南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

高斯消去法的多线程 pthread & OpenMP 编程

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 5 月 26 日

摘要

在本次实验中，我对普通高斯消去算法和特殊高斯消去算法展开了 pthread 和 OpenMP 编程实验，比较相同多线程架构的不同划分方式，不同线程建立方式的效果，并将其与 SIMD 编程技术结合，通过性能测试和 profiling 体会多线程编程的性能和编程方式。

关键字：Pthread，OpenMP，高斯消去，特殊高斯消去，并行

目录

一、 实验背景

（一） 实验目的

在本次实验将在 pthread 和 openmp 编程下探究普通高斯消去算法以及特殊高斯消去算法的优化程度，通过性能测试以及 profiling 探究 pthread 和 openmp 编程的原理以及其对高斯消去算法的优化情况，同时结合上一次实验 SIMD 编程讨论，并加以总结。该实验同时也是期末大作业的一个分支。

（二） 实验环境

本次实验主要在以下两个平台进行。了解到 O2 优化或许会降低实验结果可靠性，在本次实验中，不开启任何形式的 O2 优化。

- x86 平台：个人物理机，x86-64，Intel(R) Core(TM) i5-10200H，windows 11 系统，G++ 编译器，C++ 17 GNU。
- ARM 平台：华为鲲鹏服务器，G++ 编译器。

（三） 普通高斯消去算法

高斯消去法主要分为消去和回代两个部分，如图1。消去部分复杂度较高，为 $O(n^3)$，回代部分也有 $O(n^2)$ 的复杂度。

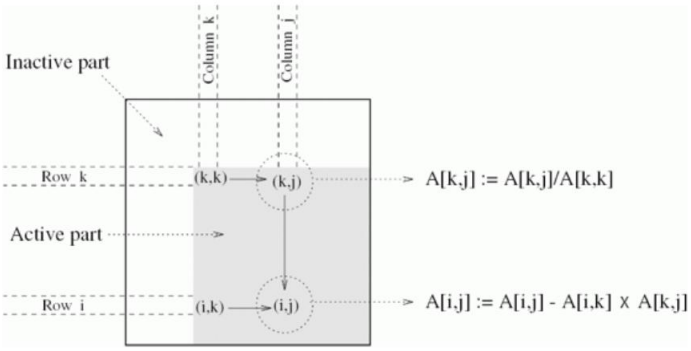


图 1: 高斯消元法示意图

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，全部结束后结果如图2所示，然后从最后一行向上回代。

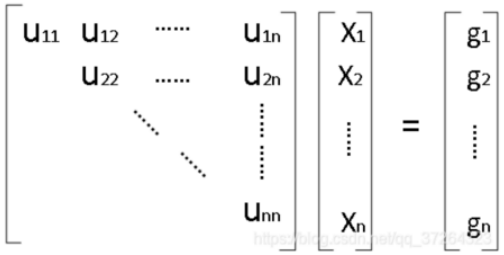


图 2: 消去过程结果示意图

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 x_{i+1},x_{i+2},..., x_n 计算出 x_i。

（四） 特殊高斯消去算法

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。

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。

考虑单次完成数据读取的简单情况，此特殊的高斯消去计算过程 (串行算法) 如下:

对于每个被消元行，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) 或首项无对应消元子。

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，消元子和被消元行共同组成结果矩阵–可能存在很多空行。

二、 普通高斯消去算法

（一） 函数框架 & 平凡算法

在本次实验中，我们采用以下计时函数对程序性能进行评价。该框架主要依赖于 chrono 库，时间精度可以达到 1ms。对于数据组较小的运算，可以通过增加循环次数取平均值的方式保证其精度。

运行及计时函数

```
void op(void (*method)(int),int n,int t){
    reset(n);
    cout<<"␣n="<<n<<"␣t="<<t;
    auto start=chrono::steady_clock::now();
    for(int i=0;i<t;i++)method(n);
    auto finish=chrono::steady_clock::now();
    auto duration=chrono::duration_cast<chrono::milliseconds>(finish-start);
    double pertime=duration.count()*double(1.0)/t;
    cout<<"␣pertime="<<pertime<<"ms\n";
}
```

在上一次 simd 实验中，我发现计算结果常常出现 nan 和 inf，这不但会大幅影响计算精度，也会对计时的可靠性造成影响。这和我的计算流程有关，更与矩阵初始化的方式有关。因此，在本次实验中，我调整了矩阵初始化函数，使得他在每一次计算中都能返得到较为平稳的结果。

初始化函数

```
void reset(int n){
    srand(time(NULL));
    for(int i=0;i<n;i++){
        for(int j=0;j<i;j++)
```

```
            A[i][j]=0;
        A[i][i]=1.0;
        for(int j=i+1;j<n;j++)
            A[i][j]=(rand()%1000+1)/10.0;
    }
    for(int k=0;k<n;k++)
        for(int i=k+1;i<n;i++)
            for(int j=0;j<n;j++)
                A[i][j]+=A[k][j];
}
```

与此同时，为了检查优化算法的正确性，我在本次实验中引入了 check 函数，将两个算法的结果进行对拍，可以更好的确定算法的可靠性，也为后续计算时间提供了保障。(在该小节中，以下每一种算法均保证通过了在 check 函数中与平凡算法的相互检查)

check 函数

```
void check(void (*m1)(int),void (*m2)(int)){
    reset(20);
    float B[20][20];
    for(int i=0;i<20;i++)for(int j=0;j<20;j++)B[i][j]=A[i][j];
    cout<<"method1 ans:\n";
    m1(20);
    for(int i=0;i<20;i++){
        for(int j=0;j<20;j++)
            cout<<A[i][j]<<" ";
        cout<<'\n';
    }
    for(int i=0;i<20;i++)for(int j=0;j<20;j++)A[i][j]=B[i][j];
    m2(20);
    cout<<"method2 ans:\n";
    for(int i=0;i<20;i++){
        for(int j=0;j<20;j++)
            cout<<A[i][j]<<" ";
        cout<<'\n';
    }
}
```

以下是本次运算采用的平凡算法（实验部分 common 组），相比于上一次的平凡算法，本次的平凡算法更好的保证了同一矩阵在多次消元后仍能保持相同的结果，而不会出现意外的 inf 与 nan，这将大幅提高计时结果的正确性。

平凡算法

```
void common(int n){
    for (int k = 0; k < n; ++k) {
        for (int j = k + 1; j < n; ++j)
            A[k][j] /= A[k][k];
        A[k][k] = 1.0;
        for (int i = k + 1; i < n; ++i) {
            for (int j = k + 1; j < n; ++j) {
```

```
                A[i][j] -= A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

（二） 多线程 pthread 编程

1. 算法设计

在该部分中，主要考虑以下四种 pthead 编程方式：

- 为第二个内层循环创建动态线程（实验部分 active 组）

  为第二个内层循环（即更新其他行的操作，对应平凡算法的 6-10 行）创建动态线程，每次外层循环迭代时，根据当前的主元行来动态地创建一组线程。每个线程将当前主元行下方的某一行进行消元操作。

- 为第二个内层循环创建静态线程，采用信号量同步的方式（实验部分 stat 组）

  在程序开始时创建一组固定数量的线程，并在整个算法执行过程中重复使用这些线程。以信号量同步的方式保证在更新某一行之前，上面的所有行均已经完成消元。

- 在上一方法的基础上，将三层循环全部纳入线程函数（实验部分 stat3 组）

  主线程进行除法操作，其他线程在收到主线程除法完成的信号进行减法操作。在收到其他线程完成减法的信号后，主线程再进行下一轮除法操作，保证了程序的可靠性

- 在上一方法的基础上，不再使用信号量同步，改为使用 barrier 同步（实验部分 statb 组）

  建立两个同步点，分别在除法和减法完成后，保证程序的并发性和正确性。

2. 编程实现

动态线程版本

```
struct threadParam_t {
    int k;
    int t_id;
    int n;
};
void* threadFunc(void* param) {
    auto* p = static_cast<threadParam_t*>(param);
    int k = p->k;
    int t_id = p->t_id;
    int n = p->n;
    int i = k + t_id + 1;
    for (int j = k + 1; j < n; ++j)
        A[i][j] -= A[i][k] * A[k][j];
    A[i][k] = 0;
    pthread_exit(NULL);
```

```
}
void active(int n) {
    for (int k = 0; k < n; ++k) {
        for (int j = k + 1; j < n; ++j)
            A[k][j] /= A[k][k];
        A[k][k] = 1.0;
        vector<pthread_t> handle(n - 1 - k);
        vector<threadParam_t> param(n - 1 - k);
        for (int t_id = 0; t_id < n - 1 - k; ++t_id) {
            param[t_id].k = k;
            param[t_id].t_id = t_id;
            param[t_id].n = n;
            pthread_create(&handle[t_id], NULL, threadFunc, &param[t_id]);
        }
        for (int t_id = 0; t_id < n - 1 - k; ++t_id)
            pthread_join(handle[t_id], NULL);
    }
}
```

信号量同步版本

```
struct threadParam_t_a{
    int t_id;
    int n;
};
sem_t sem_main;
sem_t sem_workerstart[NUM_THREADS];
sem_t sem_workerend[NUM_THREADS];
void* threadFunc_a(void* param) {
    threadParam_t_a* p = (threadParam_t_a*)param;
    int t_id = p->t_id;
    int n = p->n;
    for (int k = 0; k < n; ++k) {
        sem_wait(&sem_workerstart[t_id]);
        for (int i = k + 1 + t_id; i < n; i += NUM_THREADS) {
            for (int j = k + 1; j < n; ++j)
                A[i][j] -= A[i][k] * A[k][j];
            A[i][k] = 0.0;
        }
        sem_post(&sem_main);
        sem_wait(&sem_workerend[t_id]);
    }
    pthread_exit(NULL);
}
void stat(int n) {
    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[NUM_THREADS];
    threadParam_t_a param[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, threadFunc_a, &param[t_id]);
    }
    for (int k = 0; k < n; ++k) {
        for (int j = k + 1; j < n; ++j)
            A[k][j] /= A[k][k];
        A[k][k] = 1.0;
        for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
            sem_post(&sem_workerstart[t_id]);
        for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
            sem_wait(&sem_main);
        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);
    }
    for (int i = 0; i < NUM_THREADS; ++i) {
        sem_destroy(&sem_workerstart[i]);
        sem_destroy(&sem_workerend[i]);
    }
    sem_destroy(&sem_main);
}
```

信号量同步/三重循环版本

```
struct threadParam_t_s3{
    int t_id;
    int n;
};
sem_t sem_leader;
sem_t sem_Divsion[NUM_THREADS - 1];
sem_t sem_Elimination[NUM_THREADS - 1];
void* threadFunc_s3(void* param) {
    auto * p = (threadParam_t_s3*)param;
    int t_id = p->t_id;
    int n = p->n;
    for (int k = 0; k < n; ++k) {
        if (t_id == 0) {
            for (int j = k + 1; j < n; ++j) A[k][j] = A[k][j] / A[k][k];
            A[k][k] = 1.0;
            for (int i = 0; i < NUM_THREADS - 1; ++i)
                sem_post(&sem_Divsion[i]);
        }
```

```
        else sem_wait(&sem_Divsion[t_id - 1]);
        for (int i = k + 1 + t_id; i < n; i += NUM_THREADS) {
            for (int j = k + 1; j < n; ++j)
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            A[i][k] = 0.0;
        }
        if (t_id == 0) {
            for (int i = 0; i < NUM_THREADS - 1; ++i)
                sem_wait(&sem_leader);
            for (int i = 0; i < NUM_THREADS - 1; ++i)
                sem_post(&sem_Elimination[i]);
        } else {
            sem_post(&sem_leader);
            sem_wait(&sem_Elimination[t_id - 1]);
        }
    }
    pthread_exit(NULL);
}
void stat3(int n) {
    sem_init(&sem_leader, 0, 0);
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        sem_init(&sem_Divsion[i], 0, 0);
        sem_init(&sem_Elimination[i], 0, 0);
    }
    pthread_t handles[NUM_THREADS];
    threadParam_t_s3 param[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, threadFunc_s3, &param[t_id]);
    }
    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
        pthread_join(handles[t_id], NULL);
    sem_destroy(&sem_leader);
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        sem_destroy(&sem_Divsion[i]);
        sem_destroy(&sem_Elimination[i]);
    }
}
```


barrier 版本

```
struct threadParam_t_b{
    int t_id;
    int n;
};
pthread_barrier_t barrier_Divsion;
pthread_barrier_t barrier_Elimination;
void* threadFunc_b(void* param) {
```

```
    auto* p = (threadParam_t_b*)param;
    int t_id = p->t_id;
    int n=p->n;
    for (int k = 0; k < n; ++k) {
        if (t_id == 0) {
            for (int j = k + 1; j < n; j++)
                A[k][j] /= A[k][k];
            A[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)A[i][j] -= A[i][k] * A[k][j];
            A[i][k] = 0.0;
        }
        pthread_barrier_wait(&barrier_Elimination);
    }
    pthread_exit(NULL);
}
void statb(int n) {
    pthread_barrier_init(&barrier_Divsion, NULL, NUM_THREADS);
    pthread_barrier_init(&barrier_Elimination, NULL, NUM_THREADS);
    pthread_t handles[NUM_THREADS];
    threadParam_t_b param[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, threadFunc_b, &param[t_id]);
    }
    for (int t_id = 0; t_id < NUM_THREADS; t_id++)
        pthread_join(handles[t_id], NULL);
    pthread_barrier_destroy(&barrier_Divsion);
    pthread_barrier_destroy(&barrier_Elimination);
}
```

3. 性能测试

在矩阵规模上，我们选择 16,32,64,128,256,500,750,1000,1500,2000 十个矩阵规模。考虑到在 n 规模较小的时候运行时间较短，并进行多次实验取平均值，每个 n 的遍历次数 t 计为 5 和 $1000000/n^2$ 的较大值。

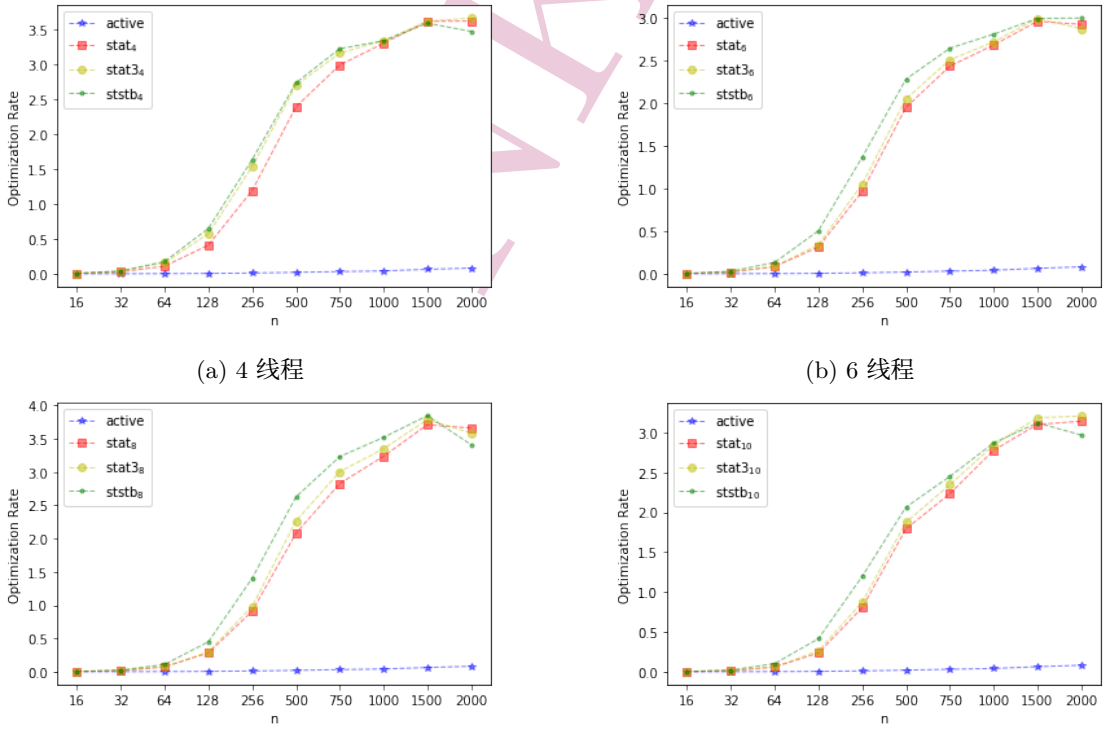
a) x86 平台

考虑到本人物理机与华为鲲鹏服务器都是 8 核的架构，首先我们以 4 线程，6 线程，8 线程，10 线程在本人物理机中进行测试，结果如表1所示。(其中，由于线程数量对平凡算法和动态线程算法不造成影响，且需要消耗极长的运行时间，这两个实验组仅测试一次)

与平凡算法相比较，得到加速比如图3所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.004	0.035	0.266	2.115	16.67	121.8	410.2	971.2	3388	7947
active	6.375	24.67	96.43	375.9	1490	5701	12826	23235	52868	96561
$stat_4$	0.766	1.294	2.455	5.148	14.00	51.00	137.4	294.6	938.6	2199
$stat3_4$	0.597	0.936	1.693	3.689	10.87	45.20	130.0	291.2	938.2	2173
$ststb_4$	0.558	0.855	1.5164	3.262	10.20	44.60	127.4	291.6	945.6	2293
$stat_6$	1.055	1.708	3.189	6.738	17.27	62.40	168.6	362.2	1145	2719
$stat3_6$	0.993	1.639	2.992	6.213	15.87	59.40	163.6	357.4	1132	2767
$ststb_6$	0.769	1.162	2.004	4.197	12.20	53.40	155.2	345.8	1132	2651
$stat_8$	1.263	2.046	3.693	7.475	18.33	58.60	145.2	300.2	913.4	2172
$stat3_8$	1.204	1.945	3.541	7.049	17.00	53.80	136.6	290.0	895.2	2218
$ststb_8$	0.941	1.434	2.426	4.705	11.87	46.40	127.0	276.0	881.2	2332
$stat_{10}$	1.552	2.496	4.471	8.967	20.73	68.00	183.8	349.8	1093	2531
$stat3_{10}$	1.479	2.381	4.168	8.082	19.07	65.00	175.0	342.0	1064	2479
$ststb_{10}$	1.107	1.578	2.602	5.098	13.93	59.00	167.6	338.6	1086	2678

表 1: x86 平台测试结果 (单位:ms)



(a) 4 线程

(b) 6 线程

(c) 8 线程

(d) 10 线程

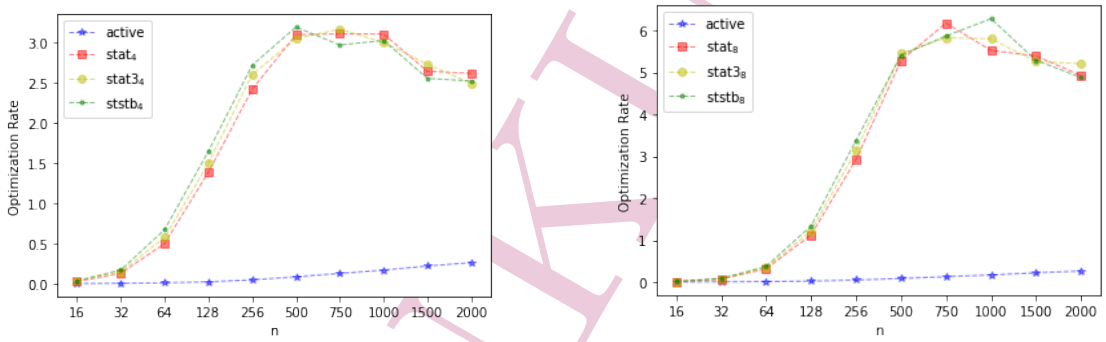
图 3: x86 平台加速比

b) ARM 平台

对 ARM 平台进行测试，分别选择 4 线程和 8 线程进行，结果如表2所示。与平凡算法相比较，得到加速比如图4所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.010	0.081	0.652	5.164	41.00	311.8	1055	2538	8556	20000
active	2.718	11.33	49.20	202.2	826.2	3548	8109	14829	38553	75624
$stat_4$	0.352	0.615	1.299	3.721	16.93	100.6	339.2	816.6	3236	7641
$stat3_4$	0.297	0.538	1.131	3.443	15.73	102.2	333.0	845.2	3129	8018
$ststb_4$	0.257	0.468	0.967	3.131	15.07	97.60	355.0	838.2	3346	7926
$stat_8$	0.674	1.128	2.053	4.590	14.00	59.00	170.8	459.4	1587	4058
$stat3_8$	0.608	1.040	1.934	4.229	13.07	57.20	181.0	437.6	1627	3838
$ststb_8$	0.578	0.931	1.738	3.852	12.13	57.60	179.8	404.0	1618	4098

表 2: ARM 平台测试结果 (单位:ms)



(a) 4 线程 (b) 8 线程

图 4: ARM 平台加速比

4. profiling

将各优化算法使用 Vtune 进行 profiling，结果如图5所示，其中 active 组创建线程数量极大，仅进行部分展示，其余进行全部展示。stat 组与 stat3 组结果没有明显差别，选择一个进行展示。最后一张子图展示 barrier 同步 statb 组的结尾细节，可以看到每个线程虽然总体结束时间相近，但仍有先后之分。

5. 总结

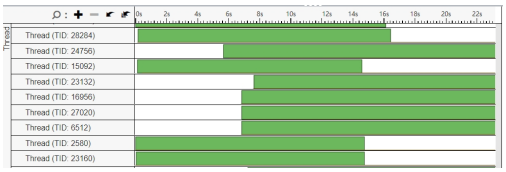
该部分所有结果均通过 check 函数检查正确性。

从性能测试结果可以看出，无论是 x86 平台还是 ARM 平台，动态分配进程的表现都不尽如人意。其分配的开销远远大于消元进行的开销（即使是在矩阵规模达到 2000 依然如此）。三种静态分配内存的表现效果都较为理想，且随着矩阵规模增大效果越发明显。在矩阵规模较小的时候，因为线程分配的开销相较于运算开销来说比较显著，所以优化比小于 1。在矩阵规模较大后，加速比逐渐趋于稳定。甚至在矩阵规模过大的情况下，或许受到访存开销的影响和平台波动影响，x86 平台和 ARM 平台甚至有了一点加速比下降的势头。

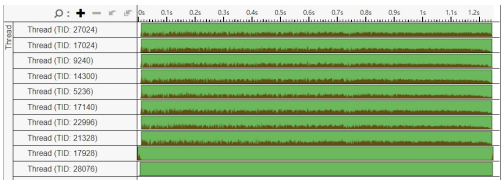
从线程角度对比来看，x86 平台上，8 线程表现最好，略好于 4 线程，而 6 线程和 10 线程相较来说效果不够理想。猜测是对于 8 核处理器来说 4 线程和 8 线程更便于分配。但 8 线程并没有高出 4 线程太多，猜测是设备 CPU 资源有限，也无法将全部电脑计算资源用于该程序的运行导致。

然而，ARM 平台上，8 线程的加速比远远高于 4 线程，最多时可以达到 6 倍以上的加速比。体现了华为鲲鹏服务器的性能之高，以及 pthread 并行化在 ARM 平台上的效果较为理想。

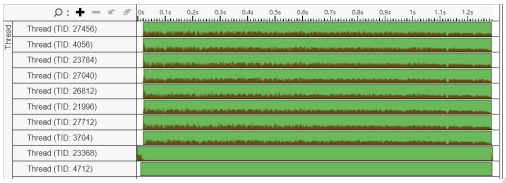
通过 profiling 结果（见图5）可以看出，静态分配线程各组线程的持续时间几乎从头到尾。从 d 图可以看到各被分配的子线程虽然结束时间相近，但也是存在不确定的先后顺序，这也与我们所期望的结果相同。动态分配的线程数量众多，持续时间也各不相同，这也体现了为什么动态分配线程造成如此大的开销以至于程序性能反向优化。



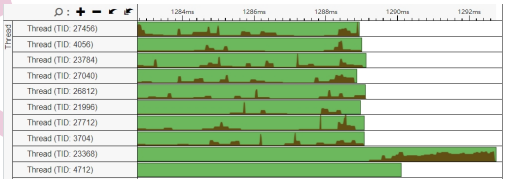
(a) 动态分配线程 active 组



(b) 信号量同步 stat3 组



(c) barrier 同步 statb 组



(d) barrier 结尾细节展示

图 5: Vtune 分析结果

（三） OpenMP 编程

1. 算法设计

在该部分中，主要讨论以在内层循环中以行来划分进程（mp 组）和以列来划分进程（mp2 组）两种不同情况，在这两种情况下，分别在 x86 平台下和 ARM 平台下进行实验，并与平凡算法进行对比。

2. 编程实现

行划分版本（mp 组）

```
void mp(int n){
    int i,j,k;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < n; ++k) {
        #pragma omp single
        {
            for (j = k + 1; j < n; ++j)
                A[k][j] /= A[k][k];
            A[k][k] = 1.0;
        }
```

```
        #pragma omp for
        for (i = k + 1; i < n; ++i) {
            for (j = k + 1; j < n; ++j) {
                A[i][j] -= A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

列划分版本（mp2 组）

```
void mp2(int n){
    int i,j,k;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < n; ++k) {
        #pragma omp single
        {
            for (j = k + 1; j < n; ++j)
                A[k][j] /= A[k][k];
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (j = k + 1; j < n; ++j) {
            for (i = k + 1; i < n; ++i) {
                A[i][j] -= A[i][k] * A[k][j];
            }
        }
        for (i = k + 1; i < n; ++i)
            A[i][k] = 0;
    }
}
```


3. 性能测试

在矩阵规模上，我们选择 16,32,64,128,256,500,750,1000,1500,2000 十个矩阵规模。考虑到在 n 规模较小的时候运行时间较短，并进行多次实验取平均值，每个 n 的遍历次数 t 计为 5 和 $1000000/n^2$ 的较大值。

在线程数量上，分别采用 4 线程和 8 线程进行测试。

a) x86 平台

测试结果如表3所示。

与平凡算法比较，优化比如图6所示。

b) ARM 平台

测试结果如表4所示。

与平凡算法比较，优化比如图7所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.004	0.035	0.266	2.115	16.67	121.8	410.2	971.2	3388	7947
mp_4	1.320	2.586	5.107	9.869	21.60	63.00	192.0	319.8	1101	2460
$mp2_4$	1.318	2.609	5.049	10.44	26.13	93.60	313.6	703.6	3085	8228
mp_8	2.566	5.036	10.05	19.74	37.20	87.60	179.0	346.8	988.8	2180
$mp2_8$	2.569	5.042	9.988	19.03	38.47	103.0	280.8	714.2	2652	6529

表 3: x86 平台测试结果 (单位:ms)

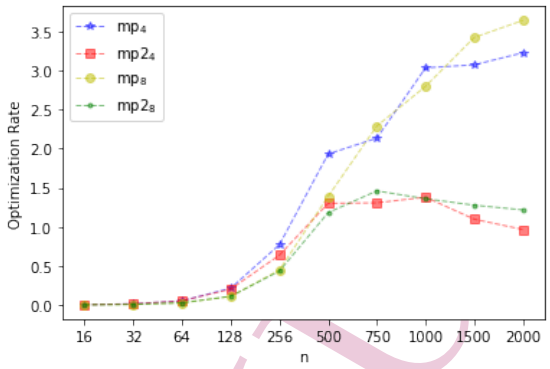


图 6: x86 平台优化比

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.010	0.081	0.652	5.164	41.00	311.8	1055	2538	8556	20000
mp_4	0.025	0.066	0.418	2.000	11.80	80.40	267.8	636.4	2162	5112
$mp2_4$	0.035	0.191	2.135	11.72	48.33	149.0	358.0	910.4	2923	7055
mp_8	0.052	0.248	0.496	1.672	7.000	42.6	139.2	322.4	1077	2525
$mp2_8$	0.090	0.342	1.533	8.984	44.80	133.6	334.6	631.4	1825	4386

表 4: ARM 平台测试结果 (单位:ms)

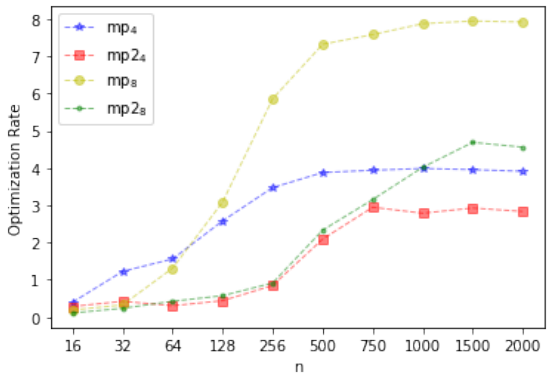


图 7: ARM 平台优化比

4. 总结

从线程分配方式来看，在 x86 平台上，在矩阵规模较小的时候按行分配和按列分配的优化比差距不大，但随着矩阵规模的增大，按行分配和按列分配则有着显著的优化差异。甚至在矩阵规模达到 1500 以上的时候，按列分配甚至造成了对程序的负优化。（因为我们的串行算法也是按行访问的）

在 ARM 平台上，虽然按列分配在矩阵规模较大的时候优化仍然很显著，但是与相同线程的按行分配方案相比，优化比仅仅只有一半多。

这是因为数组存储是行主存储的，访问同一列的元素的开销远远大于访问同一行元素的开销，因此列主访问的并行算法确实存在差于行主存储的串行算法的可能。随着矩阵规模的增大，cache 中单次存储的行数也在变少，从而造成了负优化。

从总体趋势来看，在矩阵规模较小的时候，由于分配线程的开销远高于计算的开销，所以优化比小于 1。随着矩阵规模的增加，优化比逐渐提升。最终，在矩阵规模足够大的时候趋于稳定.

从平台对比来看，x86 平台的 8 线程仍然是稍稍高于 4 线程，在矩阵规模较大后，行主优化比均在 3-4 之间，列主优化比趋于 1 左右（这说明优化结果已经很差了）。

ARM 平台整体优化比较高且增长迅速，行主 4 线程优化比接近 4，8 线程优化比接近 8。这是一个非常大的优化幅度。即使是较为笨拙的列主方法，也获得了 4 线程优化比约为 2.5，8 线程优化比超过 4 的优异成绩。

5. 与 pthread 编程对比

在 x86 平台上，比较两平台上表现最为优异的算法，无论是 4 线程还是 8 线程，pthread 编程和 OpenMP 编程的性能差异不大，均为缓慢增加，然后最后到达 3.5-4.0 的顶峰。

在 ARM 平台上,OpenMP 编程的结果相较于 pthread 编程有着明显的优势，pthread 编程的优化比往往只能达到 3（4 线程），6（8 线程），而 OpenMP 编程的优化比则可以达到接近线程数量的完美数据，这正是我们所期望的结果。

从编程难度上，OpenMP 的编程代码量和难度远远小于 pthread 编程，且不需要对原代码进行任何修改，更便于开发和维护。

（四） SIMD with pthread/OpenMP

1. 算法设计

在此部分中，我们主要对 pthread 和 OpenMP 中各选取一种性能较好且较为稳定的算法进行 SIMD 向量化优化。这里选取的是 pthread 编程的 3 重循环全部纳入线程的信号量同步方式（stat3 组）和 OpenMP 的行主划分方式（mp 组）。对三重循环部分进行 SIMD 向量化优化。分别进行 neon 向量化优化（在 x86 平台中测试）和 SSE 向量化优化（在 ARM 平台中测试）。

2. 编程实现

主要针对最内层循环进行向量化，修改如下。

SSE 向量化

```
__m128 aik=_mm_set_ps(A[i][k],A[i][k],A[i][k],A[i][k]);
for(j=k+1;j<=n-4;j+=4){
    __m128 akj=_mm_loadu_ps(A[k]+j);
    __m128 multi=_mm_mul_ps(aik,akj);
    __m128 aij=_mm_loadu_ps(A[i]+j);
```

```
    aij=_mm_sub_ps(aij,multi);
    _mm_storeu_ps(A[i]+j,aij);
}
for (;j<n; j++)
    A[i][j] -= A[i][k] * A[k][j];
A[i][k] = 0;
```

NEON 向量化

```
float32x4_t aik=vmovq_n_f32(A[i][k]);
for(j=k+1;j<=n-4;j+=4){
    float32x4_t akj=vld1q_f32(A[k]+j);
    float32x4_t multi=vmulq_f32(aik,akj);
    float32x4_t aij=vld1q_f32(A[i]+j);
    aij=vsubq_f32(aij,multi);
    vst1q_f32(A[i]+j,aij);
}
for (;j<n; j++)
    A[i][j] -= A[i][k] * A[k][j];
A[i][k] = 0;
```

3. 性能测试

在 8 线程下，分别对 neon 向量化优化（在 x86 平台）和 SSE 向量化优化（在 ARM 平台）进行测试，并与平凡算法和未进行向量化优化算法进行对比。

a) x86 平台

测试结果如表5所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.004	0.035	0.266	2.115	16.67	121.8	410.2	971.2	3388	7947
stat3	1.204	1.945	3.541	7.049	17.00	53.80	136.6	290.0	895.2	2218
stat3sse	1.213	1.955	3.459	6.607	14.60	40.20	87.20	170.2	494.4	1135
mp	2.566	5.036	10.05	19.74	37.20	87.60	179.0	346.8	988.8	2180
mpsse	2.555	5.057	10.07	20.12	38.20	77.40	132.4	227.6	586.8	1241

表 5: x86 平台测试结果 (单位:ms)

与平凡算法比较，优化比如图8所示。

b) ARM 平台

测试结果如表6所示。

与平凡算法比较，优化比如图9所示。

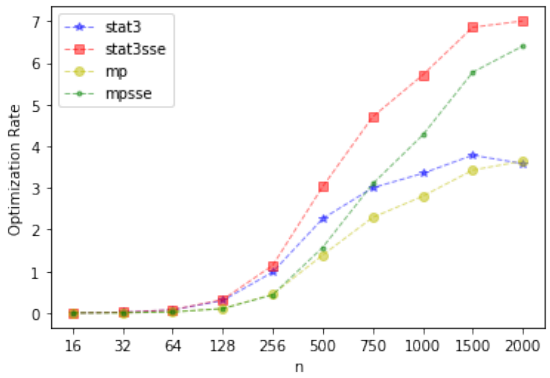


图 8: x86 平台优化比

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.010	0.081	0.652	5.164	41.00	311.8	1055	2538	8556	20000
stat3	0.608	1.040	1.934	4.229	13.07	57.20	181.0	437.6	1627	3838
stat3NEON	0.613	1.040	1.934	4.016	10.93	43.00	117.0	285.2	869.0	2221
mp	0.052	0.248	0.496	1.672	7.000	42.6	139.2	322.4	1077	2525
mpNEON	0.0996	0.220	0.475	1.262	5.067	28.60	89.40	207.8	735.4	1941

表 6: ARM 平台测试结果 (单位:ms)

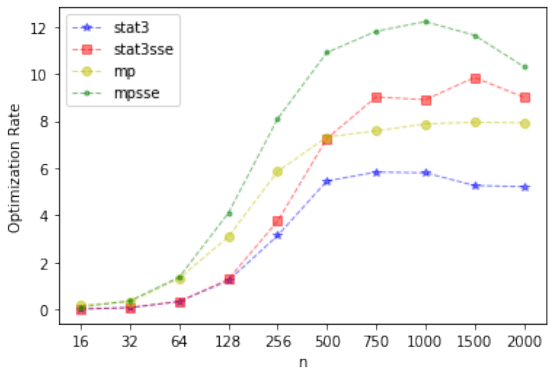


图 9: ARM 平台优化比

4. 总结

在本部分实验中，我们可以看到在 x86 平台上，sse 优化算法较平凡算法获得了 6-7 的优化比，相较于未进行 sse 优化的 pthread/OpenMP 算法也有两倍多的优化，效果与单独进行 sse 优化类似，相当可观。

在 ARM 平台上，neon+stat3 优化算法相较于平凡算法获得了 9 倍左右的优化比，neon+mp 获得了 10 倍以上的优化比，甚至在最高点接近 12 倍优化比。这一数值是相当可观的。不过若是将 neon 优化后和未进行 neon 优化之前的对比，neon+stat3 相较于 stat3 大约有 2 倍优化，neon+mp 相较于 mp 优化不足 1.5 倍。这其实远低于仅仅使用 neon 可以获得的优化比。这可能是因为程序优化已经较为完善所导致，当然，这样的优化比其实仍然是非常可观的。

三、 特殊高斯消去算法

(一) 函数框架 & 平凡算法

在解决特殊高斯消去算法的过程中，我们仍然主要考虑使用位运算的方式，即采取用 int 整型的每一位表示矩阵的每一列。函数的 io 框架如下。

变量定义及读函数

```
const int xsize=130,xlen=xsize/32+1,maxrow=10;//数值以样例1为例，xsize设置为矩阵列数，maxrow略大于被消元行数即可
int row[maxrow][xlen];int xyz[xsize][xlen];
bool is[xsize];int nrow;
void read(){
    fstream xyzfile("消元子.txt", ios::in | ios::out);
    string line;
    while (getline(xyzfile, line)){
        istringstream iss(line);int value;bool first=true;
        while (iss >> value) {
            if(first){nrow=value;is[value]=true;first=false;}
            (xyz[nrow][value/32])|=(1<<value%32);
        }
    }
    xyzfile.close();
    fstream bxyfile("被消元行.txt", ios::in | ios::out);nrow=0;
    while (getline(bxyfile, line)) {
        istringstream iss(line);int value;
        while (iss >> value) {(row[nrow][value/32])|=(1<<value%32);}
        nrow++;
    }
    bxyfile.close();
}
```

写函数

```
void write(){
    ofstream ansfile("ans.txt",ios::app);
    ansfile<<"_____\n";
```

```
    for(int i=xsize-1;i>0;i--){
        if(is[i]){
            for(int j=xlen-1;j>=0;j--){
                for(int u=31;u>=0;u--){
                    if((xyz[i][j])&(1<<u))
                        ansfile<<u+32*j<<'␣';
                }
            }
            ansfile<<'\n';
        }
    }
}
```

在本次实验中，我尝试放弃对一些较小数据集的计算，不再将读写函数以及数组清空函数纳入计时范围内，这可能会对运行时间较快的样例造成困扰，但可以更客观的体现程序的优化效率。

封装及计时函数

```
void op(void(*method)(),int t){
    double tt=0;
    for(int i=0;i<t;i++) {
        memset(row, 0, sizeof(row));
        memset(xyz, 0, sizeof(xyz));
        memset(is, 0, sizeof(is));
        read();
        auto start = chrono::steady_clock::now();
        method();
        auto finish = chrono::steady_clock::now();
        auto duration = chrono::duration_cast<chrono::milliseconds>(finish -
            start);
        tt+=duration.count() * double(1.0);
        write();
    }
    tt/=t;
    cout<<tt<<"ms\n";
}
```

程序的平凡算法如下。

平凡算法

```
void common(){
    for(int i=0;i<nrow;i++){
        bool isend=false;
        for(int j=xlen-1;j>=0 && !isend;j--){
            while(true){
                if(!row[i][j])break;
                int f;
                for(int u=31;u>=0;u--){
                    if((row[i][j])&(1<<u)){
                        f=u;break;
```

```
                    }
                }
                f+=32*j;
                if(!is[f]){
                    for(int k=0;k<xlen;k++)
                        xyz[f][k]=row[i][k];
                    is[f]=true;
                    isend=true;
                }
                else{
                    for(int k=0;k<xlen;k++)
                        row[i][k]^=xyz[f][k];
                }
            }
        }
    }
}
```

SSE 向量化范式，后续用作对比，SSE 结合 OpenMP 的时候需要将 k 用作循环变量，进行一些调整，不过总体差异不大。

SSE 向量化

```
//if分支
int k=0;
for(;k<=xlen-4;k+=4) {
    __m128i temp = _mm_loadu_si128((__m128i *) (row[i] + k));
    _mm_storeu_si128((__m128i *) (xyz[f] + k), temp);
}
for(;k<xlen;k++)xyz[f][k]=row[i][k];
//else分支
int k=0;
for(;k<=xlen-4;k+=4) {
    __m128i tx = _mm_loadu_si128((__m128i *) (xyz[f] + k));
    __m128i tr = _mm_loadu_si128((__m128i *) (row[i] + k));
    tr= _mm_xor_si128(tx, tr);
    _mm_storeu_si128((__m128i *) (row[i] + k), tr);
}
for(;k<xlen;k++)row[i][k]^=xyz[f][k];
```

（二） pthread 编程

1. 算法设计

在这一部分，我将对内层函数实现 pthread 多线程优化，通过静态分配线程的方式，完成信号量的建立，线程之间通信以及最后线程的销毁。

2. 编程实现

OpenMP 算法

```
sem_t sem_main;
sem_t sem_workerstart[NUM_THREADS];
sem_t sem_workerend[NUM_THREADS];
struct threadParam_t {
    int t_id;
    int f;
    int* current_row;
};
void* threadFunc(void* param) {
    threadParam_t* p = (threadParam_t*)param;
    int t_id = p->t_id;
    while (true) {
        sem_wait(&sem_workerstart[t_id]);
        if (p->f == -1) break;
        for (int k = t_id; k < xlen; k += NUM_THREADS) {
            if (!is[p->f]) {
                xyz[p->f][k] = p->current_row[k];
            } else {
                p->current_row[k] ^= xyz[p->f][k];
            }
        }
        sem_post(&sem_main);
        sem_wait(&sem_workerend[t_id]);
    }
    pthread_exit(NULL);
}
void stat() {
    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[NUM_THREADS];
    threadParam_t params[NUM_THREADS];
    for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
        params[t_id].t_id = t_id;
        params[t_id].f = -1;
        params[t_id].current_row = nullptr;
        pthread_create(&handles[t_id], NULL, threadFunc, &params[t_id]);
    }
    for (int i = 0; i < nrow; ++i) {
        bool isend = false;
        for (int j = xlen - 1; j >= 0 && !isend; --j) {
            while (true) {
                if (!row[i][j]) break;
                int f = -1;
                for (int u = 31; u >= 0; --u) {
```

```
                    if ((row[i][j]) & (1 << u)) {
                        f = u;
                        break;
                    }
                }
                f += 32 * j;
                if (!is[f]) {
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
                        params[t_id].f = f;
                        params[t_id].current_row = row[i];
                        sem_post(&sem_workerstart[t_id]);
                    }
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_wait(&sem_main);
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_post(&sem_workerend[t_id]);
                    is[f] = true;
                    isend = true;
                } else {
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
                        params[t_id].f = f;
                        params[t_id].current_row = row[i];
                        sem_post(&sem_workerstart[t_id]);
                    }
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_wait(&sem_main);
                    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) {
        params[t_id].f = -1;
        sem_post(&sem_workerstart[t_id]);
    }
    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
        pthread_join(handles[t_id], NULL);
    for (int i = 0; i < NUM_THREADS; ++i) {
        sem_destroy(&sem_workerstart[i]);
        sem_destroy(&sem_workerend[i]);
    }
    sem_destroy(&sem_main);
}
```

（三） OpenMP 编程

1. 算法设计

在这一部分中，由于串行算法的依赖性导致并行化优化没那么容易进行，我尝试了多种在外层循环开辟 openmp 的方法，但最终未能如愿实现，程序均以各种形式的失败告终。然而，我仍然准备了最简单的在内层开辟 openmp 线程的算法，在此部分体现，虽然它可能会造成程序的负优化，但仍然可以带来一些启示。

2. 编程实现

OpenMP 算法

```
void mp(){
    for(int i=0;i<nrow;i++){
        bool isend=false;
        for(int j=xlen-1;j>=0 && !isend;j--){
            while(true){
                if(!row[i][j])break;
                int f;
                for(int u=31;u>=0;u--){
                    if((row[i][j])&(1<<u)){f=u;break;}
                }
                f+=32*j;
                if(!is[f]){
#pragma omp parallel for num_threads(NUM_THREADS)
                    for(int k=0;k<xlen;k++)
                        xyz[f][k]=row[i][k];
                    is[f]=true;
                    isend=true;
                }
                else{
#pragma omp parallel for num_threads(NUM_THREADS)
                    for(int k=0;k<xlen;k++)
                        row[i][k]^=xyz[f][k];
                }
            }
        }
    }
}
```

（四） 性能测试

1. 样例说明

在接下来的小节中将以样例编号代指，样例具体信息如图7所示。

样例编号	1	2	3	4	5	6
矩阵列数	130	254	562	1011	2362	3799
非零消元子数	22	106	170	539	1226	2759
被消元行数	8	53	53	263	453	1953

表 7: 样例数据集说明

2. 测试结果

本部分在 x86 平台进行实验，实验组分别为 common 组（平凡算法），sse 组（SSE 向量化实现），stat 组（pthread 并行编程静态分配线程信号量同步实现），mp 组（OpenMP 并行编程实现），mpsse 组（OpenMP 并行编程 +SSE 向量化实现）。

样例编号	1	2	3	4	5	6
common	<1	<1	<1	5.3	52.6	703.0
sse	<1	<1	<1	3.0	26.6	408.0
stat	3.1	91.5	83.0	2322.9	8995	100986
mp	3.0	108.2	104.3	2814.1	10730	122113
mpsse	3.0	109.0	104.0	2812.8	10747	122063

表 8: x86 平台测试结果（单位：ms）

（五） 总结

我们可以看到我们设计的 pthread 优化算法和 OpenMp 优化算法实际上耗费了更多时间，即使是使用了 SIMD 优化。对于 OpenMP，这个大概是由于动态分配线程的开销很大，而位运算的开销本来就很小。然而，pthread 编程也没有得到较为良好的结果。我认为是位运算计算速度本来就已经很快了，即使是在较大规模的循环下，信号量传输的速度也快于计算的速度。也有可能是我对并行划分还有优化空间，或者我的串行算法不适于并行划分。

同时，注意到 simd 组的优化比相较于上次测算的优化幅度会更大一些，这也说明内存读写与消元过程相比开销是不能够被忽略的，使用本次的计时方法，对于算法优化幅度的测算更为合理。

四、 总结

（一） 实验总结

在本次实验中,我对于普通高斯消元算法和特殊高斯消元算法尝试了 pthread 编程和 OpenMp 编程，通过不同的优化方式进行实验，同时与 SIMD 编程进行结合。通过性能分析和 Vtune Profiling 更好的体会了多线程编程的内在意义，也意识到了充分利用电脑线程并行的开展实验对实验的优化效率之大。

通过并行 OpenMp 编程和 SIMD 编程结合起来，最高取得了将近 12 倍的优化比（相较于平凡算法），这也让我体会到对于相同的程序来说，并行的编程为我们的生活帮助之大。通过在 x86 平台和 ARM 平台上开展实验，我也体会到了不同平台上相同程序运行时间，优化程度的差

别之大。比如说同样是 8 核设备，x86 平台 8 线程比 4 线程仅仅好一点，但 ARM 平台就会优化特别大。因此在设计程序的时候或许也需要考虑到硬件设备的情况。

（二） GitHub 仓库链接

相关完整代码和图片可在GitHub 仓库中查看。