
南開大學
Nankai University

计算机学院

并行程序设计期末研究报告

# 特殊高斯消去并行优化

姓名： 高祎珂

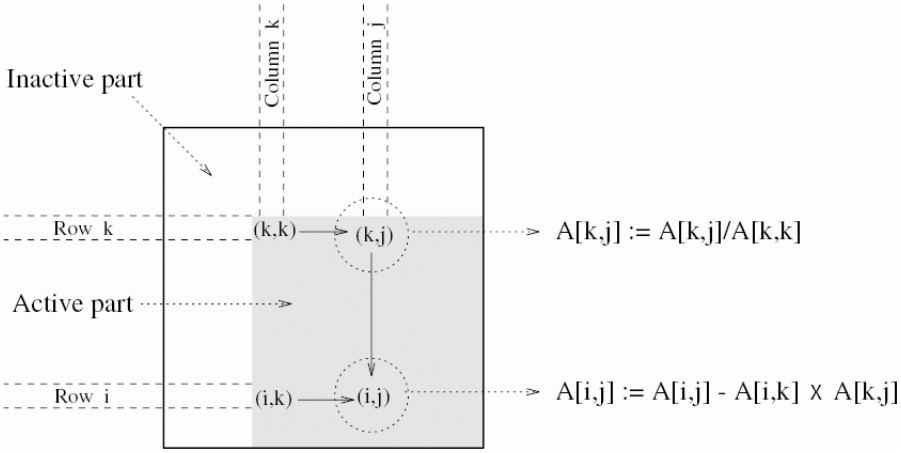
学号：2011743

专业：计算机科学与技术

2022 年 7 月 10 日

# 目录

# 1 前期工作总结




这学期前期一直进行的是普通高斯消去，进行了普通高斯消去的 SIMD, 多线程,MPI 优化，对这几个并行化方法进行了学习，主要并行优化针对上图的除法归一和消去操作。每次进行操作的对象都是二维 float 型矩阵，SIMD 部分一次 load4 个 float 型进行计算优化，当时得出的优化效果大致为 2 倍多。多线程分为自己编写 pthread 和利用 OpenMP，进行了 MPI 多进程优化，多进程优化过程中有采用了流水线方式任务划分，通过不同数据规模，不同线程数的横向纵向对比，了解了什么时候应该开辟线程，应该开辟多少线程，这不仅与机器本身性能有关，也与数据量大小有关。在前期工作中也进行了各个并行化方法的合并优化操作，体验了各种并行方法的优劣，有的编程方便，有的可移植性更好，其中 MPI 的实验的进行难度较大，当时由于消息传递机制并不是很熟悉，所以一直得不到正确的结果，最终同查阅资料得到解决。总体而言，前期的工作主要目的是对学习到的并行化方法进行实践，现在来看基本达到了这个目的。

# 2 消元子模式高斯消去实验介绍

## 2.1 实验简介

特殊高斯消去源于布尔 Gröbner 基，整个过程是在 GE(2) 域上进行，也就是参与运算的数据只有 0/1, 这样对于此高斯消去的运算就跟之前的那个有很大不同了，因为只有 0/1, 所以不用归一了，也就是相较于普通高斯消去，消元子模式的高斯消去少了一个二重循环，同时这里的消去操作，对于以位图模式存储的数据来说，也变成了异或运算，异或运算对于后续的 SIMD 和 omp 都十分友好。这里的数据整个分成了两大阵营消元子和被消元行，对于每个被消元行，都要经历以下步骤：

(1) 对每个被消元行，检查其首项，如有对应消元子，则将其按位异或对应消元子，重复此过程直至其变为空行（全 0）或无对应消元子

(2) 若为情况一，则该被消元行计算结束，则将其丢弃，不再参与后续消去计算；若无对应消元子，则将该行加入消元子参与后续运算。

但是当数据规模较大时，一次 I/O 可能无法将数据全部读进内存，例如所给样例 8 等，这个时候我们就需要分批次读入，这样我们就需要分批次执行上述任务，这里需要补充的是，如果第一步操作得到的结果为首项不在当前消元子批次覆盖范围内，那么此行在此批消元子中的运算也结束了。分批读入就要不断的 I/O，I/O 开销比较大，这个时候我们就需要加入一些外存算法以实现不断地 I/O。在

分批次读入过程中，需要保证被消元行和消元子两两之间（可以进行运算的）都进行过异或运算，这里我采用的是二重循环，每次读入被消元行的一部分，然后将该批次被消元行与分批次读入的消元子进行运算，并将新生成的消元子写入文件进行下一步运算，一次遍历整个消元子，接着进行下一批次的被消元行的运算。

### 2.2 实验设计

**Algorithm 1** 从文件中读入数据，分批次进行运算，串行伪代码

**Input:** 消元子、被消元行 txt 文件
**Output:** 消元结果

```
while 被消元行有可读数据 do
    读入 m 行数数据，并且以位图模式存入 E 数组
    while 消元子有可读数据 do
        读入固定行数数据，并且以位图模式存入 R 数组
        for i=0 to m-1 do
            while E[i]！=0 do
                if 存在首项位 E[i] 的首项的 R[j] then
                    E[i]=E[i] 与 R[j] 按位异或
                else
                    R[j]=E[i]
                    break
    return 消元结果 =0
```

在此过程中需要注意的点有

1. 在数据量较大时对于批次的大小划分要有不同，因为消元子是 I/O 操作较多的文件，对齐批次划分以及访问方式的设计尤为重要
2. 在运算过程中，R 数组是在不断更新的，即消元子是不断加入新数据的，在此批次被消元行执行完之后，要更新消元子，及此消元子要重新写入文件，这又涉及一次 I/O 操作。
3. 再分批次读入数据时，因为消元子数据在不断更新，所以初始 R 数组的大小不应该恰好开辟读入数据大小，为安全起见，应该开辟读入消元子和被消元行之和的大小。
4. 异或过程中是按位异或，因此要把二维数组中的每行每个数据拆开来对应各自进行异或

在读入数据过程中，涉及到较大的数据，因此我们需要进行外存算法的设计，来提高效率，是数据能够读入，假设被消元行数据大小为 N，一次读入块大小为 block1，消元子数据大小为 M，一次读入块大小为 block2，如果按照设计的二重循环进行 I/O，则该过程中的 I/O 代价为 O（N/block1*(M/block2+1)）

## 3 X86 平台实验

### 3.1 实验平台介绍

本次实验在本地跑的，进行了环境配置，本机的配置为

| CPU 型号 | Intel Core i7-1065G7 CPU |
|---|---|
| 内核数 | 4 |
| 线程数 | 8 |
| L1 cache | 320KB |
| L2 cache | 2.0MB |
| L3 cache | 8.0MB |
| 最大内存 | 64GB |

可以看出部分测试样例是放不进去缓存的，设计外存算法是有必要的。完整代码上传在github，下面只展示一些核心代码。

本次实验进行了特殊高斯的串行编写，并行优化，进行了 SIMD 和多线程，多线程使用了 OpenMP，对于数据量较大的样例，只是进行了异或部分的并行优化，将向量分解为不同子向量进行运算。

## 3.2 代码展示

数据读入，以被消元行为例

```
    //x是要插进来的值
void bitmap(unsigned int* a,int x)
{
  size_t index = x >> 5;
  size_t num = x % 32;
  a[index] |= (1 << num);
}
unsigned int** E=new unsigned int* [Esize];
    for (int i = 0; i < Esize; i++)
    {
      E[i] = new unsigned[size];
      for (int j = 0; j < size; j++)
      {
        E[i][j] = 0;
      }
    }

    int row = 0;
    bitmap(E[row], last);
    while (row < block && !beixiaoyuanhang.eof())
    {
      unsigned int now;
      beixiaoyuanhang >> now;
      if (now > last)
      {
        if (row == block-1)
        {
          last = now;
          break;
        }
        row++;
      }
```

```
      last = now;
      bitmap(E[row], last);
    }
    row += 1;
```

数据异或操作，核心运算部分

```
    for (int i = 0; i < row; i++)
      {
        if (isallzero(E[i],size))
        {
          int flag = 0;
          //row1行消元子，之后记得加
          for (int j = 0; j < row1; j++)
          {
            int first = firstnum(E[i],size);
            //写入
            if (firstnum(R[j], size) < first)
            {
              for (int a = row1; a > j; a--)
              {
                for (int b = 0; b <size; b++)
                  R[a][b] = R[a - 1][b];
              }
              for (int b = 0; b < size; b++)
                R[j][b] = E[i][b];
              row1++;

              break;
            }
            if (firstnum(R[j],size) == first)
            {
              int minsize = size;
              for (int a = 0; a < minsize; a++)
              {
                int num = E[i][a] ^ R[j][a];
                E[i][a] = num;
              }

              flag = 1;
              //break;
            }
          }

        }
      }
```

数据异或操作，SIMD 加入运算修改

```
if (firstnum(R[j], size) == first)
        {
          int minsize = size;
          int a;
          for (a = 0; a + 4 <= minsize; a += 4)
          {
            __m128i Ri = _mm_loadu_si128((__m128i*)(&R[j][a]));
            __m128i Ei = _mm_loadu_si128((__m128i*)(&E[i][a]));
            __m128i Result = _mm_xor_si128(Ri, Ei);
            /*int num = bxyh[i].indexnum(a) ^ xyz[j].indexnum(a);
            bxyh[i].setindex(a, num);*/
            _mm_storeu_si128((__m128i*)(&E[i][a]), Result);
          }
          while (a < minsize)
          {
            int num = E[i][a] ^ R[j][a];
            E[i][a] = num;
            a++;
          }
        }
```

数据异或操作，OMP 加入修改，线程数设置为 4

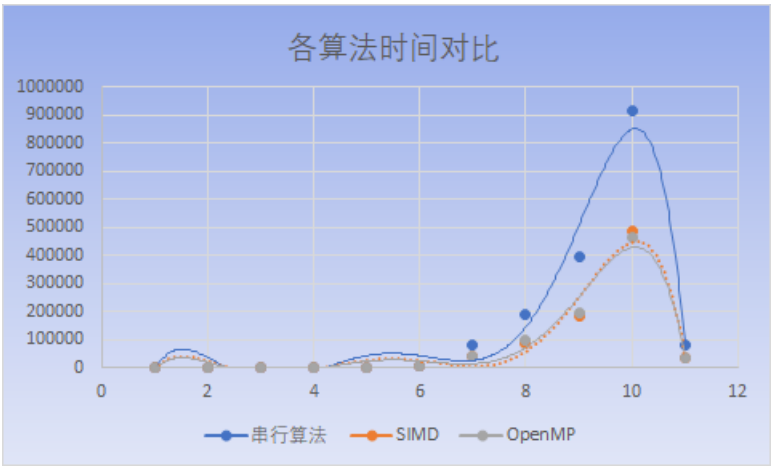
```
if (firstnum(R[j], size) == first)
        {
          int minsize = size;
          #pragma omp parallel for schedule(static,NUM_THREADS)
          for (int a = 0; a < minsize; a++)
          {
            int num = E[i][a] ^ R[j][a];
            E[i][a] = num;
          }
          flag = 1;
          //break;
        }
```

### 3.3 实验结果

文件大小统计如下

| 测试样例 | 文件大小 |
|---|---|
| 1 | 2.97KB |
| 2 | 21.3KB |
| 3 | 49.6KB |
| 4 | 321KB |
| 5 | 1.48MB |
| 6 | 7.44MB |
| 7 | 47.3MB |
| 8 | 295MB |
| 9 | 600MB |
| 10 | 1.31G |
| 11 | 260MB |

| 测试样例 | 串行/ms | SIMD 优化/ms | OpenMP 优化/ms |
|---|---|---|---|
| 1 | 5.8281 | 3.838 | 6.7277 |
| 2 | 15.4087 | 10.9479 | 15.753 |
| 3 | 32.1526 | 22.1637 | 26.2926 |
| 4 | 224.113 | 155.757 | 188.937 |
| 5 | 801.856 | 537.786 | 571.464 |
| 6 | 8581.8 | 4087.85 | 4870.2 |
| 7 | 81176.7 | 39657.2 | 43236.3 |
| 8 | 191285 | 84537 | 96542.7 |
| 9 | 396875 | 185637 | 195235 |
| 10 | 915320 | 485763 | 462189 |
| 11 | 78057.9 | 36569.1 | 37529.4 |




该数据测量并没有减去 I/O 时间，从结果可以看出，I/O 操作对结果的影响很大，在数据集大小变大时，运行时间有很大的飞跃，这还是进行了多次 I/O 之后，外存算法或许可以分割文件，构建索引，这样可以减少 I/O 开销，是时间得到进一步提升，但是整个时间消耗是不可避免的，对于样例 11，虽然矩阵列数变大了，但是数据集大小变小了，这样时间就有恢复了较低的状态，在此过程中，随着样例的矩阵列数，被消元行、消元子行数的改变，数据块的划分也需要发生变化，因完成 I/O，且以较好的效率。

SIMD 使用的是 SSE2 语言，一次 load4 个 int 型数据进行计算，理想加速比应该在 4 左右，但是从数据可以看出，接近于 2，这比普通高斯消去的 SIMD 优化效果要差一点，可能与数据格式与数

据运算有关，对于多线程，此处设置线程数为 4，理论上优化效果应该也是 4 倍左右，但是实际上并没有，大概有 2 倍左右，和 SIMD 优化效果差不多，可能因为中间消耗较大导致效果不理想。

# 4 VTune 分析

以样例 3 为例，进行 VTune 分析

| Function | 串行 | SIMD 优化 | OpenMP 优化 |
|---|---|---|---|
| L1cache MISS | 164，238 | 246，362 | 73，490 |
| L2cache MISS | 82，788 | 113，850 | 30，516 |
| L3cache MISS | 32，248 | 12，352 | 36，508 |
| Clockticks | 120，900，000 | 110，500，000 | 111，800，000 |
| Instructions Retired | 131，300，000 | 131，300，000 | 132，600，000 |
| CPI Rate | 0.921 | 0.842 | 0.843 |

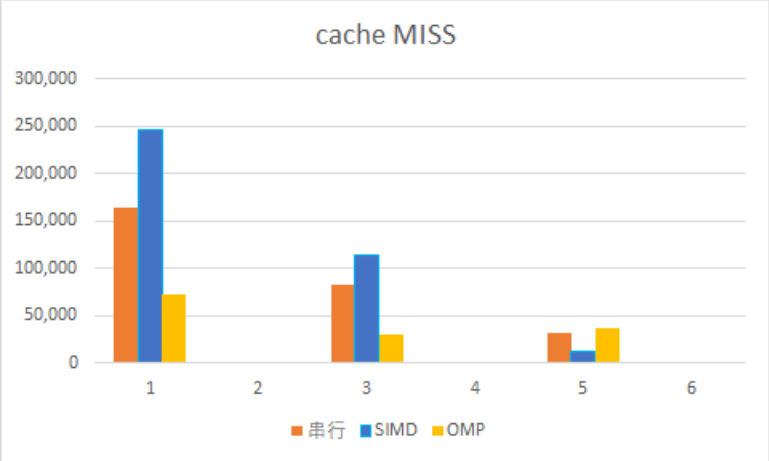
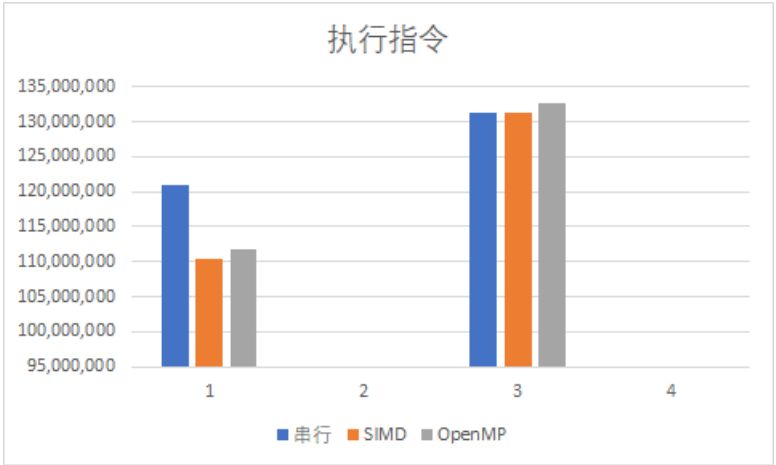
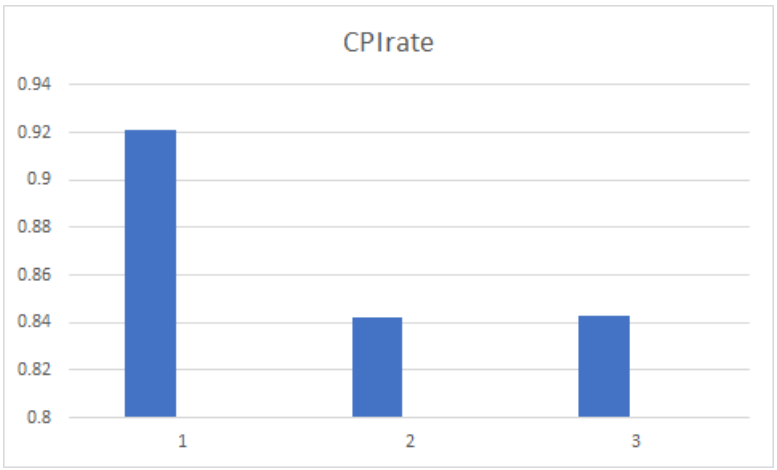



图 4.1: 三个函数 CacheMiss 对比

从上述数据可以看出 OpenMP 的 CacheMiss 最小，相较于串行，得到了很好的优化，而 SIMD 的 CacheMiss 却上升了，这也应该是优化效果没有达到预期的原因之一。






IPC（Instruction Per Clock），即每个时钟周期执行的指令数，是流水线算法的评价指标，我们测到的 CPI 是 IPC 的倒数，可以看到 SIMD 和多线程的 CPI 确实是比串行的小的，但是优化效果并不大，这也间接解释了优化效果不理想。

对于 OpenMP 进行 VTune 可视化分析，我们可以看到开辟出的线程的具体运行，可以看到开辟出了我们要并行优化的线程。

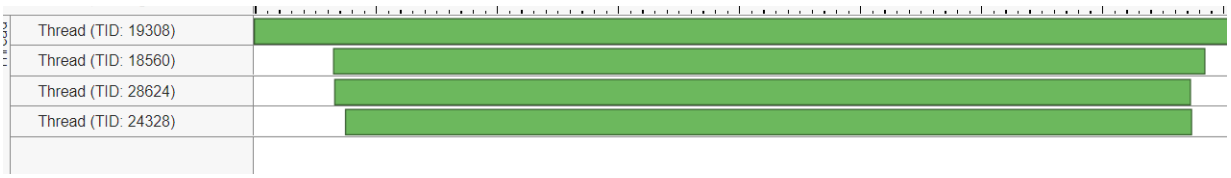



图 4.2: OMP 线程

# 5 总结

这整个学期的并行都围绕着高斯消去展开，前期工作主要围绕着普通高斯消去进行，根据实验指导书来做，难度也降低了不少，在此过程中，熟悉了一些工具的应用，了解了一些并行化方法，这里面感觉比较困难的的是自己设计并行执行函数以及消息传递机制，这里面稍不小心，就会有很多的 bug，得到正确的结果就比较难。

大作业进行的是特殊高斯消去，消元子模式跟之前不同的是数据类型，这里需要以位图模式存储，虽然在读入数据的处理操作增加了，但是后续处理变得更简单了，它简化了后续的消去操作，所以进行 SIMD 和多线程是就很顺利，拆分成子向量就行。这个过程中比较麻烦的是外存算法，数据量太大的时候数据读不进来，这时候就要分块去读，需要设计合适的算法，这块也花费了很长时间来学习，但是后续还要改进，现在效果不是特别理想，需要对计算机的外存算法有更深的理解。