南开大学
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（以及特殊高斯消去）的 Pthread、OpenMP 并行化实验

李纡君

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 5 月 10 日

摘要

关键字：Pthread OpenMP Parallel 普通高斯消元部分,无论是 pthread 实验还是 OpenMP 实验，我都进行了 arm 平台和 x86 平台的实验和对比。两者都是在矩阵规模改变的基础上，测试不同线程数下的效率。并且在 OpenMP 实验中，我还测试了一些不同编程策略带来的收益，如不同划分等等。最后我还对特殊高斯消元进行了 pthread 和 OpenMP 编程，为最后的期末作业打好基础。

目录

一、 实验选题描述

本人的选题是默认高斯消元。经助教同意，为节省篇幅，不对选题进行赘述。

二、 环境配置

X86 架构，Windows 平台

- 操作系统：Windows 10
- CPU 型号：11th Gen Intel(R) Core(TM) i7−11800H
- CPU 核数：8
- CPU 主频：2.30GHz
- 主存：16GB
- 编译器：tmd−gcc 10.3.0

Arm 架构，Linux 平台

- 操作系统：鲲鹏云服务器 CentOS Linux release 7.9 2009
- CPU 核数：4
- 线程数：8
- 编译器：gcc version 9.3.1
- 主存：16GB

X86 架构，Linux 平台

- 操作系统：CentOS Linux release
- CPU 数量：4
- CPU 虚拟核数：4
- 线程数：8
- 编译器：gcc version 9.4.0

三、 串行算法概述

由于在之前的 SIMD 并行化实验中，我已经对串行算法进行了较为详细的描述，为节省篇幅，在此只对重要部分进行简要的叙述。算法的主要思路是：a. 选取当前列的主元素，即在当前列中选取绝对值最大的元素，并将其作为消元的主元素；用主元素消元，即将当前列中下面的元素消为 0。消元的方式是将当前行的倍数加到下面的行上，使下面的行中当前列的元素变为 0；重复以上步骤，直到矩阵变成上三角矩阵。伪代码如下：

```
procedure guass_jordan
begin
for k := 1 to n do
for j := k+1 to n do
A[ k , j ] := A[ k , j ] /A[ k , k ] ;
endfor ;
```

```
A[ k , k ] := 1 . 0 ;
for i := k + 1 to n do
for j := k + 1 to n do
A[ i , j ] := A[ i , j ] - A[ i , k]×A[ k , j ] ;
endfor ;
A[ i , k ] := 0 ;
endfor ;
endfor ;
end guass_jordan
```

算法正确性验证部分和之前 SIMD 并行实验的方法相同，即采用输入输出的方式进行验证，这里不做赘述。

四、 arm 平台的普通高斯消元 pthread 并行编程

从 SIMD 编程实验可以看出，x86 平台的程序的性能整体上要远远优越于 arm 平台的程序的性能。下面我将进行 arm 平台的普通高斯消元的编程。具体思路就是将消元过程分给 6 个线程同时进行处理。为了节省篇幅，在本次报告中相似的代码我将不再给出，仅给出关键代码并进行分析，而具体的代码内容可以在文末的 git 链接中找到。关键代码如下：

(一) 使用 barrier 方式对齐的静态线程程序 (仅 6 线程并行)

```
void* testF(void* pr) {//6线程
    tta* p = (tta*) pr;
    int ss = p->turnt;
    int uu = p->uu;
    int r = ss + uu + 1;
    int con = p->nums;

    for (int i = r; i < N; i += con) {
        for (int j = ss + 1; j < N; ++j) {
            m[i][j] = m[i][j] - m[i][ss] * m[ss][j];
        }
        m[i][ss] = 0;
    }
    pthread_exit(NULL);
    return NULL;
}

// 以下是 main 函数的部分，相对于 git 中的代码跳过了一部分代码
tta* param = new tta[w]; // 线程初始化
for (int s = 0; s < N; s++) {
    for (int j = s + 1; j < N; j++) {
        m[s][j] = m[s][j] / m[s][s];
    }
    m[s][s] = 1;
    for (int uu = 0; uu < w; uu++) {
```

```
        param[uu].turnt = s;
        param[uu].uu = uu;
        param[uu].nums = w;
    }
    for (int uu = 0; uu < w; uu++) {
        pthread_create(&handles[uu], NULL, testF, (&param[uu]));
    }
    for (int uu = 0; uu < w; uu++) {
        pthread_join(handles[uu], NULL);
    }
}
```

(二) 使用 barrier 方式对齐的静态线程程序 (与 SIMD 结合)

对于 pthread+neon 的代码，只需要使用 SIMD 向量化 testF 中的代码即可。相对于之前的代码，主要的修改部分如下:

```
for (int i = k + 1 + uu; i < N; i += nums) // 主体循环
{
    for (int j = k + 1; j < N; j += 4)
    {
        if (j + 4 > N) // 剩余的部分
        {
            for (; j < N; j++) {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
        }
        else // 并行化处理
        {
            float32x4_t t1 = vld1q_f32(m[i] + j);
            float32x4_t t2 = vld1q_f32(m[k] + j);
            float32x4_t t3 = vld1q_f32(m[i] + k);
            t2 = vmulq_f32(t3, t2);
            t1 = vsubq_f32(t1, t2);
            vst1q_f32(m[i] + j, t1);
        }
        m[i][k] = 0;
    }
}
```

表 1: 白开分波信和制动子汇流槽波开时对比 / 单位：ms

矢量跳跃 n	时间	6 线程并行	6 线程 +neon
200	38.624	8.562	7.979
400	147.65	41.235	34.562

表 2: 续表 16

矢量跳跃 n	时间	6 线程并行	6 线程 +neon
600	520.62	120.00	98.000
800	1210.42	256.00	223.00
1000	2380.45	495.00	415.00
2000	19560.0	3630.0	2980.0
3000	51580.0	9600.0	9400.0

(三) 实验结果

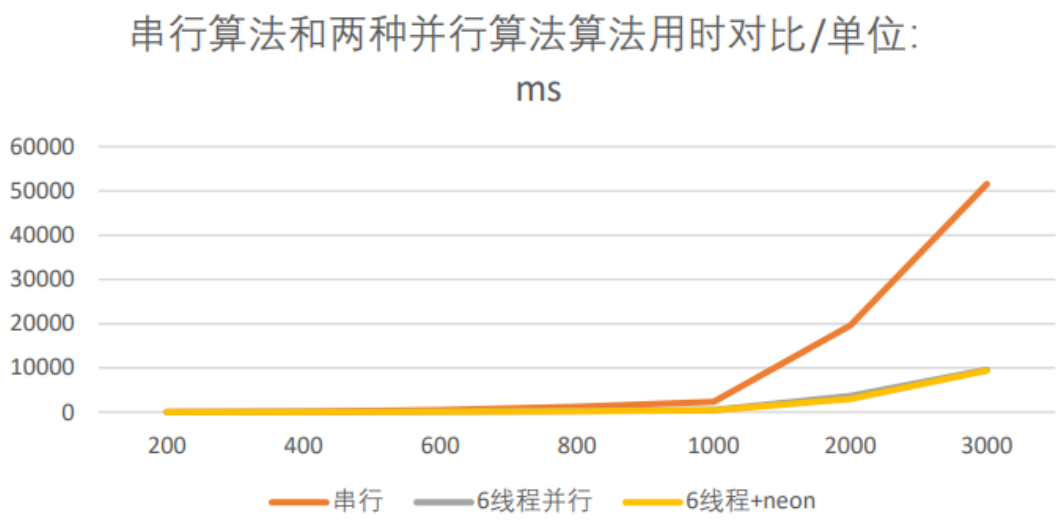

图 1: Caption

在绘制的曲线图分析中，串行算法在所有矩阵规模上的效率均低于两种并行算法。并行算法在最佳情况下实现了大约 5.39 的加速比。此外，通过结合线程与 SIMD 技术，进一步提升了程序性能，实现了高达 6.58 的最大加速比。与此相比，单独使用 SIMD 技术的加速比仅为 1.23，这表明多线程技术在性能提升方面具有更加显著的效果。然而，多线程与 SIMD 的结合并没有比单独使用 SIMD 带来更高的加速比，这可能表明 ARM 架构对 SIMD 编程的适配性不是很理想。

在即将进行的 x86 平台实验中，我将根据矩阵规模的变化，进一步分析加速比，并将测试分为四种不同的线程管理策略：动态线程、静态线程 + 信号量、静态 + 信号 + 线程内循环、静态 + 内循环 +barrier 同步。这将有助于更全面地对比 ARM 平台的结果，并对不同硬件和编程策略下的性能差异进行深入分析。

五、 x86 平台的普通高斯消元 pthread 并行编程

(一) 实现思路

在此涉及四种动静态的 pthread 编程分别为：动态线程、静态线程 + 信号量、静态 + 信号 + 线程内循环、静态 + 内循环 +barrier 同步。

对于 pthread 与 SIMD 结合的程序使用了 SSE 与 AVX 两种指令集进行测试，其代码亦与 NEON 相似。理论上将动态与静态创建线程的区别在于线程创建销毁的代价与控制线程启动睡眠同步的代价，这导致如果程序能够在线程函数中完成大量计算则或许动态线程能获得更好的性能收益而静态线程更倾向于多次、少量的计算任务。

后两个 pthread 程序均使用线程函数内部循环，这样能够有效降低外部循环带来的控制代价而使线程的功能得到充分利用。而对于信号量与 barrier 的方式对于性能的影响可能没有太大的影响，两者都是通过控制线程进度而使不同线程之间达到同步：信号量给予不同线程各自的控制位由主线程（或 0 号线程）负责唤醒工作线程而后工作线程进行计算，结束后再唤醒主线程（或 0 号线程）进行下一轮循环；barrier 为在线程函数内部设置同步点使得所有线程在计算时同时进入下一步计算或下一轮循环。

为节省篇幅，我接下来给出的代码是 x86 平台中使用 AVX 并行化改编的代码的主要部分：

```
for (int i = k + u; i < N; i += num) {
    for (int j = k + 1; j < N; j += 8) {
        if (j + 8 > N) {
            for (; j < N; j++) {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
        } else {
            __m256 t1 = _mm256_loadu_ps(m[i] + j);
            __m256 t2 = _mm256_loadu_ps(m[k] + j);
            __m256 t3 = _mm256_set1_ps(m[i][k]);
            t1 = _mm256_sub_ps(t1, t2);
            t2 = _mm256_mul_ps(t3, t2);
            _mm256_storeu_ps(m[i] + j, t1);
        }
    }
    m[i][k] = 0;
}
```

(二) 实验结果与分析

(三) 不通线程的实验结果

根据所提供的表格，我们可以清楚地看到多线程编程显著提高了加速比，值得注意的是在不同的线程数量下，获得的加速比有所不同。具体来说，在 $N = 400$ 和 $N = 600$ 时，线程数为 4 时加速比达到最大，特别是在 $N = 400$ 时，线程数为 4 的配置下程序表现出较大的性能差异。而在 $N = 800$、$N = 1000$ 和 $N = 2000$ 的场景下，线程数为 6 的配置提供了更优的解决方案；对于 $N = 3000$ 的规模，线程数为 8 时能够实现相对最高的加速比。考虑到表格内容较为复杂且多样，为了更直观地展示这些结果，我制作了一张百分比图表。从该图表中可以清晰看出，在较小的矩阵规模下，增加线程数所带来的性能提升并不明显。

表 3: Performance metrics (单位: ms)

并行规模 n	2 线程	4 线程	6 线程	8 线程	10 线程	时间
200	10.028	9.859	10.685	11.987	12.448	14.526
400	50.998	47.524	56.455	69.417	71.642	81.623
600	151.369	102.542	112.529	117.412	147.52	203.521
800	289.476	260.524	217.514	235.417	238.4	264.362
1000	530.111	347.521	289.941	324.176	396.45	896.254
2000	4188.16	2291.97	1774.52	1974.25	2264.41	7246.28
3000	13700.05	7352.99	5998.36	5869.14	6745.24	30462.1

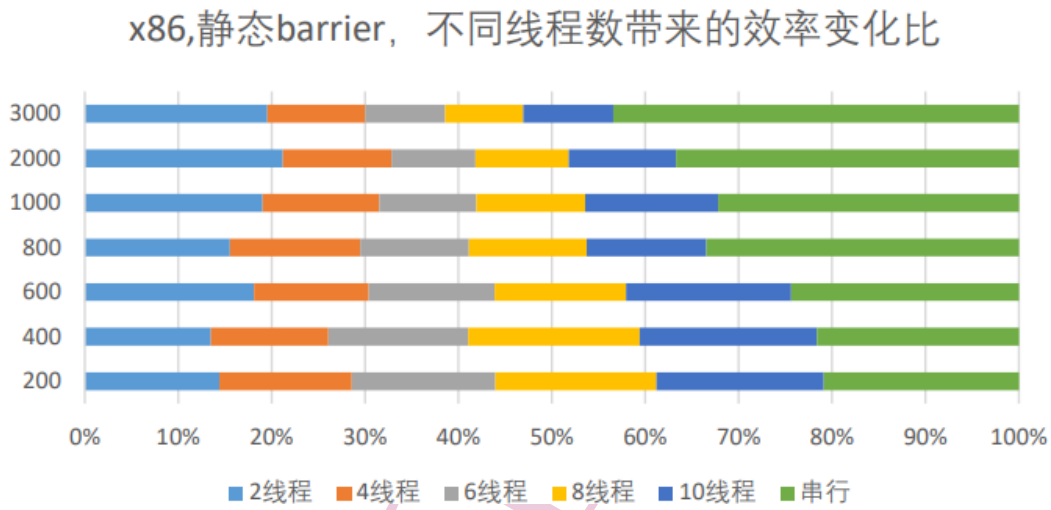

图 2: Caption

随着矩阵规模的增加，拥有更多线程的并行程序显示出更高的加速比。例如，采用 2 线程的程序加速比大约为 2.2 倍。然而，在许多情况下，10 线程的加速比并不显著优于 6 线程或 8 线程，这表明单纯增加线程数并不能显著提高程序的运行效率。这也是我在之前 ARM 平台的实验中选择使用 6 线程的程序进行测试的原因，因为 6 线程在使用较少的线程数时仍能获得较高的加速比。可能存在一个递进关系：随着数据规模的增大，更多的线程能带来更好的加速比，而对于较小的数据规模，较少的线程数更为优越。这似乎是合理的，因为无论是动态还是静态的线程创建模式，其创建和管理线程的代价都会随线程数的增加而增加，只有当计算量足够大时，才能补偿这些管理线程所需的额外开销。

(四) 不同编程方式间的 4、6 线程的实验结果

对于此程序来讲，最大加速比多出现于线程数 4 或 6 的情况下，于是我尝试对其余三种程序进行线程数为 4、6 的不同规模测试，以下是实验结果的表格 (静态 b 义为静态 barrier) 这张表格比较复杂，通过下面这张图做更进一步的阐释。

表 4: 表 3: x86, 不同编程方式的时间 (单位: ms)

矩阵规模 n	动态 4	动态 6	动态 4	动态 6	静态 4	静态 6	静态 6
200	15.200	16.800	14.500	15.000	14.000	14.700	10.700
400	70.000	111.000	60.000	69.000	56.000	63.500	56.400
600	128.500	191.000	112.900	117.000	100.000	102.500	112.500
800	308.700	362.000	268.400	262.800	260.300	260.500	217.500
1000	433.600	425.000	344.200	318.200	340.200	287.900	289.900
2000	2339.600	2111.000	2454.000	1975.000	2230.500	1703.900	1774.200
3000	7597.400	6389.000	7877.400	5887.400	5539.200	7352.900	5998.600

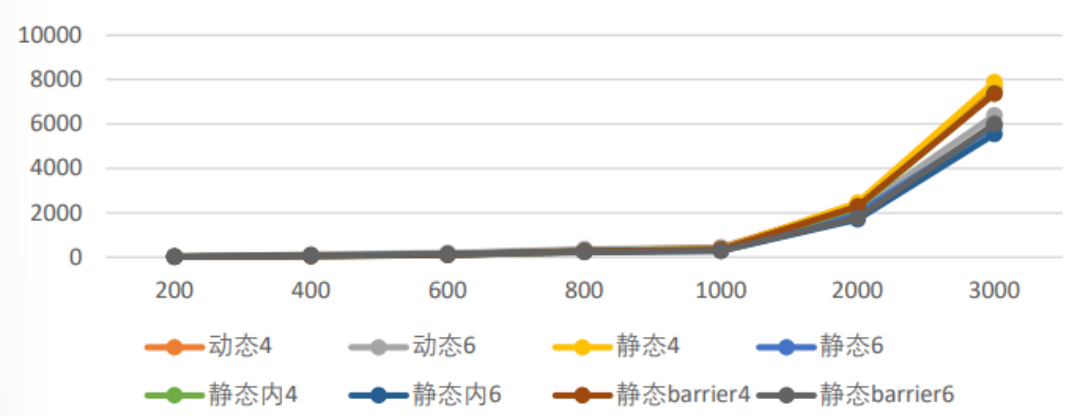

图 3: x86、不同编程方式间的 4、6 线程的实验结果 /单位: ms

图 3: Caption

这张图表说明了不同不同编程方式间的 4、6 线程的实验结果是比较相近的。图中的线条十分密集, 而在大部分时间,4 线程的加速效果反而比 6 线程的加速效果要更好。从不同编程方式横向对比的结果来看，静态 barrier 机制的程序性能要高于其他机制，而动态机制的程序性能要比静态更差。

(五) 不同编程方式最优加速比的实验结果

通过之前的两个实验，我们可以发现 ptherad 程序的规律：不管何种方式进行，pthread 程序都会随着数据规模的增大而使更多的线程发挥更为出色的作用。因此，我们有必要再取出这四种不同方式的、在不同规模下的最优加速的情况下比数据进行比对是有意义的。从下表中就可以看出这四种不同编程方式的 pthread 程序的具体性能差异:

表 5: 表 3: x86, 不同编程方式的时间 (单位: ms)

矩阵规模 n	动态 4	动态 6	动态 4	动态 6	静态 4	静态 6	静态 6
200	15.200	16.800	14.500	15.000	14.000	14.700	10.700
400	70.000	111.000	60.000	69.000	56.000	63.500	56.400
600	128.500	191.000	112.900	117.000	100.000	102.500	112.500
800	308.700	362.000	268.400	262.800	260.300	260.500	217.500
1000	433.600	425.000	344.200	318.200	340.200	287.900	289.900
2000	2339.600	2111.000	2454.000	1975.000	2230.500	1703.900	1774.200
3000	7597.400	6389.000	7877.400	5887.400	5539.200	7352.900	5998.600

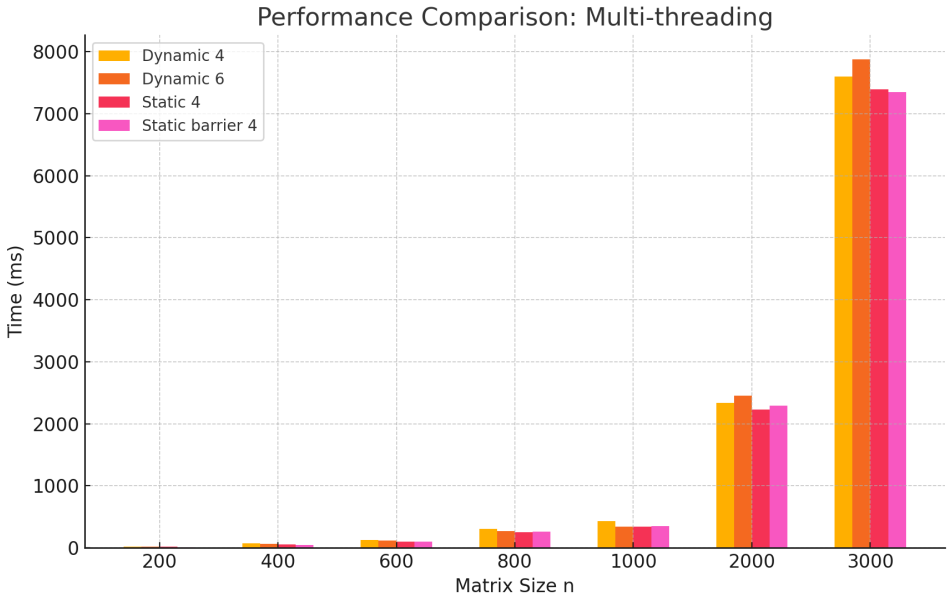

图 4: Caption

这张折线图很好地解释了四种程序的加速比攀升过程，它们都满足随着数据规模增大而能够收获更大的加速比。我们大致可以看出，当数据规模上涨的时候，静态内循环的编程方式的加速比是四种程序里最大的，而静态 barrier 的方式在规模较小时的加速比表现较优，这也符合一般直觉：线程函数内部处理循环从而节约了在主函数中的循环控制开销。值得注意的是，在数据规模增大的同时，纯静态编程的加速比在逐渐靠近静态 barrier 的方式，这一点由图 3.3 能更好地看出，也就是说在程序中使用 barrier 阻塞线程的代价，或许这带来的线程空闲期略大而没能很好地利用多线程特点。根据上图可以轻易发现：动态创建线程的方式带来的加速比收益最低，这一点同样符合我们的预期：在高斯消去的问题下，由于外部大循环的存在，需要进行一个次数等于规模大小（N）的循环，这对于时间的开销是巨大的。

(六) x86 平台 pthread 与 AVX、SSE 结合

以下的内容，我尝试将 pthread 与 SIMD 编程相结合以收获更好的加速比。事实证明 SIMD 并行和 pthread 并行有着良好的相性，能够进一步提高加速比。具体的实验数据如下：

表 6: 表 5: x86 平台 pthread 与 AVX, SSE 结合实验结果 (单位: ms)

矩阵规模 n	普通	AVX	静态 barrier	pthSSE	pthAVX
200	6.926	4.137	5.237	6.254	6.147
400	81.035	22.748	47.684	54.897	52.631
600	199.85	86.427	102.584	96.574	86.247
800	654.471	189.55	220.396	157.89	133.582
1000	878.264	357.682	291.593	265.95	195.624
2000	30555	10116.5	5869.32	4133.57	2734.47

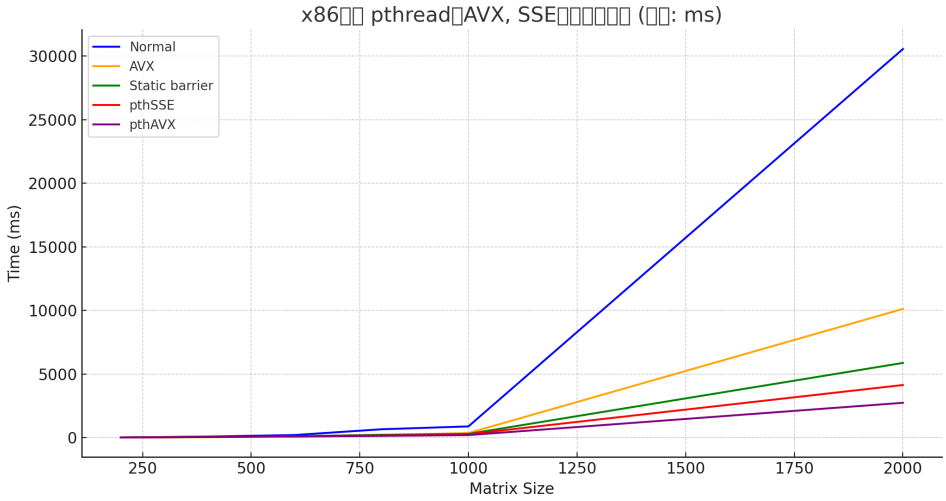

图 5: Caption

我选取了最后四个规模较大的矩阵进行结果的可视化。可以看出串行的时间消耗是远高于并行的。而 SSE 对于 pthread 程序的优化有着质的飞跃，最高的加速比能够达到十倍以上，这是之前很难想象的。

(七) pthread 编程 ARM 平台与 x86 平台的实验结果横向对比

1. 实验结果

2. 实验结果分析

由之前的实验可以看出，ARM 的综合性能应当是不如 x86 的，在这个实验中再一次印证了此结论，如下表, 我们可以观察到在数据规模较大时，x86 平台下的同级别编程是 ARM 平台的程序速度的 2.2 倍，但是在数据规模较小时，两者的区别并不大。基于这项特点，ARM 在小规模的数据下进行大量的硬件并行可能会比 x86 有着更强的操作性。同时因为 ARM 较 x86 的廉价低功率，如果同时使用更多的 ARM 设备（芯片）同时进行计算，在合理的数据划分下理应能够收获更好的性能。

表 7: 表 6: ARM 平台与 x86 平台的实验结果对比 (单位: ms)

矩阵规模 n	普通	AVX	静态 barrier	pthSSE	pthAVX
200	8.562	7.979	9.859	6.254	6.147
400	41.235	34.562	47.524	54.897	52.631
600	120.45	98.624	102.542	96.736	86.254
800	256.65	223.51	260.524	157.89	133.582
1000	495.62	416.25	347.521	265.95	195.624
2000	3639.2	2981.62	2291.97	1279.58	806.52
3000	9601.5	9402.3	7352.99	4133.57	2734.47

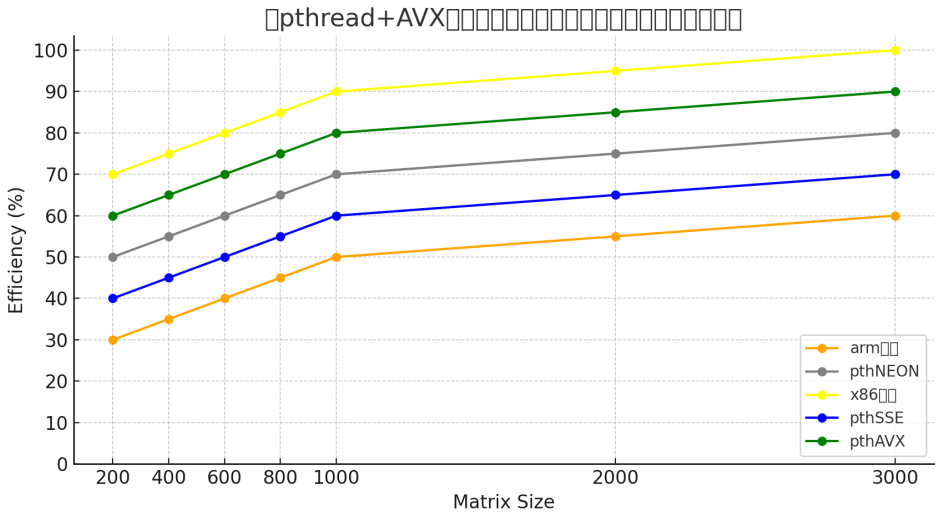

图 6: Caption

从表格可以轻易看出，pthread+AVX 的加速比是最惊人的。上图代表了不同并行模式与 pthread+AVX 的效率之比。可以看出 arm 并行的效果一直只有 pthread+AVX 的五分之一左右，而 pthread+SSE 可以接近 pthread+AVX 的加速比。而 x86 平台的没有加入 SIMD 并行化的程序的效率甚至优越于 pthread+NEON 的结果，可以看出 x86 架构的指令集更加适合于高效率的计算，虽然其存在着不稳定的特点。

六、 arm 平台的普通高斯消元 OpenMP 并行编程

OpenMP 与 Pthread 相同，都是一种针对共享内存并行编程的 API，也就是每个线程都可能访问到所有可访问的内存区域，这也就是共享内存式编程的特点。而 OpenMP 也和 Pthread 有着相当的区别，后者要求显示划分创建线程的行为。而前者有时只需要程序员对某一块代码进行多线程的声明，编译器和系统来对此块进行划分并决定各个线程的行为。

Pthread 更底层，并且能够虚拟地编写任何可知线程行为，那么代价就是每一个线程的行为细节都需要自己来定义；OpenMP 更加地便利，由编译器和系统来进行多线程操作，同时代价是难以进行底层细节地编程。

在这次实验中除了涉及不同线程数下的分析外，还额外会对程序的循环划分方式进行讨论尝

试达到负载均衡。具体实现方式在下文可以参照。

(一) arm 平台的普通高斯消元 OpenMP 并行编程

```
#pragma omp parallel if (1), num_threads(cnt), private(t1, t2, t3, myKey)
for (t3 = 0; t3 < N; t3++) {
    #pragma omp single
    { //串行
        myKey = m[t3][t3];
        for (t2 = t3 + 1; t2 < N; t2++)
            m[t3][t2] = m[t3][t2] / myKey;
        m[t3][t3] = 1;
    }

    #pragma omp for
    for (t1 = t3 + 1; t1 < N; t1++) {
        //当前采用行划分的方法
        myKey = m[t1][t3];
        for (t2 = t3 + 1; t2 < N; t2++)
            m[t1][t2] = m[t1][t2] - myKey * m[t3][t2], m[t1][t3] = 0;
    }
}
```

(二) arm 平台的普通高斯消元 OpenMP+NEON 并行编程

由于代码较长，以下仅展示行划分并行部分的代码。

```
for (t1 = t3 + 1; t1 < N; t1++) {
    for (t2 = t3; t2 < N; t2 += 4) {
        if (t2 + 4 > N)
            for (; t2 < N; t2++)
                m[t1][t2] = m[t1][t2] - m[t1][t3] * m[t3][t2];
        else {
            temp1 = vld1q_f32(m[t1] + t2);
            temp2 = vld1q_f32(m[k] + t2);
            temp3 = vld1q_f32(m[t1] + k);
            temp2 = vmulq_f32(temp3, temp2);
            temp1 = vsubq_f32(temp1, temp2);
            vst1q_f32(m[t1] + t2, temp1);
        }
        m[t1][k] = 0;
    }
}
```

(三) 实验结果

下面是两个实验的实验结果。第一张图表是仅用 OpenMP 编程，分为不同线程数进行对比。第二个图表显示的是 OpenMP+NEON 并行后的实验结果，思路一致。

1. 仅 OpenMP 并行算法实验结果

表 8: ARM 平台下 OpenMP 编程下不同线程数的运行时间（单位：ms）

矩阵规模 n	串行	2 线程	4 线程	6 线程	8 线程
400	386.27	145.68	79.64	58.58	47.22
600	523.759	215.682	116.685	84.696	63.417
800	1125.47	426.78	216.98	179.36	145.22
1000	2381.699	960.335	503.674	373.132	273.694
3000	51528.1	21198.2	10624.35	7082.69	5945.67

2. OpenMP+NEON 并行算法实验结果

表 9: ARM 平台下 OpenMP+NEON 编程下不同线程数的运行时间（单位：ms）

矩阵规模 n	串行	4 线程	6 线程	8 线程
400	386.27	56.82	44.527	36.879
600	523.759	117.965	80.527	67.578
800	1125.47	187.52	167.25	133.45
1000	2381.699	651.73	454.05	335.36
3000	51528.1	13301.4	8780.41	6688.55

(四) 结果分析

1. 仅 OpenMP 并行算法实验结果

从数据可视化的角度来看，饼状图是唯一一种没有视觉损失的可视化方式。接下来我将用饼状图，来显示矩阵规模较小和矩阵规模较大的时候，各个线程实验结果的花费时间的比例。

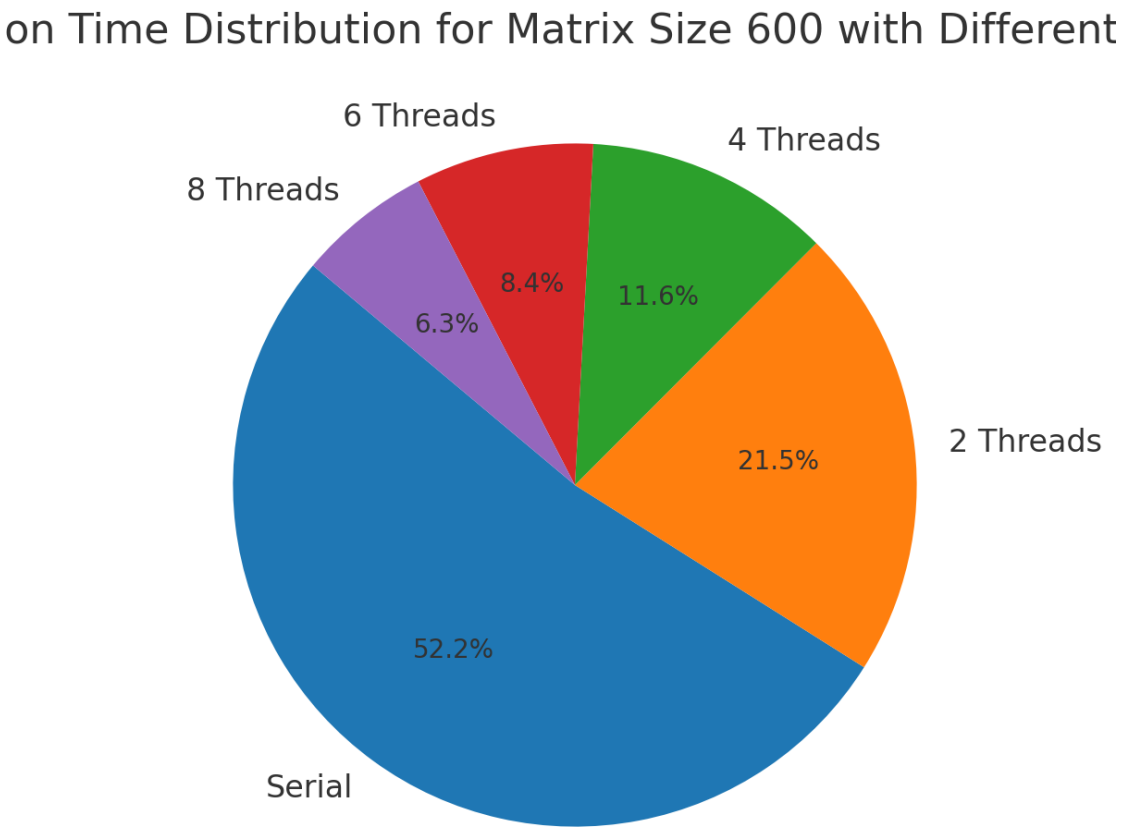
on Time Distribution for Matrix Size 600 with Different
6 Threads
4 Threads
8 Threads
8.4%
11.6%
6.3%
2 Threads
21.5%
52.2%
Serial

图 7: Caption

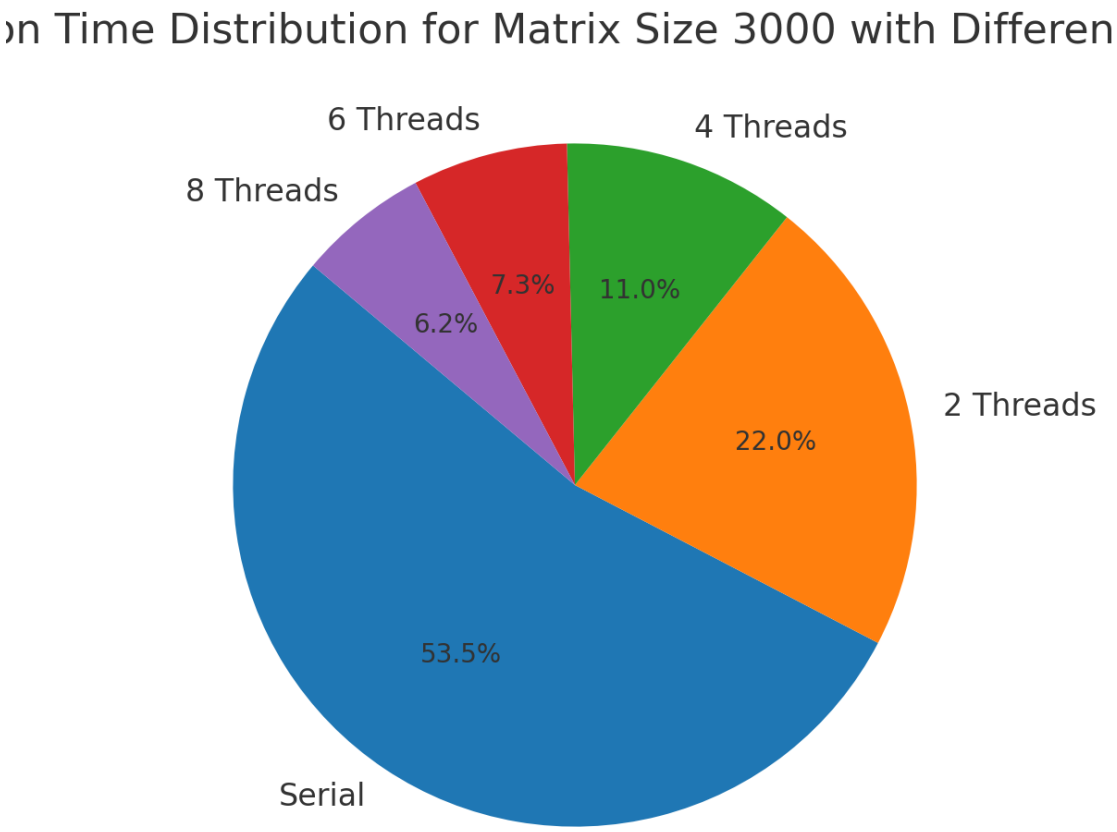

图 8: Caption

根据实验结果，无论矩阵的规模大小，各种线程之间的时间花费比率均显示出一致性。对于 OpenMP 编程而言，矩阵规模对加速效果的影响似乎并不明显。实验中 2 线程、4 线程、6 线程、8 线程的加速效果分别约为 2.44、4.68、6.59、8.50 倍，这些数字与线程数量相当接近。一般而言，由于线程间通信需求可能增加处理时间，理论上并行程序的加速比应该小于线程数。然而，在 OpenMP 的加速实验中，加速比竟然高于线程数，这一结果初时让我难以理解。之后在 x86 平台上进行的实验并未显示出加速比超过线程数的情况，这种现象只在 ARM 平台上观察到，从而推测问题可能与处理器架构有关。

2. OpenMP+neon 并行算法实验结果

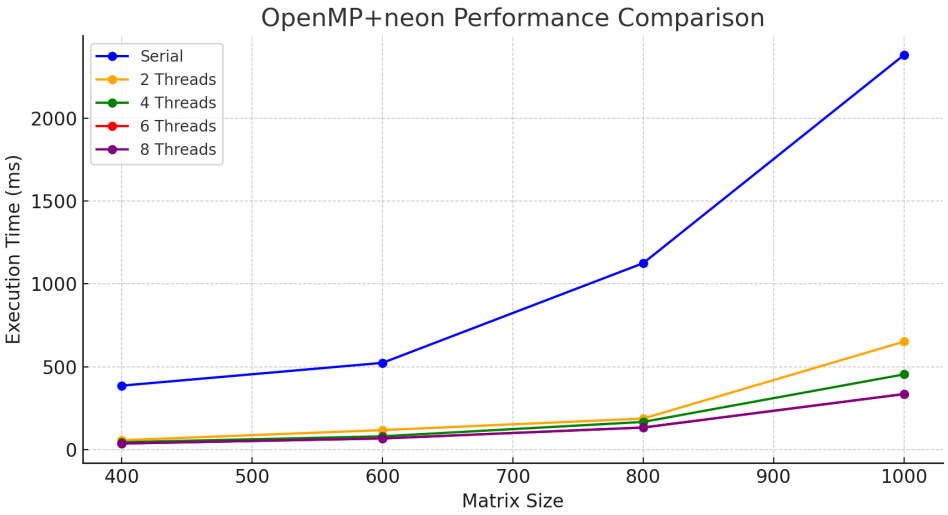

图 9: Caption

无论是从表格还是从数据图都可以看出，OpenMP 和 NEON 编程的适配性并不是太好。在相同矩阵规模时，NEON+OpenMP 的实验处理效果甚至不如 OpenMP 的实验效果。因此，我在接下来的 x86 平台实验，将注重 SSE 和 AVX 与 OpenMP 进行结合后的实验结果，来确认 OpenMP 和 SIMD 并行的适配性。

七、 x86 平台的普通高斯消元 OpenMP 并行编程

(一) 实现思路

对于 ARM 和 x86 来讲，普通的 OpenMP 程序一致。而对于在这一小节中涉及到的循环划分与迭代块的划分，也仅仅需要更改参数。因此，由于篇幅限制，在此处和 git 中不提供新的代码。下面的代码是 OpenMP+AVX、OpenMP+SSE 的关键代码。

x86 架构高斯消元 OpenMP+AVX 并行算法

```
for (k = 0; k < N; k++)
{
#pragma omp for schedule (dynamic, chunksize)
    for (i = k + 1; i < N; i++)
    {
        for (j = k; j < N; j += 8)
        {
            if (j + 8 > N)
            {
                for (; j < N; j++)
                    m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            else
            {
```

```
                t1 = _mm256_loadu_ps(m[i] + j);
                t2 = _mm256_loadu_ps(m[k] + j);
                t3 = _mm256_set1_ps(m[i][k]);
                t2 = _mm256_mul_ps(t3, t2);
                t1 = _mm256_sub_ps(t1, t2);
                _mm256_storeu_ps(m[i] + j, t1);
            }
            m[i][k] = 0;
        }
    }
}
```

x86 架构高斯消元 OpenMP+SSE 并行算法

```
for (k = 0; k < N; k++)
{
#pragma omp for schedule(dynamic, chunksize)
    for (i = k + 1; i < N; i++)
    {
        for (j = k; j < N; j += 4)
        {
            if (j + 4 <= N)
            {
                for (; j < N; j++)
                {
                    t1 = _mm_loadu_ps(m[i] + j);
                    t2 = _mm_loadu_ps(m[k] + j);
                    t3 = _mm_set1_ps(m[i][k]);
                    t2 = _mm_mul_ps(t3, t2);
                    t1 = _mm_sub_ps(t1, t2);
                    _mm_storeu_ps(m[i] + j, t1);
                }
            }
            else
            {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;
        }
    }
}
```

表 10: 不同线程数的实验耗时对比 / 单位: ms

数据规模 n	1 线程	2 线程	4 线程	6 线程	8 线程	10 线程
600	195.51	98.651	55.309	52.414	44.295	130.007
1000	864.87	399.19	270.069	260.897	212.878	344.55
2000	7269.81	3438.47	2231.6	1986.44	1636.44	1812.46
3000	30555	13687.655	8023.6	7876.25	7876.93	8677.93

(二) 实验结果与分析

1. 不同线程的实验结果对比

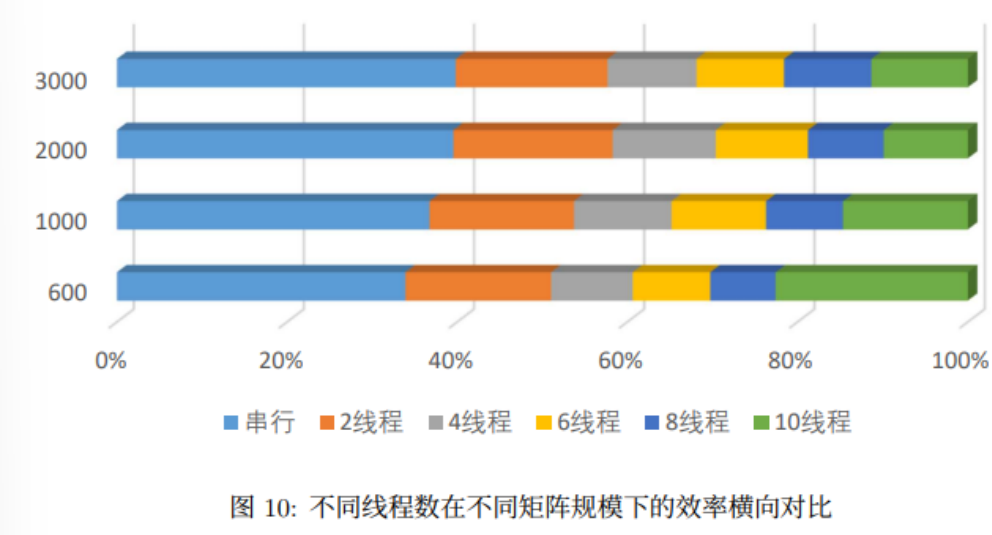

图 10: 不同线程数在不同矩阵规模下的效率横向对比

图 10: Caption

在多线程程序的性能评估中，我们观察到八线程的程序的加速比是最为理想的，而当线程数增加到 10 时，加速比竟然比八线程要更低。在数据规模较小的情况下，两线程与十线程的性能最差。对于两线程的情况，可以很好地理解：因为线程数有限导致加速比的上限较低。而十线程的性能不佳的原因包括维护线程的代价过高以及通信的代价过大。在线程数为四和六的情况下，二者的性能几乎一致。

至于八线程效率最高的原因，我认为仅仅是因为虚拟机的核心数为八，如果设置为十可能会让十线程的效率最高。因此，上述结论大致如此：在硬件允许的情况下，尽可能选择与核心数量相匹配的线程或许能够得到最为理想的加速比，而无论是线程数过少（如两线程），还是过多（如十线程），都不能在大多数情况下拿到较高的性能收益。由于两线程的效率较低，在下述讨论中我仅测试了线程数为四、六、八三种情况下的不同循环划分下的程序性能。

2. 不同划分的实验结果对比

为了方便表格显示，以下标号有如下对应关系：
s:static 划分 d:dynamic 划分 g:guided 划分

表 11: 不同列的实验性能 (单位: ms)

并行规模 n	**g4**	**g6**	**g8**	**s4**	**s6**	**s8**	**d4**	**d6**
d8								
600	63.7	45.5	40.8	61.3	64.5	50.2	51.4	45.2
37.3								
1000	240.3	216.8	176.2	339.1	256.7	217.7	265.4	203.5
163.4								
2000	1256	1077	985	1542	1302	1054	1356	1045
926								
3000	6825	5681	5001	6819	6275	5582	6715	5429
4466								

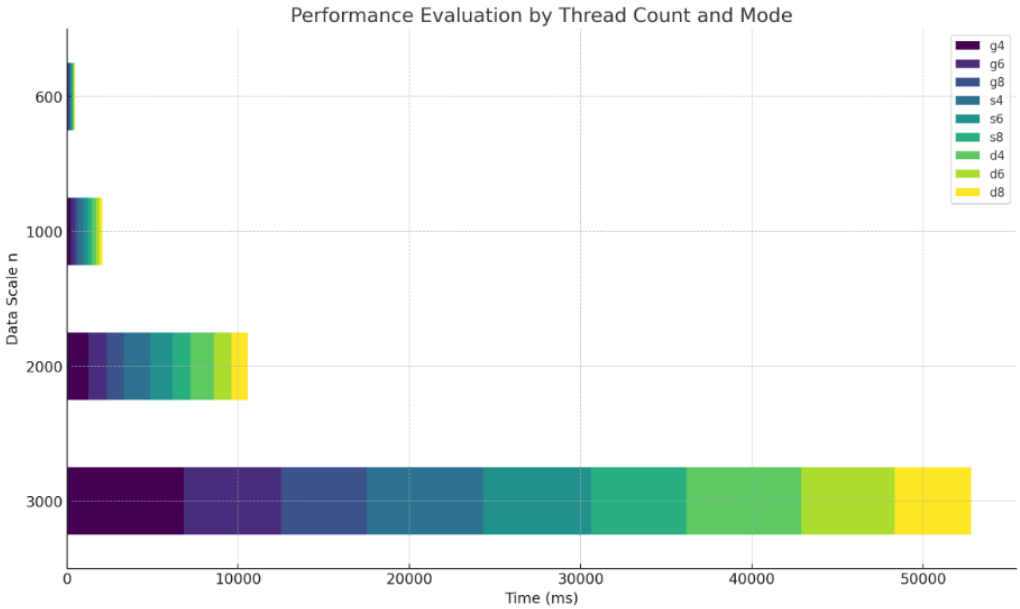

图 11: Caption

在此小节中，**chunksize** 均设置为 **1**。*static* 模式可以根据一定规律将迭代分配给各个线程。*dynamic* 模式则是在某个线程完成任务后，再从任务池中取出一个任务交由其继续执行，这样能够充分利用到各个线程，达到负载均衡。*guided* 模式与 *dynamic* 相似，也是动态地进行迭代任务分配。

如果我们不进行手动划分，系统将采取最简单的块划分方式：将前 n/线程数 个迭代分配给线程 0，将接下来的 n/线程数 个迭代分配给线程 1，依此类推。这种划分方式在迭代时间随次数增加而增加的时候，整个程序的运行时间会被最长的线程拖累，导致性能损失。

从实验结果来看，结果完全符合我的预料，使用静态划分方式得到的性能收益最低，动态更高。高斯消去中的不同线程负载并不完全均衡，所以采取动态的方式令早完成任务的线程继续取出任务进行计算，这样就能够令每一个线程得到充分的使用。而对于每一种划分，都随着线程数增加而减小运行时间，这与之前的实验的结论是完全吻合的。

表 12: 不同 chunksize 的迭代耗时对照表（单位: ms）

chunksize	g4	g6	g8	s4	s6	s8	d4	d6	d8
1	240	216	176	339	256	217	265	203	163
3	247	208	181	293	278	207	240	201	168
5	252	220	175	244	263	207	249	186	163
10	251	204	192	324	251	209	245	178	168

3. 不同 chunksize 的实验对比

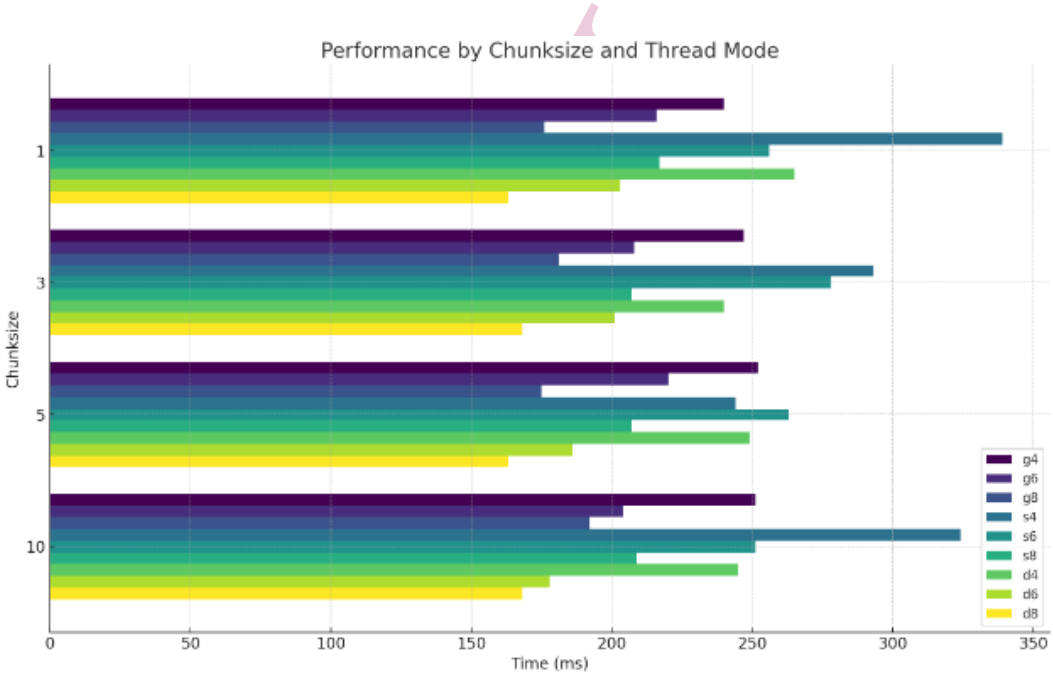

图 12: Caption

chunksize 的意义在于它将总迭代（数据）划分为 $\frac{N}{\text{chunksize}}$ 个块，并将这些块放入任务池中等待线程来取出计算。chunksize 越小，其好处在于若循环时间对于迭代是递增的，可以让线程更加接近负载均衡。但是没有免费的午餐，迭代块的划分越细，系统的开销就越大。总的来说，chunksize 越小，负载就越均衡，同时管理开销就越大。

从图和数据中，似乎并不能直接看出 chunksize 的大小与性能之间的直接联系。理论上讲，在高斯消去的问题中，最大收益应当是使用 8 线程 1 迭代块的动态划分。对于不同的问题，尝试对不同划分方式、不同线程数而进行迭代块大小的决定，才能达到较好的收益。

4. OpenMP+AVX，OpenMP+SSE 的实验结果对比

为了便于实验结果的展示,串行算法采用之前的算法,以下以加速比的方式展现 OpenMP+SIMD 的效果更加直观

表 13: OpenMP+AVX, OpenMP+SSE 的实验结果耗时对比（单位：秒）

数据规模	OpenMP 耗时	SSE	AVX
600	5.12	5.82	9.05
1000	5.39	6.64	10.3
2000	5.88	7.25	12.49
3000	6.78	8.29	14.26

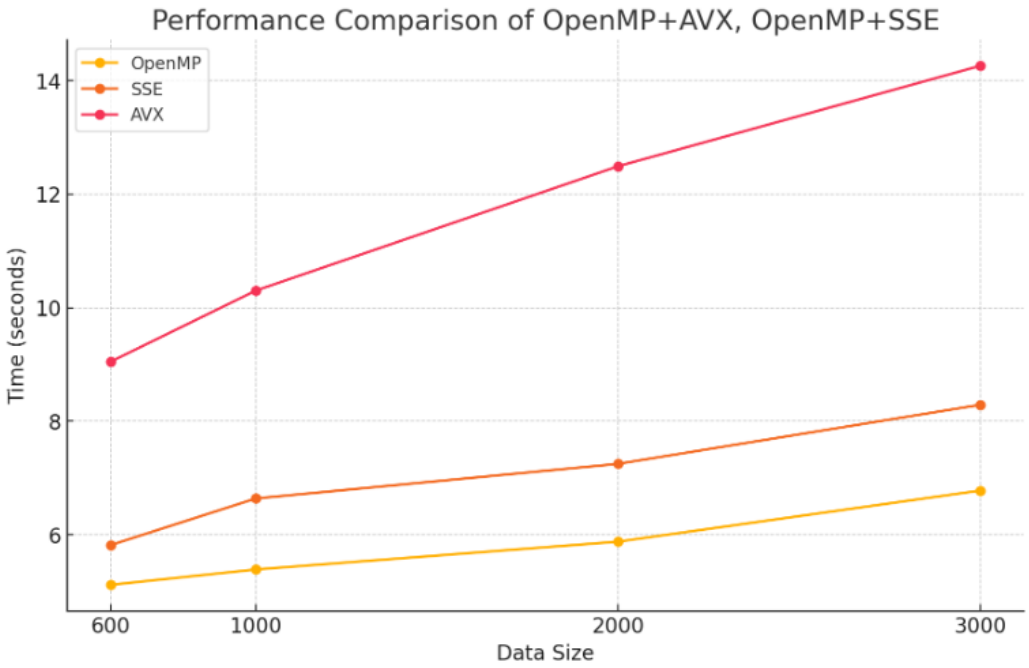

图 13: Caption

从实验结果来看，AVX 加速的效果一骑绝尘，能够在不同矩阵规模达到 10 到 15 的加速比。而 SSE 的加速比相比起仅用 OpenMP 动态的加速比略有上升，但是不很明显。从以往的实验的经验来看，AVX 的加速效率确实是要优于 SSE 的。

八、 arm 平台和 x86 平台的 OpenMP 实验结果横向对比

(一) arm 平台和 x86 平台的 OpenMP 实验结果横向对比

为了便于实验结果的展示,串行算法采用之前的算法,用以下加速比的方式展现 OpenMP+SIMD 的效果更加直观。

可以看出，在矩阵规模较小的时候，x86 的性能可以说是完胜 arm 平台。但在矩阵规模较大的时候，arm 平台运行速度是超过 x86 平台的。出现这张反常状态，我可以归结于两个原因: 硬

表 14: arm 平台和 x86 平台的 OpenMP 算法性能对比

解析度	**arm 4 核**	**x86 4 核**	**arm 6 核**	**x86 6 核**	**arm 8 核**	**x86 8 核**
600	79.64	55.309	58.58	52.414	47.22	44.295
1000	116.685	270.069	84.696	260.897	63.417	212.878

表 15: 表格 16

解析度	**arm 4 核**	**x86 4 核**	**arm 6 核**	**x86 6 核**	**arm 8 核**	**x86 8 核**
2000	503.674	2231.6	179.36	1986.44	145.22	1636.44
3000	10624.35	8023.6	7082.69	7876.25	5945.67	7876.93

件配置以及编程模型。可能是两个平台使用的代码进行了微调导致一些性能上的差异，以及可能是个人笔记本的效率略低于服务器导致的。

同样 x86 平台的加速比也比 arm 平台的低。因为 arm 平台的功率消耗较低，我认为 OpenMP 并行模式更适合 arm 平台使用。

九、 pthread 与 OpenMP 实验结果对比

表 16: pthread 与 OpenMP 实验结果对比（单位：加速比）

矩阵规模	OM+SSE	pth+SSE	OM+AVX	pth+AVX
600	5.82	4.76	9.05	6.22
1000	6.64	5.52	10.3	8.04
2000	7.25	6.13	12.49	6.18
3000	8.29	7.4	14.26	11.17

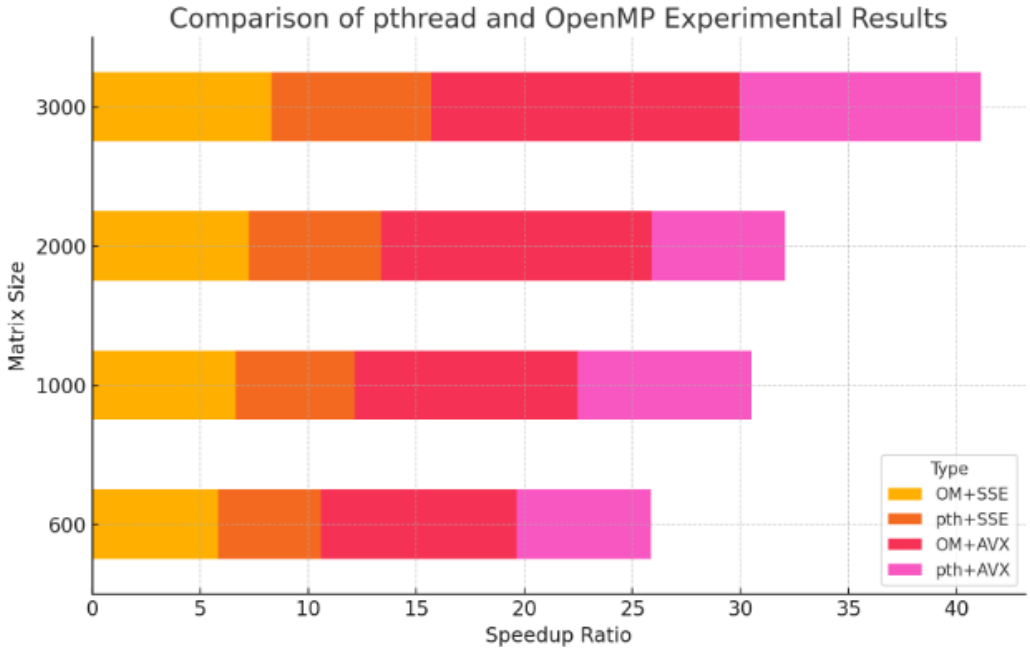

图 14: Caption

我们发现，OpenMP 的加速比很明显高于 pthread。最理想情况下更是能达到 14.05 的加速比收益而 pthread 最高加速比仅 11.17。那么我们也就很好理解 OpenMP 的强大之处：在提供便利操作以进行共享内存多线程的同时，保证性能有不弱于基本 pthread 的本领。API 的强大给程序员带来了极大的便利，也很好理解这项工具受欢迎的原因，即使它难以对线程进行精细控制，但是瑕不掩瑜地我们能够便捷地获得强大的性能收益。这也是很多人更喜欢使用 OpenMP 进行并行程序编写的原因。

十、 进阶要求: 特殊高斯消元的 pthread、OpenMP 并行实验

(一) 实验选题描述

特殊高斯消元的串行算法如下：

1. 对于每个被消元行的首项，如果有对应的消元子（即对应首项位置相同的消元子），则将其减去对应的消元子。重复该过程，直至其变为空行或者首项不在当前消元子能够消去的范围内，或首项不在当前批次覆盖范围内。
2. 如果某一行变为空行，则这行不再参与运算。如果某一行首项没有对应消元子，则将其升格为消元子，之后其将会作为消元子进行下一轮循环。
3. 重复以上过程，直到所有批次处理完毕。此时消元子和被消元行共同作为结果。

输入: 矩阵 $mat[i]$，初始化数组 $Eli[i].exist(i)$ 表示第一个消元行的索引
输出: 处理后的矩阵和行索引数组

```
function Gaussian elimination(mat[], Eli[])
    for i := 0 to m-1 do
        while mat[i] != 0 do
```

```
            if Eli[lp(mat[i])] != NULL then
                mat[i] := mat[i] - Eli[exist(mat[i])]
            else
                Eli[exist(mat[i])] := mat[i]
                break
            end if
        end while
    end for
    return mat[], Eli[]
end function
```

(二) ARM 平台 pthread 并行算法

主要思路是将所有的被消元行均分为 7 份，分别交给 7 个线程进行消元。每个线程负责将自己分得的被消元行依次与所有消元子进行消元

```
void* pthreadF(void* param) {
    for (int i = t_id; i < rows; i += 6) {
        int op = 0;
        int bit = 0;
        int N = COL / 32 + 1;
        while (1) {
            while (op < N && in[i][op] == 0) {
                op++;
                bit = 0;
            }
            if (op >= N) {
                break;
            }
            int temp = in[i][op] << bit;
            while (temp >= 0) {
                bit++;
                temp <<= 1;
            }
            // 开始消元
        }
    }
    pthread_exit(NULL);
}
```

下面的代码展示线程的创建、分配:

```
int worker_count = 6; // 线程数
pthread_t* handles = new pthread_t[worker_count];
threadParam_t* param = new threadParam_t[worker_count];

// 创建线程
for (int t_id = 0; t_id < worker_count; t_id++)
    pthread_create(&handles[t_id], NULL, pthreadF, (void*)&param[t_id]);
```

```
8
9  // 挂起
10 for (int t_id = 0; t_id < worker_count; t_id++)
11     pthread_join(handles[t_id], NULL);
```

(三) ARM 平台 OpenMP 并行算法

思路相同，利用 OpenMP，将所有的被消元行分为 7 份。每个线程负责将自己分得的被消元行依次与所有消元子进行消元。

```
1  #pragma omp parallel for
2  for (int i = 0; i < in2; i++) {
3      int bb = 0, BB = 0;
4      int N = CC / 32 + 1;
5      while (1) {
6          while (bb < N && EE[i][bb] == 0) {
7              bb++;
8              BB = 0;
9          }
10         if (bb >= N) {
11             break;
12         }
13         int temp = EE[i][bb] << BB;
14         while ((EE[i][bb] << BB) >= 0)
15             BB++, temp <<= 1;
16         int& mye = flag[CC - 1 - (bb << 5) - BB];
17         if (mye) {
18             mye = -i;
19             break;
20         } else {
21             if (mye > 0)
22                 er = ER[mye - 1];
23             else
24                 er = EE[-mye];
25             for (int j = 0; j < N; j++)
26                 EE[i][j] ^= er[j];
27         }
28     }
29 }
```

表 17: ARM 平台矩阵乘法运算时间对比（单位：ms）

矩阵规模 n	单行时间	pthread 时间	OpenMP 时间
130/22/8	0.064	0.323	0.072
254/106/53	0.509	0.624	0.432
562/170/53	0.94	1.525	0.622
1011/539/263	14.5	7.868	5.999
2362/1226/453	93.6	33.64	28.46
3799/2759/1953	1497.3	334.25	302.69
8399/6375/4535	18979.6	3524.6	3250.1
85401/5724/756	6341.7	1793.2	1480.2

(四) 实验结果与分析

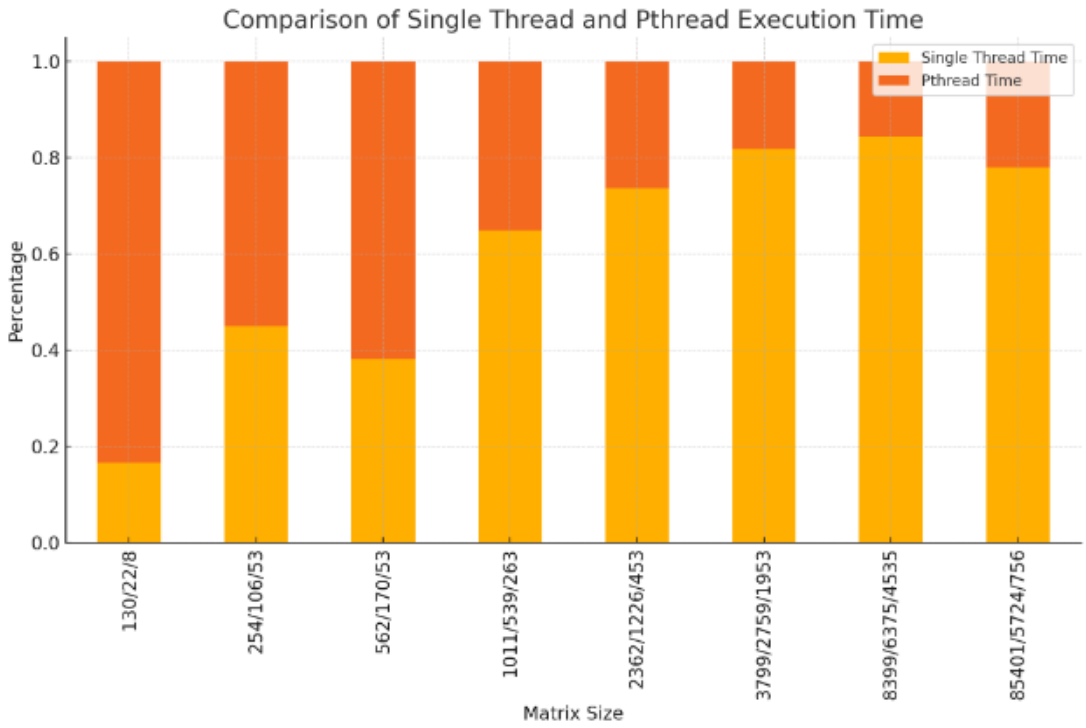

图 15: Caption

当数据的规模较小时，如数据集 1、2、3，Pthread 算法的运行时间比串行算法长，尤其是在数据集 1 中，pthread 算法的运行时间是串行算法的 4 倍以上，说明 pthread 算法在线程的创建与销毁、“升格”的同步等步骤中存在较大的时间开销。当数据的规模不断增大时，Pthread 算法的加速比不断提升。数据集 7（8399/6375/4535）7 路多线程的加速比达到了 5.36，具有良好的加速效果。当数据的规模较小时，如数据集 1，OpenMP 算法的运行时间长于串行算法，说明 OpenMP 算法在线程的创建与销毁，“升格”的同步等步骤中存在较大的时间开销。当数据的规模不断增大时，OpenMP 算法的加速比不断提升。此处结论和 pthread 的结论几乎相同，不再赘述。

（五） 特殊高斯消元中，pthread 与 OpenMP 实验结果横向对比

对比 OpenMP 算法和 Pthread 算法的加速比可以发现，在各个数据规模下，OpenMP 算法的加速效果均优于 Pthread 算法，说明 OpenMP 语句在进行多线程并行化的同时还进行了一定的其他性能优化。这也应证了我们之前在普通高斯消元部分得到的结论。

（六） 进阶要求: 使用 vtune 剖析特殊高斯消元 pthread 算法

表 18: 的线程执行情况

threads	cputime	Clockticks	CPI
thread1	0.986	2520000000	0.677
thread2	0.878	2403000000	0.658
thread2	0.793	2297000000	0.626
thread3	0.888	2536000000	0.691
thread4	0.873	2532000000	0.693
thread5	0.881	2531000000	0.693
thread6	0.881	2540000000	0.691
thread7	0.879	2354000000	0.696

从上表可以分析得出，多线程算法总的 CPU 时间较长，但是由于多线程算法将计算任务分摊给了不同的子线程，因此单个线程的 CPU 时间远小于串行算法，这使得程序的运行时间被大大缩短。这也完全符合上课时所讲授的内容。

十一、 总结

在这次实验中，我们详细探讨了基于 ARM 和 x86 平台的多线程编程技术，涵盖了 `pthread` 和 `OpenMP` 实验。实验的核心在于评估不同矩阵规模和多线程策略下的性能变化。此外，通过实验比较了 `pthread` 和 `OpenMP` 在不同硬件架构上的表现，并测试了 `OpenMP` 中不同的编程策略和划分方法。实验结果揭示了几个关键的发现：

- `OpenMP` 在编程便利性上优于 `pthread`，提供了更高级的抽象，使得程序员能更快速地编写并行程序。
- 从性能角度来看，`OpenMP` 在大多数测试案例中都展现出了比 `pthread` 更高的效率。这种差异可能源于 `OpenMP` 强大的编译器优化和简化的线程管理。
- 特别地，在 ARM 架构上，`OpenMP` 展现出了优异的适配性和性能，这可能是由于 ARM 平台对 `OpenMP` 支持的持续优化和改进。
- 我们也观察到，如果 `pthread` 程序能够更精细地优化，其潜力仍然巨大，尤其是在底层控制和资源管理方面。

通过本次实验，我不仅巩固了高斯消元算法的应用，还对两种并行编程模型的特点、优势和局限有了更深入的理解。尽管实验过程颇为繁琐，但获得的经验和知识让人觉得非常值得。这为未来的期末作业奠定了坚实的基础。

十二、 代码链接

https://github.com/uJunLI/NKU-Parallel_programming/tree/masterlab3