南開大學
Nankai University

计算机学院

并行程序设计实验报告

高斯消去算法的 pthread & OpenMP 并行研究

姓名：王浩

学号：2013287

专业：计算机科学与技术

2024 年 5 月 26 日

目录

1 问题介绍 **2**

2 pthread 多线程编程实现 **2**

2.1 实验设计 . . . 2
2.1.1 总体思路 . . . 2
2.1.2 同步机制 . . . 2
2.1.3 pthread 编程范式 . . . 3
2.1.4 任务划分方法 . . . 3
2.2 理论分析 . . . 4
2.2.1 平凡算法 . . . 4
2.2.2 动态线程 . . . 4
2.2.3 静态线程 + 信号量 . . . 4
2.2.4 静态线程 + barrier . . . 5
2.3 代码实现 . . . 5
2.4 结果分析 . . . 5
2.4.1 动态线程的结果分析 . . . 5
2.4.2 静态线程的结果分析 . . . 6
2.4.3 不同任务划分对时间的影响 . . . 7

3 OpenMP 多线程编程实现 **8**

3.1 实验设计 . . . 8
3.1.1 总体思路 . . . 8
3.1.2 研究方案 . . . 8
3.2 理论分析 . . . 10
3.2.1 单个线程划分数据量分析 . . . 10
3.2.2 划分方式 . . . 10
3.3 代码实现 . . . 11
3.4 结果分析 . . . 11
3.4.1 线程数与问题规模的影响 . . . 11
3.4.2 数据划分方式 . . . 12
3.4.3 循环调度方式 . . . 12
3.4.4 其他算法策略 . . . 13
3.4.5 OpenMP 自动向量化 . . . 13
3.4.6 与 pthread 的对比 . . . 14

1 问题介绍

高斯消去法求解线性方程组的原理是, 对于一个由 n 元未知数, n 个一次方程组成的线性方程组, 将 $n \times n$ 个系数组成一个矩阵, 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, 使每行第一个元素为 1, 然后对矩阵该行右下角 $(n-k+1) \times (n-k)$ 的子矩阵进行消去:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2,n-1} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & a'_{12} & \cdots & a'_{1,n-1} & a'_{1n} \\ 0 & 1 & \cdots & a'_{2,n-1} & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & a'_{n-1,n} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

2 pthread 多线程编程实现

2.1 实验设计

2.1.1 总体思路

在设计高斯消元算法的并行实现之前，首先需要深入理解整个程序的依赖关系结构，从而识别出可以串行处理和可以并行处理的部分。普通的串行高斯消元算法主要包括一个外层的大循环，其内部嵌套着一个二重循环用于执行行的除法操作，以及一个三重循环用于实施行的消去操作。这个最外层的循环负责逐行处理矩阵，即使用当前行进行归一化（除法）后，再利用该行消去下方行中的对应元素。

在这个过程中，操作的依赖性非常明显：必须首先对当前行进行除法操作，确保行首元素为 1，然后才能使用这一行对其下方的行进行消去处理。虽然除法和消去操作在每一行内部是依序进行的，但这两者之间并不存在必然的顺序关系，这为并行计算提供了可能。

考虑到线程的创建和同步通常会带来一定的开销，我们的多线程设计策略是：在进行每一行的处理时，首先由一个或多个线程完成除法操作，此时负责行消去的其他线程处于等待状态。一旦除法完成，立即唤醒负责消去的线程开始其任务。所有线程必须等待当前全部消去任务完成后，才能共同进入下一轮循环，重复上述步骤。

此外，从 SIMD 实验的讨论中我们了解到，尽管除法操作在每一行的处理中所占的时间比较短，但由于消去操作对总体性能的影响较大，优化消去部分的执行效率是提升程序性能的关键。因此，对于除法操作，我们可以选择在特定情况下不进行线程并行化，以避免不必要的线程管理与同步开销。

在深入具体实现之前，还需要详细考虑任务如何划分、信息如何交互与同步，并探索如何将 SIMD 技术与多线程技术有效结合。在接下来的章节中，我们将详细讨论这些技术细节，并提出具体的实现方案和优化策略。这种综合考虑将帮助我们更有效地实现高斯消元算法的并行版本，从而在保证算法正确性的同时，显著提升计算效率。

2.1.2 同步机制

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。

根据上述分析，在假设有一个除法线程的情况下，共需要一个 + 两组信号量。一个信号量对应于一个除法线程，即上述讨论中的信号量 2；两组信号量对应于多个消去线程，即上述讨论中的信号量 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。

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。

2.1.3 pthread 编程范式

1. 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。

2. 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。

2.1.4 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。

2.2 理论分析

2.2.1 平凡算法

对于单线程下的平凡算法, 高斯消去法的时间分为两部分: 每行除以首元素的除法时间和消去当前行下方下三角的消去时间, 可以用公式表示如下:

$$n \times (\text{time}_{alg-divide} + \text{time}_{alg-eliminate})$$

其中 $\text{time}_{alg-divide}$ 对每个行元素计算, 因此是 $O(n^2)$ 的 (考虑到最外层的 n 次循环); $\text{time}_{alg-eliminate}$ 需要对下三角的每个元素计算, 因此是 $O(n^3)$ 的. 所以在接下来的优化考虑中, 因优先考虑 $\text{time}_{alg-eliminate}$ 部分的优化.

2.2.2 动态线程

假设线程数为 $\theta+1$, 对于除法部分使用 1 个线程, 对于消去部分使用 θ 个线程. 矩阵有 n 行, 每轮循环处理一行的消去, 所以一共需要进行 n 次除法操作; 由于采用动态线程, 每轮循环创建, 销毁 θ 个线程. 所以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:

$$n \times (\theta \times (\text{time}_{create} + \text{time}_{destroy}) + \text{time}_{alg-divide} + \frac{\text{time}_{alg-eliminate}}{\theta})$$

相对于平凡算法, 动态线程方法下消去部分时间 $\text{time}_{alg-eliminate}$ 为原来的 $\frac{1}{\theta}$, 同时增加了线程创建和销毁的时间, 但由于 $\text{time}_{create} + \text{time}_{destroy}$ 是 $O(n)$ 的, 而 $\text{time}_{alg-eliminate}$ 是 $O(n^3)$ 的, 因此随着 n 的增加理论上该算法的加速比能达到 θ.

2.2.3 静态线程 + 信号量

1. 主线程做除法, 工作线程做消去

在这种设计模式下, 每轮循环的工作模式如下: 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。

2. 主线程不做运算, 工作线程做除法 + 消去

在这种设计模式下, 每轮循环的工作模式如下: 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 T_0 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 T_0 先做除法，其他线程等待；除法完成后，T_0 通知其他线程结束等待，然后 $_T0$ 与其他线程一起进行消去，T_0 消去之后等待其他线程都完成消去再一起进入下一轮。

对比两种设计模式, 方式 1 完成了除法与消去线程的分离, 方式 2 减少了一个线程的通信, 即 T_0 线程无需在” 除法完成-> 消去开始” 和” 消去结束-> 下一轮开始” 之间通知自己.

2.2.4 静态线程 + barrier

使用 barrier 不使用信号量则意味着只需要将整个循环纳入线程函数, 同时加入两个同步点即可, 可以使代码和逻辑更为简洁, 并且不影响程序最终 θ 的理论最大加速比.

2.3 代码实现

根据以上分析, 我们主要实现了以下版本的代码:

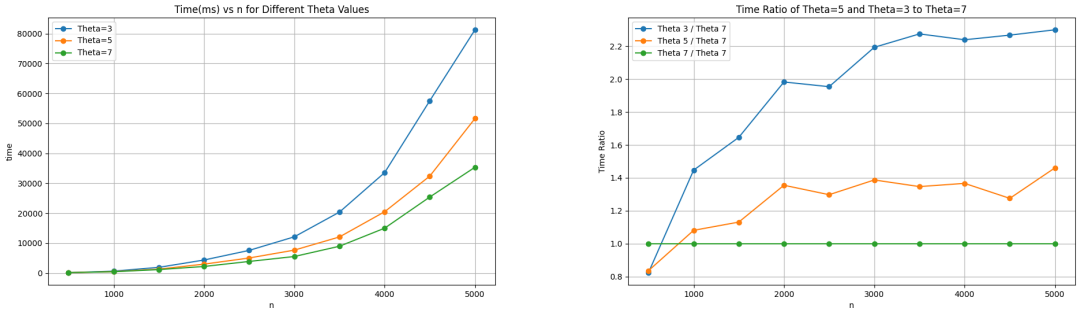
1. pthread_dynamic_pthread.cpp: 高斯消去的动态线程实现
2. pthread_static_semaphore_main_do_nothing.cpp: 高斯消去的静态线程 + 信号量 + 三重循环全部纳入线程函数实现，主线程只做创建和挂起销毁，其余线程做除法和消去
3. pthread_static_semaphore_main_do_divide.cpp: 高斯消去的静态线程 + 信号量实现，主线程做除法，其余线程消去
4. pthread_static_barrier.cpp: 普通高斯消去的静态线程 +barrier 实现

具体代码可以见github 仓库.

2.4 结果分析

2.4.1 动态线程的结果分析

接下来, 我们编写 python 脚本, 在动态线程下分别测试当 $\theta = 3, 5, 7$, $n \in [500, 5000], \text{step} = 500$ 下各个程序的运行时间, 如图2.1所示.



(a) 不同 θ, n 下的程序运行时间.

(b) 其他 θ 相对于 $\theta = 7$ 的时间比.

图 2.1: 动态线程下程序运行时间

从图2.1中我们可以发现, 在问题规模较小 ($n = 500$) 时, $\theta = 7$ 的运行时间 $> \theta = 5$ 的运行时间 $> \theta = 3$ 的运行时间, 意味着此时创建与销毁线程的时间占用比例较大, 而真正用于除法-消去计算的时间占用比例较小. 随着问题规模增大, 时间复杂度为 $O(n^3)$ 的 $\text{time}_{alg-eliminate}$ 成为程序运行时间的瓶颈, 不同线程数下总运行时间之比开始接近线程数之比.

基于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：

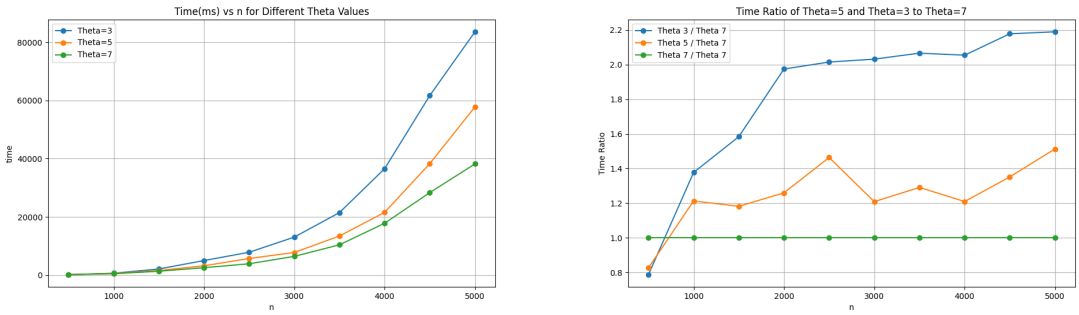
$$\text{Time} = n \times (\theta \times (\text{time}_{create} + \text{time}_{destroy}) + \text{time}_{alg-divide} + \frac{\text{time}_{alg-eliminate}}{\theta})$$

对于问题规模 n 而言，动态线程的方法减少了 $\frac{\theta - 1}{\theta}$ 的消去时间，增加了 $n \times \theta$ 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，θ 越大，$\theta \times (\text{time}_{create} + \text{time}_{destroy})$ 越大，但 $\frac{\text{time}_{alg-eliminate}}{\theta}$ 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，总时间越长；n 较大时，消去时间占主导，θ 越大，总时间越短；所以才会出现不同问题规模下，线程数时间比的不同规律。

2.4.2 静态线程的结果分析

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2.2所示.



(a) 不同 θ, n 下的程序运行时间.

(b) 其他 θ 相对于 $\theta = 7$ 的时间比.

图 2.2: 静态线程 + 信号量下程序运行时间

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：

$$\text{Time} = n \times (\text{time}_{alg-divide} + \frac{\text{time}_{alg-eliminate}}{\theta})$$

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 $\frac{\theta}{1}$ 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2.3是规模为 5000 的各线程运行情况，图2.4是规模为 500 的各线程运行情况。可以发现，规模为 5000 时，ID 为 66860 的线程和规模为 5000 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 5000 的时候，各线程的运行时间几乎一致，验证了我们的分析。

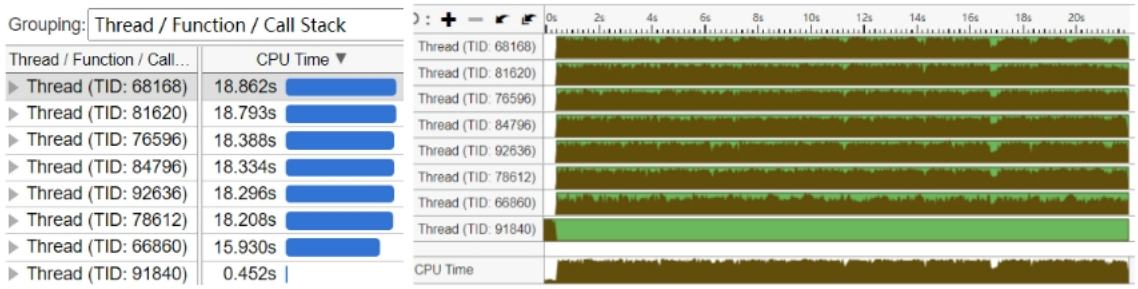


图 2.3: $n = 5000$ 下各线程的 CPU 时间

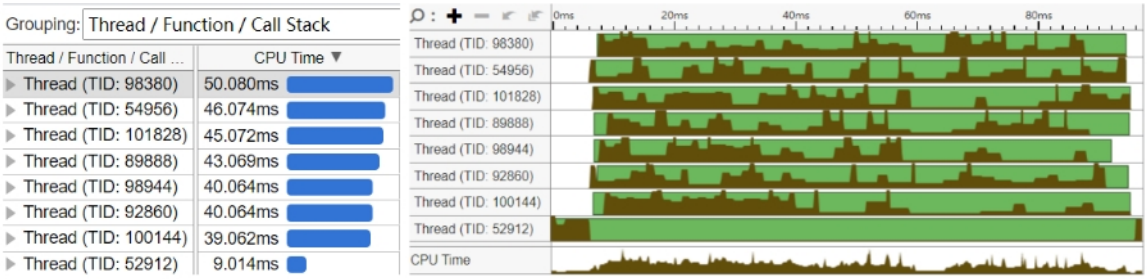


图 2.4: $n = 500$ 下各线程的 CPU 时间

2.4.3 不同任务划分对时间的影响

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）	51.25	311.61	2160.74	19757.7
水平划分（块 +SIMD）	43.38	254.93	1903.36	16196.2
水平划分（穿插 + 无 SIMD）	69.1	403.11	2918.77	28055
垂直划分（穿插 + 无 SIMD）	75.1	526.82	5481.78	28516.2
垂直划分（块 +SIMD）	62.89	334.47	2484.14	24166.4

表 1: 不同配置和数据规模下的运行时间（单位：ms）

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。

3 OpenMP 多线程编程实现

3.1 实验设计

3.1.1 总体思路

分析高斯算法的执行过程，如图3.5，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（假设某次循环到第 k 行），其中的二重循环表示对矩阵的第 k 行进行除法，三重循环表示用第 k 行对其右下角的 $(n-k+1)\times(n-k)$ 的子矩阵进行消去。

在每轮大循环中存在着先后关系：需要首先对该行进行除法，然后再对它右下角子矩阵消去。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，OpenMP 程序的整体思路为：取一个线程（或多个线程并行）先对矩阵第 k 行进行除法，除法结束后所有线程才可以进入并行消去环节。当所有线程完成消去后，一起进入下一轮，重复上述步骤。

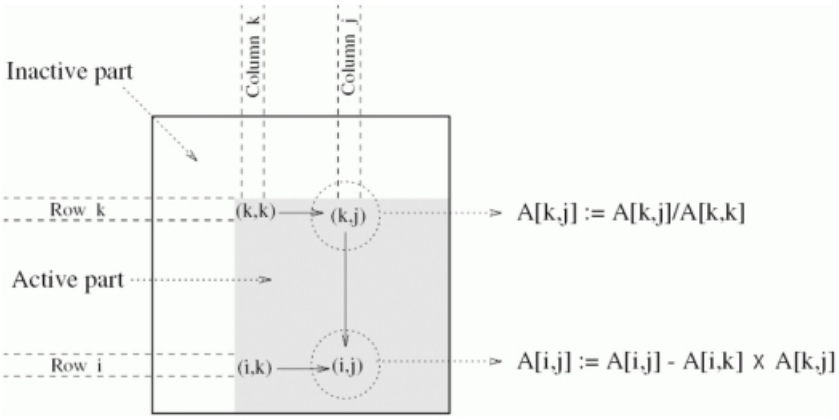


图 3.5: Caption

在 pthread 部分，我们已经编写了该算法的 pthread 多线程程序，完成了对信息交互、同步机制、多线程的具体实现等内容的探究，因此在 OpenMP 部分我们将将侧重于对任务划分、循环调度、负载均衡、体系结构、编程范式等内容的探究。

3.1.2 研究方案

对于 OpenMP，有如下方向可以进行探究:

1. 数据划分

   - 粗粒度划分：适合于计算密集型的任务，可以减少线程之间的同步和通信开销。通过将大块数据分配给每个线程处理，可以减少线程切换和调度的代价，优化体系结构的利用率。例如，可以将大型数组或矩阵均匀分配给不同的线程。

- 细粒度划分：适合于 I/O 密集或访问模式不均匀的应用。通过细粒度划分，可以更好地实现负载均衡，尤其是在数据访问模式不可预测时。例如，可以对数据库的查询结果进行行级或记录级的划分处理。
- 动态数据划分：当无法事先知道数据分布或工作量时，动态划分能够在运行时根据实际情况调整数据划分策略，以实现更优的负载均衡和降低访存代价。

2. 循环调度

- 静态调度：将循环迭代在线程之间静态均匀分配，适用于迭代之间无依赖且执行时间大致相等的情况。
- 动态调度：在运行时动态分配迭代到线程，更适合迭代执行时间不均或不可预测的情况。这种方式虽然增加了调度的开销，但能够更有效地处理不均匀的工作负载。
- 指导性调度：是一种折衷方案，开始时分配较大的块，随着执行逐渐减小块的大小，适合于逐渐减少的工作量。

3. 体系结构 & 访存

为了最大限度地利用硬件资源，特别是缓存, 可以研究：

- Cache 优化：设计任务和数据分布，使得常用数据能够尽可能地保持在 cache 中，减少访问主内存的延迟。通过调整数据结构和访问模式，优化局部性原理。
- 多线程与 Cache 共享：合理安排线程与处理器核心的绑定（affinity），以利用多级缓存体系。例如，让频繁交互的线程运行在共享同一级缓存的核心上。
- 考虑不同问题规模和线程数：设计实验，考察不同的问题规模和线程数量对缓存利用率的影响，以找到最优配置。

4. 不同平台

OpenMP 支持多种操作系统，不同平台的性能表现可能有所不同, 对此可以研究：

- 跨平台性能比较：在 Linux 和 Windows 系统上运行相同的 OpenMP 程序，观察并分析两者的性能差异。
- 硬件优化指令：利用特定平台支持的 SIMD 指令集，如 SSE、AVX，来加速计算。评估在不同平台下这些指令对性能的提升。

5. 与 SIMD 的结合

结合 SIMD 可以显著提升数据并行的效率, 包括：

- 手动优化：在 OpenMP 程序中手动插入 SIMD 指令，如使用 OpenMP 4.0 及以上版本中的 #pragma omp simd，优化循环的向量化。
- 编译器优化：探索编译器自动向量化的效果，使用如-O3、-ftree-vectorize 等编译选项，比较自动与手动向量化的效率和结果。

3.2 理论分析

3.2.1 单个线程划分数据量分析

在设计多线程程序时，可以通过充分适应 CPU 的体系结构，来达到更好的实验效果。根据体系结构和组成原理的知识，服务器的一个节点可以有多个 CPU，一个 CPU 可以有多个 CPU 核心，一个 CPU 核心可以有一个以上的线程。一个核心在某个时间点只能执行一个进程，一个程序可以调用多个进程，一个进程可调用至少一个线程，如果一个进程同时调用的线程数超过 CPU 核心的线程数，则需要调用其他 CPU 核心实现并行。一个进程只能在本节点运行，线程是进程派生的并共享进程资源，所以多线程并行不能跨节点运行。分析实验使用的华为鲲鹏服务器，如表2，鲲鹏服务器有两个物理 CPU，每个物理 CPU 有 48 个核心，共有 96 颗逻辑 CPU。服务器有 4 个 node。鲲鹏每个 CPU 核心有独自的 L1 cache（64K），L2 cache（512K），共享 L3 cache（49152K）。在鲲鹏服务器上运行 OpenMP 程序时，总共最多开启 8 个线程。

CPU 型号	华为鲲鹏 920 处理器
CPU 主频	2.6GHz
L1 cache	64KB
L2 cache	512KB
L3 cache	48MB
指令集	Neon
核心数	1
线程数	8

表 2: 华为鲲鹏服务器配置

对于 64k 大小的 L1 cache 来说，我们进行如下分析：每个 float 类型的变量占 4 字节，L1 cache 的大小为 64k，当问题规模 n 足够大时，可忽略其他变量的空间，设 float 型变量总数为 N，有 n^2 约等于 N。因此，计算得出，占满 L1 cache 的 n 在 120 附近，占满 L2 cache 的 n 在 350 附近。

因此，当我们设计的问题规模较大时，如果能够使得每次分配给每个线程的数据量合适，不过于小也不超过其 L1 cache 大小，就可能达到更好的命中率和更快的速度。

以处理的矩阵规模为 $\theta \times \theta$ 为例，将 $\theta \times \theta$ 的矩阵按行划分成若干任务，假设每个任务 h 行，$h \times \theta$ 接近 float 型变量总数为 N 即可。

但是，高斯算法每轮处理的消元矩阵大小为 $(n-k)\times(n-k)$，随着轮数 k 的增大而递减，因此，我们取其运算矩阵最大的规模 $n \times n$ 来计算任务行数，以提高整体命中率。

综上，我们得出结论：当每次划分的数据量过大，线程读取的数据会超出其 L1 cache 容量大小，降低命中率，速度减慢；当划分的数据量过小，空间读取不连续，由于空间局部性也会降低性能。因此，我们分析，当划分的数据大小接近 cache 容量可能会更好。

3.2.2 划分方式

- 静态划分

  openMP 默认的调度方式为静态划分，且默认的划分块大小为 ceil(iterations / threads)。这种情况下，每个线程分配到的数据量不均衡，容易造成负载不均，时间浪费。如果减小划分粒度，极限情况下，每个线程负责的数据块只有一行/一个元素，会导致一个线程负责计算的数据几乎没有连续的，缺乏局部性，高度交互，通信量大。因此，高维块循环分配可能达到更好的效果。此时每个线程分配的数据量大小可以参考上一小节数据划分给出的计算结果。

- 动态划分

  逻辑上相当于有一个任务池，包含所有迭代步。先分配给每个线程一个迭代步，一个线程完成任务后再为其分配迭代步。动态划分的好处是不会让线程有空闲等待的时间。

- guided 动态划分

  划分方式与普通的动态划分相同，但划分过程中，最初的组中的循环体执行数目较大，而后指数减小。

3.3 代码实现

在 OpenMP 部分, 我们实现了如下代码:

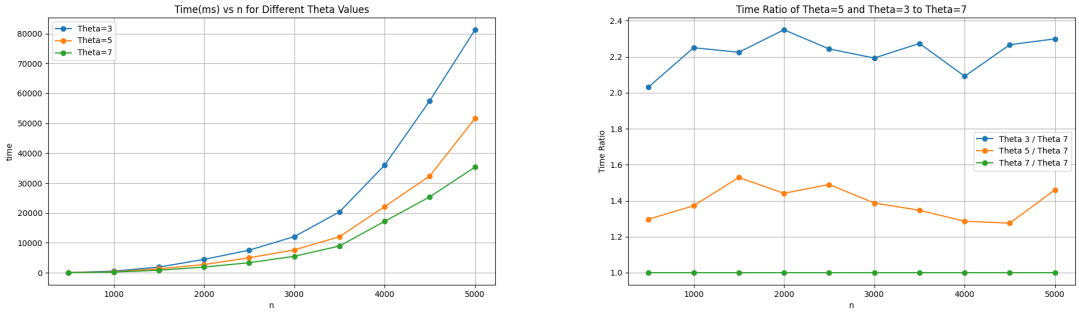
1. openmp.cpp: 高斯消去的 OpenMP 实现, 基于静态数据划分.
2. openmp_more.cpp: 高斯消去的 OpenMP 实现, 比较动态划分, guided 划分, barrier, master 范式下的效果, 并进行了除法优化. 此外，结合 pthread 编程的动态线程和静态线程区别，我们发现将 pragma omp parallel... 语句放到整个大循环的外部，而在大循环内部再划分 pragma omp single 串行部分和 pragma omp for 并行部分，可以减少创建、销毁循环的开销.

具体代码可以见github 仓库.

3.4 结果分析

3.4.1 线程数与问题规模的影响

基于 OpenMP 静态划分算法, 我们运行 $\theta = 3, 5, 7$, $n \in [500, 5000], \text{step} = 500$ 下的实验, 结果如图3.6所示:



(a) 不同 θ, n 下的程序运行时间.

(b) 其他 θ 相对于 $\theta = 7$ 的时间比.

图 3.6: OpenMP 静态划分算法下运行时间

图3.6的左侧展示了不同规模和不同线程数下，OpenMP 算法运行的具体时间，右图展示了各规模下，不同线程数的运行时间比。分析数据我们可以发现，在各个规模下，不同线程数的时间比都近似等于线程个数之比。

再与串行算法对比，结合了 neon 的 OpenMP 多线程算法比纯串行算法快了 8 倍以上，达到了很好的加速效果，且加速效果在各个问题规模下都很明显。

3.4.2 数据划分方式

在理论分析一节中，我们结合体系结构，对数据划分的粗细粒度进行了大致计算。下面我们以问题规模 1000 为例，进行了实验。

根据我们之前的计算，问题规模为 1000 时，在最大的 $n \times n$ （1000 x 1000）的矩阵中，能够填满 L1 cache 的划分粒度是 14 行，而随着最外层大循环的迭代，每次要处理的矩阵的大小会递减，能够填满 L1 cache 的划分行数也会增大。由于 schedule 编程范式只能设置 [chunk] 为常数值，我们不能更改大循环每次到达并行小循环的 [chunk]（即使使用 guided 范式，也只是针对并行小循环自己内部的划分粒度改变，不能改变大循环每次到达并行小循环的 [chunk]），因此我们取填满 $n \times n$ 的划分行数来实验。

因此，下面我们以问题规模为 1000，[chunk] 分别为 7, 14, 30 进行实验，测试结果如表3所示：

chuck	7	14	30
时间/ms	315.26	303.57	325.68
cache 命中率	82.59%	84.12 %	81.98%

表 3: 不同 chuck 划分下的结果

可以看出，chunk = 14 的划分方式，要快于 chunk = 7 和 30 的，为了进一步分析其产生原因，我们使用 perf 分析工具，查看它们的 cache 命中率。在 perf 的结果中，chunk=14 的 L1-dcache 命中率 > chunk=30 和 7 的 L1-dcache 命中率，它们都达到了 80% 以上，彼此相差 5% 以下。这说明我们之前对体系结构和粒度划分的分析是较合理的。

3.4.3 循环调度方式

在问题规模为 1000，线程数 =7 的情况下，我们在消去的循环上分别采用 static、dynamic、guided 划分，结果如表4:

调度方式	static	dynamic	guided
时间/ms	305.2	278.1	282.3

表 4: 不同循环调度方式下的结果

可以看到，动态划分线程和 guided 方式划分线程要快于静态划分，我们利用 VTune 分析工具进行进一步分析，得到 static, dynamic, guided 的结果如图3.7, 3.8, 3.9所示:

Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
Thread (TID: 43980)	232.371ms	1,017,500,000	1,877,500,000	0.542	97.5%
Thread (TID: 47716)	218.349ms	825,000,000	1,577,500,000	0.523	100.0%
Thread (TID: 42728)	211.337ms	927,500,000	1,680,000,000	0.552	60.2%
Thread (TID: 26308)	210.336ms	830,000,000	1,422,500,000	0.583	100.0%
Thread (TID: 44180)	207.331ms	892,500,000	1,652,500,000	0.540	66.0%
Thread (TID: 41632)	198.317ms	890,000,000	1,520,000,000	0.586	100.0%
Thread (TID: 36008)	182.291ms	720,000,000	1,555,000,000	0.463	100.0%

图 3.7: static VTune 结果

Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
Thread (TID: 43076)	225.360ms	967,500,000	1,735,000,000	0.558	54.5%
Thread (TID: 47512)	211.337ms	882,500,000	1,645,000,000	0.536	49.2%
Thread (TID: 32708)	210.336ms	857,500,000	1,580,000,000	0.543	100.0%
Thread (TID: 30676)	208.333ms	825,000,000	1,530,000,000	0.539	78.9%
Thread (TID: 46444)	205.328ms	852,500,000	1,532,500,000	0.556	100.0%
Thread (TID: 29764)	204.326ms	855,000,000	1,530,000,000	0.559	68.9%
Thread (TID: 42860)	202.323ms	897,500,000	1,770,000,000	0.507	48.4%

图 3.8: dynamic VTune 结果

Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
Thread (TID: 32708)	240.384ms	972,500,000	1,760,000,000	0.553	28.7%
Thread (TID: 3860)	228.364ms	905,000,000	1,515,000,000	0.597	82.2%
Thread (TID: 37264)	207.331ms	885,000,000	1,717,500,000	0.515	56.0%
Thread (TID: 46524)	207.331ms	892,500,000	1,600,000,000	0.558	100.0%
Thread (TID: 46980)	203.324ms	862,500,000	1,682,500,000	0.513	97.0%
Thread (TID: 19068)	195.312ms	855,000,000	1,570,000,000	0.545	87.0%
Thread (TID: 1584)	184.294ms	827,500,000	1,530,000,000	0.541	74.9%

图 3.9: guided VTune 结果

可以明显看出，dynamic 动态调度方式下，每个线程的运行时间条明显更整齐，说明这种方法下的负载更加均衡。这是因为：动态划分策略下，程序运行时的 7 个线程动态轮流抽取任务，使得负载更加均衡，减少了资源闲置浪费的情况。再看 static 的结果，会有几个线程的运行时间明显少于其他线程，这是因为资源分配是一开始就决定好的，当一个线程运行完它负责的任务后，只能闲置等待。在本实验的情境下，guided 的运行时间比 static 快但却没有 dynamic 快，这是因为 guided 也是一种动态划分的方式，要比 static 负载更均衡，但它是随着循环的遍历，划分的块大小递减。这样的划分方式对于随着循环的遍历而规模减小的循环更有利。但是本实验比较特殊，虽然消去块是随着外层大循环的遍历而逐渐变小，但是回观我们之前写的代码，openmp 的 for schedule 语句是写在并行消去小循环的外面，只能控制消去循环内部的划分粒度，不能控制外层大循环对每次消去部分的划分粒度，因此，这里的 guided 并没有更好的效果，反而因为其刚开始划分的块过大，导致最后的效果还没 dynamic 好。

3.4.4 其他算法策略

对于 OpenMP 程序，我们考虑并尝试了其他多种算法策略，介绍如下:

1. ” 动态” 地创建线程:

   将 OpenMP 创建线程且分配任务的语句放到大循环内部，相当于每次运行到并行部分，都要创建、销毁线程。

2. 其他 OpenMP 编程范式:

   使用 barrier 和 master 的范式编程。

3. 将除法并行化:

   把除法部分也分为多线程运算，并且要注意除法和消去的同步。

结果如表5所示:

3.4.5 OpenMP 自动向量化

OpenMP4.0 以上支持自动结合 SIMD 并行编程。编译器通过分析程序中控制流和数据流的特征，识别并选出可以向量化执行的代码，并将标量指令自动转换为相应的 SIMD 指令的过程。即向量化的

问题规模	**1000**	**2000**	**4000**
初始参数设置	242.48	1900.43	15936.8
“动态” 初始设置	246.07	1919.28	16256.9
barrier 调式	241.77	1909.56	16016.3
除法也并行化	230.87	1854.22	15052.7

表 5: 不同策略下的结果

过程由编译器自动完成，我们只要编写正常的 C++ 代码，让编译器会自动分析代码结构，将适合向量化的代码部分自动生成 SIMD 指令的向量化代码。而且这些代码可以跨平台编译，针对不同的平台生成不同的 SIMD 指令。

查看鲲鹏服务器的 GCC 版本所对应的 OpenMP 已经可以支持自动向量化了，因此我们进行了自动向量化的实验。当 $\theta = 7$ 时，不同问题规模下的结果如表6:

问题规模	500	1000	2000
OpenMP 无 SIMD	37.87	272.42	2233.33
OpenMP 手动 SIMD	33.03	242.17	1909.36
OpenMP 自动化 SIMD	34.67	265.38	2010.72

表 6: 自动向量化

可以看到，虽然相比于没有 SIMD 的程序，OpenMP 的自动化 SIMD 算法有一定的加速效果，但效果并没有手动编写的代码好。

3.4.6 与 pthread 的对比

OpenMP 提供了一种更高层次的抽象，通过简单的编译器指令来实现多线程，这使得代码更加简洁，易于编写和理解，特别是对于循环并行化这一常见需求。例如，OpenMP 允许开发者通过几个简单的指令，如 #pragma omp parallel for，来自动处理线程的创建、执行和销毁，而不需要像在 pthread 中那样手动管理这些细节。

另一方面，pthread 提供了更细粒度的控制，允许开发者详细指定线程行为、同步机制如条件变量和互斥锁，以及线程间的通信。这种控制的精细化使 pthread 在需要精确控制线程行为的场景中更为合适，例如在高度依赖精确线程同步和复杂线程间通信的应用程序中。

在性能方面，由于 OpenMP 的抽象层次更高，它可能在某些情况下引入额外的开销，尤其是在对循环进行自动并行化处理时。然而，由于 OpenMP 可以利用编译器的优化能力，如自动向量化，它在某些数值密集型的任务中可能表现得更好，特别是在使用了编译器优化选项后。

具体到我们的实验，OpenMP 在数据划分和循环调度方面的灵活性带来了明显的便利。例如，通过调整 schedule 类型，可以简单地改变任务的负载均衡策略，从而适应不同的工作负载模式。相比之下，使用 pthread 实现相同的动态负载均衡则需要更多的编程工作和复杂的逻辑控制。