报告中的性能测量在笔记本电脑上通过 WSL 进行，WSL 系统为 Debian11，内核版本 5.10.16.3-microsoft-standard-WSL2，GCC 编译器版本 10.2.1 20210110，MPI 实现为 MPICH，版本 3.4.1。

硬件方面，电脑的 CPU 为英特尔 12 代 i7-12700H，为 6 大核 +8 小核的混合架构，L1d cache 480 KiB，L1i cache 320 KiB，L2 cache 12.5 MiB，L3 cache 24 MiB。

4.2 性能测试

本次实验的性能测试通过编译时宏定义改变数据规模，并通过 mpirun 命令的-n 参数指定不同的线程数，将不同情况的接轨导出为 csv 格式的数据，绘制折线图，本次的编译均使用了-O2 优化选项。测试 Shell 脚本中特殊高斯消去的编译与运行命令如下：

```
mpic++ -march=native -w -pthread -DDATA=\"${data_path}${file}/\" \
    -DCOL=${attr[1]} -DELE=${attr[2]} -DROW=${attr[3]} \
    -O2 -fopenmp -DSEPR=\",\" -DNUM_THREADS=1 \
    ./groebner.cpp -o ./groebner
mpirun -n 1 ./groebner >>groebner_$timestr.csv
mpirun -n 2 ./groebner >>groebner_$timestr.csv
mpirun -n 4 ./groebner >>groebner_$timestr.csv
mpirun -n 8 ./groebner >>groebner_$timestr.csv
```

4.3 结果分析

4.3.1 Xsimd 库与 O2 优化

使用 Xsimd 库编写向量化的矩阵行间异或算法时，笔者发现 SIMD 优化后的程序运行时间比未优化的程序运行时间还长，为此，笔者使用 perf 对程序进行了性能热点分析：

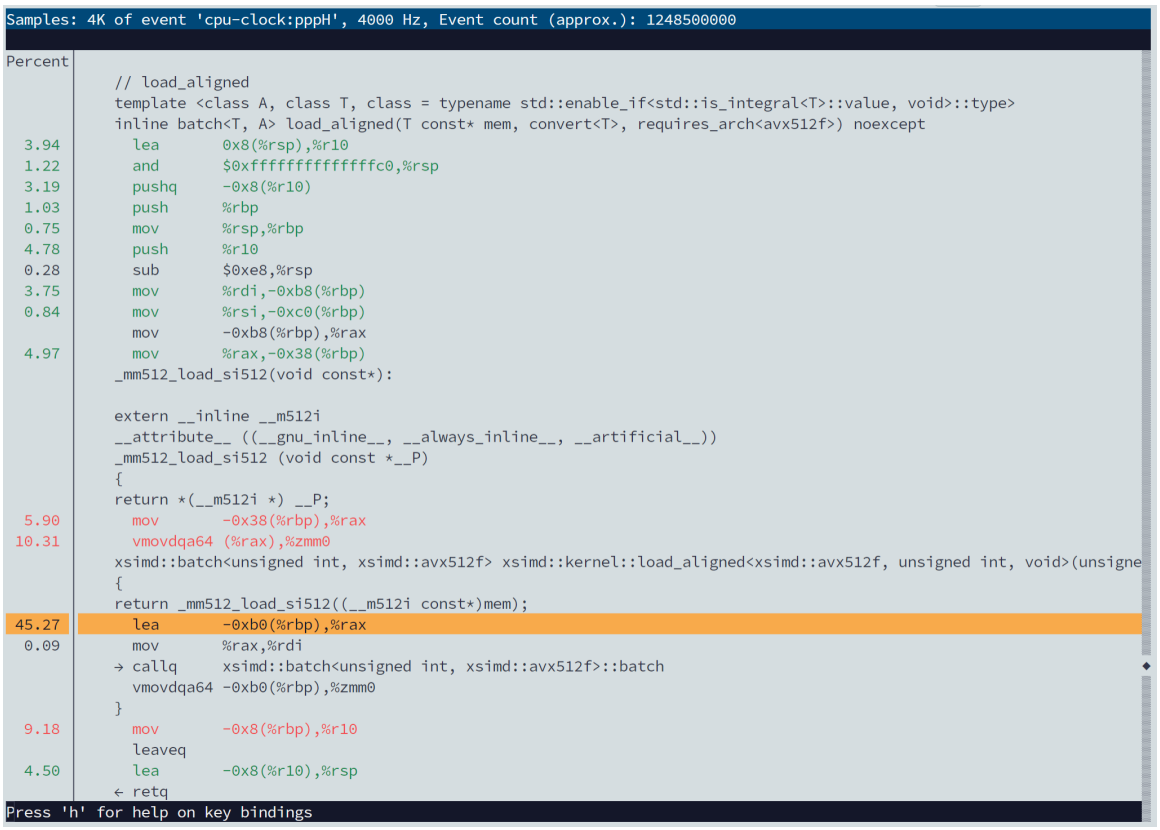

图 4.4: perf 找到的程序性能热点 load_aligned 函数的分析视图

perf 的分析结果显示，程序的性能热点在 xsimd 的 load_aligned 函数上，打开具体的汇编代码热点视图，我们便会发现 xsimd 的 load_aligned 函数并不像原生的 Intrinsic 函数 _mm512_load_si512 一样，是一个内联函数，其调用过程占用了大量的资源，导致了程序性能不佳。

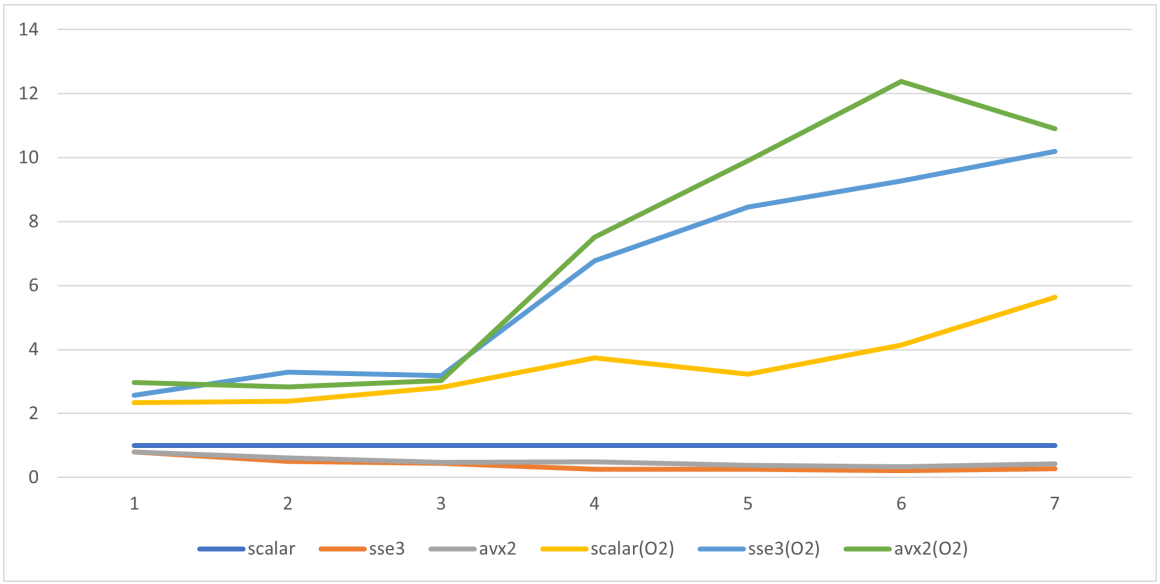

图 4.5: SIMD 加速的 Gröbner 基高斯消去消去算法加速比（以-O0 编译的串行算法为标准）

为了解决这一问题，笔者搜寻了 Xsimd 的文档和社区文档，但是并未找到相似的问题描述或解决方案。之后笔者又尝试开启了 O2 优化，如图4.5所示，加速比得到了大幅提升，再次 perf 分析，发现

函数的调用过程已经被优化掉了。

开启 O2 优化选项编译出的程序的反汇编结果显示，编译器帮助我们调用了 SSE 指令集来对程序进行加速，然而从图4.5中可以发现，即便编译器已经自动进行了向量化，手动进行的 SIMD 优化依然有不小的性能收益。

4.3.2 位图矩阵算法

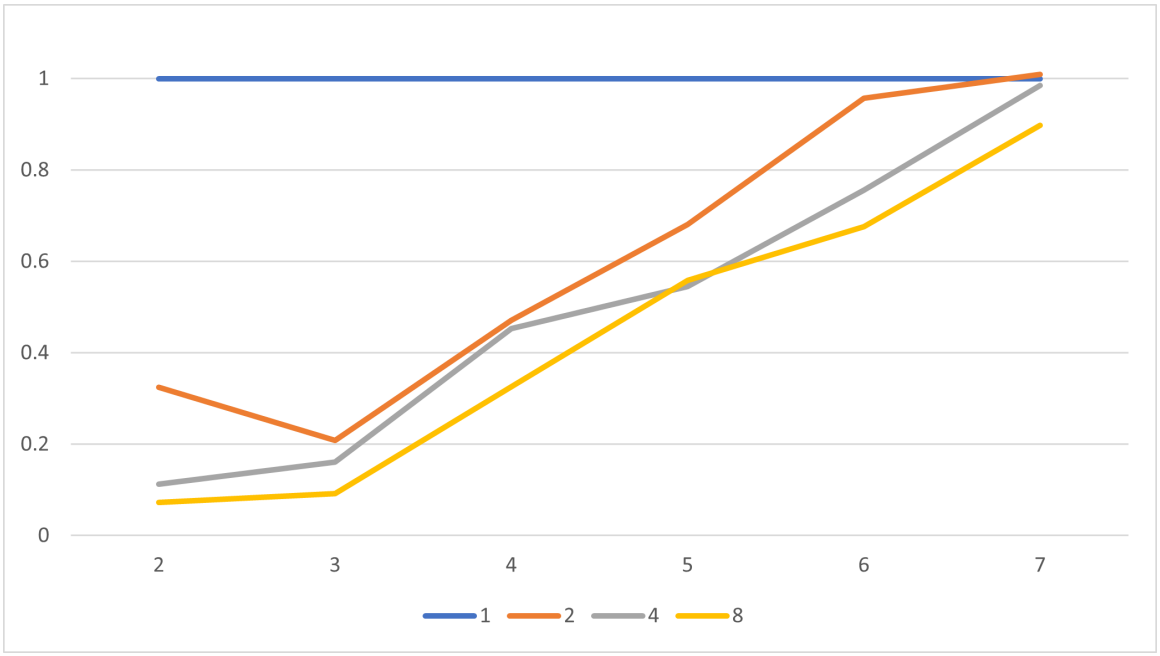

图 4.6: 旧特殊高斯消去算法加速比-输入矩阵行数-节点数

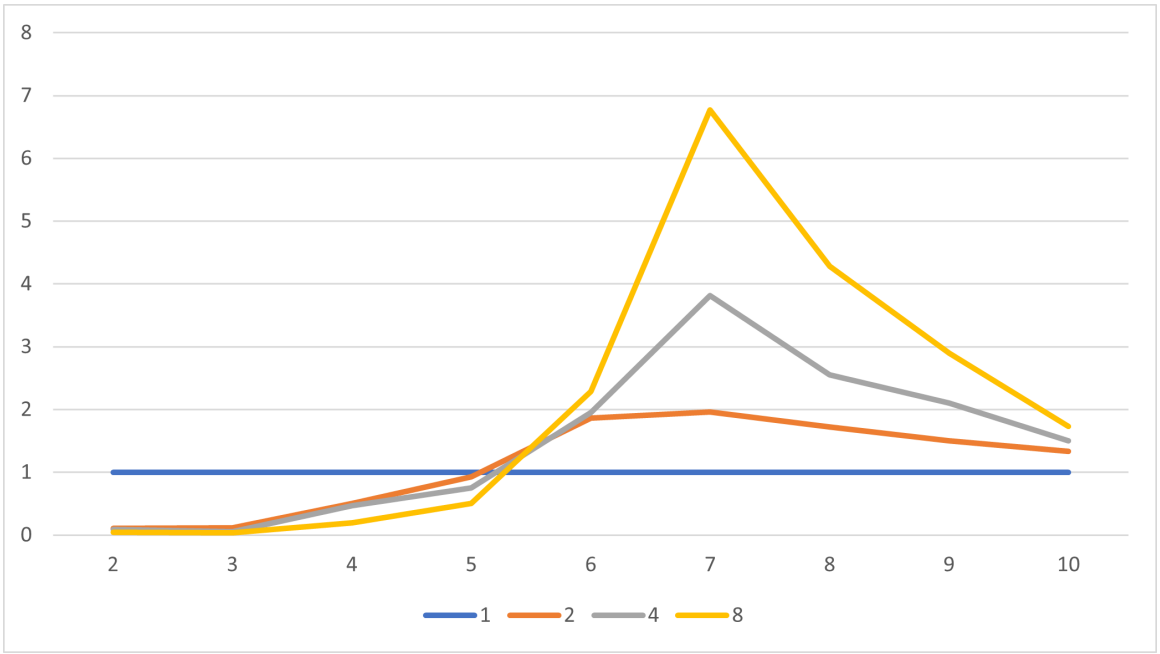

图 4.7: 新特殊高斯消去算法加速比-输入矩阵行数-节点数

从图4.6和图4.7中我们可以看出，经过改进降低了通讯的时间复杂度的新并行化实现，相较之前的实现有更好的性能和可扩展性：数据达到一定规模后，随着线程的增加，加速比呈现上升趋势。

不过我们也观察到，在 8、9 号数据集上，并行实现的加速比有一定的下降，这是不和常理的。之所以会出现这样的情况，是因为虽然优化后的算法最好情况下时间复杂度为 $O(n)$，但是当最差情况，也就是消元子矩阵为空、遍历消元子的每一步都需要进行升格的情况下，通讯的整体复杂度依然为 $O(n^3)$。

此外，笔者因此通过 top 对程序进行了性能分析，发现 8、9 号数据集在被处理时会导致程序占用大量内存，甚至导致 swap 区的虚拟内存开始被大量使用，造成访存代价的极大提高。之所以会出现这样的情况，是因为主节点在一开始就向每个节点发送了完整的消元子数据，造成了不必要的内存浪费。

4.3.3 稀疏矩阵算法

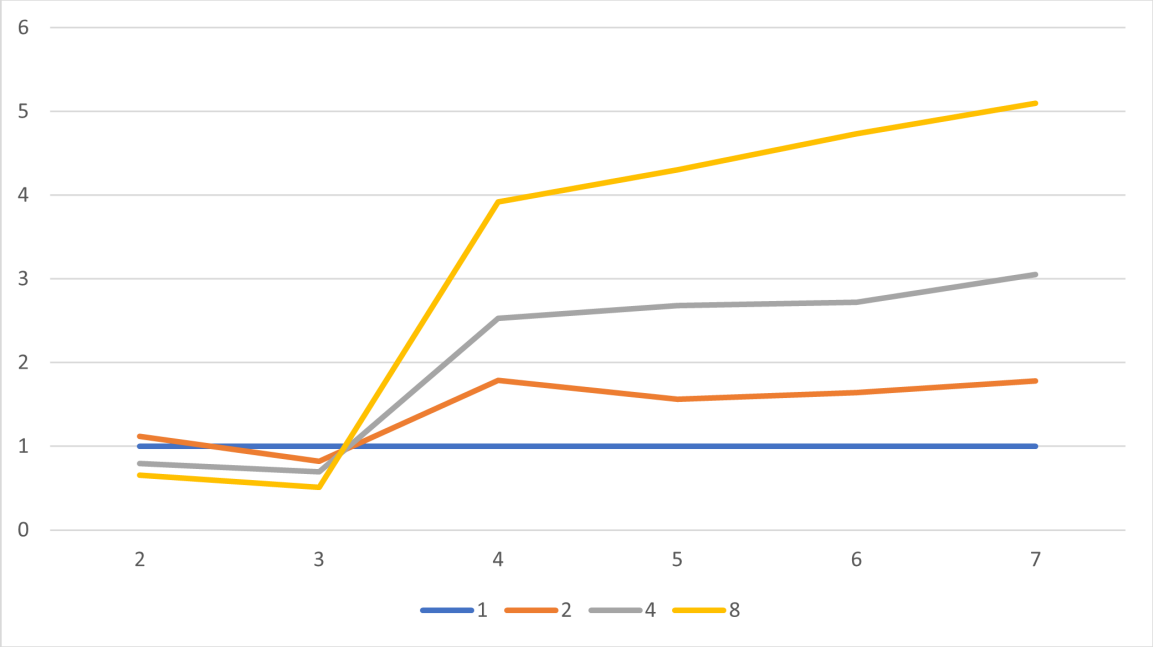

图 4.8: 稀疏矩阵特殊高斯消去算法加速比-输入矩阵行数-节点数

如图4.8所示，稀疏矩阵的特殊高斯消去算法在并行化上有着较好的表现，这其实主要是因为稀疏矩阵的特殊高斯消去算法在串行实现时效率就要低一些，以 7 号数据集为例，串行实现下的位图矩阵消去算法运行耗时为 725 毫秒，而稀疏矩阵算法则需要 88 秒，性能差距达到了上百倍。

造成这样性能差异的最关键原因在于输入矩阵的非 0 元素占比：在测试使用的数据集中，大部分输入数据的非 0 元素占比在 6% 到 12% 左右，而如果采用位图方式存储在 uint32_t 型的变量中，一次异或操作就平均可以处理 1.92 到 3.84 个非 0 元素，而在系数矩阵中处理 1 个非 0 元素，就需要进行一个循环步。

数据结构 \ 线程数	1	2	4	8
sparse	31890.6	23836.1	15558.4	10313.6
bitmap	5063.31	4019.63	3467.37	8841.61

表 1: 稀疏矩阵和位图矩阵的算法在 HFE80 数据集上的运行时间对比（单位：ms）

如表1所示，在非 0 元素占比约为 1.8% 左右的 HFE80 数据集上，稀疏矩阵算法的性能有了较大的改观，性能差距从超过百倍降低到了六倍左右，如果考虑到位图算法 SIMD 向量化的加成，两者的

性能几乎没有差异了。此外，稀疏矩阵算法的并行化可扩展性也要优秀一些。这些现象证明了我们关于非 0 元素占比对稀疏矩阵算法性能影响的猜测。

除了非 0 元素占比的影响外，稀疏矩阵算法性能较差的另一原因也包括了其无法充分利用处理器流水线、分支预测技术的特性。在稀疏矩阵算法的核心——异或这一过程中，我们采用了类似双指针合并有序链表的算法，这导致每一个循环步中，都需要依据从内存中数组取出的数据来判断对哪个指针进行自增操作，此外对于结果向量的写入也需要进行判断因而不能连续的进行写入。除此之外，稀疏矩阵算法的实现中大量使用了 vector 这一 STL 容器，然而在进行异或操作时可能需要 vector 进行动态扩容，这需要转移整块的内存。以上这些因素都导致了稀疏矩阵算法性能的低下。

5 总结

在本篇报告中，我们使用了 Xsimd、Boost.MPI 等工具，对 Gröbner 基高斯消去消去算法的稀疏矩阵实现和位图矩阵实现进行了并行化，并对这些并行化实现进行了性能测试和分析。我们通过对通讯的优化，成功大幅提升了 Gröbner 基高斯消去消去算法并行化实现的加速比和可拓展性。在本报告中所完成的工作使我们对算法、并行程序设计，以及在这一学期中学到的核心内、核心间、节点间的并行计算模型有了更深入的了解。

这一学期中，教学工作都受到了疫情极大的影响。十分感谢本门课程的老师和助教，能在这种情况下排除万难，为我们带来内容丰富的课程。笔者作为一名辅修的学生，在本门课程中收获良多。

参考文献

[1] Douglas Gregor and Matthias Troyer. Chapter 24. boost.mpi - 1.79.0, 2007. https://www.boost.org/doc/libs/1_79_0/doc/html/mpi.html.

[2] Johan Mabille and Sylvain Corlay. Introduction —xsimd documentation, 2016. https://xsimd.readthedocs.io/en/stable/.