南開大學
Nankai University

计算机学院

并行程序设计作业报告

pthread 多线程编程

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 5 月 5 日

目录

1 普通高斯消去算法

1.1 实验介绍

高斯消元法（Gaussian Elimination）是数学上线性代数中的一个算法，可以把矩阵转化为行阶梯形矩阵。高斯消元法可用来为线性方程组求解，求出矩阵的秩，以及求出可逆方阵的逆矩阵。

本实验将在 ARM 平台上利用 pthread 编程技术进行普通高斯算法的多线程优化实验，从而对 pthread 多线程编程有更加深刻的认识与理解。

1.2 程序设计思路

1.2.1 算法分析

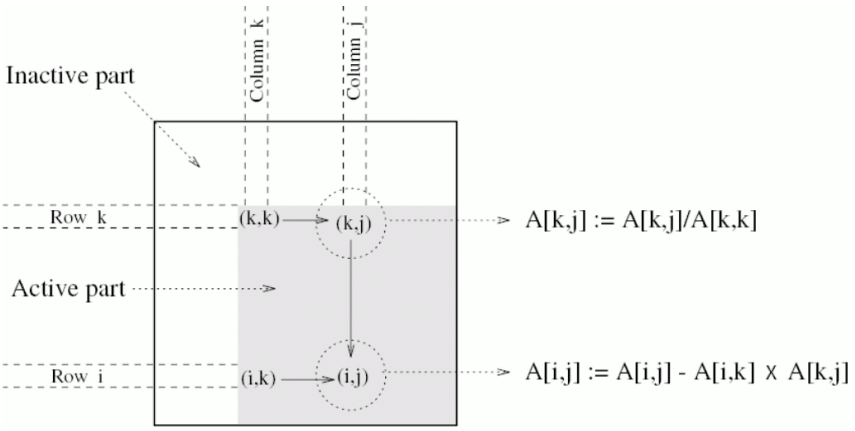

图 1.1: 高斯消去算法示意图

高斯消去的计算模式如图1.1所示，在第 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，串行算法如下面伪代码所示。

```
1  procedure LU (A)
2  begin
3      for k := 1 to n do
4          for j := k + 1 to n do
5              A[k, j ] := A[k, j] / A[k, k];
6          endfor;
7          A[k, k] := 1.0;
8          for i := k + 1 to n do
9              for j := k + 1 to n do
10                 A[i, j ] := A[i, j ] - A[i, k] * A[k, j];
11             endfor;
12             A[i, k] := 0;
13         endfor;
14     endfor;
15 end LU
```

由高斯消去伪代码容易推出该算法时间复杂度为 $O(n^3)$。观察高斯消去算法，可以对内层除法循环后的内层二重循环进行 pthread 多线程优化。具体代码见文末 GitHub 链接。

1.3 实验环境说明

实验中的 ARM 部分为鲲鹏服务器的 ARM 平台，利用 ARM 平台的 g++ 编译器进行程序的 pthread 多线程并行优化；实验中的 x86 部分为个人 PC，CPU 参数如表1

CPU Name	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
NumberOfCores	4
NumberOfLogicalProcessors	8

表 1: 个人 PC CPU 参数

1.4 实验方案设计

pthread 编程中，程序的运行时间与问题规模和线程数有直接关联。同时，采取不同的多线程开辟机制（动态多线程和静态多线程）、线程同步机制（信号量和 barrier）以及不同的任务分配方式（行划分、列划分）也会对程序性能造成一定的影响。在多线程中适当的加入 SIMD 编程也有可能对使程序性能得到较好的提升。因此，本次实验将分为如下四部分进行：

- Part 1 探究不同的线程开辟机制和线程同步机制以及不同线程数对程序性能的影响。具体方法为对比平凡算法、动态线程算法、静态线程信号量同步算法、静态线程 barrier 同步算法在不同问题规模和不同线程数下程序的运行性能的差异。
- Part 2 探究不同的任务分配方式对多线程程序性能的影响。具体方法为基于第一部分得到的较优的多线程编程方法，比较程序在矩阵行划分、列划分的任务分配模式下运行性能的差异。
- Part 3 第三部分我们探究加入 SIMD 优化对多线程程序性能的影响，根据上一次 SIMD 编程实验的经验，我们只需对普通高斯消去算法的内层二重循环进行 SIMD 优化便可得到较好的性能提升，我们将对比普通多线程程序和加入了 SIMD 优化的多线程程序运行性能的差异。
- Part 4 在最后一部分中，我们在 x86 平台上对多线程算法及多线程结合 SIMD 算法进行性能测试，并比较不同平台程序性能的差异。

1.5 实验结果呈现及分析

1.5.1 Part 1 不同的线程开辟机制和线程同步机制以及不同线程数对程序性能的影响

首先，我们分别固定线程数为 2、4、6、8，探究线程数固定的情况下不同程序的性能随问题规模增大的变化（多线程算法均采用行划分）。程序运行时间随问题规模变化趋势如图1.2和图1.3所示。

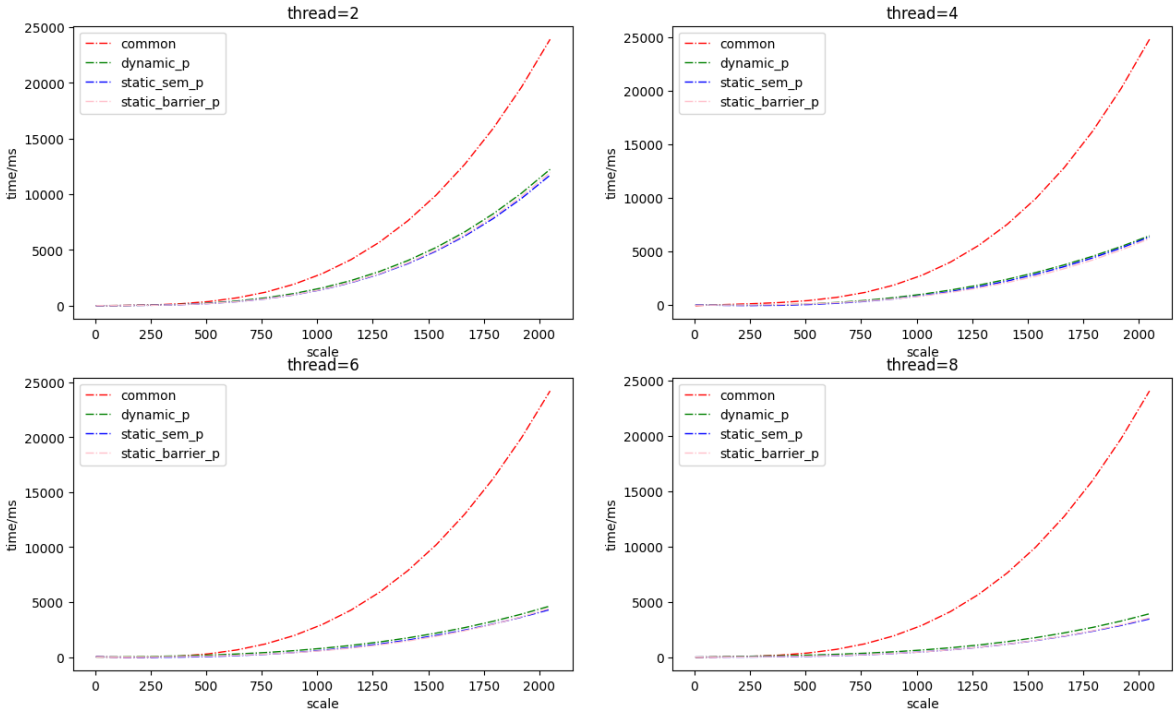

图 1.2: 不同程序固定线程数下运行时间随问题规模变化

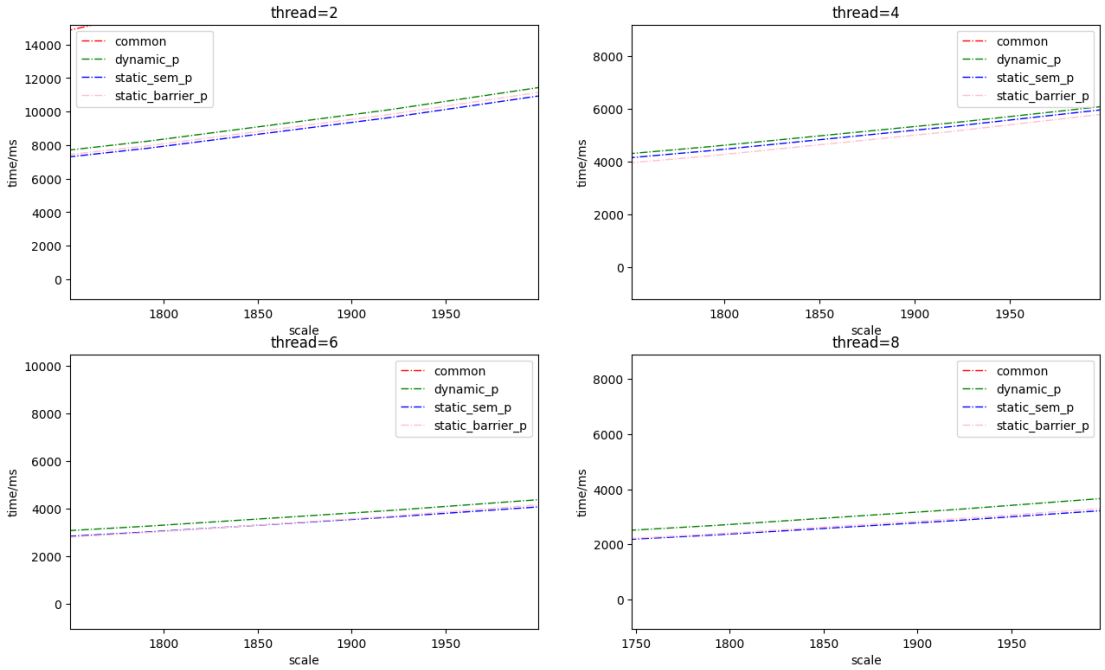

图 1.3: 不同程序固定线程数下运行时间随问题规模变化（大规模下小范围）

对上图结果进行分析，我们可以大致得到以下几点信息：

- 1 问题规模较大的情况下，对于每组给定线程数的程序，pthread 多线程编程均可以起到较为明显的优化效果。
- 2 问题规模较大的情况下，动态开辟线程算法、静态信号量线程同步算法和静态 barrier 线程同步算法运行时间差别不是很大，但动态开辟线程算法运行时间均最大。

- 3 问题规模较大的情况下，随着线程数的增加，多线程程序运行时间似乎也在减少。

下面，我们对程序运行时间进行更细致的分析。

首先，由于图1.2中问题规模跨度较大，问题规模较小时不同算法运行时间的差异不太明显，因此，我们取出线程数为 2 且问题规模较小时程序运行所得时间进行对比分析，结果如表2所示。

算法\规模	32	64	128	256	512
common	0.0844411	0.676742	5.36255	42.8311	356.503
dynamic	2.15934	4.62152	11.5305	40.0969	221.499
static_sem	0.364857	0.929819	3.98568	22.613	176.573
static_barrier	0.461076	1.14316	4.25342	23.2629	179.497

表 2: 小规模下不同算法程序运行时间（单位:ms）

可以看到，当问题规模较小时，动态开辟线程算法相比于其他算法有着较大的运行时间，显然，这是因为频繁地进行线程的开辟和销毁造成的。而随着规模逐渐增大，动态开辟线程算法运行时间开始小于平凡算法运行时间，这可能是因为多线程所带来的性能增益开始大于线程频繁地开辟和销毁所带来的额外开销。且随着问题规模越来越大，多线程带来的性能增益远大于动态开辟线程算法中线程频繁开辟和销毁带来的额外开销（如果问题规模扩大 10 倍，则动态线程多线程算法的线程开辟开销将增长到原来的 10 倍，但由于高斯消去算法的时间复杂度为 $O(n^3)$，此时程序运行时间增长幅度可能远大于 10 倍。此时，算法中的多线程带来的增益可能会占主导地位）。这也解释了问题规模较大时为什么动态开辟线程算法和两种静态多线程算法运行时间差别不是很明显。但是由于频繁开辟线程的额外开销，动态开辟线程算法运行时间始终要大于两种静态多线程算法运行时间。

从表2中我们还可以发现，问题规模较小时，两种静态多线程算法也并未取得较好的加速效果，甚至运行时间大于平法算法的运行时间，这可能是因为线程间同步带来的额外开销大于多线程带来的性能增益。而随着问题规模增大，增益效果也远远大于额外开销。

接下来分析线程数的不同对多线程程序的影响。考虑到三个多线程算法的相似性，这里我们只对静态信号量同步多线程算法进行分析，得到不同规模不同线程数下程序运行时间如图1.4和图1.5所示。

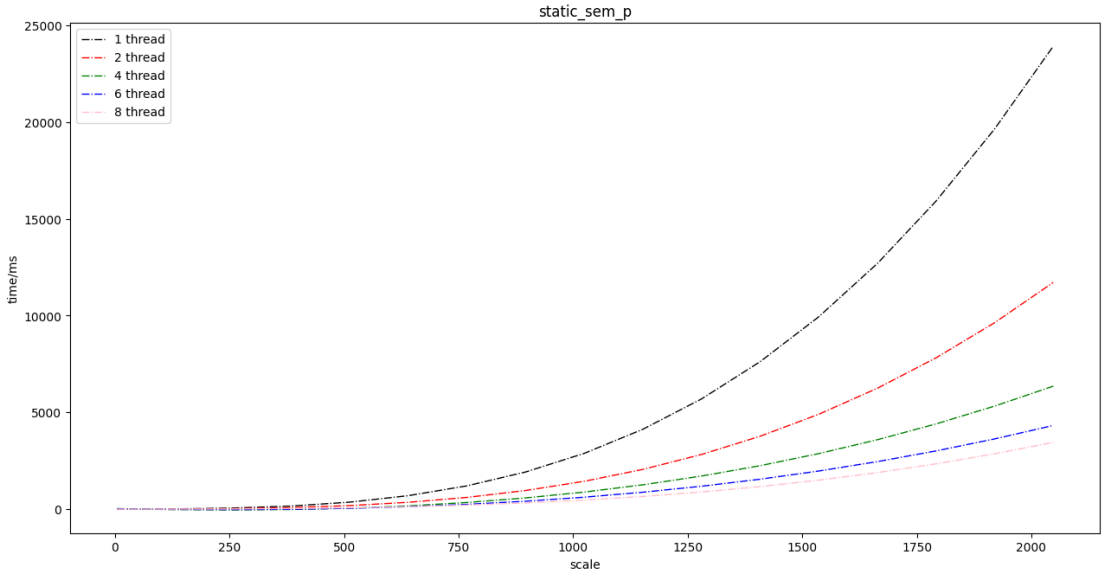

图 1.4: 不同规模不同线程数下静态信号量同步多线程程序运行时间

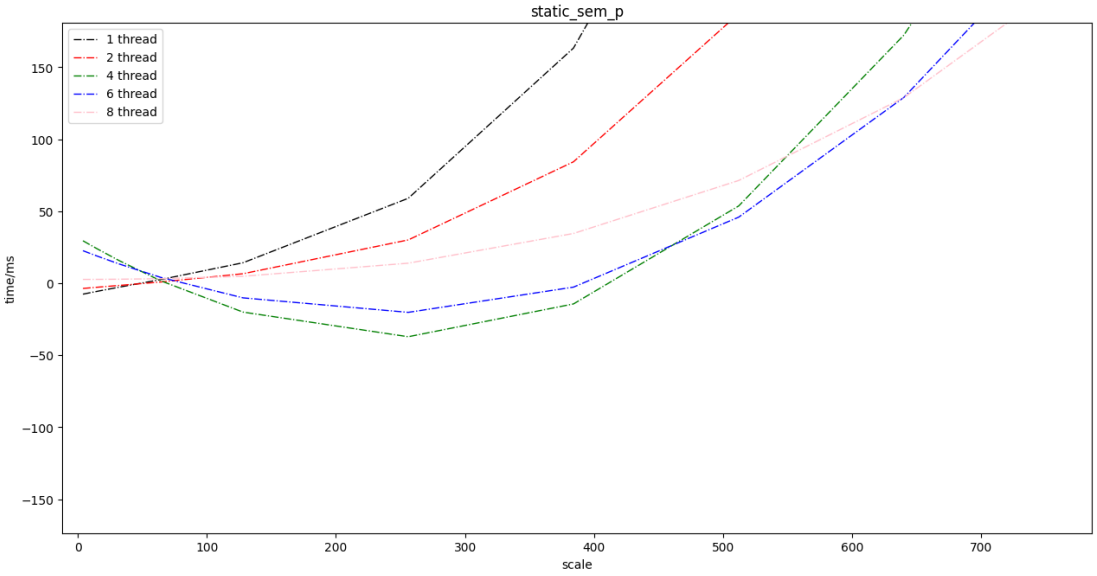

图 1.5: 不同规模不同线程数下静态信号量同步多线程程序运行时间

从图1.4中可以看到，当问题规模较大时，随着线程数的增加，程序的运行时间随之减少，这符合我们的预期；从图1.5中可以看到，当问题规模较小时，多线程程序运行时间要大于平凡算法运行时间，而当问题规模大于大约 70 时，各个线程数下的多线程程序性能开始优于平凡算法程序性能，这也和我们上述的分析一致。

最后的一个问题，随着多线程程序线程数的增加，加速比是如何变化的呢？绘制不同线程数下静态信号量同步多线程程序加速比随问题规模变化图，如图1.6所示。

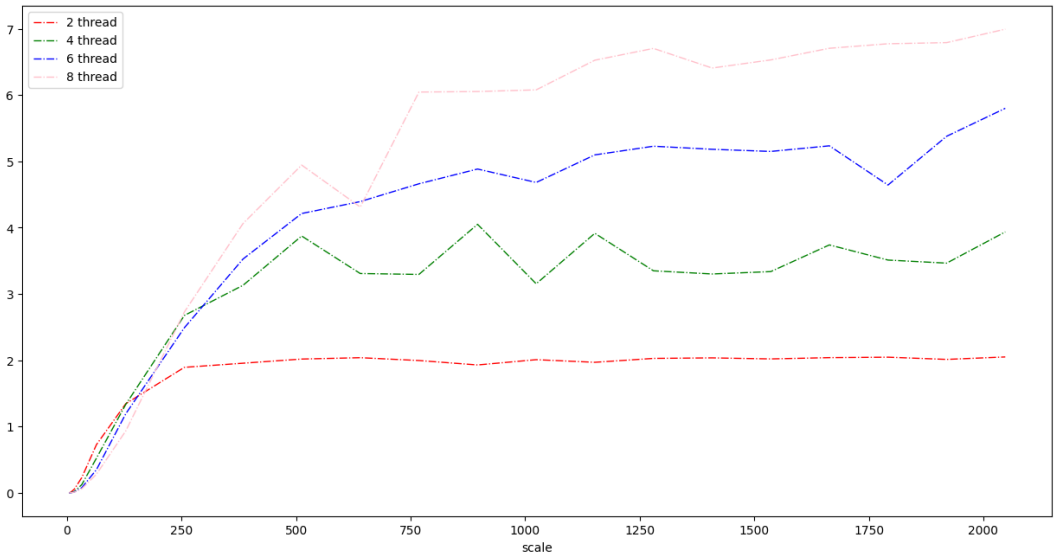

图 1.6: 不同规模不同线程数下静态信号量同步多线程程序运行时间

可以看到，问题规模较大时，每个线程数下的多线程程序加速比趋于稳定，但大体均小于线程数

目。且当线程数为 2 时，加速比与 2 非常接近，这可能是因为线程数少，线程同步开销较小造成的。

1.5.2 Part 2 不同的任务分配方式对多线程程序性能的影响

根据 Part1 的实验结果，两个静态多线程算法性能相近且优于动态多线程算法，本部分我们主要基于静态信号量同步多线程算法探究不同的任务分配方式对多线程程序性能的影响。分别利用行划分和列划分的任务分配方式进行实验。

不同规模不同线程数下行划分与列划分任务分配方式程序运行时间如图1.7所示。

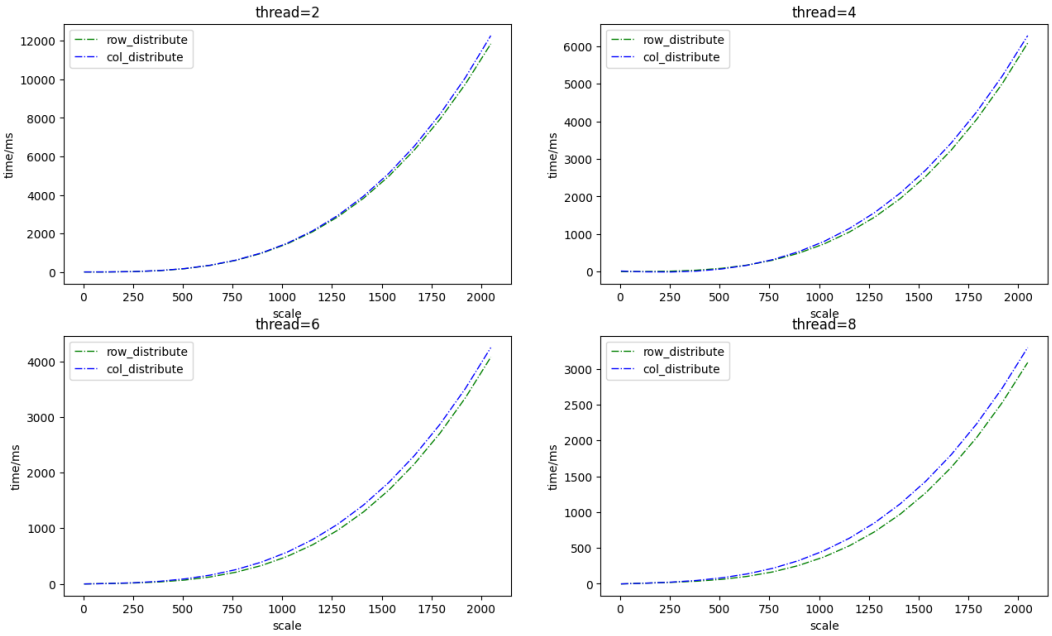

图 1.7: 不同规模不同线程数下不同任务分配方式程序运行时间

结合图1.7和程序运行数据我们可以得到，行划分和列划分的性能差异并不是十分显著，二者应具有相近的加速比。同时，我们发现，在各个给定线程数下，行划分程序的运行时间要略微优于列划分程序的运行时间。考虑负载均衡问题，本次实验中，测试数据为方阵，因此，每一步消去过程中，行划分和列划分各个线程的负载应该是相同的。继续思考，还有一种可能原因是行划分与列划分的缓存利用存在些许差异，因此我们利用 perf 分析具体问题规模下二者的缓存利用情况。

表3为线程数为 2 时行划分和列划分多线程程序中线程函数的 cache miss 数目。可以看到，在各个测试规模下，列划分线程函数的 cache miss 数目均大于行划分行划分线程函数的 cache miss 数目，这与我们的推测相符。由此可见不同的任务划分方式可能会影响到 cache 利用效率等因素。

算法\规模	128	256	512	1024	2048
row_distribute	38,725	279,496	2,413,052	12,873,968	92,243,807
col_distribute	73,390	343,934	2,514,380	13,910,598	98,104,181

表 3: 线程数为 2 时行划分和列划分多线程程序中线程函数的 cache miss 数目

1.5.3 Part 3 SIMD 优化对多线程程序性能的影响

最后，我们探究加入 SIMD 优化对多线程程序性能的影响，在本部分，我们将编写静态信号量同步多线程算法的 SIMD 优化版本，并对两种算法的性能进行对比分析。不同线程数不同问题规模下程

序运行时间如图1.8所示。

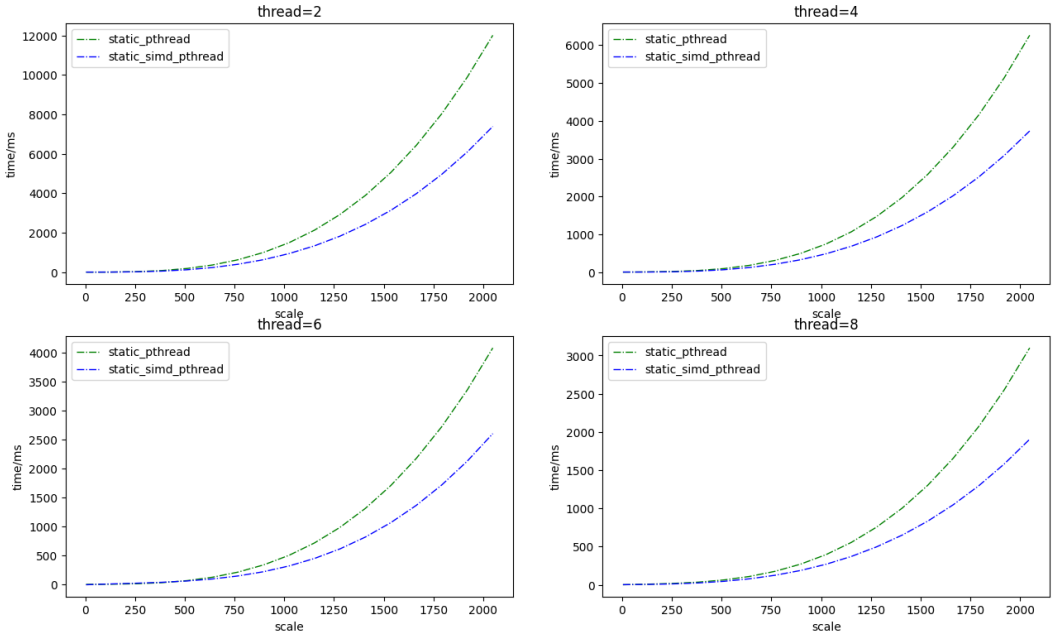

图 1.8: 静态多线程算法与静态多线程结合 SIMD 优化算法运行时间对比

从上图可知，SIMD 优化具有显著的加速效果。显然，这是因为 SIMD 并行化提升了程序的 IPC。图1.9为不同线程数下静态多线程结合 SIMD 优化后的算法的加速比结果。与图1.6作比较，我们发现，较大问题规模下，结合了 SIMD 优化的静态信号量同步多线程算法的加速比已经超过了线程数目，达到了超线性加速比。这也提示我们在对程序进行并行化处理时要综合利用可能的并行化方法以达到较好的加速效果。

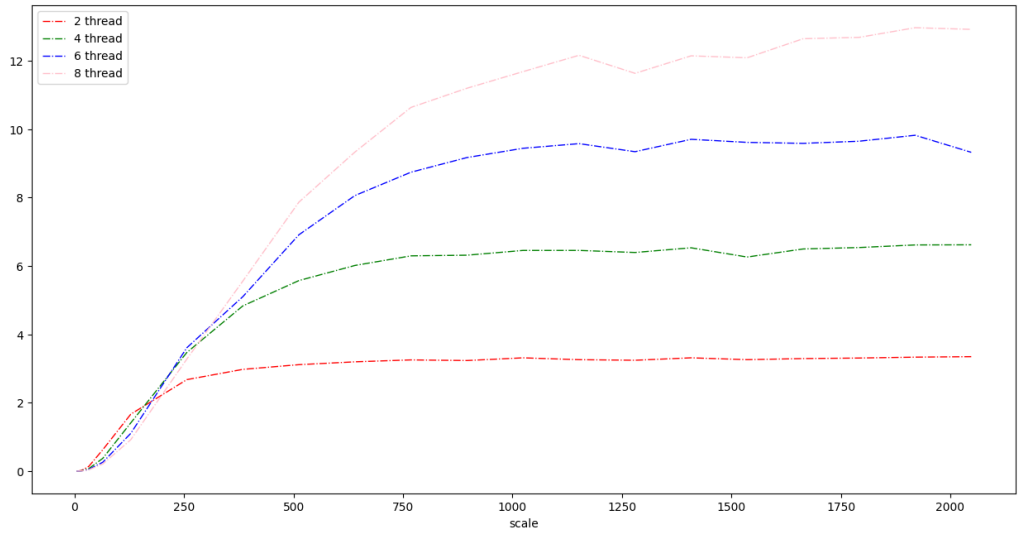

图 1.9: 不同线程数下静态多线程结合 SIMD 优化算法加速比

1.5.4 Part 4 x86 平台程序性能测试

在 x86 平台中，我们主要测试静态信号量同步算法和静态信号量同步结合 SSE SIMD 优化算法的性能。考虑到两平台中程序运行时间随线程数及问题规模变化的变化应具有相似性，我们这里直接对 x86 平台中多线程算法以及多线程结合 SSE 优化算法的加速比进行分析，如图1.10。与 ARM 平台相比，x86 平台中的多线程程序加速比较小，且线程数为 4、6、8 时的加速比区分不是很明显。即使是结合了 SSE 优化，程序也未取得超线性加速比。

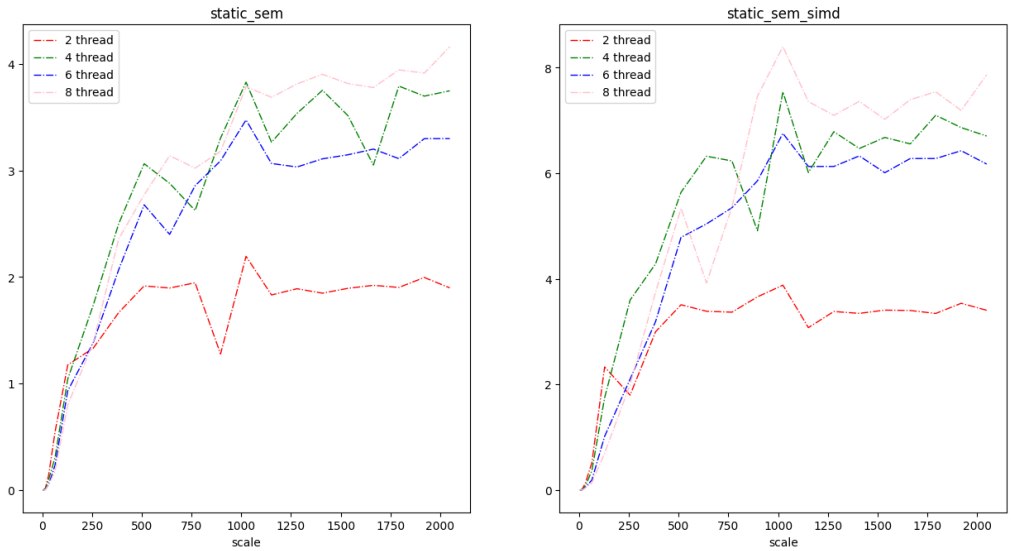

图 1.10: 不同线程数下多线程算法加速比

接下来我们对两平台下不同算法的性能差异进行比较，仍取线程数为 2，如表4所示。

算法\规模		128	256	512	1024	2048
平凡算法	ARM	5.50598	44.0139	367.728	3087.17	24733.8
	x86	2.7924	17.951	157.578	1405.98	10174.5
普通多线程算法	ARM	4.17806	23.7781	175.44	1508.79	12030.1
	x86	2.376	13.473	82.204	640.7	5359.76
多线程 SIMD 优化算法	ARM(NEON)	3.30729	16.4541	118.142	931.977	7391.8
	x86(SSE)	1.1968	9.976	44.882	362.142	2986.63

表 4: 线程数为 2 不同平台算法性能对比（单位：ms）

由上表，程序运行时间随问题规模变化的增长趋势是合理的。但是对比相同算法不同平台上的运行时间差异，显然，x86 平台程序具有更短的运行时间。ARM 平台下程序运行时间甚至超过了 x86 平台下程序运行时间的两倍，这个结果令人感到惊讶。

2 特殊高斯消去算法

2.1 实验介绍

本实验将利用 pthread 编程对特殊高斯消去算法进行优化，并在鲲鹏服务器提供的 ARM 平台中进行程序性能测试。ARM 平台使用 g++ 编译器对程序进行编译。

2.2 程序设计思路

2.2.1 算法分析

首先考虑算法的执行流程，我们需要循环遍历每一个消元行对其进行消元。对于每一个消元行的消元，消元过程首先寻找是否有消元子与该行具有相同的首项，若存在满足条件的消元子，则用相应的消元子对该消元行进行消去，即将消元子的每一位和消元行进行异或操作，之后判断消元行是否消元为 0，若仍未被消元为 0，则重复寻找消元子继续进行消元；若没有满足条件的消元子，则该消元行“升格”为消元子，参与后续消元行的消元过程，并结束该消元行的消元过程。

伪代码如下：

```
1  //R: 所有消元子构成的集合
2  //R[i]: 首项为 i 的消元子
3  //E: 所有被消元行构成的数组
4  //E[i]: 第 i 个被消元行
5  //lp(E[i]): 被消元行第 i 行的首项
6  for i := 0 to m - 1 do
7      while E[i]! = 0 do
8          if R[lp(E[i])]! = NULL then
9              E[i] := E[i] - R[lp(E[i])]
10         else
11             R[lp(E[i])] := E[i]
12             break
13         end if
14     end while
15 end for
16 return E
```

2.2.2 算法设计

首先我们观察平凡算法以寻找算法中可以进行 pthread 多线程优化的部分。平凡算法如下所示，不难发现，平凡算法中只有第 8 到 10 行的对消元行消元的循环可以进行 pthread 优化，即每个线程负责进行消元行中一部分的消元。但是，显然，特殊高斯消去算法有着较为复杂的控制流。因此，对该算法进行多线程并行化可能需要付出较大的用于线程同步的额外开销，有可能多线程并行算法无法取得好的加速效果，经过我们的探索发现，事实也确实如此（不排除设计的算法效率低下的可能）。

```
1  //平凡算法
2  for (int i = 0; i < NUME; i++) {
3      while (lp[i] > -1) { //lp[i] 为-1 则表示该行被消为零
4          if (!(lpE2R.find(lp[i]) == lpE2R.end())) { //寻找符合的消元子
5              int rowR = lpE2R[lp[i]];
6              bool lpHasChanged = false;  //该 bool 变量用于判断本消元行是否消元为零
7              int p = totalCols - 1;
```

```
            for (p; p >= 0; p--) {
                A[i][p] ^= A[rowR][p]; //消元过程
            }
            for (p = totalCols - 1; p >= 3; p -= 4) {
            //此循环及接下来的循环为更新该消元行的首项值
            //此处为了加快速度使用循环展开
                int x = ((A[i][p] | A[i][p - 1]) | (A[i][p - 2] | A[i][p - 3]));
                if (x == 0)
                    continue;
                break;
            }
            for (p; p >= 0; p--) {
                if (A[i][p] != 0) { //消元行的首项值一定在第一个不为零的 32 个 bit 中
                    lpHasChanged = true;
                    for (int k = 31; k >= 0; k--) {
                        if ((A[i][p] & (1 << k)) != 0) {
                            lp[i] = p * 32 + k;
                            break;
                        }
                    }
                    break;
                }
            }
            if (lpHasChanged == false) //判断该消元行是否被消元为零
                lp[i] = -1;
            }
        else {
            lpE2R[lp[i]] = i;   //找不到合适的消元子，该消元行成为消元子
            break;
        }
    }
}
```

以下为我设计的特殊高斯消去多线程算法代码。可以看到，为了进行同步，算法中使用了四个 barrier。

```
void *thread_func(void *param) { //线程函数
    long long t_id = (long long)param;
    for (int i = 0; i < NUME; i++) {
        while (lp[i] > -1) {
            if (!(lpE2R.find(lp[i]) == lpE2R.end())) {
                int rowR = lpE2R[lp[i]];
```

```
                int p;
                for (p = t_id; p < totalCols; p += thread_count) {
                    A[i][p] ^= A[rowR][p];
                }
                pthread_barrier_wait(&barrier1);
                if (t_id == 0) {
                    bool lpHasChanged = false;
                    for (p = totalCols - 1; p >= 3; p -= 4) {
                        int x = ((A[i][p] | A[i][p - 1]) | (A[i][p - 2] | A[i][p - 3]));
                        if (x == 0)
                            continue;
                        break;
                    }
                    for (p; p >= 0; p--) {
                        if (A[i][p] != 0) {
                            lpHasChanged = true;
                            for (int k = 31; k >= 0; k--) {
                                if ((A[i][p] & (1 << k)) != 0) {
                                    lp[i] = p * 32 + k;
                                    break;
                                }
                            }
                            break;
                        }
                    }
                    if (lpHasChanged == false)
                        lp[i] = -1;
                }
                pthread_barrier_wait(&barrier2);
            }
            else {
                pthread_barrier_wait(&barrier4);
                if (t_id == 0)
                    lpE2R[lp[i]] = i;
                pthread_barrier_wait(&barrier3);
                break;
            }
        }
    }
    return NULL;
}

```

```
49  void static_gausst_p() { //主函数
50      pthread_barrier_init(&barrier1, NULL, thread_count);
51      pthread_barrier_init(&barrier2, NULL, thread_count);
52      pthread_barrier_init(&barrier3, NULL, thread_count);
53      pthread_barrier_init(&barrier4, NULL, thread_count);
54      pthread_t *handles = new pthread_t[thread_count];
55      for (int i = 0; i < thread_count; i++)
56          pthread_create(handles + i, NULL, thread_func, (void *)(long long)i);
57      for (int i = 0; i < thread_count; i++)
58          pthread_join(handles[i], NULL);
59      delete[] handles;
60      pthread_barrier_destroy(&barrier1);
61      pthread_barrier_destroy(&barrier2);
62      pthread_barrier_destroy(&barrier3);
63      pthread_barrier_destroy(&barrier4);
64  }
```

2.3 实验环境说明

实验在鲲鹏服务器中进行，使用 g++ 编译器编译程序。

2.4 实验方案设计

实验原计划在课程提供的测试数据的前七组数据集上进行平凡算法和多线程算法以及多线程结合 SIMD 优化算法的性能测试。但由于在实验过程中发现多线程算法的性能要远逊于平凡算法的性能，因此不再继续尝试进行多线程程序的 SIMD 优化实验。

2.5 实验结果呈现及分析

实验结果数据如表5所示。可以看到，在各个数据集下，多线程算法的运行时间均远大于平凡算法的运行时间，在某些数据集下运行时间甚至为平凡算法的 10 倍左右，可见线程间同步开销很大。此外，观察多线程算法不同线程数目下程序运行时间，我们发现，随着线程数目的增多，程序运行时间并没有像我们期望的那样减少，而是随着线程数目的增加而增长，原因可能为线程数目越多同步开销越大而带来的增益却并不显著。

算法\数据集编号		1	2	3	4	5	6	7
common		0.0183503	0.70837	0.877486	25.6882	148.768	2475.54	27832.6
pthread	thread=2	0.388963	11.3767	10.1758	243.428	1203.14	11894.9	101095
	thread=4	0.634828	14.8964	15.0408	423.547	1649.8	20868.8	138333
	thread=8	1.2467	34.3823	32.7167	889.394	3594.53	42285.6	262860

表 5: 特殊高斯消去不同算法运行时间（单位：ms）

得到如上的实验结果，我认为原因可能有两个：一是特殊高斯消去算法不适合进行 pthread 多线程优化；二是我设计的 pthread 多线程算法太过丑笨低效（此原因的可能性更大）。接下来我会继续对特殊高斯消去算法的多线程优化进行尝试和探索，希望可以找到一个好的实现方法。

3 实验源码项目 GitHub 链接

实验源码项目 GitHub 链接