



南開大學
Nankai University

计算机学院
并行程序设计实验报告

Pthread&OpenMP 优化高斯消元算法

姓名：张天歌

学号：2211123

专业：计算机科学与技术

2024 年 5 月 26 日

目录

1 概述	2
1.1 实验平台	2
1.2 正确性验证说明	2
1.3 实验用时测量说明	2
2 普通高斯消元	3
2.1 串行算法	3
3 普通高斯消元 Pthread	4
3.1 实验设计	4
3.2 划分方式和同步机制	4
3.2.1 动态线程版本	4
3.2.2 静态线程 + 信号量同步版本	6
3.2.3 静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数	8
3.2.4 静态线程 + barrier 同步	9
3.2.5 对比总结	11
3.3 ARM 平台与 x86 平台对比	11
3.3.1 分配方式 + 同步机制	12
3.3.2 线程数目对比	13
3.3.3 结合 SIMD	14
4 普通高斯消元 OpenMP	15
4.1 实验设计 (ARM 平台与 x86 平台对比)	15
4.2 OpenMP 任务分配方式	16
4.3 线程数目	17
4.4 与 SIMD 结合	18
5 普通高斯消元多线程优化总结	19
5.1 Pthread 和 OpenMP 性能差异	19
5.2 cache 优化和按列划分	19
6 基于 Gröbner 基计算的特殊高斯消元	20
6.1 问题分析	20
6.2 pthread 优化思路	20
6.3 OpenMP 优化思路	21
6.4 实验结果	21
7 总结与反思	23

Abstract

本次实验主要完成了普通高斯消元和特殊高斯消元的 Pthread、openMP 并行化，分析了不同任务划分方式和同步机制，不同数据划分方式、以及 pThread 与 openMP 的实现差异，同时结合 SIMD 进行优化，测试了不同问题规模/线程数目/参数变化对执行时间的影响，对 cache 命中等指标进行 profiling；此外，实验还在 Windows x86 环境下进行了实验，并与 ARM 环境下的性能进行了对比。

关键字：并行；高斯消元优化；Pthread；OpenMP

1 概述

基于高斯消元并行优化的期末选题报告，本次实验采用 Pthread 和 OpenMP 优化方法设计系列算法策略和不同同步机制，结合 SIMD 的指令集，分别对普通高斯消元和特殊高斯消元进行不同程度的优化尝试。

1.1 实验平台

本次实验中，ARM 基于华为鲲鹏云平台进行，x86 平台中的实验基于 Code::Block 进行。两个平台处理器信息如下图1.1、1.2所示：


Processor					
Name	Intel Core i5 12500H				
Code Name	Alder Lake	Max TDP	45.0 W		
Package	Socket 1744 FCBGA				
Technology	10 nm	Core VID	1.273 V		
Specification	12th Gen Intel®Core™i5-12500H				
Family	6	Model	A	Stepping	3
Ext. Family	6	Ext. Model	9A	Revision	L0
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3, SHA				
Clocks (P-Core #0)					
Core Speed	4489.02 MHz				
Multiplier	x 45.0 (4.0 - 31.0)				
Bus Speed	99.76 MHz				
Rated FSB					
Cache					
L1 Data	4 x 48 KB + 8 x 32 KB				
L1 Inst.	4 x 32 KB + 8 x 64 KB				
Level 2	4 x 1.25 MB + 2 x 2 MB				
Level 3	18 MBytes				
Selection					
Socket #1		Cores		4P + 8E	
		Threads		16	

图 1.1: x86 处理器信息

1.2 正确性验证说明

在进行高斯消元算法的并行化程序开发时，确保结果的准确性是至关重要的。由于高斯消元过程中的数据具有强烈的时序依赖性，验证最终输出矩阵的正确性可以为我们提供算法正确执行的强有力证据。在实际操作中，用具有确定性特征的矩阵，如对角矩阵、上三角矩阵或下三角矩阵，验证算法的正确性；如果测试程序输出正确，此后重复测试过程中，仅比对输出矩阵各元素均值，若无误，则认为并行化程序正确性得到检验，数据可以采信。

1.3 实验用时测量说明

在时间性能的测量上，x86 平台与 ARM 平台分别选用高精度计时函数 QueryPerformanceCounter() 与 time.h 来计时，并通过重复多次取平均值的策略计算单次程序/函数运行时间。且由于精确计时函

选项	信息
内核版本	Linux master 4.14.0-115.el7a.0.1.aarch64
发行版版本	OpenEuler(CentOS)
编译器	HUAWEI 毕升编译器 (clang 12.0.0)
CPU 型号	鲲鹏 920 服务器版
性能分析工具	Perf
CPU 核心数	96 核 96 线程
CPU 主频	2.6GHz
L1 一级缓存	96 64KB 数据缓存 96 64KB 指令缓存
L2 二级缓存	96 512KB
L3 三级缓存	48MB
内存	20GB

表 1: 鲲鹏服务器的 CPU 相关参数

图 1.2: arm 处理器信息

数开销巨大，实验中统一在循环体外进行计时，忽略循环条件判断等微小开销。

2 普通高斯消元

2.1 串行算法

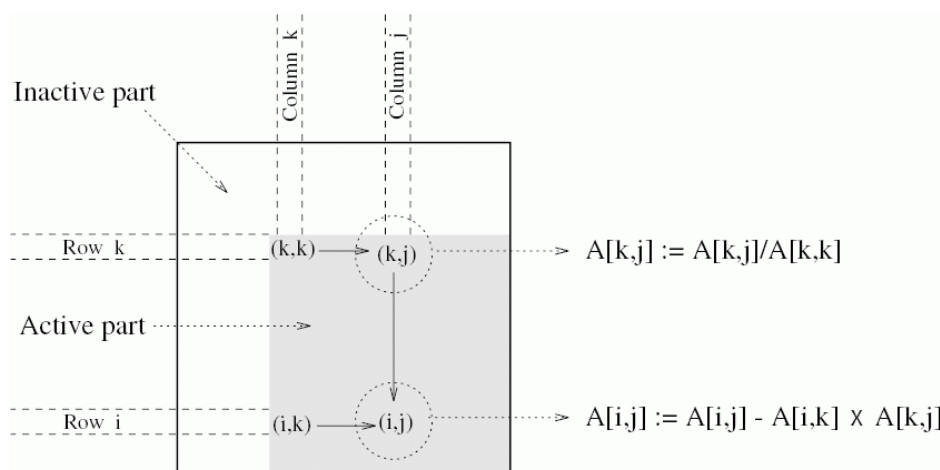


图 2.3: 高斯消去法示意图

高斯消去的计算模式如图2.3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。串行代码的算法如下面伪代码（算法 1）所示：

对于本问题，在初始化数据时，我们设计了一个 `reset` 函数，令主对角线上的元素 $m1[i][i] = 1$ ，令上三角区域的数据 $m1[i][j] = \text{rand}() \% 1000 + 1$ ，下三角区域的数据 $m1[i][j] += m1[k][j]$ 。这种初始化可以重复生成样例、避免产生无穷数、检查程序正确性等。

Algorithm 1 Gaussian Elimination 串行算法

```

1: procedure GAUSSIAN__ELIMINATION(A,B)
2:    $n \leftarrow \text{size}(A)$  ▷ 消元过程
3:   for  $k$  from 1 to  $n$  do
4:     for  $i$  from  $k + 1$  to  $n$  do
5:        $\text{factor} \leftarrow \frac{A[i,k]}{A[k,k]}$ 
6:       for  $j$  from  $k + 1$  to  $n$  do
7:          $A[i,j] \leftarrow A[i,j] - \text{factor} \times A[k,j]$ 
8:       end for
9:        $B[i] \leftarrow B[i] - \text{factor} \times B[k]$ 
10:    end for
11:  end for
12: end procedure

```

3 普通高斯消元 Pthread

3.1 实验设计

伪代码第 5 行第一个内嵌循环中的 $\text{factor} := A[k, j]/A[k, k]$ 以及伪代码第 6, 7, 8 行双层 for 循环中的 $A[i, j] := A[i, j] - \text{factor} \times A[k, j]$ 可以考虑使用不同的同步机制（如信号量和 barrier）和线程管理策略（动态线程和静态线程）并行化，并结合 SIMD 编程，讨论并行优化加速比随问题规模和线程数量的变化情况。本次实验分别在 ARM 架构和 x86 架构两个平台实验，对比不同平台上多线程编程的性能差异；并且讨论 Pthread 和 OpenMP 的程序性能差异。具体实验步骤如下：

1. 讨论同步机制和线程管理策略时，固定程序线程数目为 8。实验中分别使用动态分配方式和静态分配方式，分别运用信号量同步机制和 barrier 同步机制，为了达到较好的线程分配效果，考虑将三重循环整体抽取出来构成线程函数，最外层循环每个循环步结束时，使用 barrier 同步所有线程一起进入下一个循环步（简单、低开销的同步方法）。
2. 结合 SIMD 指令集，分析 x86 的 AVX 和 arm 的 neon 指令集对静态分配线程的性能影响，实现三重纳入 +neon/AVX、静态 barrier+neon/AVX、静态信号量同步 +neon/AVX 的性能优化对比。
3. 在线程分配中，线程太少，会使得加速效果不明显，而线程太多，导致管理线程、分配任务的开销又增大，使得其加速效果下降，因此找到合适的线程数目极为重要。我们假设固定一种线程管理方式和同步机制，分析 4、6、8、10、12、16 不同线程数对于 pthread 性能的影响。

3.2 划分方式和同步机制

3.2.1 动态线程版本

通过每轮消去步骤前动态创建和销毁线程，实现高斯消元的并行计算。

动态线程版本

```

1 // 主函数负责初始化矩阵的高斯消元步骤，并创建线程
2 void dynamicMain(void* (*threadFunc)(void*)) {
3   for (int  $k = 0$ ;  $k < \text{ROW}$ ; ++ $k$ ) {
4     // 计算第  $k$  行的消元值

```

```

5     float32x4_t diver = vld1q_dup_f32(&a[k][k]);
6
7     int j;
8     for (j = k + 1; j < ROW && ((ROW - j) & 3); ++j)
9         a[k][j] = a[k][j] / a[k][k];
10    for (; j < ROW; j += 4)
11    {
12        float32x4_t divee = vld1q_f32(&a[k][j]);
13        divee = vdivq_f32(divee, diver);
14        vst1q_f32(&a[k][j], divee);
15    }
16    a[k][k] = 1.0;
17
18    // 计算工作线程数量
19    int worker_count = ROW - 1 - k;
20    // 分配线程句柄和参数数组
21    pthread_t* handles = new pthread_t[worker_count];
22    threadParam_t* param = new threadParam_t[worker_count];
23
24    // 初始化线程参数
25    for (int t_id = 0; t_id < worker_count; t_id++) {
26        param[t_id].k = k;
27        param[t_id].t_id = t_id;
28    }
29
30    // 创建工作线程
31    for (int t_id = 0; t_id < worker_count; t_id++) {
32        pthread_create(&handles[t_id], NULL, threadFunc, &param[t_id]);
33    }
34
35    // 等待工作线程完成
36    for (int t_id = 0; t_id < worker_count; t_id++) {
37        pthread_join(handles[t_id], NULL);
38    }
39 }
40 }
41

```

分别采取 500、1000、1500、2000 数据规模，对于规模小的程序，运行时以 1 秒为限进行循环，循环结束后除以循环次数来获取平均值；对于规模大的程序，则采用重复运行 5 次来取平均值。结果如下图所示：

- 线程管理开销：动态线程的创建和销毁可能会引入显著的开销，尤其是在数据规模较小的情况下，线程管理开销可能超过了并行计算带来的性能提升。
- 负载不均衡：动态线程可能导致工作负载分配不均，特别是在数据规模较小时，这可能导致某些线程空闲等待，降低了并行效率。

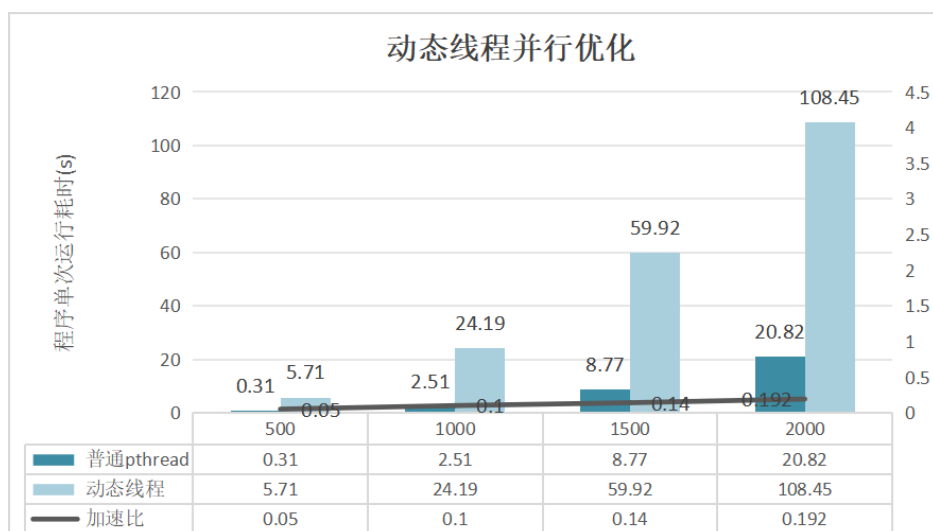


图 3.4: 动态线程版本的优化效果

3.2.2 静态线程 + 信号量同步版本

固定分配 8 个线程，利用静态创建的线程池和sem_t信号量进行同步，以减少线程创建销毁开销，并实现高斯消元的并行处理。

静态线程 + 信号量

```

1 // 信号量定义
2 sem_t sem_main;
3 sem_t sem_workerstart[NUM_THREADS]; // 每个线程有自己专属的信号量
4 sem_t sem_workerend[NUM_THREADS];
5
6 // 线程函数定义
7 void* threadFunc(void* param) {
8     threadParam_t* p = (threadParam_t*)param;
9     int t_id = p->t_id;
10
11     for (int k = 0; k < n; ++k) {
12         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）
13
14         // 循环划分任务
15         for (int i = k + 1 + t_id; i < n; i += NUM_THREADS) {
16             // 消去
17             for (int j = k + 1; j < n; ++j) {
18                 A[i][j] = A[i][j] - A[i][k] * A[k][j];
19             }
20             A[i][k] = 0.0;
21         }
22         sem_post(&sem_main); // 唤醒主线程
23         sem_wait(&sem_workerend[t_id]); // 阻塞，等待主线程唤醒进入下一轮
24     }
25     pthread_exit(NULL);
26 }

```

```

27
28 //Funcmain 函数
29     for (int k = 0; k < n; ++k) {
30         // 主线程做除法操作
31         for (int j = k + 1; j < n; j++) {
32             A[k][j] = A[k][j] / A[k][k];
33         }
34         A[k][k] = 1.0;
35
36         // 开始唤醒工作线程
37         for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
38             sem_post(&sem_workerstart[t_id]);
39         }
40
41         // 主线程睡眠（等待所有的工作线程完成此轮消去任务）
42         for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
43             sem_wait(&sem_main);
44         }
45
46         // 主线程再次唤醒工作线程进入下一轮次的消去任务
47         for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
48             sem_post(&sem_workerend[t_id]);
49         }
50     }
51
52     for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
53         pthread_join(handles[t_id], NULL);
54     }
55

```

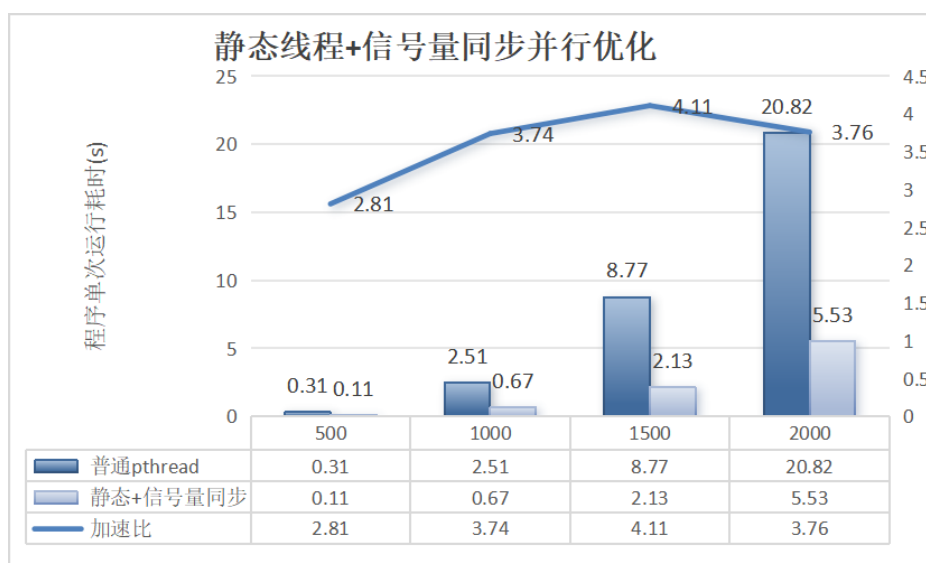


图 3.5: 静态线程 + 信号量同步版本的优化效果

- 信号量开销: 信号量的等待和通知操作可能会引入额外的开销, 尤其是在数据规模较小时, 这种

开销可能更加明显。

- 资源浪费: 静态线程可能在小规模数据上造成资源浪费, 因为线程可能在没有足够工作量的情况下仍然占用 CPU 资源。

3.2.3 静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数

将三重循环整体抽取出来构成线程函数, 并使用静态线程池和信号量同步, 全程只有一次线程创建和销毁开销, 以提高计算效率和同步灵活性。

纳入三重循环的静态线程信号量同步版本

```

1 // 线程函数 staticFunc2
2 void* staticFunc2(void* param) {
3     long t_id = (long)param; // 从参数中获取线程 ID
4
5     for (int k = 0; k < ROW; ++k) {
6         // 如果是主线程 (t_id 为 0), 执行除法操作
7         if (t_id == 0) {
8             // 执行第 k 行的除法操作
9             float32x4_t diver = vld1q_dup_f32(&a[k][k]);
10            // ... (除法操作代码)
11            a[k][k] = 1.0;
12        }
13        // 非主线程等待除法操作完成
14        else sem_wait(&sem_Division[t_id - 1]);
15
16        // 主线程通知其他线程除法操作完成
17        if (t_id == 0) {
18            for (int t_id = 0; t_id < NUM_THREADS - 1; ++t_id)
19                sem_post(&sem_Division[t_id]);
20        }
21
22        // 工作线程执行消去操作
23        for (int i = k + 1 + t_id; i < ROW; i += NUM_THREADS) {
24            // 执行消去操作
25            float32x4_t mult1 = vld1q_dup_f32(&a[i][k]);
26            // ... (消去操作代码)
27            a[i][k] = 0.0;
28        }
29
30        // 线程间同步以等待消去操作完成
31        if (t_id == 0) {
32            // 主线程等待所有工作线程完成消去操作
33            for (int t_id = 0; t_id < NUM_THREADS - 1; ++t_id)
34                sem_wait(&sem_leader);
35            // 然后唤醒所有工作线程进入下一轮
36            for (int t_id = 0; t_id < NUM_THREADS - 1; ++t_id)
37                sem_post(&sem_Elimination[t_id]);
38        }
39        else {

```

```

40     // 工作线程通知主线程消去操作完成，并等待下一轮
41     sem_post(&sem_leader);
42     sem_wait(&sem_Elimination[t_id - 1]);
43 }
44 }
45 pthread_exit(NULL);
46 }
47

```

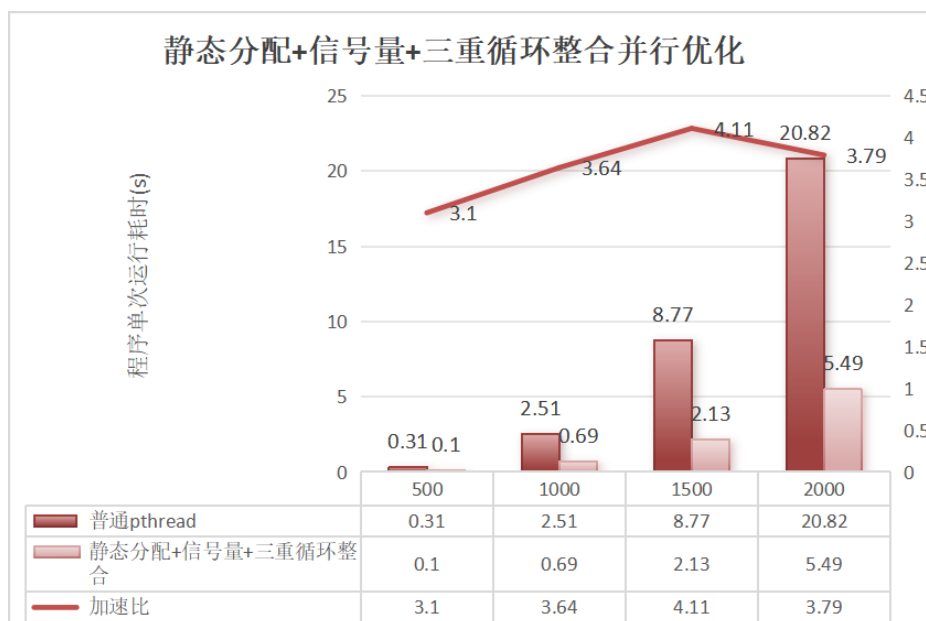


图 3.6: 纳入三重循环的静态线程信号量同步版本的优化效果

- 减少同步: 通过将三重循环纳入并行化，可能减少了同步点的数量，降低了同步开销。
- 较高的加速比: 在所有数据规模下，加速比都较高，表明并行化效率较好，在拐点 1000、2000 加速比斜率陡升，使用 perf 工具进行深入分析后，我们发现在 $n=3000$ 时，串行算法的缓存未命中 (cache miss) 率有所下降，这是其性能提升的原因。

3.2.4 静态线程 + barrier 同步

采用静态线程池和 barrier 同步机制简化线程间的协调，实现高斯消元算法的高效并行执行。



图 3.7: “静态线程 + barrier 同步” 逻辑图

静态线程 + barrier 同步

```

1 // barrier 定义
2 pthread_barrier_t barrier_Division;
3 pthread_barrier_t barrier_Elimination;
4

```

```

5 // 线程函数定义
6 void* threadFunc(void* param) {
7     threadParam_t* p = (threadParam_t*)param;
8     int t_id = p->t_id;
9
10    for (int k = 0; k < n; ++k) {
11        // t_id 为 0 的线程做除法操作, 其它工作线程先等待
12        if (t_id == 0) {
13            for (int j = k + 1; j < n; j++) {
14                A[k][j] = A[k][j] / A[k][k];
15            }
16            A[k][k] = 1.0;
17        }
18        // 第一个同步点
19        pthread_barrier_wait(&barrier_Division);
20        // 循环划分任务
21        for (int i = k + 1 + t_id; i < n; i += NUM_THREADS) {
22            // 消去
23            for (int j = k + 1; j < n; j++) {
24                A[i][j] = A[i][j] - A[i][k] * A[k][j];
25            }
26            A[i][k] = 0.0;
27        }
28        // 第二个同步点
29        pthread_barrier_wait(&barrier_Elimination);
30    }
31    pthread_exit(NULL);
32    return NULL; // 注意: 实际中 pthread_exit 无需返回值
33 }
34
35 int main() {
36     // 初始化 barrier
37     pthread_barrier_init(&barrier_Division, NULL, NUM_THREADS);
38     pthread_barrier_init(&barrier_Elimination, NULL, NUM_THREADS);
39
40     // 创建线程
41     pthread_t handles[NUM_THREADS];
42     threadParam_t param[NUM_THREADS];
43     for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
44         param[t_id].t_id = t_id;
45         pthread_create(&handles[t_id], NULL, threadFunc, (void*)&param[t_id]);
46     }
47
48     // 等待所有线程结束
49     for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
50         pthread_join(handles[t_id], NULL);
51     }
52
53     // 销毁所有的 barrier

```

```

54 pthread_barrier_destroy(&barrier_Division);
55 pthread_barrier_destroy(&barrier_Elimination);
56 }

```

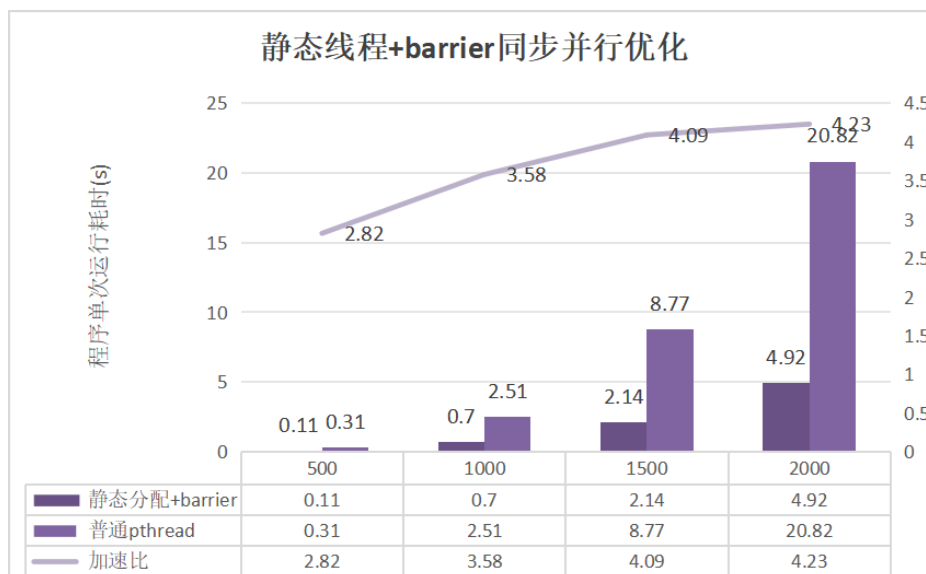


图 3.8: 静态线程 +barrier 同步的优化效果

- Cache 优化: 在小规模数据上, 由于数据量较小, 可能无法充分利用 CPU 缓存, 导致 cache 未命中增多, 降低了并行效率。
- 同步开销: Barrier 同步可能在小规模数据上引入了额外的开销, 因为所有线程都需要在 barrier 处等待, 增加了同步的时间。

3.2.5 对比总结

- 小规模加速比小于 1: 创建和销毁线程的开销在小数据规模时可能超过了并行计算所带来的好处, 开销代价大过收益; 负载不均衡: 线程之间工作负载分配不均, 导致一些线程空闲等待; 指导书中说明 CPU 核心数有限, 过多的线程数可能会因为上下文切换和资源竞争而导致效率降低。
- 加速比: 所有方法的加速比都随着数据规模的增加而提高, 但增长速度和最终值各有不同。三重循环纳入 + 静态信号量同步并行优化在大部分数据规模下显示出最高的加速比。
- 线程管理: 动态线程方法可能会因为线程创建和销毁的开销而在小规模数据上表现不佳。静态线程方法通过减少这些开销提高了效率。
- 同步机制: 使用屏障和信号量作为同步机制的方法在大规模数据上显示出较好的性能, 但具体的同步开销和效率取决于实现细节和数据特性。

3.3 ARM 平台与 x86 平台对比

x86 平台上的 pthread 优化方法和同步策略仍然如上文所述, 仅仅只是平台的不同, 以及所结合的 SIMD 指令集不同。在 x86 平台上, 选择 AVX 指令集结合 pthread 进行优化。具体代码可看代码仓库。

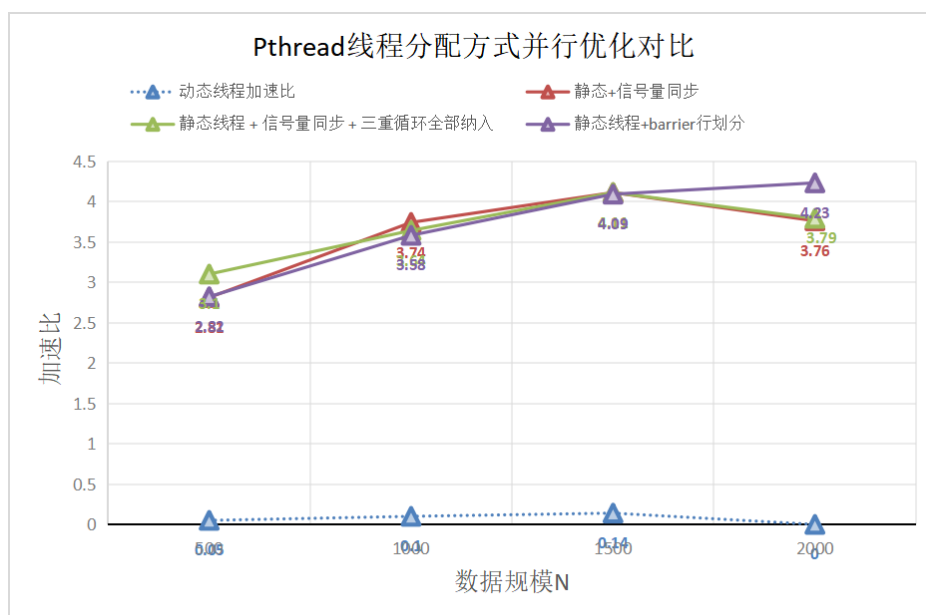


图 3.9: Pthread 线程管理对比

编译选项：

- Have g++ follow the C++17 GNU C++ language standard
- Target x86(32bit)
- Optimize even more(for speed) [-O2]
- -march = native

3.3.1 分配方式 + 同步机制

x86 运行结果：

初步分析可见，出现了与 ARM 平台相同的问题，即动态分配线程这种方法下耗时会出奇的慢，其负优化程度相比 ARM 平台更甚，仅有 0.03 左右的加速比，也就是说耗时是原先的 30-40 倍，可见 O(n²) 次的线程创建和销毁带来的时间开销是巨大的，尤其是在数据规模 n 较大时，因此可知这种方法是不切实际的，是无法提升程序性能的。

在小数据规模下，动态线程可能因为线程管理开销较大而性能不佳；中等数据规模下，静态信号量方案开始展现出较好的性能，但可能仍受同步开销的限制。大规模数据规模下，静态屏障方案和静态信号量 + 三重循环方案表现最佳，因为它们能够有效地管理线程同步和数据局部性。

因此，在大规模数据下，静态屏障方案可能是最佳选择，因为它提供了高效的线程同步。

x86 和 ARM 对比图：

用 perf 对 x86 程序进行 profiling, 详细数据如表3：

1. 线程同步机制的选择：这对性能有显著影响。**静态同步机制（如信号量和 barrier）可能会增加线程间的协调开销，但可以减少线程间的竞争。**

2. 线程管理开销：多线程执行需要额外的线程调度和管理开销，这可能抵消了并行带来的一些好处，**cache miss 概率增高，cache hit 概率变低。**

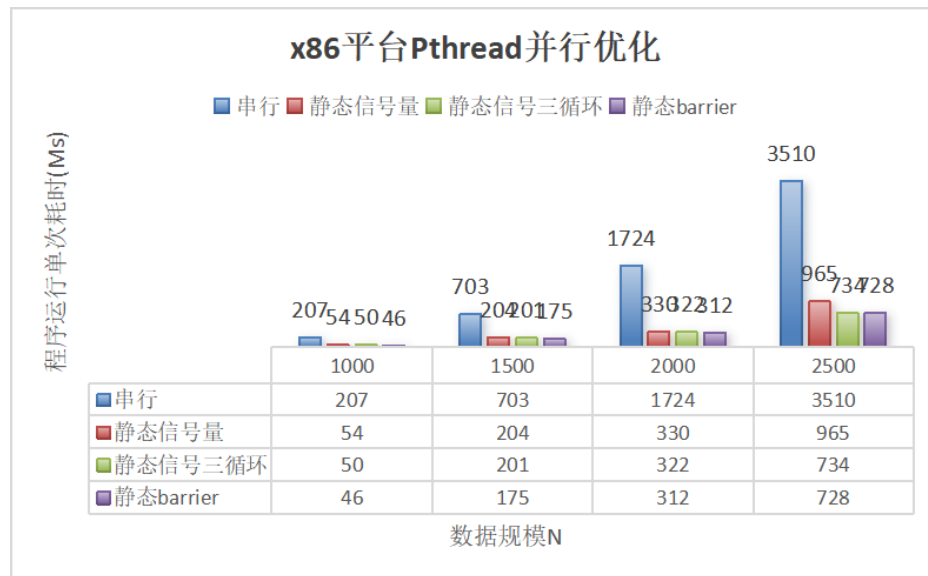


图 3.10: x86 平台 Pthread 并行优化

	L1-loads	L1-misses	miss 的概率	L1-load-stores	cache 命中率	CPI
串行	36310296866	639794535	0.017620196	35670502331	0.982379804	0.396296002
动态分配	7822398470	773305938	0.098857907	7049092532	0.901142093	6.285635291
静态信号量	7302221068	716645591	0.098140769	6585575477	0.901859231	6.744104566
静态信号三循环	7269205001	723051561	0.099467763	6546153440	0.900532237	6.928883775
静态 barrier	7215566552	421017005	0.058348433	6794549547	0.941651567	6.791952618

表 1: x86 特殊高斯消元的对齐并行化

总结: 尽管动态线程分配可能在理论上提供更好的性能,但实际上可能因为线程管理开销和资源竞争而导致性能下降。静态同步机制可能在减少线程间竞争方面更有效,但需要仔细设计以避免过高的同步开销。

3.3.2 线程数目对比

在 x86 平台下,使用静态线程和 barrier 信号同步,在 1000, 2000, 2500 规模下,讨论不同线程数目对 pThread 并行化的影响。

根据图3.12所示,我们观察到随着线程数从单线程增加到十六线程,程序的性能呈现出先提升后下降的趋势。在较低线程数时,性能提升较为显著,特别是从四线程增加到八线程时,这表明并行化开始发挥效果,最后到八线程性能最好。但当线程数增加到一定程度后,在较大规模例如 2500 之后,性能增益开始减少,这可能是由于硬件资源(如 CPU 核心数、内存带宽)的限制。

观察三条线的整体趋势,规模越大在多线程中性能越好,2500 规模的 8 线程加速比可达到最高 5.45, 16 线程加速比可达到 5.68。

在不同平台对线程数目因素的检测,加速比曲线趋势大致相同,又因为八线程具有独特的性能表现优势,故比较 ARM 平台八线程、不同规模的静态 +barrier 同步即可。在 2000 数据规模八线程,ARM 的加速比为 4.23,低于 x86 的 5.42;在 1000 数据规模八线程,ARM 加速比为 3.58,低于 x86 的 4.05。可以得出结论, **x86 性能更加优越,更适合做并行优化,程序性能更优。**

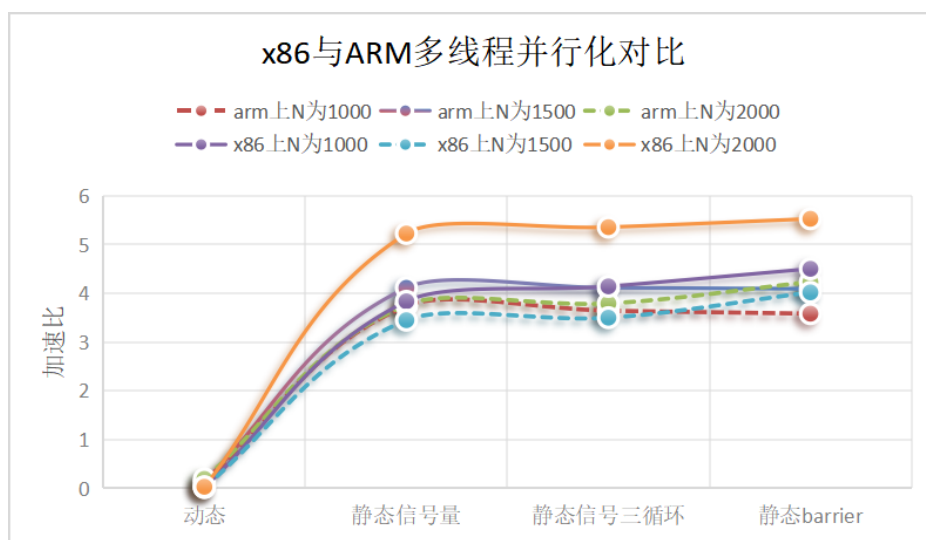


图 3.11: x86 与 ARM 多线程并行化对比

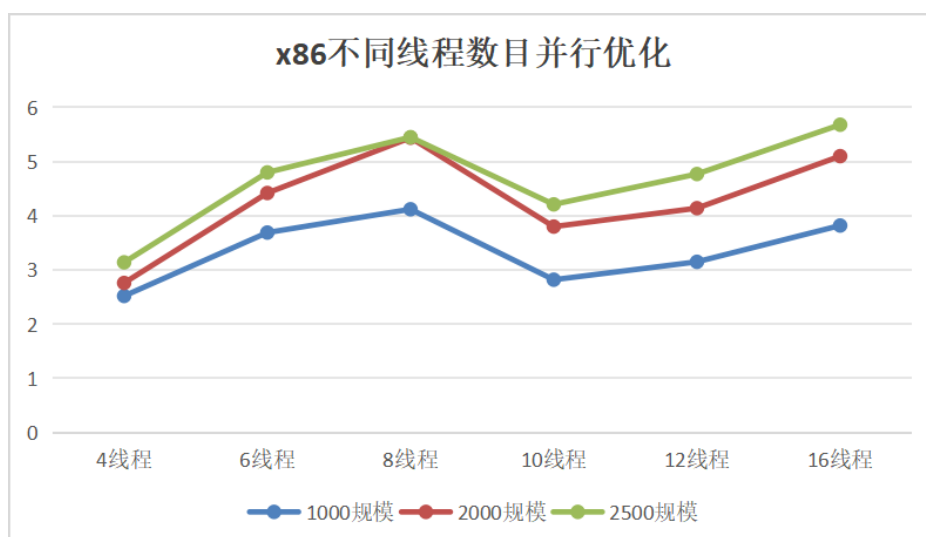


图 3.12: x86 不同线程数目并行优化

3.3.3 结合 SIMD

将三种划分方式做横向比较，三者最高加速比在 2000 规模达到 10 以上，可以证明集合 SIMD 对 pthread 多线程优化有积极作用。静态分配 + barrier 结合 SIMD 的效果最好，在 2000 规模可达 12.49，比静态 + 信号量同步的 10.45 高，比三重循环整合的 10.77 高。

单独做纵向比较，对比 AVX 指令集的提高一倍加速比，NEON 指令集对程序优化很不明显，证明了之前 SIMD 实验中 NEON 只可达到一点多的优化效果，AVX 性能远高于 NEON。

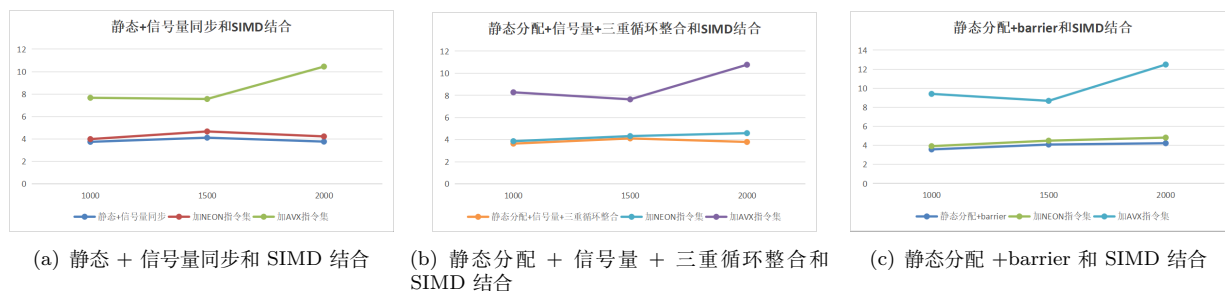


图 3.13: 不同并行优化算法的执行时间与准确率对比

4 普通高斯消元 OpenMP

4.1 实验设计 (ARM 平台与 x86 平台对比)

在 OpenMP 中, 编程范式较为一致简单, 因此在两个平台可以实现同步对比, 将任务划分方式、静态算法优化、不同线程数对比、结合 SIMD 两两比较分析, 在单平台内分析算法优化, 在多平台比较架构差异。

OpenMP 优化思想与 pthread 相同, 都是对行消去部分进行并行优化, 不同的是 OpenMP 只用几行简单的编译语句即可转换成并程序, 不用像 pthread 一样麻烦地管理线程。

此次要用到的编译语句大致有几条:

- `#pragma omp parallel num_threads(NUM_THREADS), private(i, j, k, tmp)`, 在外循环之外创建线程, 避免线程反复创建销毁, 同时设置意共享变量和私有变量, 防止线程冲突;
- `#pragma omp single`, 只用一个线程进行该部分代码的处理, 可用在除法部分;
- `#pragma omp for schedule(static/dynamic/guided)`, 指定行消去部分任务的划分方式, 这也是接下来要比较的内容。

负载均衡任务划分方式

OpenMP 使用简单, 比较容易探究多线程并行优化中任务划分方式带来的影响。对于 for 循环的多线程并行, 大致有 static (chunksize)、dynamic(chunksize)、guided(chunksize) 三种方式, 它们的区别在:

1. `#pragma omp for schedule` 默认采用 static 划分, static 指每个线程划分 chunksize 个任务量, 如果 chunksize 没有人为指定, 则默认为

$$\frac{N}{\text{NUM_THREADS}}$$

N 是任务总量。这样可以达到均衡的任务负载, 没有空出来的线程, 且各线程间工作量差别不会有很大差异。对于能够事先得知总任务量的高斯消元算法而言, 这是一种合适的任务划分方式。

2. dynamic 方式可以动态分配任务量, 每当一个工作线程空出来时会为其分配 chunksize 的工作量, chunksize 为 3。这样的方法比较灵活, 但是可能会出现各线程之间负载不均衡的情况, 甚至会出现有的线程没有参与工作。

3. guided 方式与 dynamic 相近, 不过每次分配的任务量不同, 开始比较大, 之后逐渐减小至 chunksize 不再减小, chunksize 为 3。这种方式一开始多分配一些任务量, 避免了多次的任务划分, 也是一种很好的划分方式。

尝试改变 chunksize

在任务执行中某个线程结束了先前的任务之后，又会从任务队列中取出新的任务，而并不依赖于线程号与行号的某种映射关系。改变 chunksize 个数分别为：1、3、5、10、15，在二线程规模为 1024 下，观察循环调度的多线程管理是否对负载均衡有帮助，找到合适的 chunksize。

改变线程数目

分别在 x86 和 arm 使用静态划分，在 128、256、512、1024、2048、4096 规模下，讨论不同线程数目对 pThread 并行化的影响。

与 SIMD 结合

使用八线程，chunksize=3，在 x86 平台分别实现动态分配 +avx、动态分配 +sse，静态分配 +avx、静态分配 +sse，分析程序耗时和加速比，实现并行化最好效果。

4.2 OpenMP 任务分配方式

编译选项：

- Have g++ follow the C++17 GNU C++ language standard
- Target x86(32bit)
- Optimize even more(for speed) [-O2]
- Other Compiler options: -march = native -fopenmp
- Other linker options: -fopenmp

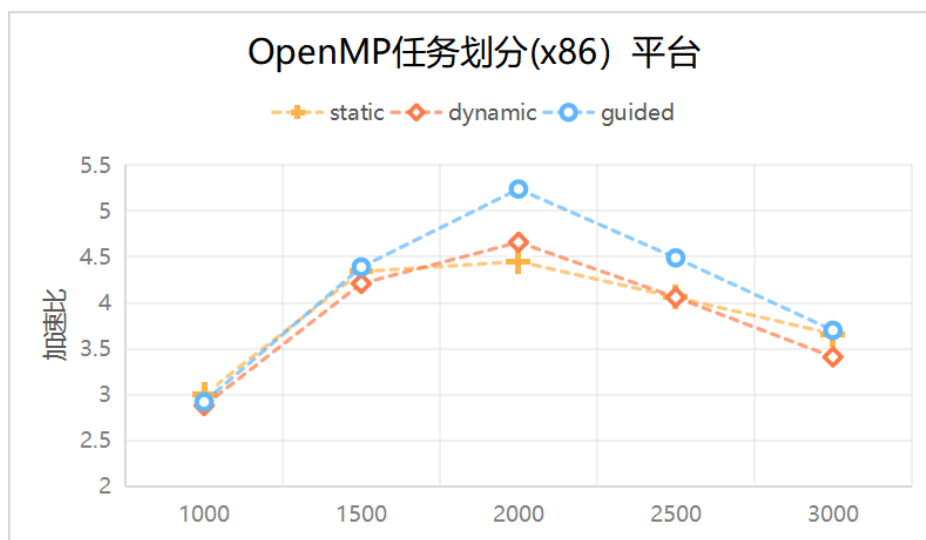


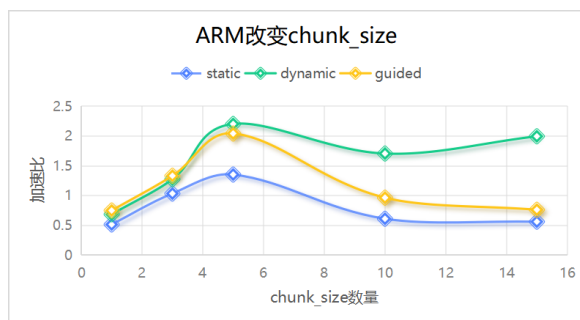
图 4.14: OpenMP 任务分配 (x86) 平台

总体而言，dynamic 方式的优化效果在三种方式中表现较差，分析是因为 dynamic 方式需要不断分配空闲线程，导致效果不佳，甚至可能会出现有的线程没有参与工作，或线程任务量分配不均的情况，不能较好的利用线程资源。guided 划分方式表现最好，在所有数据规模中，guided 总是达到最高的加速比。这与 guided 划分方式有关，**guided 相比 dynamic 更加灵活**，最初为每个线程分配较多任务量，然后指数下降到 chunksize，能够有效的均衡线程之间的负载。用 **perf 测量指令数、周期数**，计算 **CPI**，可以更细粒度地分析出 static 划分是一种更平稳、有效的划分方法。

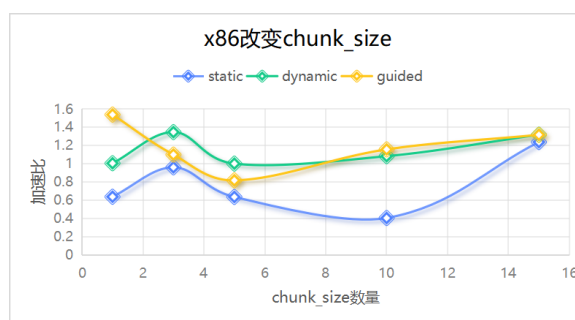
	指令数	周期数	CPI
static	8826664293	6032198764	0.683406388
dynamic	8798756157	6593784243	0.749399588
guided	8792903322	6224766053	0.707930683

表 2: 对 OpenMP 静态、动态、guided 进行 profiling

改变 chunksize



(a) ARM 改变 chunksize



(b) x86 改变 chunksize

图 4.15: arm 与 x86 对比 chunksize 变化

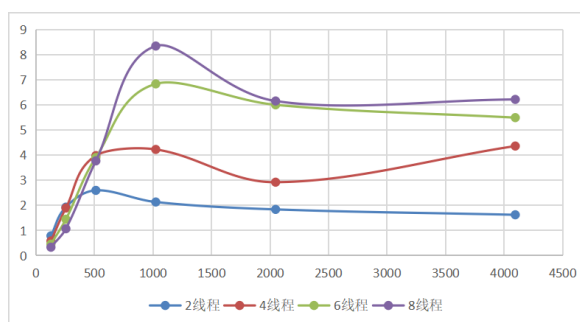
找到合适的 chunksize 有益于程序性能提升，在消去过程可以更灵活的使用循环划分来平衡负载。

1. 较小的 chunksize 可以帮助平衡负载，避免某些线程过早完成它们的迭代而闲置，而其他线程还在处理它们的迭代，例如在 x86，chunksize 为 1 时，guided 分配方式加速比较大。

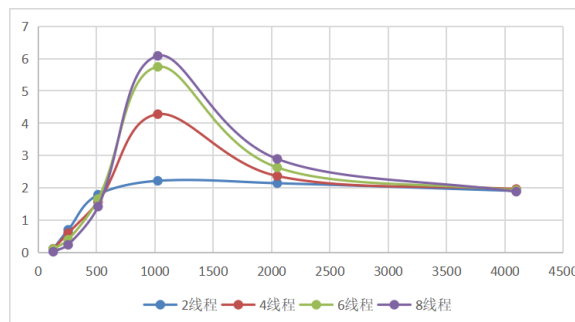
2. 然而，过小的 chunksize 也会增加线程之间的同步开销，x86 平台适合的 chunksize=3，arm 平台适合的 chunksize=5。

3. 本次比较在数据规模较小情况下完成，不同划分方式的性能可能与预期结果有出入，误差不大，且只需增大数据规模可看到更高的加速比和明显的曲线走势。

4.3 线程数目



(a) arm 不同线程数对比



(b) x86 不同线程数对比

图 4.16: arm 与 x86 对比线程数不同

OpenMP 和 Pthread 差不多，根据图4.16所示，观察到随着线程数从单线程增加到八线程，程序的性能呈现出先提升后下降的趋势，八线程性能最好。但当数据规模增加到一定程度后，在较大规模

例如 2048 之后，性能增益开始减少，这可能是由于硬件资源（如 CPU 核心数、内存带宽）的限制。下面是平台对比：

1. 加速比趋势：ARM 平台在较大规模下加速比趋于稳定，而 x86 平台在较小规模下加速比提升较为明显。
2. 最佳线程数：对于 ARM，最佳线程数在 8 线程左右，1000 规模尤其明显。对于 x86，随着线程数的增加，加速比提升较为平稳，但 8 线程时加速比最高。
3. 线程扩展性：ARM 平台在较小规模下线程扩展性较好，但在较大规模下加速比提升有限。x86 平台的线程扩展性较为一致，随着线程数增加，加速比稳步提升。

4.4 与 SIMD 结合

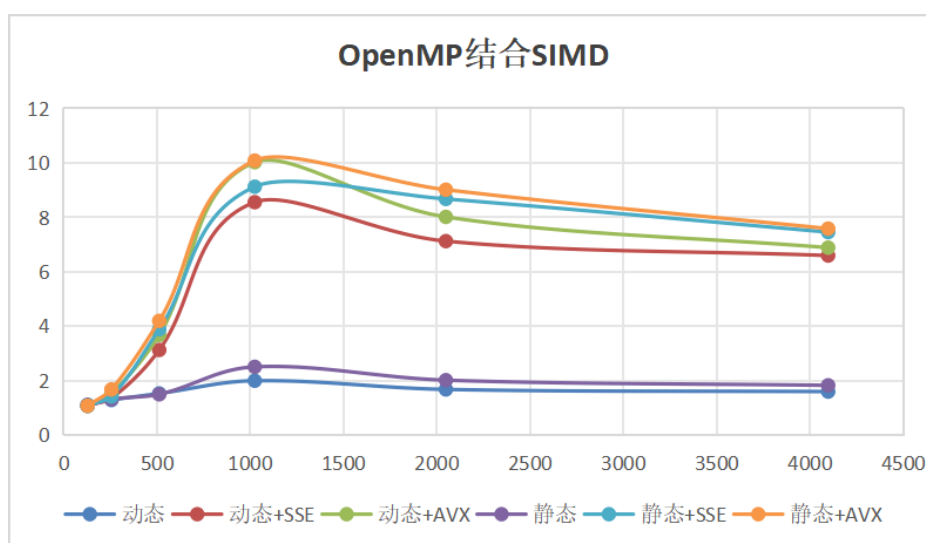


图 4.17: OpenMP 结合 SIMD

与 Pthread 实验同理，AVX 结合效果最好，SSE 次之，可以从 perf 数据中细致观察，静态 +AVX 的 CPI 最高 4.73，结合 SSE 的动态分配和静态分配的 CPI 仅为 1.4 和 1.96。

	指令数	周期数	CPI
串行	2973209222	1096552996	0.368811245
动态	8821796942	6373696096	0.722494083
动态 SSE	798060768	1119310306	1.40253769
动态 AVX	444581465	825302684	1.856358731
静态	8760436200	5282370667	0.602980325
静态 SSE	715644849	1399838921	1.956052535
静态 AVX	730587486	3462901448	4.73988607

表 3: OpenMP 结合 SIMD 的 profiling

5 普通高斯消元多线程优化总结

5.1 Pthread 和 OpenMP 性能差异

综合上述 Pthread 和 OpenMP 的实验结果，我们观察整体趋势，并且找到性能最优的拐点。

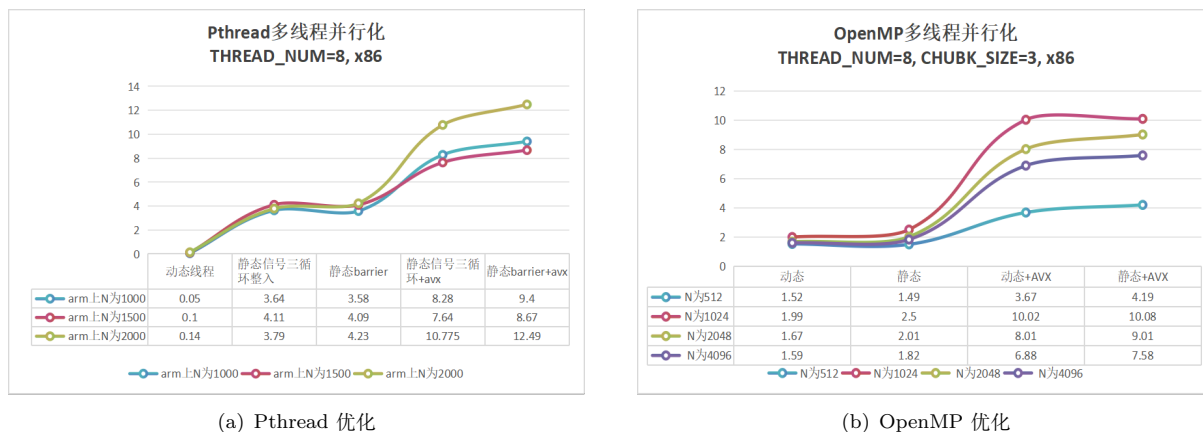


图 5.18: Pthread 与 OpenMP 对比

Pthread 最高性能：N=2000，结合 avx，八线程，静态分配 +barrier 同步，加速比 12.49

OpenMP 最高性能：N=1024, chunksize=3，结合 AVX，八线程，static 分配，10.08

总体来说，Pthread 性能和 OpenMP 差不多，在结合 barrier 同步机制达到了 12.49，两者差异有以下几点。第一，性能开销：OpenMP 的运行时库针对并行循环和区域进行了优化，可以减少线程创建和管理的开销。pthread 的开销可能会更高，特别是如果程序员没有有效地管理线程生命周期和同步的话。第二，可扩展性：OpenMP 在处理大规模并行问题时通常表现良好，特别是在多核处理器上，它可以自动地将工作分配给可用的核心。pthread 也可以很好地扩展，但这需要手动进行更多的工作，例如动态地创建和销毁线程。

5.2 cache 优化和按列划分

在普通高斯消去中，对于算法的消去部分，既可采用水平划分（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。

按列划分减少不同线程间的同步需求和数据竞争，所以对比普通静态分配按行划分和按列划分，可以发现按列是非常低耗时耗时的方法，和普通串行算法相比，加速比接近 0。同时对按列划分进行 cache 优化算法，考虑到了缓存的层次结构和数据访问模式，这可以提高缓存命中率，因为同一列的数据更可能被加载到同一缓存行中，加速比到 6 点多。如图 5.19

在与 SIMD 结合时，SIMD 只能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，故按行划分结合 neon 指令集，性能比按列划分 cache 优化要好，最高可接近 9。

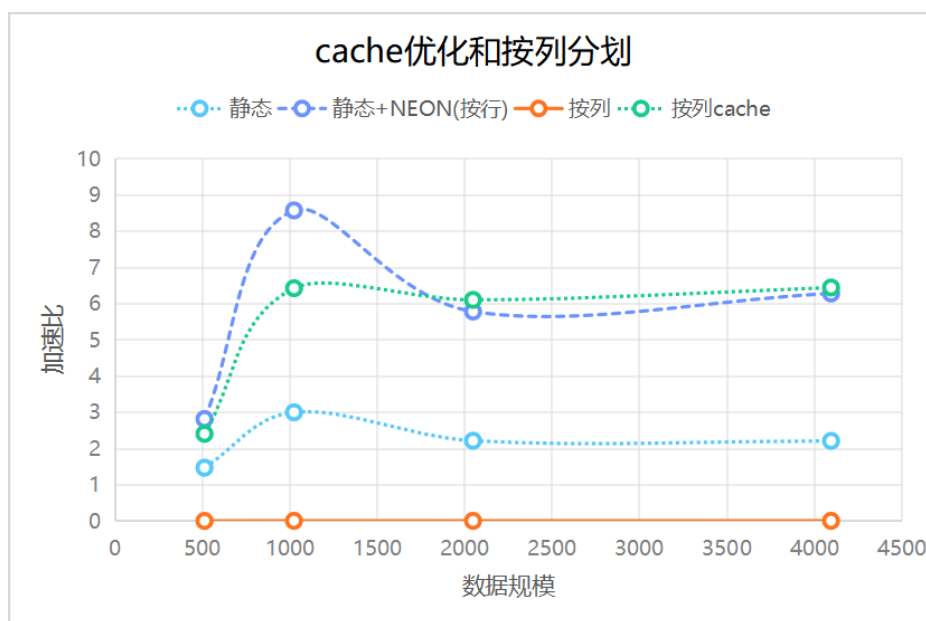


图 5.19: 按列划分和 cache 优化

6 基于 Gröbner 基计算的特殊高斯消元

6.1 问题分析

特殊高斯消元算法在有限域 $GF(2)$ 上进行，主要涉及两个操作：消元子和被消元行的操作。该算法的流程是：

1. 将文件中的消元子和被消元行读入内存。
2. 对每个被消元行，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）或首项在范围内但无对应消元子或该行。
3. 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。
4. 重复第二步和第三步的过程，直至所有被消元行都处理完毕。

其串行算法的代码如下所示，其中 $R[i]$ 代表首项为 i 的消元子， $E[i]$ 代表第 i 个被消元行， $lp(E[i])$ 代表被消元行第 i 行的首项。

6.2 pthread 优化思路

在多线程高斯消元算法中，我们的目标是同时处理多行以加速计算。但这样做可能会引起冲突，特别是当消元后多行的首项系数相同时，若直接将任一行设为消元行，会导致错误。为解决这个问题，我采用了互斥锁（Mutex）来管理线程对消元行的访问：

- 当一行准备成为消元行时，它会先锁定互斥锁，防止其他线程同时修改消元行。
- 锁定后，该线程会更新消元行，并在完成后释放锁。

Algorithm 2 串行特殊高斯消元算法

```

1: for  $i$  from 0 to  $m - 1$  do
2:   while  $E[i] \neq 0$  do
3:      $x \leftarrow \text{lp}(E[i])$ 
4:     if  $R[x] \neq \text{NULL}$  then
5:        $E[i] \leftarrow E[i] - R[x]$ 
6:     else
7:        $R[x] \leftarrow E[i]$ 
8:     end if
9:     break
10:  end while
11: end for
12: return  $E$ 

```

- 其他线程在锁被释放后，会再次检查是否有重复的首项系数，如果有，它们将重新进行消元；如果没有，它们将更新消元行并锁定互斥锁。

简而言之，互斥锁确保了消元行的更新是顺序且互斥的，从而避免了多线程环境下的数据冲突和错误。

6.3 OpenMP 优化思路

OpenMP 优化思路与 pthread 相同，从此前探究中能够得知 guided 的任务划分方式在大规模的循环运算中相较 static 和 dynamic 有更好的表现，因此选用 guided 任务划分方式。

使用 `#pragma omp for ordered schedule(guided)` 结合 `#pragma omp ordered` 既保证了没有数据冲突冒险——只有一个线程执行写入消元子，又保证了写入消元子的顺序依赖不被打破。经测试，这样的方式能准确得到结果。代码见特殊高斯消元算法的 OpenMP 优化。

此外，我还尝试了 pthread、OpenMP 与 SIMD 指令集的结合，探究如何进一步提升加速比，选取的是目前更新、更有运用价值的 AVX 指令集。

6.4 实验结果

耗时对比

优化类型	例编号										
	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10	例 11
普通算法	0	3	3	81	355	3999	24745	186101	279961	878052	485
平凡算法 (pthread 优化)	0	1	1	46	238	2704	16967	123238	185819	578223	198
平凡算法 (OpenMP 优化)	0	0	1	8	39	533	3141	26111	44197	132280	36
AVX 优化 (pthread)	0	1	1	12	60	734	4664	32222	53451	222136	80
AVX 优化 (OpenMP)	0	0	0	7	30	505	3716	27781	46044	151274	35

表 4: 特殊高斯消元的 pthread、OpenMP 优化，结合 AVX 指令集

由于例 1 至例 3 加速到已经无法测得具体耗时，因此将从例 4 开始讨论加速比。仅从数据表 4 而言，可见运用 pthread 和 OpenMP 进行多线程优化后运行耗时明显降低了许多，且由于 pthread 和 OpenMP 均用 8 线程进行并行加速，所以二者在很多时候表现相近。

加速比对比

各样例下的加速比如下；

Optimization	Example Number							
	4	5	6	7	8	9	10	11
Single Thread AVX	1.76	1.49	1.48	1.46	1.51	1.51	1.52	2.45
Pthread Plain	4.50	4.55	5.68	5.62	6.01	5.49	5.47	5.84
Pthread AVX	10.12	9.10	7.50	7.88	7.13	6.33	6.64	13.47
OpenMP Plain	6.75	5.92	5.45	5.31	5.78	5.24	3.95	6.06
OpenMP AVX	11.57	11.83	7.92	6.66	6.70	6.08	5.80	13.86

表 5: 特殊高斯消元多线程优化加速比

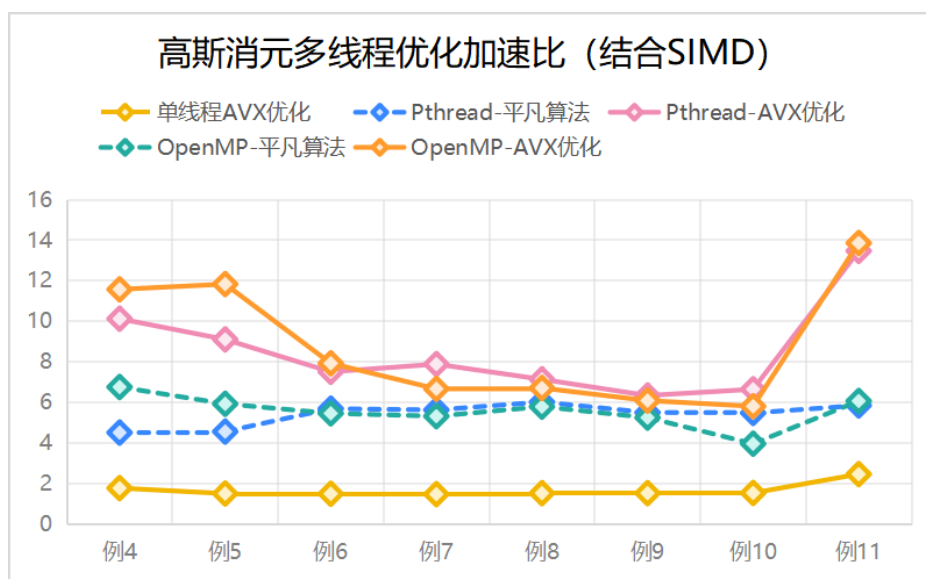


图 6.20: 高斯消元多线程优化加速比 (结合 SIMD)

可以看见，相比于单纯的 AVX 指令集的 SIMD 优化，加入 pthread 和 OpenMP 后性能获得了很大的提升，从实验结果表5和图像6.20中可以获得以下结论：

1. 多线程优化的重要性：实验显示，单纯的 AVX SIMD 优化带来的性能提升有限，而结合多线程优化后，性能提升显著，加速比可高达 7 倍甚至更多。
2. 线程库的选择：在轻量级计算任务中，OpenMP 优化效果普遍优于 pthread，显示出其在任务调度和执行上的高效性。对于计算量大的任务，pthread 优化表现更佳，这可能是因为 pthread 提供了更接近底层的线程管理能力。
3. 多线程与 SIMD 优化的比较：即便对于平凡算法，多线程优化也能带来显著的加速效果，有时甚至超过 SIMD 优化，这表明多线程优化在某些情况下更为有效。
4. SIMD 优化的局限性：SIMD 优化主要针对数据并行性，对于循环的串行执行部分无法提供加速，这是其性能提升有限的原因之一。
5. 优化策略的选择：根据任务的计算量和特性，选择合适的优化策略至关重要。轻量级任务可能更适合 OpenMP，而重计算任务则可能需要 pthread 来实现更优的性能。

总之，此次特殊高斯消元多线程加速达到了很好的效果。相比上一次的 SIMD 优化，多线程达到了更好的提升幅度，而二者结合之后效果更加显著，在面临巨大计算量时也能有 6 到 7 的加速比，而在任务量较轻时能达到超过 10 的加速比，说明这一次的多线程加速达到了很好的效果。

7 总结与反思

本次实验尝试了 Pthread、OpenMP 两种多线程编程方式，均证明了多线程优化的性能优越性，且在 x86 和 arm 上分别对比了多线程实现的平台差异性，在普通高斯消元最后总结 Pthread、OpenMP 手动和自动的区别，结合上个实验 SIMD 的指令集优化，探究线程数目对程序性能的影响。本次实验在 cache 优化和行划分、列划分方法中做了讨论，并且对特殊高斯消去进行多线程优化。

本次实验的细致度较上一个实验又有所提升，但是由于刚开始对多线程编程的不熟悉，导致进行实验的时候 debug 较为困难，得到实验数据的准确性不高，返工情况较多。在实验中逐渐养成了规划好再动手，扎实编程基础的习惯。

代码详见：[Pthread、OpenMP 并行优化实验](#)