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

Pthread 编程实验

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 5 月 5 日

摘要

本实验基于两种高斯消去算法，进行 **pthread** 并行实验。分别对**普通高斯消去**和**特殊高斯消去**两种算法，实现了串行算法和 **pthread** 的并行优化。

关键字：pthread，普通高斯消去，特殊高斯消去

目录

一、 普通高斯消去

(一) 问题简介

给定一个满秩矩阵，从上到下依次对它的每行进行除法（除以每行的对角线元素），然后对矩阵该行右下角 (n−k+1)×(n−k) 的子矩阵进行消去，最后得到一个三角矩阵。

(二) 实验设计

1. 总体思路

在设计之前，首先应该从总体上理清普通高斯消去程序的依赖关系，并据此分别找出串并行的部分：

分析普通的串行程序，程序由外层一个大循环和它内部的一个二重循环和一个三重循环构成。最外层大循环控制对矩阵每行的操作，表示对矩阵的该行进行除法并用该行对其他行进行消去，大循环之中有一个二重循环和一个三重循环，二重循环进行除法，三重循环进行消去。

在每轮操作中存在着先后关系：需要首先对该行进行除法，然后对它右下角的矩阵分别用该行进行消去。消去依赖于除法，而除法和消去内部没有必然的顺序关系。

因此，pthread 程序的整体思路为：对矩阵每行先进行除法，此时负责消去的线程部分睡眠等待，除法结束后唤醒消去部分的线程进行消去操作，等待所有线程完成消去后，一起进入下一轮，重复上述步骤。如图1所示。

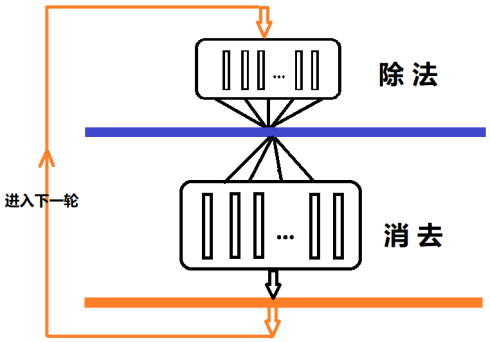

图 1: 总体思路

在《SIMD》实验中，我们已经讨论过除法和消去两部分对总体性能的影响，结果证明，消去部分对程序整体性能影响占比很大，而除法对整体性能影响很小。并且考虑到分配线程、同步机制需要耗费一定时间，所以除法部分的多线程并行操作在某些情况下可以省略。下面的章节里，我们将对此进行更多探讨。

再具体到细节问题，需要考虑如何对任务进行划分、如何进行信息的交互与同步等，并且要把 SIMD 与多线程结合起来。具体分析见下面小节。

2. 不同 Pthread 编程范式

- 动态线程

动态线程是指，在程序运行过程中，反复地创建线程——> 并行操作——> 销毁线程。在没有并行计算需求时不会占用系统资源，在程序达到并行运算部分时创建线程，在并行部分结束销毁线程。

在普通高斯消去的程序中，考虑到如果对除法部分也进行多线程化，则在除法和消去部分都要创建、销毁线程，开销过大；并且此时程序本身的并行度会很高、串行度很低，导致动态线程只是在反复地创建、销毁线程，不能发挥它“没有并行计算需求时不会占用系统资源”的优势，反而放大了它“有较大的线程创建和销毁开销”的缺点。因此，动态线程实验只对消去部分多线程化。

具体设计时，在线程函数里只用写消去操作，而在主程序写外层大循环以及第一个除法操作。在主程序控制大循环每轮的执行，每次循环都要进行除法操作——创建线程——分配任务——多线程并发进行消去——销毁线程，然后进入下一轮。

- 静态线程

静态线程需要在程序初始化的时候就创建好所有会用到的线程。在整个程序中，需要并行时，将任务划分给这几个线程，不需要并行时，并不结束线程，等待下一个并行部分继续为线程分配任务。静态线程相比于动态线程，重点需要考虑线程间的通信与同步，在实验时，将会对以下几种同步机制进行实现与分析。

3. 同步机制

- 3.1 信号量

对于普通高斯的程序而言，需要进行两次同步，一个是在除法结束的时候，另一个是在每一轮结束的时候。对于第一个同步，在除法线程进行除法前，要让其余线程先 wait，等到除法线程结束，除法线程发出 post 信号通知消去线程。这就需要一组信号量 1（每个线程有自己对应的信号量）。对于第二个同步，每个消去线程完成消去后 post 信号量 2 并进入睡眠等待，等待所有消去线程都完成消去并 post 信号量 2 进入睡眠后，除法线程苏醒，然后除法线程 post 信号量 3 唤醒休眠的消去线程。进入下一轮。

（下面的讨论都是基于一个除法线程的情况，多个除法线程在后面讨论）

根据上述分析，共需要一个 + 两组信号量。一个信号量：是对于一个除法线程而言的，对应于上面讨论的信号量 2；两组信号量：是对于多个消去线程而言的，对应于上面的信号量 1 和 3。

因此，在程序执行前，首先应该定义并初始化需要的这些信号量，值得注意的是，我们要将信号量初始化为 0，使得线程先进入等待，只有有了 post 出现，信号量才能结束等待，执行下面的操作。

- 3.2 barrier

barrier 可以理解为原地集合后解散。对于一群线程，barrier 意味着任何线程执行到此后必须等待，直到所有线程都到达此点才可继续执行下面的操作。

对于普通高斯消去程序，共需要两个 barrier。具体思路是把整个大循环都放入线程函数中，在第一个除法循环后面，添加第一个 barrier，在一轮大循环结束时，添加第二个 barrier。

4. 不同任务划分方法

程序分为除法和消去两个循环，对于除法循环而言，只涉及对矩阵里面一行的操作，只可能采用垂直划分。对于消去循环，考虑如下两种方法：

- （1）水平划分

消去循环涉及右下角的子矩阵，水平划分可以穿插着划分（例如第 0、7、14 行... 为一个任务，1、8、15... 为另一个任务）或者按块划分（例如第 0　n/7 行一组、第 2n/7 行　3n/7 行一组...），从

负载均衡的角度考虑，如果按块划分，会导致最后一组过多或过少，而穿插着划分，可以使得负载相对均衡。从体系结构的角度考虑，块划分的 cache 命中率应该更高，空间利用率会更好。

并且，无论是穿插着划分还是按块划分，都是对每行进行操作，都可以与 SIMD 结合，将行内连续元素打包向量化运算。

- （2）垂直划分

垂直划分也可以分为穿插划分和块划分，但是如果采用穿插划分或者块划分的块较小，则不能与 SIMD 结合，因为没有每行连续的元素可以进行打包。而对于块划分来说，若问题规模较大且线程数适当，划分出的块大小合适，则也可以与 SIMD 结合，但负载均衡性可能会有所下降。

但是，相比于水平划分，垂直划分对 cache 的利用率有所下降，cache 需要不断地将矩阵整行载入、移出或者每次操作到的矩阵元素不在最近的 cache 里，都会导致性能不佳。

(三) 理论分析

在此我们先讨论以下几种方法，对于其他的策略，将在后文讨论。

1. 动态线程

以线程数 $= \theta$ 为例，对于动态线程，我们采用除法单线程、消去多线程的方法。共有 n 个外层大循环，故进行 n 次除法操作，每个循环都会创建 θ 个线程、销毁 θ 个线程。在最理想的负载均衡以及忽略通信、函数调用等开销的情况下，动态线程用时可以大致估算为：

$n \times \theta \times$ 销毁时间 $+ n \times \theta \times$ 创建时间 $+ n \times$ 除法时间 $+ n \times$ 消去时间 $/ \theta$

相比于平凡算法而言，动态线程的方法缩短消去时间到原来的 $1 / \theta$，即减少了 θ - $1 / \theta$ 的消去时间，增加了 $n \times \theta$ 次创建销毁线程的开销。分析得到：当问题规模 n 越大，消去和创建、销毁线程的时间都会增大。虽然减少的和增加的时间都呈现上升趋势，但创建销毁线程是 O(n) 级别的增长，而消去时间是 $O(n^3)$ 级别的增长，所以推测总体呈现，规模越大，加速比越高的趋势。

2. 静态线程 + 信号量

- （1）主线程除法，工作线程消去

如图2，主线程先进行除法，工作线程等待；主线程完成除法后，通知工作线程开始消去，同时自己进入等待；工作线程完成消去之后依次通知主线程并进入等待，等到所有工作线程都完成消去并等待的时候，主线程被唤醒，依次通知所有工作线程一起进入下一轮。

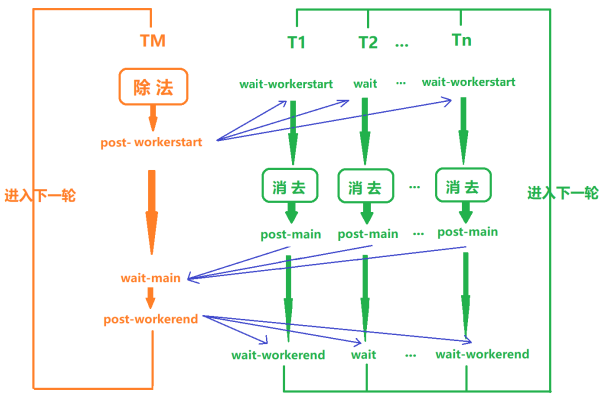

图 2: 信号量 1

- （2）主线程不做运算，工作线程除法 + 消去

如图3，主线程只需要控制所有工作线程的创建和销毁。其中 1 个工作线程既进行除法又进行消去，其余的工作线程只进行消去。程序运行时，线程 T0 先做除法，其他线程等待；除法完成后，T0 通知其他线程结束等待，然后 T0 与其他线程一起进行消去，T0 消去之后等待其他线程都完成消去再一起进入下一轮。

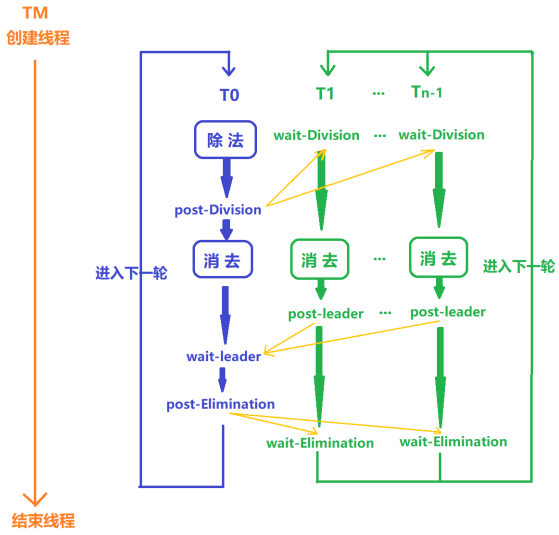

图 3: 信号量 2

对比（1）和（2），当创建的工作线程数都是 θ 时，二者的区别在于（1）的除法线程与消去线程分离，（2）的除法和 1 / θ 的消去可以在同一个工作线程完成。两者的消去时间都减少到了 1 / θ，而除法都是一个线程进行。而对于每次同步来讲，（2）的线程间通信要比（1）减少一次，减少了“T0 除法完成——T0 消去开始”和“T0 消去结束——T0 下一轮开始”的通信。

3. 静态线程 + barrier

barrier 只需整个循环纳入线程函数，同时加入两个同步点即可。理论上讲，创建的工作线程数是 θ 时，它的消去时间也可以减少到原来的 1 / θ。

（四） 实验内容及算法实现

首先，结合 SIMD，我们在不同线程数、不同问题规模下采用水平划分矩阵的方法对动态线程、静态线程 + 信号量、静态线程 +barrier 算法进行了实现，并且对比了这几种方法。

而后，我们针对【静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数】这一版本的算法，实现块划分和穿插划分这两种垂直划分的方法。由于代码较多，不在报告里具体展示，可以在https://github.com/hanmaxmax/parallel_homework3中详见代码。

此外，针对不同的指令集和平台，我们用【静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数】这一版本的算法，测试了 x86 平台下，结合 sse、avx 指令集的程序性能。

（五） 实验结果及分析

1. 基于问题规模和线程数

动态线程的实验结果如图4。左图为不同规模和线程数下的运行时间，右图为不同规模下，不同线程数时间之比

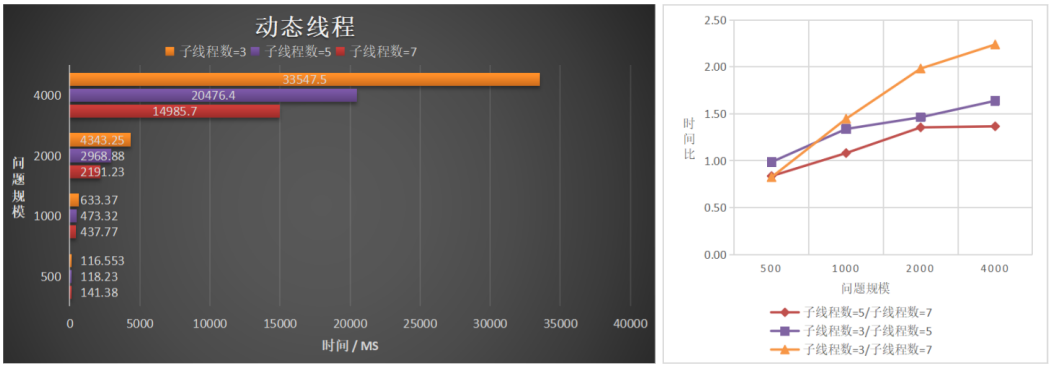

图 4: 动态结果

在问题规模较小时（例如 500），子线程数为 7 的运行时间 (141.38ms) > 子线程数为 5 的 (118.23ms) > 子线程数为 3 的 (116.55ms)，这与理论上的线程数越多，并行运行的时间越短相矛盾；而随着问题规模增大，到了 4000 的规模时，明显表现为线程数越多，运行时间越短。也可以从右图直观地看出，在规模较小时，不同线程数的时间比甚至都下降到了 1 以下，而随着问题规模的增大，时间比才逐渐接近线程数之比。

基于上文《理论分析》一节对动态线程的讨论，对于上述结果产生的原因，我们可以解释如下：

Time = n × θ × 销毁时间 + n × θ × 创建时间 + n × 除法时间 + n × 消去时间 / θ

对于问题规模 n 而言，动态线程的方法减少了 θ - 1 / θ 的消去时间，增加了 n × θ 次创建销毁线程的开销。当问题规模 n 越大，创建销毁线程的时间以 O(n) 级别的增长，而消去时间以 $O(n^3)$ 的级别增长，总体呈现，规模越大，加速比越高的趋势。

对于子线程数 θ 而言，是一个对勾函数的态势，θ 越大，n × θ × 销毁时间 + n × θ × 创建时间越大，但 n × 消去时间 / θ 越小，因此主要取决于“销毁 + 创建时间”和“消去时间”的大小关系。当 n 较小时，创建销毁线程占主导，θ 越大，时间越大；n 较大时，消去时间占主导，θ 越大，时间越小；所以才会出现不同问题规模下，线程数时间比的不同规律。

静态线程的三个版本在各种情况下用时近似，其中信号量的两个版本在误差允许的范围内可认为近似相等，而 barrier 的版本要比信号量版本用时普遍减少 20% 左右。由于三个版本的趋势、时间相似，故只拿出信号量版本进行分析。如图5。

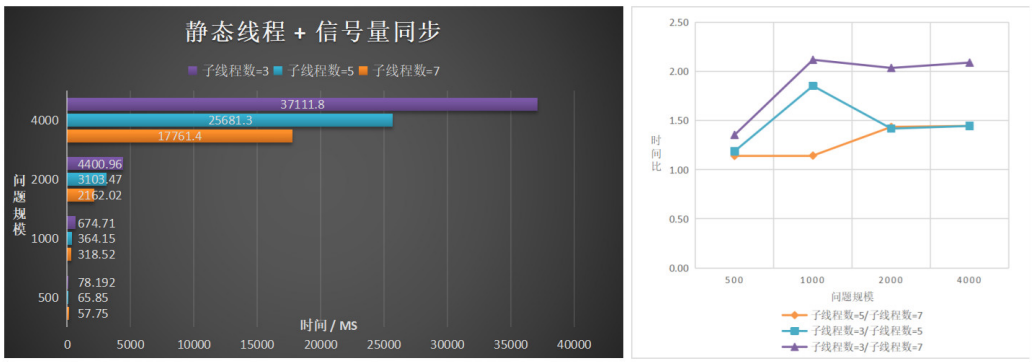

图 5: 静态结果

由图中看出，当问题规模较小时，不同线程数的时间比未能达到理想的线程数之比，而当问题规模变大时，时间比越来越接近线程数之比，几近相等。

从最理想的情况下分析，只考虑占大部分比例的除法和消去这两块时间开销，静态线程的时间如下：Time = n × 除法时间 + n × 消去时间 / θ

除法的复杂度是 O(n^2)，而消去的复杂度在 O(n^3)，当 n 较小时，除法和消去的时间相差不大，此时 1 / θ 并行部分的减少量不能冲淡串行的消去时间，而当 n 较大时，消去的占比要远大于除法，因此会表现为时间比接近线程数之比。这说明问题规模小时，资源闲置的情况较明显，我们应尽量避免这种情况的出现。

为了验证上述分析，在【静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数】这一版本下，我们利用 vtune 进行了 profiling。在线程数都为 7 的条件下，图6是规模为 4000 的各线程运行情况，图7是规模为 500 的各线程运行情况。可以发现，规模为 4000 时，ID 为 66860 的线程和规模为 4000 时 ID 为 100144 的线程都明显比其他工作线程时间短，这是由于在任务划分时，负载不均衡，该线程是划分的最后一个线程，处理的循环数稍少。

在其他可以保证负载均衡的线程中，我们发现，规模为 500 时，ID 为 98380 的线程先占用 CPU 资源且占用 CPU 时间最长，它应为进行除法 + 消去的线程，它对 CPU 时间的占比要比其他线程多 10% 左右。而在规模为 4000 的时候，各线程的运行时间几乎一致，验证了我们的分析。

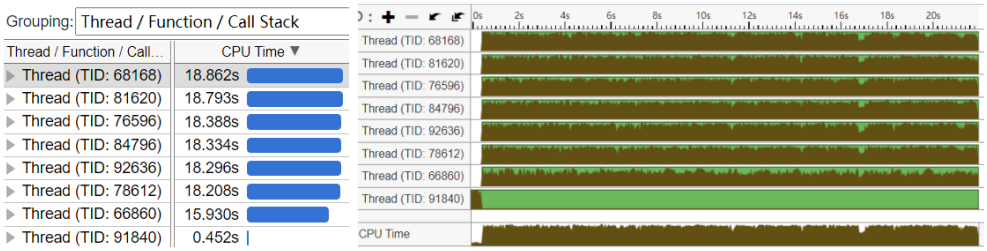

图 6: 规模为 4000 的 vtune 分析结果

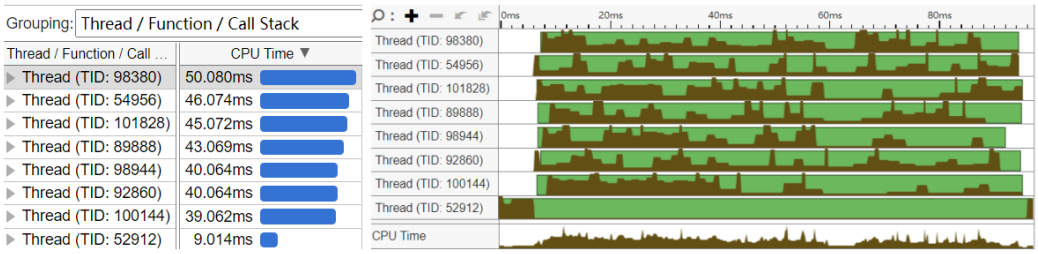

图 7: 规模为 500 的 vtune 分析结果

2. 基于不同算法策略

结果如图8，几种静态线程方法里，barrier 的时间要比信号量的时间普遍更低，而两种信号量的方法相差很小。在规模小的时候，动态线程比静态线程时间长，而当规模变大，差距逐渐缩小，甚至动态线程的方法要优于静态线程。

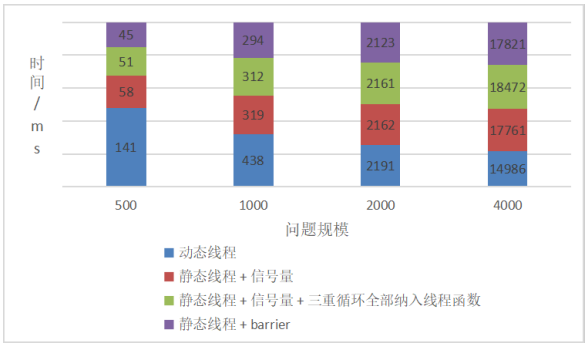

图 8: 不同策略的结果

为了解释动态线程在规模更大的情况下优于静态的原因，我们用 vtune 工具进行了进一步分析，在问题规模为 4000 的条件下，发现动态线程的 CPU frequency 几乎都能达到最高点 4.2Hz，而静态线程的 CPU frequency 在 3-4.2Hz 之间浮动，如图9对应动态线程、图10对应静态线程。

图 9: 动态线程的 CPU frequency

图 10: 静态线程的 CPU frequency

3. 基于平台和指令集

对于 AVX 和 SSE 的指令集，我们在 x86 架构下进行了实验。与鲲鹏的 ARM 平台相比，x86 平台的用时普遍偏大，但是与 arm 不同的是，在 x86 平台 +windows 上的动态线程版本要明显慢于静态线程。例如，问题规模为 1000，工作线程为 7 时，使用 avx256 指令集，在 x86 平台下的动态线程运行时间是 9803ms，而静态线程只有 107ms。Windows 操作系统对线程创建和销毁的开销要明显大于 Linux 系统。

除此之外，不同指令集体现的运行时间：x86 平台下，avx < sse，arm 平台的 neon 普遍快于 x86 平台。

4. 基于任务划分（cache）

对于任务划分问题，我们分别对水平、垂直划分方法采取了穿插着划分和块划分的方法，结果如表1所示：

问题规模	500	1000	2000	4000
水平划分（穿插 +SIMD）/ms	51.25	311.61	2160.74	19757.7
水平划分（块 +SIMD）/ms	43.38	254.93	1903.36	16196.2
水平划分（穿插 + 无 SIMD）/ms	69.1	403.11	2918.77	28055
垂直划分（穿插 + 无 SIMD）/ms	75.1	526.82	5481.78	28516.2
垂直划分（块 +SIMD）/ms	62.89	334.47	2484.14	24166.4

表 1: 任务划分策略实验结果

可以发现，从总体上来看，时间大小关系为：水平 + 块划分 +SIMD < 水平 + 穿插划分 +SIMD < 垂直 + 块划分 +SIMD < 水平 + 穿插划分 + 无 SIMD < 垂直 + 穿插划分 + 无 SIMD。呈现块划分优于穿插划分，水平划分优于垂直划分的结果。与我们之前讨论的分析结果一致，体现了水平划分、块划分对 cache 的利用率更高。

我们在鲲鹏服务器上使用 perf 工具进行了 profiling，结果表明，对于垂直块划分的 cache miss 比垂直穿插划分的 cache miss 小 5% 左右，而水平穿插划分和块划分又比垂直划分 cache miss 低 2% 左右，证实了我们的分析结果。

(六) 不同策略的尝试

分析目前已经实现的这几种算法，发现都存在着主线程利用率不高的情况，因此考虑让主线程承担一部分工作量，并且在之前的算法中没有对除法进行任务划分，因此考虑将除法垂直划分给不同的线程。算法具体思路如下：

创建 7 个工作线程，加上主线程总共 8 个线程，其中除法垂直划分给八个线程共同完成，消去水平划分给 8 个线程，由主线程控制与其他所有线程的同步，控制等所有线程完成除法，发送信号量一起进入消去，等所有线程完成消去，发送信号量一起进入下一轮。

具体代码见https://github.com/hanmaxmax/parallel_homework3 ，不再赘述。

最后得到的结果发现，在 1000 的规模下，时间是 332ms，与上文静态线程的几种方法结果相近，根据 perf 分析发现，它的 cache 命中率要低于之前的算法，故产生这种现象的原因在于将除法划分任务时只能垂直划分，而垂直划分会导致与 SIMD 的结合和 cache 的利用率下降，从而并不会提高性能。

二、 特殊高斯消去

(一) 问题简介

给定消元子和被消元行，依次使用消元子消去对应的被消元行，若某行消元子为空而有其对应的被消元行，则将被消元行升格成消元子，进入到后续的消元中。

(二) 实验设计

在《SIMD》一节中，我们完成了对特殊高斯消去的串行设计，方法如下：

1) 逐批次读取消元子 Act[ ]。

2) 对当前批次中每个被消元行 Pas[ ]，检查其首项（Pas[row][last]）是否有对应消元子；若有，则将与对应消元子做异或并更新首项（Pas[row][last]），重复此过程直至 Pas[row][last] 不在范围内。

3) 运算中，若某行的首项被当前批次覆盖，但没有对应消元子，则将它“升格”为消元子，copy 到数组 Act 的对应行，并设标志位为 1 表示非空，然后结束对该被消元行的操作。

运算后若每行的首项不在当前批次覆盖范围内，则该批次计算完成；

4) 重复上述过程，直至所有批次都处理完毕。

在当时的设计中，我们使用分批对消元子进行读取，然后使用当前这批消元子对所有被消元行进行消去，若被消元行没有对应的消元子则直接将其升格，作为消元行进入到后续的消元中。然而，这种方法会在程序运行中间对某些被消元行进行升格，而升格成的这些消元子会影响到后续的消元环节，即”升格“的存在会造成程序前的后依赖关系。因此，对特殊高斯消去的设计难点集中于两点——如何任务划分以及如何处理造成依赖关系的“升格”。

我们发现，如果不考虑升格，则消元子对不同的被消元行进行处理互不冲突，可以将被消元行划分给不同的线程，每个线程负责对一部分的被消元行进行处理。若采用划分被消元行的方式，如果按照串行的思路，将升格放到线程函数内部，若对”升格“用互斥量加锁，会导致某线程正在升格的消元子无法对其他线程之前出现的或正在处理的被消元行进行操作，影响到程序的正确性。

因此，我们考虑将升格操作单独拿出来，不参与到并行运算。具体方法如下：

1) 每一轮将被消元行划分给不同线程，多线程不升格地处理被消元行；

2) 所有线程处理完之后，一个线程对可能升格的被消元行升格，其他线程等待；

3) 该线程升格完之后，通知其他线程使用更新后的消元子进入下一轮的消去；

4) 直到所有被消元行都不用再升格，说明消去完成，退出程序。

(三) 理论分析

从上面的设计中可以看出，该方法对应的串行程序与之前设计的串行程序相比，由于升格的单独化，消去的循环次数会增多，对被消元行的遍历升格次数也会增多，性能预计会下降。但是其对应的 pthread 并行算法相对于较好的串行算法可能会达到优化的效果。即性能：不单独处理升格的串行算法 > 单独处理升格的串行算法；单独处理升格的并行算法 > 单独处理升格的串行算法；单独处理升格的并行算法? 不单独处理升格的串行算法。

(四) 实验内容及算法实现

结合 SIMD，我们对上面的设计进行实现。由于消去（并行）部分的复杂度是 O(n^3)，升格（串行）的复杂度是 O(n)，串行比并行占比低，因此采用静态线程的方法可以减少创建线程的开销，且更多地利用系统资源。对于同步的策略，我们使用信号量的方法。对于任务划分来说，只能对被消元行采用水平划分的方法。代码具体思路如图11。因代码较长，受文章篇幅限制，可在

https://github.com/hanmaxmax/parallel_homework3查看具体代码。

（注：因在《SIMD》的报告中阐述了特殊高斯消去的矩阵读取转换、数据结构和算法步骤，在本实验中不做重点解释。）

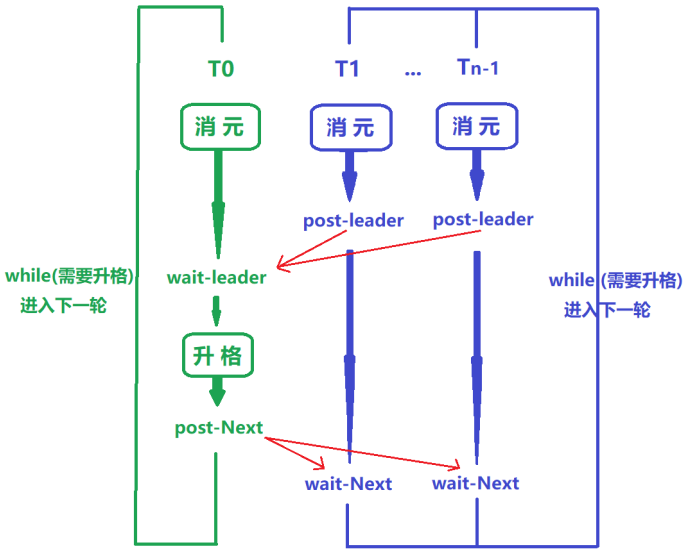

图 11: 特殊高斯 pthread

（五） 实验结果及分析

在工作线程个数为 7、测试环境为鲲鹏服务器并且结合 neon 编程的条件下，我们对以下样例进行实验，结果如表2。

测试样例 8：矩阵列数 23045，非零消元子 18748，被消元行 14325

测试样例 9：矩阵列数 37960，非零消元子 29304，被消元行 14921

测试样例 10：矩阵列数 43577，非零消元子 39477，被消元行 54274

测试样例	8	9	10
不单独处理升格的串行算法/ms	227.28	475.28	1984.17
单独处理升格的串行算法/ms	627.93	1158.85	4890.63
单独处理升格的 pthread 算法/ms	92.98	189.26	1521.16

表 2: 特殊高斯实验结果

从表2中可看出，预期结果与我们理论分析的结果一致，单独处理升格的串行算法要比不单独处理升格的串行算法慢，但是这种“较差”的串行算法要比“较好”的串行算法跟适合应用于 pthread 编程，且其 pthread 处理过的程序相比于“较好”的串行算法要更快。可以达到 2 左右的加速比。

三、 源代码

GitHub 仓库地址：https://github.com/hanmaxmax/parallel_homework3