南開大學
Nankai University

计算机学院

并行程序设计实验报告

Pthreads&OpenMP 编程实验

姓名：陆遥

学号：2211843

专业：计算机科学与技术

2024 年 5 月 24 日

目录

1 问题描述

数学上，高斯消元法（或译：高斯消去法），是线性代数规划中的一个算法，可用来为线性方程组求解。但其算法十分复杂，不常用于加减消元法，求出矩阵的秩，以及求出可逆方阵的逆矩阵。不过，如果有过百万条等式时，这个算法会十分省时。一些极大的方程组通常会用迭代法以及花式消元来解决。当用于一个矩阵时，高斯消元法会产生出一个“行梯阵式”。高斯消元法可以用在电脑中来解决数千条等式及未知数。亦有一些方法特地用来解决一些有特别排列的系数的方程组。

2 普通高斯消去算法

2.1 串行算法设计

高斯消去的计算模式如图 2.1 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，全部结束后，得到如图 2.2 所示的结果。而回代过程从矩阵的最后一行开始向上回代，对于第 i 行，利用已知的 xi+1, xi+2, ..., xn 计算出 xi。串行算法如下面伪代码所示。

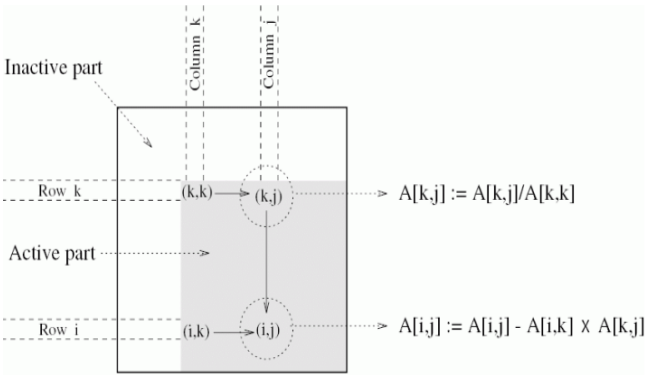

图 2.1: 高斯消去法示意图

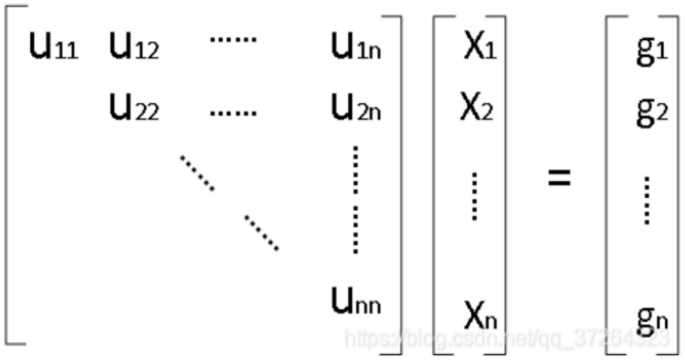

图 2.2: 高斯消去法消去过程结束后，结果的示意图

2.2 Pthreads 共享内存对比实验设计

2.2.1 问题规模大小

问题规模的大小主要体现在矩阵规模上，对于不同的并行方案，不同的问题规模会体现不同的并行优化效果，有时甚至会出现负优化的现象，这都是非常正常的。在实验中设置不同的问题规模，有助于在各规模范围内找到最优的并行优化方案。

2.2.2 动态线程还是静态线程？

Pthread 编程有两种范式：

- 静态线程：程序初始化时创建好线程（池）。对于需要并行计算的部分，将任务分配给线程执行。但执行完毕后并不结束线程，等待下一个并行部分继续为线程分配任务。直至整个程序结束，才结束线程。优点是没有频繁的线程创建、销毁开销，性能更优；缺点是线程一直保持，占用系统资源，可能造成资源浪费。
- 动态线程：在到达并行部分时，主线程才创建线程来进行并行计算，在这部分完成后，即销毁线程。在到达下一个并行部分时，再次重复创建线程——并行计算——销毁线程的步骤。优点是再没有并行计算需求时不会占用系统资源；缺点是有较大的线程创建和销毁开销。

对动态线程和静态线程进行分别的实验，分析在不同条件下应该如何选择，可以达到更佳的优化性能。

2.2.3 临界区访问同步：忙等待还是信号量还是 barrier？

在并行算法中，经常会遇到需要多个线程修改全局变量，而这些线程可能需要修改同一位置的情况，这时就需要考虑使用临界区：在多线程编程中，临界区是指一段代码，在这段代码中共享资源的访问受到限制，只有一个线程可以进入临界区执行代码，而其他线程必须等待。这是为了避免多个线程同时对共享资源进行写操作而导致的竞争问题。

在本实验中，有三种比较合适的方法能够有效解决临界区访问问题：忙等待、信号量和 barrier。这里将探究三者哪个能够达到更优的性能。

2.2.4 任务划分策略：按行还是按列？

可以看到，高斯消去过程中的计算集中在 5-8 的第一个内层循环（除法）和 11-17 行的第二个内层循环（双重循环，消去），对应矩阵右下角 (n−k + 1)×(n−k) 的子矩阵。因此，任务划分可以看作对此子矩阵的划分。对于除法部分，因为只涉及一行，只可能采用垂直划分（列划分）。而对于消去部分，即可采用水平划分（将其外层循环拆分，即每个线程分配若干行），也可采用垂直划分（将其内层循环拆分，即每个线程分配若干列）。两种划分策略在负载均衡上可能会有细微差异，而在同步方面会有差异，cache 利用方面也会有不同。

2.2.5 并行区域：仅内层还是总外层？

程序主体是一个双重循环，外层循环内、内层循环之外基本没有实质性计算，主要计算操作都在内层循环内，我们选择将内层循环拆分分配给工作线程。此时如果机械地从程序结构角度出发，只将内层循环放入线程函数中，外层循环还保留主函数中，则有两种情况：要么采用动态线程范式，外层循环的每步执行都创建、销毁所有工作线程，带来严重的额外开销；采用静态线程范式的话，主线程和工作线程之间的通信（同步）就会很复杂，程序易读性、可维护性大大下降。而好的方式是，从程序

动态执行的角度看，其实外层循环执行的全过程都是处于并行部分中（因为内外层循环之间并无计算，外层循环的执行实际上只是在持续执行内层循环而已），因此将双重循环都置于线程函数中，对外层循环保持不动，将内层循环拆分分配工作线程即可。

2.2.6 寻求更快性能：与 SIMD 结合

上一次实验进行了 SIMD 并行化的相关编程。SIMD 将行内连续元素的运算打包进行向量化，即只能对最内层循环进行展开、向量化。这里将 Pthreads 共享内存与 SIMD 相结合，以寻求更快的计算性能。

2.3 代码实现

篇幅所限，部分函数仅展示部分代码，省略与其他函数重复部分。

2.3.1 动态线程

```
void *threadFunc1(void *param)
{
    for (int j = k + 1; j < n; ++j){
        m[i][j] = m[i][j] - m[i][k] * m[k][j];
    }
    m[i][k] = 0;
    pthread_exit(NULL);
    return NULL;
}

void pthread_1(int n)//动态线程
{
    for (int k = 0; k < n; ++k)
    {
        for (int j = k + 1; j < n; j++)
            m[k][j] = m[k][j] / m[k][k];
        m[k][k] = 1.0;
        int worker_count = n - 1 - k;
        pthread_t *handles = new pthread_t[worker_count];
        threadParam_t1 *param = new threadParam_t1[worker_count];
        for(int t_id = 0; t_id < worker_count; t_id++)
            pthread_create(&handles[t_id],NULL,threadFunc1,(void*)(&param[t_id]));
        for(int t_id = 0; t_id < worker_count; t_id++)
            pthread_join(handles[t_id],NULL);
    }
    return;
}
```

2.3.2 静态线程 + 信号量同步 + 二重循环

```
void *threadFunc2(void *param) {
```

```
    for (int k = 0; k < n; ++k)
    {
        sem_wait(&sem_workerstart[t_id]); //
            阻塞，等待主线完成除法操作（操作自己专属的信号量）
        //循环划分任务
        for(int i=k+1+t_id; i < n; i += NUM_THREADS){
            for (int j = k + 1; j < n; ++j)
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            m[i][k]=0.0;
        }
        sem_post(&sem_main); // 唤醒主线程
        sem_wait(&sem_workerend[t_id]); //阻塞，等待主线程唤醒进入下一轮
    }
    pthread_exit(NULL);
    return NULL;
}

void pthread_2(int n)//静态线程 + 信号量同步 + 二重循环
{
    sem_init(&sem_main, 0, 0);
    for (int i = 0; i < NUM_THREADS; ++i){
        sem_init(&sem_workerstart[i], 0, 0);
        sem_init(&sem_workerend[i], 0, 0);
    }
    pthread_t *handles = new pthread_t[NUM_THREADS];
    threadParam_t2 *param = new threadParam_t2[NUM_THREADS];
    for(int k = 0; k < n; ++k){
        //主线程做除法操作
        for (int j = k+1; j < n; j++)
            m[k][j] = m[k][j] / m[k][k];
        m[k][k] = 1.0;
        //开始唤醒工作线程
        for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
            sem_post(&sem_workerstart[t_id]);
        //主线程睡眠（等待所有的工作线程完成此轮消去任务）
        for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
            sem_wait(&sem_main);
        // 主线程再次唤醒工作线程进入下一轮次的消去任务
        for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
            sem_post(&sem_workerend[t_id]);
    }
    for(int t_id = 0; t_id < NUM_THREADS; t_id++)
        pthread_join(handles[t_id],NULL);
    return;
}
```

2.3.3 静态线程 + 忙等待同步 + 二重循环

```
int flag[NUM_THREADS] = {0};//为0时不阻塞，为1时阻塞
void *threadFunc3(void *param) {
    for (int k = 0; k < n; ++k)
    {
        while(flag[t_id] == 0);//用来阻塞线程，等到主线程完成除法后解锁
        //循环划分任务
        for(int i=k+1+t_id; i < n; i += NUM_THREADS)
        {
            for (int j = k + 1; j < n; ++j)
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            m[i][k]=0.0;
        }
        flag[t_id] = 0;
    }
    pthread_exit(NULL);
    return NULL;
}

void pthread_3(int n)//静态线程 + 忙等待同步 +二重循环
{
    pthread_t *handles = new pthread_t[NUM_THREADS];
    threadParam_t2 *param = new threadParam_t2[NUM_THREADS];
    for(int t_id = 0; t_id < NUM_THREADS; t_id++)
    {
        param[t_id].t_id = t_id;
        param[t_id].n = n;
        pthread_create(&handles[t_id],NULL,threadFunc3,(void*)(&param[t_id]));
    }
    for(int k = 0; k < n; ++k)
    {
        //主线程做除法操作
        for (int j = k+1; j < n; j++)
            m[k][j] = m[k][j] / m[k][k];
        m[k][k] = 1.0;
        for (int j = 0; j < NUM_THREADS; j++)
            flag[j] = 1;
        while(1)//阻塞主线程，等到多线程完成消去后解锁
        {
            int temp = 0;
            for (int i = 0; i < NUM_THREADS; ++i)
            {
                if(flag[i] == 1)
                    temp = 1;
            }
            if(temp == 0)
                break;
        }
    }
```

```
    for(int t_id = 0; t_id < NUM_THREADS; t_id++)
        pthread_join(handles[t_id],NULL);
    return;
}
```

2.3.4 静态线程 + 信号量同步 + 三重循环

```
void *threadFunc4(void *param)
{
    for (int k = 0; k < n; ++k){
        if (t_id == 0){
            for (int j = k+1; j < n; j++)
                m[k][j ] = m[k][j] / m[k][k];
            m[k][k] = 1.0;
        }
        else
            sem_wait(&sem_Divsion[t_id-1]); // 阻塞，等待完成除法操作
        if (t_id == 0){
            for (int i = 0; i < NUM_THREADS-1; ++i)
                sem_post(&sem_Divsion[i]);
        }
        for(int i=k+1+t_id; i < n; i += NUM_THREADS){
            for (int j = k + 1; j < n; ++j)
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            m[i][k]=0.0;
        }
        if (t_id == 0){
            for (int i = 0; i < NUM_THREADS-1; ++i)
                sem_wait(&sem_leader); // 等待其它 worker 完成消去

            for (int i = 0; i < NUM_THREADS-1; ++i)
                sem_post(&sem_Elimination[i]); // 通知其它 worker 进入下一轮
        }
        else{
            sem_post(&sem_leader);// 通知 leader，已完成消去任务
            sem_wait(&sem_Elimination[t_id-1]);
        } // 等待通知，进入下一轮

    }
    pthread_exit(NULL);
    return NULL;
}


void pthread_4(int n)//静态线程 + 信号量同步 + 三重循环
{
    for(int t_id = 0; t_id < NUM_THREADS; t_id++){
        param[t_id].t_id = t_id;
```

```
        param[t_id].n = n;
        pthread_create(&handles[t_id],NULL,threadFunc4,(void*)(&param[t_id]));
    }
    for(int t_id = 0; t_id < NUM_THREADS; t_id++)
        pthread_join(handles[t_id],NULL);
     return;
}
```

2.3.5 静态线程 + barrier 同步 + 三重循环

```
void *threadFunc5(void *param)
{
    for (int k = 0; k < n; ++k){
        if (t_id == 0){
            for (int j = k+1; j < n; j++)
                m[k][j] = m[k][j] / m[k][k];
            m[k][k] = 1.0;
        }
        pthread_barrier_wait(&barrier_Divsion);
        for(int i=k+1+t_id; i < n; i += NUM_THREADS){
            for (int j = k + 1; j < n; ++j)
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            m[i][k]=0.0;
        }
        pthread_barrier_wait(&barrier_Elimination);

    }
    pthread_exit(NULL);
    return NULL;
}

void pthread_5(int n)//静态线程 + barrier同步 + 三重循环
{
    pthread_t *handles = new pthread_t[NUM_THREADS];
    threadParam_t2 *param = new threadParam_t2[NUM_THREADS];
    for(int t_id = 0; t_id < NUM_THREADS; t_id++){
        param[t_id].t_id = t_id;
        param[t_id].n = n;
        pthread_create(&handles[t_id],NULL,threadFunc4,(void*)(&param[t_id]));
    }
    for(int t_id = 0; t_id < NUM_THREADS; t_id++)
        pthread_join(handles[t_id],NULL);
    return;
}
```

2.3.6 SIMD 部分示例代码

```
void m_gauss_simd(int n)
{
    __m128 vt, va, vaik, vakj, vaij, vx;
    for(int k = 0; k < n; k++){
        vt = _mm_set_ps1(m[k][k]);
        int j;
        for(j = k+1; j+4 <= n; j+=4){
            va = _mm_loadu_ps(&m[k][j]);
            va = _mm_div_ps(va, vt);
            _mm_storeu_ps(&m[k][j], va);
        }
        m[k][k] = 1.0;
        for(int i = k+1; i < n; i++){
            vaik = _mm_set_ps1(m[i][k]);
            int j;
            for(j = k+1; j+4 <= n; j+=4){
                vakj = _mm_loadu_ps(&m[k][j]);
                vaij = _mm_loadu_ps(&m[i][j]);
                vx = _mm_mul_ps(vakj, vaik);
                vaij = _mm_sub_ps(vaij, vx);
                _mm_storeu_ps(&m[i][j], vaij);
            }
            m[i][k] = 0;
        }
    }
}
```

2.4 实验结果

2.4.1 实验平台运行

在 Code::Blocks 中运行实验程序。

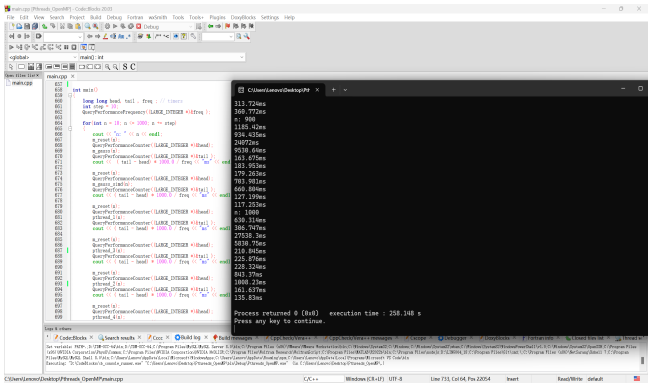

图 2.3: 实验平台运行图

2.4.2 不同并行方案和问题规模下运行用时表

单位：ms

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0013	0.0969	0.3905	0.695	76.2036	323.595	625.226
SIMD	0.0012	0.0484	0.1786	0.3352	33.6881	149.48	304.402
动态线程	3.0834	44.995	145.75	228.31	5323.79	13566.2	21498.6
静态 + 忙等待 + 二重	85.772	319.58	547.453	619.79	3092.09	6435.31	5872.44
静态 + 信号量 + 二重	2.8516	4.265	8.5643	10.657	49.2564	129.543	227.072
静态 + 信号量 + 三重	2.0239	3.833	6.6285	7.3809	53.3872	127.993	221.345
静态 +barrier+ 三重	1.6835	3.0111	4.5567	5.6104	80.6835	145.797	215.301
信号量 + 二重 + 按列	1.7596	3.9506	6.6718	8.7122	245.173	532.88	870.292
barrier+ 三重 + 按列	1.3483	3.2329	5.4542	6.398	215.72	501.441	857.792
信号量 + 二重 +SIMD	1.5683	3.7492	6.19	6.9614	34.1793	94.9785	145.072
barrier+ 三重 +SIMD	1.2705	2.8958	4.9652	5.4084	40.3477	96.6436	140.362

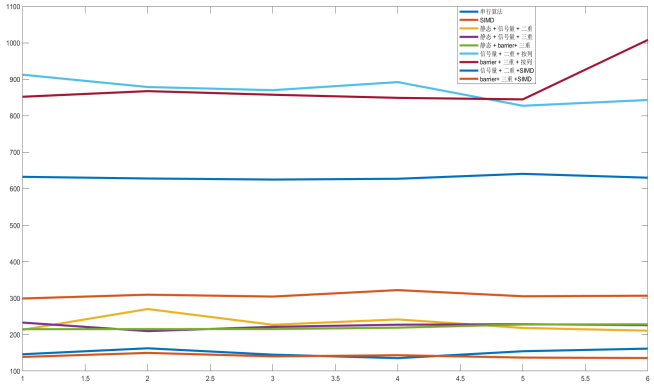

图 2.4: 问题规模为 1000，不同算法下运行用时折线图

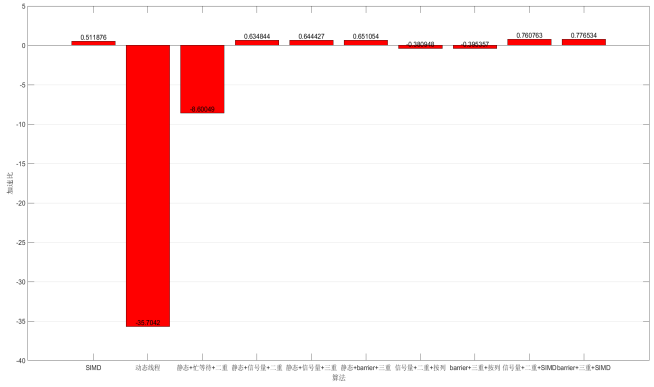

图 2.5: 不同算法平均加速比柱状图

2.4.3 实验结果总结

1、问题规模的影响：问题规模较小时，总体并行优化算法相较于传统串行用时普遍较长。这是因为线程创建和销毁以及静态线程中线程的保持和各种同步量的加入等因素都会延长算法运算的时间，而因为问题规模较小，其优化的效果没有很好地表现出来。随着问题规模的增大，并行算法的优化效果才开始显现。

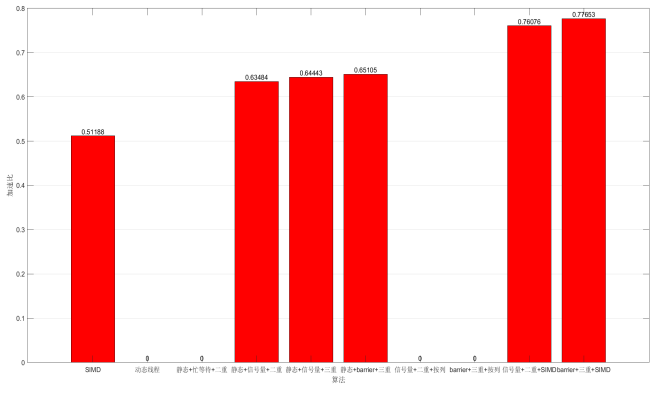

图 2.6: 不同算法平均加速比 (去除负数) 柱状图

2、动态线程与静态线程的影响：动态线程没有并行计算需求时不会占用系统资源；缺点是有较大的线程创建和销毁开销。静态线程优点是没有频繁的线程创建、销毁开销，性能更优；缺点是线程一直保持，占用系统资源，可能造成资源浪费。

从运行时间角度来看，动态线程反复的线程创建、销毁开销很大，使其运算始终慢于静态线程，也始终慢于传统串行算法，因此，如果在运行性能的角度下，静态线程是更优的选择。我们在进行并行优化时，也要尽量量避免频繁的创建和销毁线程。

3、临界区访问同步量的比较：实验结果，通过忙等待来实现同步始终大幅慢于用信号量来实现同步。忙等待的性能劣于信号量的性能是符合已知的理论规律，在后面的性能分析中，我会给出比较详细的比较。

4、任务划分策略的选择：在按行分配与按列分配的问题上，我们一直都认为按行分配是更优的选择，而事实也确实如此，两个按列分配方案的运行时间都要慢于按行分配的时间，这其中主要是 cache 高速缓存行内存存储访问的问题，使得在进行数据查找能更快地获取同一行的数据。这一点在之前的实验中有所涉及。

在后面的性能分析中，我们也可以从 CPI 的角度去发现行分配与按列分配的性能区别。

5、并行区域的划分：高斯消去算法有存在着外层循环与内存循环。从实验结果来看，将三重循环都进行并行处理的方案显然要快于仅将内层循环进行并行处理的方案。同时，在三重循环都进行并行处理的方案中，采用了信号量与 barrier 分别实现了同步，barrier 体现了略优的性能。

6、SIMD 的影响：我们在用 Pthreads 共享内存时，没有改动最内层循环进行具体数据运算部分，在这里加上 SIMD 编程可以进一步优化性能。实验结果也体现了这一点，两个加上 SIMD 编程优化的方案在所有方案中的运行时间最短。

2.5 性能分析

对问题规模 n=1000 时各算法的性能进行具体的分析。

2.5.1 忙等待性能不佳的原因

在图中可以看到，最上面的线程是主线程，而下面线程均为忙等待函数创建的线程，从 CPU 时间角度看，线程实际占据 CPU 运算时间只在后面小半部分，而前面因为要设置了 while 循环，一直在等待主线程的工作。

而在下图 sleep 函数中，我们可以看到这些线程的睡眠时间，可以看出它们在空闲时间进行了等待，由于线程数目为 16，因此这样的等待对于整体的性能影响较大。

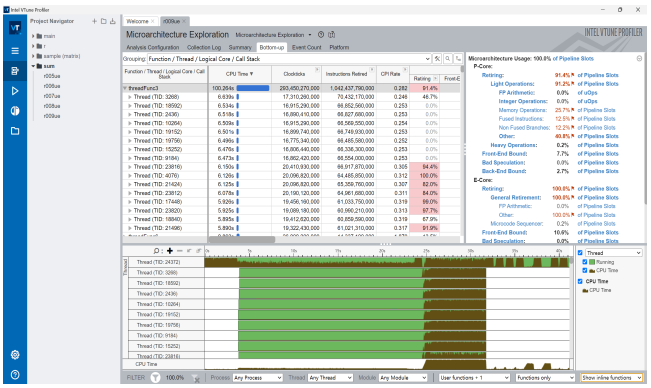

图 2.7: VTune Proflier 分析图 1

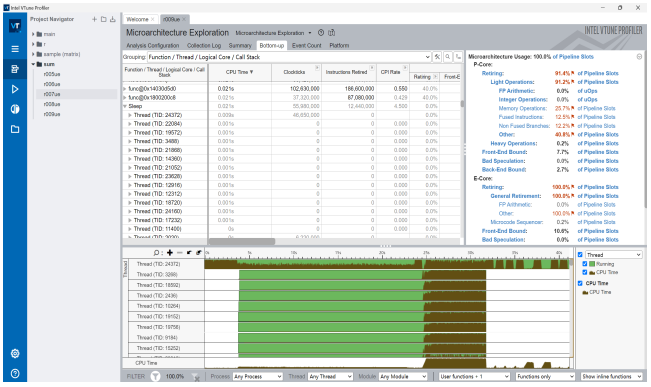

图 2.8: VTune Proflier 分析图 2

2.5.2 按行与按列分配 CPI 的比较

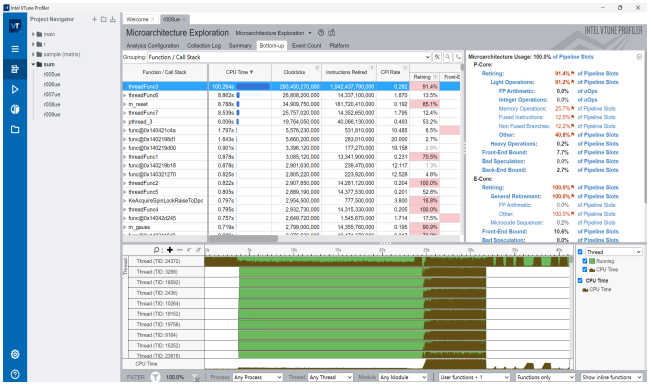

图 2.9: VTune Proflier 分析图 3

上图中，第 6 与第 7 个方案的函数用时较长，我们可以关注它们的 Instructions Retired 和 CPI Rate 这两个指标。它们的指令数相较于其他方案并没有什么差异，问题出在 CPI Rate。它们的 CPI Rate 相当高，说明平均每时钟周期内执行的指令数相对较少，这对应了之前提及的 cache 高速缓存行内存存储访问的问题，按列查找数据比按行查找需要花费更长的时间，从而导致性能的下降。

2.6 OpenMP 共享内存对比实验设计

2.6.1 问题规模大小

问题规模的大小主要体现在矩阵规模上，对于不同的并行方案，不同的问题规模会体现不同的并行优化效果，有时甚至会出现负优化的现象，这都是非常正常的。在实验中设置不同的问题规模，有助于在各规模范围内找到最优的并行优化方案。

2.6.2 任务划分策略：按行还是按列？

可以看到，高斯消去过程中的计算集中在 5-8 的第一个内层循环（除法）和 11-17 行的第二个内层循环（双重循环，消去），对应矩阵右下角 (n−k + 1)×(n−k) 的子矩阵。因此，任务划分可以看作对此子矩阵的划分。对于除法部分，因为只涉及一行，只可能采用垂直划分（列划分）。而对于消去部分，即可采用水平划分（将其外层循环拆分，即每个线程分配若干行），也可采用垂直划分（将其内层循环拆分，即每个线程分配若干列）。两种划分策略在负载均衡上可能会有细微差异，而在同步方面会有差异，cache 利用方面也会有不同。

2.6.3 并行区域划分：除法部分还是消去部分

高斯消去法中有两个部分可以进行并行化，我们可以对比一下这两个部分 (除法部分和消去部分) 进行 OpenMP 共享内存对程序速度的影响。

2.6.4 寻求更快性能：与 SIMD 结合

上一次实验进行了 SIMD 并行化的相关编程。SIMD 将行内连续元素的运算打包进行向量化，即只能对最内层循环进行展开、向量化。这里将 OpenMP 共享内存与 SIMD 相结合，以寻求更快的计算性能。

2.6.5 负载均衡问题：设置 chunk size 大小

在为线程划分任务的时候，需要避免负载不均，以高斯消去为例，对于倒数第 k 行，消去过程的计算量为 O(k2)，使用纯块划分会使得处理头几行的线程负载要高于处理最后几行的线程，在 OpenMP 的 schedule 子句默认的划分方式 (static) 就是这样，可以通过设置 chunk size，使用循环划分或者块循环划分来均衡负载。

2.7 代码实现

篇幅所限，部分函数仅展示部分代码，省略与其他函数重复部分。

2.7.1 除法串行 + 消去并行 + 行划分

```
1 void m_OpenMP_1(int n)//除法串行+消去并行+行划分
2 {
3     #pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
4     for(k = 0; k < n; ++k)
5     {
6         #pragma omp single
7         {
```

```
            tmp = m[k][k];
            for(j = k + 1; j < n; ++j)
                m[k][j] = m[k][j] / tmp;
            m[k][k] = 1.0;
        }
        #pragma omp for
        for(i = k + 1; i < n; ++i)
        {
            tmp = m[i][k];
            for(j = k + 1; j < n; ++j)
                m[i][j] = m[i][j] - tmp * m[k][j];
            m[i][k] = 0.0;
        }
    }
}
```

2.7.2 除法串行 + 消去并行 + 列划分

```
void m_OpenMP_2(int n)//除法串行+消去并行+列划分
{
    #pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
    for(k = 0; k < n; ++k)
    {
        #pragma omp single
        {
            ......
        }
        #pragma omp for
        for(j = k + 1; j < n; ++j)
        {
            ......
        }
    }
}
```

2.7.3 除法并行 + 消去串行 + 行划分

```
void m_OpenMP_3(int n)//除法并行+消去串行+行划分
{
    #pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
    for(k = 0; k < n; ++k)
    {
        tmp = m[k][k];
        #pragma omp for
            ......
        #pragma omp single
        {
```

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

2.7.4 除法并行 + 消去串行 + 行划分

```
void m_OpenMP_3(int n)//除法并行+消去串行+行划分
{
    #pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
    for(k = 0; k < n; ++k)
    {
        tmp = m[k][k];
        #pragma omp for
            ......
        #pragma omp single
        {
            ......
        }
    }
}
```

2.7.5 除法并行 + 消去并行 + 行划分

```
void m_OpenMP_4(int n)//除法并行+消去并行+行划分
{
    #pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
    for(k = 0; k < n; ++k)
    {
        tmp = m[k][k];
        #pragma omp for
            ......
        #pragma omp for
            ......
    }
}
```

2.7.6 除法并行 + 消去并行 + 负载均衡

```
void m_OpenMP_5(int n)//除法并行+消去并行+行划分+负载均衡
{
    #pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
    for(k = 0; k < n; ++k)
    {
        tmp = m[k][k];
```

```
7           #pragma omp for schedule(static,2)
8                 . . . . . .
9           #pragma omp for schedule(static,2)
10                . . . . . .
11      }
12  }
```

2.8 实验结果

2.8.1 实验平台运行

在 Code::Blocks 中运行实验程序。

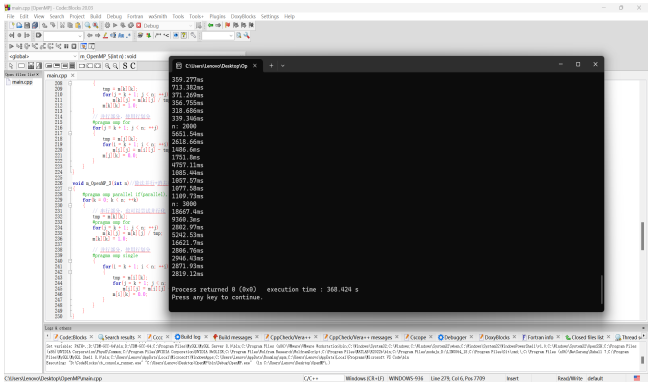

图 2.10: 实验平台运行图

2.8.2 不同并行方案和问题规模下运行用时表

单位：ms

问题规模	n=10	n=50	n=100	n=500	n=1000	n=2000	n=3000
传统串行算法	0.0011	0.093	0.6704	76.869	1540.6	5328.5	18667.4
SIMD	0.0012	0.0435	0.3379	34.942	885.68	2586.6	9360.3
除法串行 + 消去并行 + 行划分	3.2553	9.5179	18.463	107.29	610.43	1385.2	2802.9
除法串行 + 消去并行 + 列划分	1.8593	9.49	19.178	114.80	795.20	1687.6	5242.5
除法并行 + 消去串行 + 行划分	1.8589	9.572	18.858	135.41	1917.2	4574.7	16621.7
除法并行 + 消去并行 + 行划分	1.8589	9.2586	18.961	106	832.59	1145.2	3126.05
负载均衡（size=2）	1.8656	9.5013	18.650	110.99	1564.3	1194.3	2946.43
负载均衡（size=4）	1.8737	9.4974	21.301	110.34	832.17	4762.1	2871.93
负载均衡（size=8）	1.8846	9.3685	22.123	102.85	1309.7	1046.6	2819.12

2.8.3 实验结果总结

1、问题规模的影响：问题规模较小时，总体并行优化算法相较于传统串行用时普遍较长。这是因为线程创建和销毁以及静态线程中线程的保持和各种同步量的加入等因素都会延长算法运算的时间，而因为问题规模较小，其优化的效果没有很好地表现出来。随着问题规模的增大，并行算法的优化效果才开始显现。

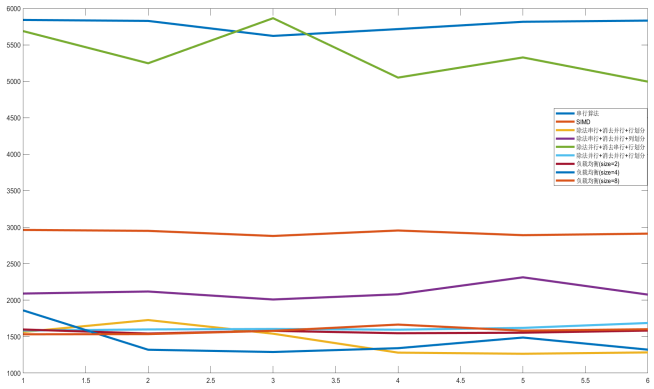

图 2.11: 问题规模为 1000，不同算法下运行用时折线图

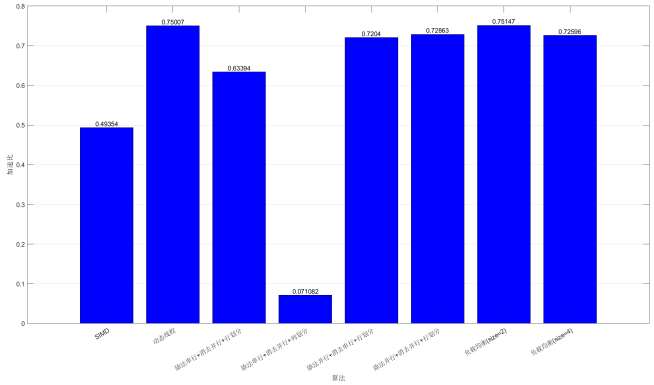

图 2.12: 不同算法平均加速比柱状图

2、任务划分策略的选择：在按行分配与按列分配的问题上，我们一直都认为按行分配是更优的选择，而事实也确实如此，两个按列分配方案的运行时间都要慢于按行分配的时间，这其中主要是 cache 高速缓存行内存存储访问的问题，使得在进行数据查找能更快地获取同一行的数据。这一点在之前的实验中有所涉及。

3、并行区域划分问题：有关除法部分与消去部分的并行化，在问题规模足够大的情况下，两部分并行化肯定是最好的选择。而在两部分的比较中，消去部分的并行化能体现更好的优化效果，因为其为二重循环，时间复杂度更高，因此相应的可优化行也更强。

4、负载均衡问题：在实验中，我们设置了三组 chunk size，分别为 2，4，8。具体的实验结果中，我们也看到了一定的优化效果，其加速比也更大一些。但是并不明显，后续可以进一步探究其原因，并在此问题上探索更好的优化方案。但此次实验并不再作讨论。

2.9 性能分析

对问题规模 n=1000 时各算法的性能进行具体的分析。

上图中，时间最长的函数为按列分配的方案对应函数，我们可以关注它的 Instructions Retired 和 CPI Rate 这两个指标。它的指令数相较于其他方案并没有什么差异，问题出在 CPI Rate。它们的 CPI Rate 相对较高，说明平均每时钟周期内执行的指令数相对较少，这对应了之前提及的 cache 高速缓存行内存存储访问的问题，按列查找数据比按行查找需要花费更长的时间，从而导致性能的下降。

其余方面，由于 OpenMP 是自动化的工具，其对程序的并行化自动完成，我们很难在不知道具体内部具体的并行步骤的情况下进行清楚地分析。

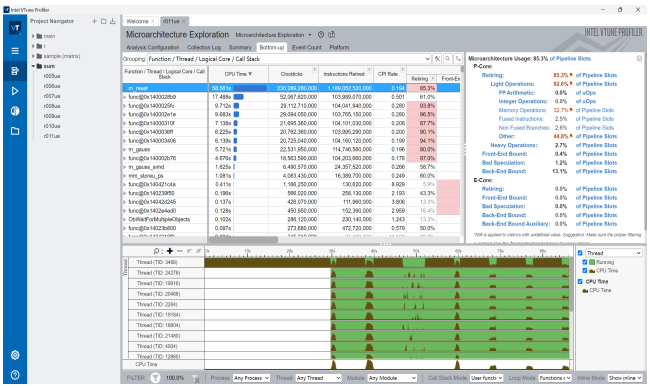

图 2.13: VTune Proflier 分析图

3 特殊高斯消去计算

3.1 算法介绍

特殊高斯消去计算来自一个实际的密码学问题–Gröbner 基的计算与普通高斯消去计算的区别如下:

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。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。

3.2 算法设计

3.2.1 传统串行算法

a) 实际问题中矩阵规模很大，消元子和被消元行的数量很多 (可能达到百万级)，大大超出内存容量，一种处理方式是逐批次将消元子和被消元行读入内存，执行下面步骤 b)-c);

b) 对当前批次中每个被消元行，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) 或首项不在当前批次覆盖范围内、或首项在范围内但无对应消元子或该行，若为情况 2 则该行此批次计算完成;

c) 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; 如其首项被当前批次覆盖，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; 重复上述过程，直至所有批次都处理完毕，此时消元子和被消元行共同组成结果矩阵——可能存在很多空行，

3.2.2 AVX 向量化算法：16 路向量化

采用了 AVX 指令集，可以将数据进行 16 路向量化，同时也和 Pthreads 与 OpenMP 相结合，探寻更优的优化效果。

3.2.3 Pthreads 共享内存：互斥量同步

将整个特殊高斯消去计算都纳入了 Pthreads 共享内存范围，在这里，用之前未使用过的互斥量来控制访问临界区，采用静态线程的方法，实现 Pthreads 共享内存。

3.2.4 OpenMP 共享内存

OpenMP 是一种更方便的工具，自动进行并行的分配，同时也和 AVX 向量化相结合，探寻更优的优化效果。

3.3 代码实现

为了实现特殊高斯消去计算，程序中有多个辅助函数，但是由于篇幅有限，因此不作展示。

3.3.1 传统串行算法

参见上次 SIMD 编程作业相关部分。

3.3.2 AVX

```
void AVX_GE() {
  int begin = 0;
  int flag;
  flag = readRowsFrom(begin);       //读取被消元行
  int num = (flag == -1) ? maxrow : flag;
  for (int i = 0; i < num; i++) {
    while (findfirst(i) != -1) {
      int first = findfirst(i);
      if (ifBasis[first]==1) {  //存在该消元子
        //int* basis = iToBasis.find(first)->second;
        int j = 0;
        for (; 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];
        }
      }
      else {
        int j = 0;
        for (; j + 8 < maxsize; j += 8) {
```

```
            __m256i vij = _mm256_loadu_si256((__m256i*) & gRows[i][j]);
            _mm256_storeu_si256((__m256i*) & gBasis[first][j], vij);
          }
          for (; j < maxsize; j++) {
            gBasis[first][j] = gRows[i][j];
          }
          //iToBasis.insert(pair<int, int*>(first, gBasis[first]));
          ifBasis[first] = 1;
          ans.insert(pair<int, int*>(first, gBasis[first]));
          break;

        }
      }
    }
}
```

3.3.3 Pthreads

```
void* GE_lock_thread(void* param) {
  for (int i = t_id; i < num; i += NUM_THREADS) {
    while (findfirst(i) != -1) {
      int first = findfirst(i);
      if (ifBasis[first]==1) {  //存在首项为first消元子
        //int* basis = iToBasis.find(first)->second;  //找到该消元子的数组
        for (int j = 0; j < maxsize; j++) {
          gRows[i][j] = gRows[i][j] ^ gBasis[first][j];
            }
      }
      else {   //升级为消元子
        pthread_mutex_lock(&lock); //如果第first行消元子没有被占用，则加锁
        if (ifBasis[first]==1)
        {
          pthread_mutex_unlock(&lock);
          continue;
        }
        for (int j = 0; j < maxsize; j++) {
          gBasis[first][j] = gRows[i][j];      //消元子的写入
        }
        ifBasis[first] = 1;
        ans.insert(pair<int, int*>(first, gBasis[first]));
        pthread_mutex_unlock(&lock);           //解锁
        break;
      }
    }
  }
  pthread_exit(NULL);
  return NULL;
}
```

```
31
32 void GE_pthread() {
33   int begin = 0;
34   int flag;
35   flag = readRowsFrom(begin);      //读取被消元行
36   int num = (flag == -1) ? maxrow : flag;
37   pthread_mutex_init(&lock, NULL);  //初始化锁
38   pthread_t* handle = (pthread_t*)malloc(NUM_THREADS * sizeof(pthread_t));
39   threadParam_t* param = (threadParam_t*)malloc(NUM_THREADS * sizeof(threadParam_t));
40   for (int t_id = 0; t_id < NUM_THREADS; t_id++) {//分配任务
41     param[t_id].t_id = t_id;
42     param[t_id].num = num;
43     pthread_create(&handle[t_id], NULL, GE_lock_thread, &param[t_id]);
44   }
45   for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
46     pthread_join(handle[t_id], NULL);
47   }
48   pthread_mutex_destroy(&lock);
49 }
```

3.3.4 OpenMP

```
1  void GE_omp() {
2          int begin = 0;
3    int flag;
4    flag = readRowsFrom(begin);      //读取被消元行
5    int t_id = omp_get_thread_num();
6    int num = (flag == -1) ? maxrow : flag;
7          #pragma omp parallel num_threads(NUM_THREADS)
8    {
9          #pragma omp for schedule(guided)
10   for (int i = 0; i < num; i++)
11         {
12        . . . . . .
13             #pragma omp critical
14        . . . . . .
15     }
16   }
17   }
18 }
```

3.3.5 Pthreads+AVX

```
1  void* AVX_lock_thread(void* param) {
2    for (int i = t_id; i  < num; i += NUM_THREADS) {
3      while (findfirst(i) != -1) {
4        int first = findfirst(i);
```

```
        if (ifBasis[first]==1) {  //存在该消元子
          int j = 0;
          for (; j + 8 < maxsize; j += 8) {
              . . . . . .
                }
        }
        else {
          pthread_mutex_lock(&lock); //如果第first行消元子没有被占用，则加锁
          if (ifBasis[first]==1)
          {
            pthread_mutex_unlock(&lock);
            continue;
          }
          int j = 0;
          for (; j + 8 < maxsize; j += 8) {
              . . . . . .
          }
          ifBasis[first] = 1;
          ans.insert(pair<int, int*>(first, gBasis[first]));
          pthread_mutex_unlock(&lock);
          break;
        }
        }
    }
    pthread_exit(NULL);
    return NULL;
}

void AVX_pthread() {
    int begin = 0;
    int flag;
    flag = readRowsFrom(begin);      //读取被消元行
    int num = (flag == -1) ? maxrow : flag;
    pthread_mutex_init(&lock, NULL);  //初始化锁
    pthread_t* handle = (pthread_t*)malloc(NUM_THREADS * sizeof(pthread_t));
    threadParam_t* param = (threadParam_t*)malloc(NUM_THREADS * sizeof(threadParam_t));
    for (int t_id = 0; t_id < NUM_THREADS; t_id++) {//分配任务
      param[t_id].t_id = t_id;
      param[t_id].num = num;
      pthread_create(&handle[t_id], NULL, AVX_lock_thread, &param[t_id]);
    }
    for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
      pthread_join(handle[t_id], NULL);
    }
    pthread_mutex_destroy(&lock);
}
```

3.3.6 OpenMP+AVX

```
void AVX_GE_omp() {
  int begin = 0;
  int flag;
  flag = readRowsFrom(begin);      //读取被消元行
  int num = (flag == -1) ? maxrow : flag;
  int i = 0, j = 0;
        #pragma omp parallel  num_threads(NUM_THREADS),private(i,j)
        #pragma omp for schedule(guided)
  for (i = 0; i < num; i++) {
          . . . . . .
      else {
                #pragma omp critical
        {
                    . . . . . .
        ifBasis[first] = 1;
        ans.insert(pair<int, int*>(first, gBasis[first]));
        }
      }
    }
  }
}
```

3.4 实验结果

3.4.1 实验平台运行

在 Code::Blocks 中运行实验程序。

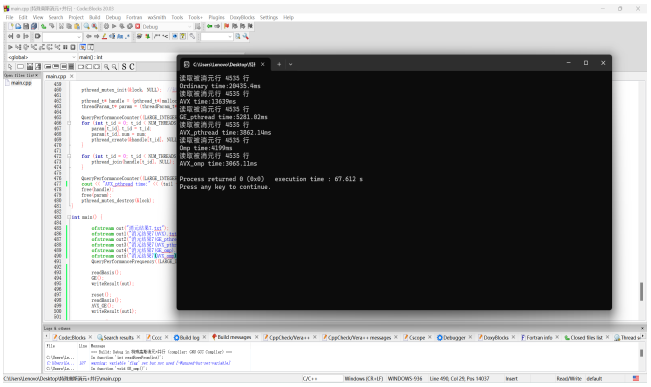

图 3.14: 实验平台运行图

3.4.2 不同算法和测试样本下运行用时表

单位：ms

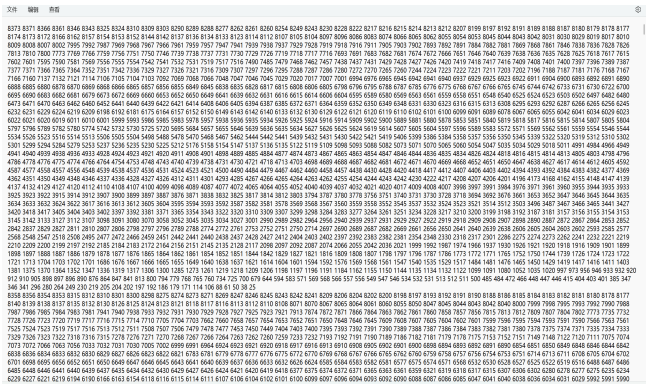

图 3.15: 消元结果

测试样本编号	1	2	3	4	5	6	7
传统串行算法	0.1908	3.448	2.8393	50.9145	131.562	2855.92	20435.4
AVX	0.1206	2.2192	1.9057	32.0717	84.8288	1912.35	13639
Pthreads	0.6083	2.7345	2.0903	24.3165	37.1499	813.565	5281.02
Pthreads+AVX	1.3217	2.2082	1.2888	14.0496	29.9639	573.906	3862.14
OpenMP	0.8586	2.3912	2.4081	21.2157	40.9589	607.6	4199
OpenMP+AVX	0.9735	1.8732	1.1824	11.6593	22.1933	464.713	3065.11

3.4.3 Pthreads 和 OpenMP 对比分析

随着测试样例编号的增长，问题规模也在增长。首先，在 AVX 编程问题上，我们可以发现无论是传统串行算法上，还是在 Pthreads 和 OpenMP 并行优化方案上，与 AVX 的结合都能起到不错的性能优化效果。

在 Pthreads 与 OpenMP 的比较上，我们发现 OpenMP 共享内存并行方案比 PthreadsP 共享内存并行方案展现了更好的性能优化效果。这展现了 OpenMP 虽然作为相较顶层的编程模块，却拥有更高级的 API，能够更好地完成并行优化任务。

4 链接

github 项目链接:https://github.com/TimeIsAPlace/Pthreads-OpenMP.git