南開大學
Nankai University

计算机学院

并行程序设计实验报告

高斯消去算法和 LU 分解算法的 pthread 与 OpenMP 并行化研究

姓名：蔡沅宏 黄明洲

学号：2213897 2211804

专业：计算机科学与技术

2024 年 5 月 26 日

目录

1 任务描述

1.1 研究任务及报告说明

本研究旨在分别对 gauss 方法和 LU 算法进行并行化优化，运用 pthread 和 openmp 方法进行性能提升。通过实验数据分析，比较不同优化方法的性能表现，并根据实验结果对两种方法进行横向对比，分析它们适用的情况。

本报告主要展示小组成员蔡沅宏针对 LU 算法展开的一系列研究与分析，报告将在结尾模块展示小组成员完成各自任务后针对这两种算法应用的合作分析结果。

1.2 研究问题说明

1.2.1 Guass 消去

在进行科学计算的过程中，经常会遇到对于多元线性方程组的求解，而求解线性方程组的一种常用方法就是 Gauss 消去法。即通过一种自上而下，逐行消去的方法，将线性方程组的系数矩阵消去为主对角线元素均为 1 的上三角矩阵。Gauss 消去可用来进行线性方程组求解，矩阵求秩，以及逆矩阵的计算。

高斯消元法计算公式：

- 将方程组表示为增广矩阵 [A|b]，通过消元操作将系数矩阵 A 化为上三角矩阵；
- 利用回代求解出方程组的解 x；

Gauss 消去示意图如1.1所示：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1\ n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2\ n-1} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1\ 1} & a_{n-1\ 2} & \cdots & a_{n-1\ n-1} & a_{n-1\ n} \\ a_{n\ 1} & a_{n\ 2} & \cdots & a_{n\ n-1} & a_{n\ n} \end{bmatrix} => \begin{bmatrix} 1 & a'_{12} & \cdots & a'_{1\ n-1} & a'_{1n} \\ 0 & 1 & \cdots & a'_{2\ n-1} & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & a'_{n-1\ n} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

图 1.1: Gauss 消去示意图

1.2.2 LU 算法

LU 分解是一个重要的数值分析方法，它将矩阵分解为下三角矩阵 L 和上三角矩阵 U 。在科学计算、工程技术等领域中具有广泛应用。通过 LU 分解将原始线性方程组转化为两个简单的三角形方程组，从而可以更容易地求解线性方程组。同时可以用 L 和 U 的对角线元素相乘得到原矩阵的行列式。此外，通过 LU 分解后，可以快速求解矩阵的逆矩阵，从而可以方便地进行矩阵运算。

LU 分解算法计算公式：

- 将系数矩阵 A 分解为两个矩阵 L 和 U 的乘积，其中 L 为下三角矩阵，U 为上三角矩阵；
- 利用 LU 分解可以简化方程组求解的过程，提高计算效率；

LU 算法示意图如1.2所示：

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix}$$

图 1.2: LU 算法示意图

2 实验环境

此实验将对 LU 算法在 x86 和 ARM 两个平台下进行算法并行优化及分析，下表是对于实验环境详细展示，如表10所示：

	ARM	**x86**
CPU 型号	华为鲲鹏 920 服务器	12th Gen Intel Core i7-12700H
CPU 主频	2.6Ghz	2.3Ghz
一级缓存	64KB	1.2MB
二级缓存	512KB	11.5MB
三级缓存	48MB	24.0MB
指令集	NEON	SSE、AVX、AVX512

表 1: 实验环境

3 实验设计

本次实验将使用 pthread 和 OpenMP 进行并行编程，pthread 和 OpenMP 作为并行编程工具，都能够实现并行化处理，充分利用多核处理器和多线程架构，将任务分解为子任务，并行执行，从而实现并发处理，降低任务处理的总时间，提高程序的性能和效率。

LU 分解算法中涉及大量的矩阵乘法、除法和求和等计算，这些计算可以地被很好地分解为独立的子任务，并行执行。无论是 pthread 还是 OpenMP 都能够很好地实现这种任务并行。同时，在 LU 分解算法中，矩阵的行/列之间的计算是相互独立的，具有可以通过并行化加快计算速度的性质。这种数据并行的特性也正是 pthread 和 OpenMP 擅长处理的内容。

所以，本次实验中，我仍将针对 LU 分解算法进行 pthread 和 OpenMP 的并行编程尝试，同时，结合上一章节 simd 编程内容，对 LU 算法进行多线程与 simd 结合的并行加速尝试，并对实验结果进行分析。

另外，我们也应注意到 pthread 与 OpenMP 也具有不同之处，OpenMP 提供了相比于 pthread 更简便的方式来实现并行化，而 pthread 则提供了更大的灵活性和控制能力，可以根据需要精细调整线程的行为，实现更复杂的并行算法。因此，在实验中也将根据实验结果对于这两种编程方式进行比较分析。

此次实验规模较大，为更好的梳理出实验进行顺序，以下是完成编程优化的代码的内容和顺序：

一、pthread

1. pthread 方法探索与分析：这一部分包括对于线程管理方式与数据划分方式的研究和分析，最终选择出效果最好的方式继续进行以下实验；

2. 普通 LU 算法 pthread 实现：实验首先针对普通的 LU 算法展开研究；
3. pthread 方法与 simd 方法结合：在实现对于普通 LU 算法的 pthread 的并行化之后，在此基础上加入 simd 并行加速方法；
4. 串行优化算法 count_LU 的 pthread 实现：之后讨论进阶的串行算法与 pthread 结合的效果；
5. 串行优化算法 count_LU 的 pthread 实现 +simd：同样在以上基础上加入 simd 进行尝试。

二、openmp

1. openmp 方法探索与分析：这一部分包括对于线程管理方式与数据划分方式的研究和分析，最终选择出效果最好的方式继续进行以下实验；
2. 普通 LU 算法 openmp 实现：实验首先针对普通的 LU 算法展开研究；
3. openmp 方法与 simd 方法结合：在实现对于普通 LU 算法的 openmp 的并行化之后，在此基础上加入 simd 并行加速方法；
4. 串行优化算法 count_LU 的 openmp 实现：之后讨论进阶的串行算法与 openmp 结合的效果；
5. 串行优化算法 count_LU 的 openmp 实现 +simd：同样在以上基础上加入 simd 进行尝试。

实验将根据以上代码在 ARM 和 x86 平台下的运行结果，进行多方面的比较分析，并针对多种现象进行探究，还将使用 VTune 等工具对实验现象进行辅助分析。同时，还会在其它方面（如把任务卸载到 gpu 上），对算法性能优化进行尝试。

本实验涉及的所有代码都已上传至Github。

4 实验数据及结果分析

为方便测算与分析，我在实验中使用的矩阵维度均为 2^n，以下报告中涉及维度时所述均为 n（表示是 2 的 n 次方）。

我们通过逐步改变矩阵大小，测算不同算法完成 LU 分解所需时间，来对不同算法的性能进行比较。同时，对不同规模下，相同算法的优化比的变化趋势进行分析；对相同规模下，不同算法的实际与理论优化比的不同进行分析，以进一步分析深层原因。

同时需要注明的是，除对于并行方法使用不同线程数的效果的分析外，其余模块的分析均建立在使用 8 线程并行算法的测试数据基础上。

4.1 编程方法选择的探索

4.1.1 pthread 方法探索

一、线程管理

根据实验指导书上有关 pthread 线程管理的相关指导意见，为探究针对 LU 算法最好的线程管理方法，我在此次实验中尝试使用动态线程、静态线程和静态线程并将双重循环皆置于线程函数中三种方法进行 LU 算法的 pthread 并行编程，并根据实验数据对三种方法的优劣进行探究。

首先我对结果进行了预测分析，由于动态线程创建的方式就是在每一轮消元的过程中，重新创建线程，可能带来严重的额外开销；而静态线程创建的方式则是在最初一次性创建好全部线程，然后在每一轮中重新进行任务的划分。所以估计静态线程会比动态线程更加适合这次任务，又因为如果将双

重循环都置于线程函数中，对外层循环保持不动，将内层循环拆分分配工作线程。这样的方式可以避免一直创建和销毁线程，又可以避免线程间更加复杂的通信（同步），所以猜测静态线程并且将双重循环皆置于线程函数中的效果最好，静态线程居于次席，动态线程效果最差。

将使用不同线程管理方法的代码在华为鲲鹏服务器上（ARM 平台）运行后得到数据（代码均使用 8 线程进行并行处理），与未优化的 LU 算法对比后得到如下结果，如图4.3所示：

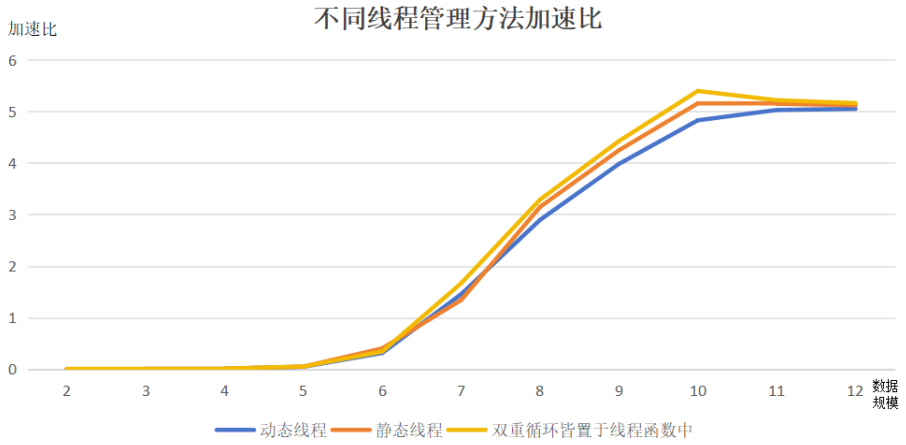


图 4.3: 不同线程管理方法在 ARM 平台上运行的加速比折线图

从图中可以看到，将双重循环皆置于线程函数中的方法的加速比在数据规模为 2-12 时均为最高，而静态线程方法基本居于次席，动态线程方法加速效果最差，符合预期结果。

同时，我们根据折线图也可以发现，当数据规模较小时，三者优化比均不明显，所以差异也不是很明显；当数据规模逐渐增大时，优化比提升，这时差异逐渐显现出来，这是由于创建与销毁线程的开销以及线程间通讯造成的不同导致优化比的差异；而当数据规模再进一步提升，有关线程的时间开销占比逐渐降低，我们可以看到此时三种方法的加速比逐渐趋于稳定且逐渐接近。

所以，根据以上结果，在接下来的实验中，我将使用将双重循环皆置于线程函数中的线程管理方法继续进行实验。

二、数据划分

本次实验中，除了对不同的线程管理方法进行了尝试和分析意外，还从数据划分的角度出发，考虑不同的数据划分方式，对于并行算法优化效果的影响，并分析不同的原因。在这一部分，我对比了对于矩阵的行划分和列划分两种不同方式，并测出它们在不同问题规模下的表现效果，比较并分析了它们差异的原因。情况如下图4.4所示：

从图中可以看到，行划分的加速效果远好于列划分的加速效果，我认为可能是由于在内存中，矩阵的数据是按行存储的，因此行划分更符合数据访问模式，可以更好地利用缓存，减少缓存未命中的情况，从而提高内存访问效率。同时，在 LU 分解算法中，存在数据依赖性，后续计算需要依赖前面的计算结果。使用行划分的方法，使得每一整行均由某一个线程处理，更好地利用了这种数据依赖性，减少了线程间的同步开销。

由于列划分可能导致极低的内存访问效率和极高的线程间的同步开销，所以我们可以在图中看到，使用列划分的方式，数据加速比非常低，当数据规模足够大时，加速比也都低于 1.5，这与使用的 8 线程所希望达到的加速比相差甚远。

根据以上数据和分析结果，在接下来的实验中，将使用行划分的数据划分方法对 pthread 并行优化的其他方面进行研究。

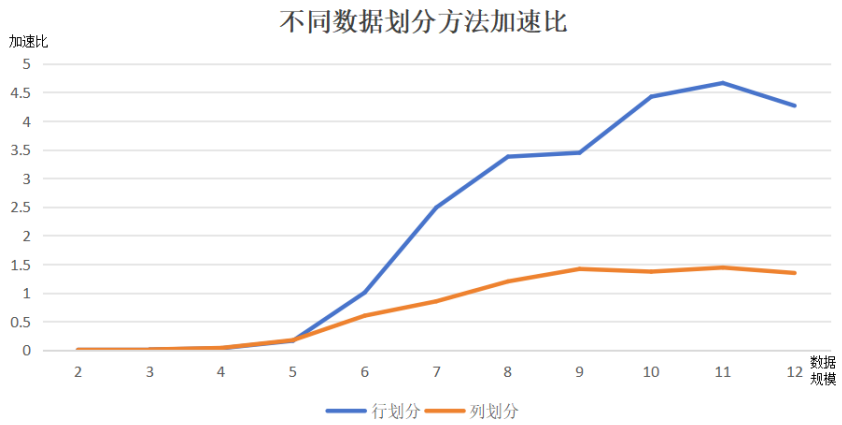


图 4.4: 不同数据划分方法在 x86 平台上运行的加速比折线图

4.1.2 openmp 方法探索

openmp 为我们在任务划分提供了多种方式，即在进行数据划分 schedule 的时候，为我们提供了 static、dynamic 和 guided 三种不同的任务划分方法。

在静态划分中，任务被分配给线程的方式是在编译时确定的，每个线程被分配一个或多个连续的任务块。这些任务块在执行过程中不会改变分配给线程的顺序。在动态划分中，任务在运行时动态地分配给线程，每个线程在完成一个任务后会请求下一个任务。引导划分结合了静态和动态划分的优点，任务被动态地分配给线程，但任务块的大小逐渐减小。初始时，任务块的大小较大，随着任务的进行，任务块的大小逐渐减小。

为探究不同的任务划分方式对于 LU 算法性能的影响，选择最好的任务划分方法，我在此次实验中对三种方法都进行了尝试，并进行了比较分析。根据三种方法的算法在华为鲲鹏服务器（ARM 平台）下的运行结果，我计算算出了三种方法对应的加速比，如图4.5所示：

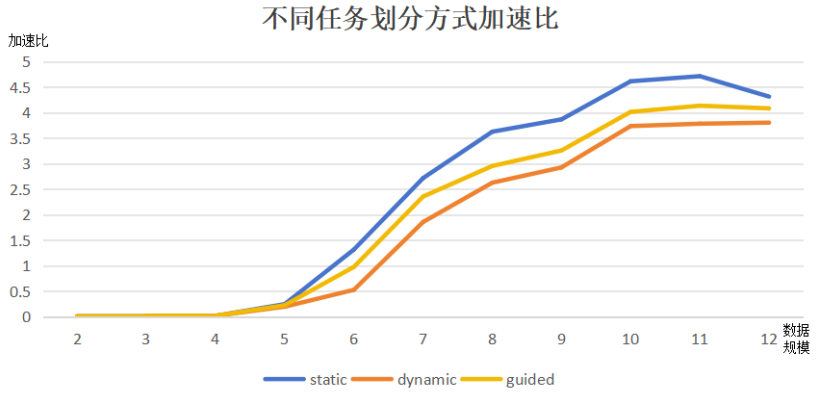
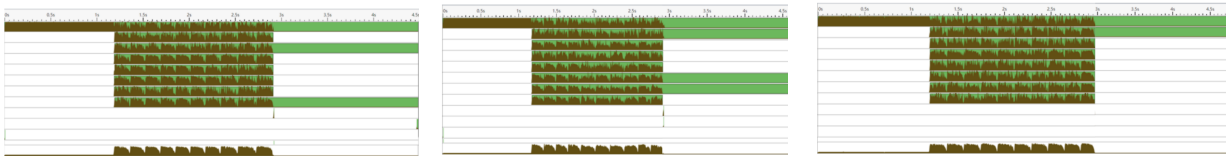


图 4.5: 不同任务划分方法在 ARM 平台上运行的加速比折线图

根据上面的折线图，我们可以看到静态划分的效果最好，这是因为 LU 算法所涉及的矩阵运算是大小相对均匀且已知的，这个通过静态划分的方法一次性划分完并分配给各线程也能保证不错的负载均衡效果。而动态线程方法是在运行过程中不断调整，这确实可以更好地处理负载不平衡，但在 LU 算法的问题上就显得没有什么必要，这还会带来一定的线程调度开销，导致时间开销更大，所以在三种方法中优化效果垫底。引导划分也会进行动态调整，但是相比于动态划分，线程调度开销减小，所

以优化效果居于静态划分和动态划分中间。

为了更好地体现三种方法间的差异，并进行探究，我在 x86 平台下对比了这三种方法在数据规模 n=10（矩阵维度为 1024）时的性能表现，并将使用 VTune 性能分析工具对于这三种方式的 CPU 占用时间进行检测，结果如图4.6所示。



(a) 静态划分 (b) 动态划分 (c) 引导划分

图 4.6: x86 平台下 openmp 不同任务划分方式的 CPU 占用时间对比

从图中可以看出，静态划分方式的 CPU 占用时间分配与其他两种方法相比比较不均衡，这说明静态分配还是会带来一定的负载不均问题，但是其占用时间是最短的，说明在这个问题下，静态分配是最优方案；动态划分方式的 CPU 占用时间分布最为均衡，但是占用时间长；而引导划分的方法其性能与预想中的效果一样，介于静态划分和动态划分之间，说明其在一定程度上可以减轻负载不均的影响。

根据以上数据和分析结果，在接下来的实验中将使用静态划分的任务划分方法对 openmp 并行优化的其他方面进行研究。

4.2 pthread 性能探究

在以上对于 pthread 的方法进行探索后，使用已经选择出的效果最好的 pthread 方法进行接下来的实验。

4.2.1 在多平台上针对不同数据规模的分析

首先，我们仍然在 ARM 和 x86 两个平台测试出相应的实验数据。实验数据如表2所示：

规模	x86 运行时间（ms）		ARM 运行时间（ms）	
	未优化	pthread	未优化	pthread
2	0.001623	0.320776	0.00097413	0.319809
3	0.003654	0.361904	0.0016665	0.326855
4	0.012284	0.375121	0.00420614	0.375484
5	0.062131	0.380398	0.0190396	0.3722
6	0.408137	0.406388	0.12656	0.371093
7	2.82359	1.13409	1.01232	0.607588
8	21.8062	6.45598	8.46296	2.5744
9	173.723	50.4093	69.7063	15.7661
10	1441.2	325.725	619.605	114.888
11	12126.7	2647.35	5763.13	1105.2
12	143345	33591.1	93122.6	18055.308

表 2: 实验数据

根据以上实验数据，我们可以看到无论在 ARM 还是 x86 平台上，当数据规模较小时，pthread 优化的效果都远不如未优化的 LU 算法，这是因为，当数据规模较小时计算并不复杂，而 pthread 方法

带来的线程创建、销毁和通信的开销远高于 pthread 所带来的优化，所以此时优化产生负面效果。

随着数据规模的增加，pthread 带来的优化逐渐明显，这是由于数据规模增加并行优化带来的时间优化占比增多，线程创建、销毁和通信的开销在总时间中的占比逐渐下降。

为进一步发现数据中隐藏的现象，我们计算出相应的优化比并作出相应的折线图，如下表3和图4.7所示：

数据规模	2	3	4	5	6	7	8	9	10	11	12
x86 下优化比	0.00506	0.0101	0.032747	0.16333	1.0043	2.4897	3.37767	3.44624	4.42459	4.66411	4.26736
ARM 下优化比	0.003046	0.0051	0.011202	0.51154	0.34104	1.6661	3.2874	4.42128	5.37571	5.21456	5.15763

表 3: 不同平台下优化比

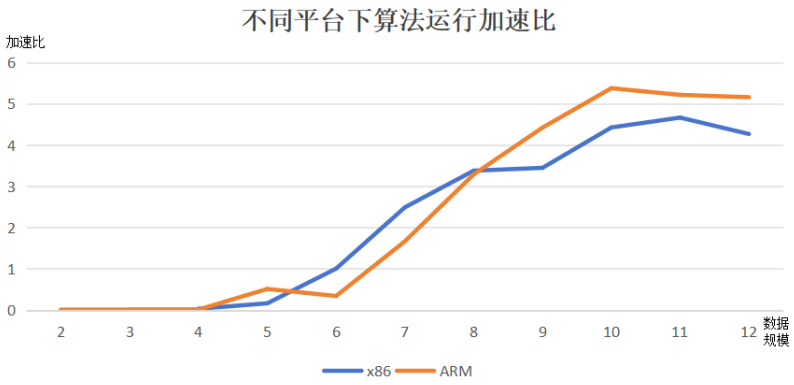


图 4.7: pthread 并行算法在 ARM 和 x86 平台上运行的加速比折线图

我们发现，当数据规模足够大时，优化效果并不会不断上升至我们采用线程数（8）附近，而是会停留在 5 左右，这是因为采用 pthread 方法在并行优化计算的同时，也会带来线程创建和销毁的开销，此外，采用静态任务划分的方法可能导致任务划分的并非绝对均匀，这导致线程间负载不均衡，影响优化效果。同时，多线程的并行计算可能会涉及到对同一行或同一列的数据访问，在并行计算过程中可能会引入额外的内存访问冲突，降低了性能的提升。因此，优化比无法达到 8，在实验中停留在 5 左右。

此外，我们还能注意到随着问题规模的增加，运行时间呈指数级增长。特别是在规模 n=11 和规模 n=12 上，运行时间增长迅速。这是因为当数据规模达到 11 时，缓存已经无法容纳下一次运算所需的数据，运行过程中被迫得去内存中找寻数据，这会带来极大的访存开销。这也是当数据规模太大时，优化比停滞不前甚至略有下降的原因之一。

接下来，根据实验数据进行横向对比，通过对比 ARM 和 x86 架构上使用 pthread 方法的运行时间和优化效果，我们发现在小规模问题下（如规模 2 至规模 6），在 x86 架构下的运行时间和优化效果都要略低于 ARM，但随着问题规模的增加，x86 上的运行时间开始显著高于 ARM，优化效果也会低于在 ARM 平台下。出现这种现象可能是因为 x86 架构通常具有更高的时钟频率和更复杂的指令集，所以 x86 处理器在处理小规模数据时具有更好的性能。相比之下，ARM 架构的处理器可能在处理小规模数据时性能稍逊一筹。但是，当数据规模增大时，ARM 架构的处理器可能会展现出更好的性能。这可能是因为 ARM 架构在处理大规模数据时能更好地利用其多核心架构或者更高效的内存访问方式。

4.2.2 针对使用不同线程数的分析

接下来，我们对不同线程数的情况进行讨论。

基于上面的分析，可以预测，无论使用的线程数是多少，当数据规模足够大时，其相应的优化比都无法达到使用的现成的数量。以下是基于 x86 平台测试的结果，如表4所示：

数据规模	x86 运行时间（ms）			
	4 线程	6 线程	8 线程	10 线程
2	0.221321	0.248354	0.320776	0.397242
3	0.195163	0.324254	0.361904	0.401231
4	0.205214	0.319342	0.375121	0.40527
5	0.241633	0.278574	0.380398	0.398912
6	0.575982	0.627625	0.406388	0.809909
7	2.23563	2.0982	1.13409	1.47397
8	12.6711	9.2314	6.45598	6.24257
9	105.354	57.6472	50.4093	48.2456
10	513.248	352.972	325.725	297.176
11	5467.88	3571.73	2647.35	2251.54
12	69255.23	42579.3	33591.1	27019.23

表 4: 实验数据

根据以上数据，我们可以看到，无论使用的线程数设置为多少，其运算的优化比都始终无法达到相应的线程数。这样的现象符合预期结果，原因已在前一部分分析。

此外，我们根据表中数据还可以发现一个很有意思的现象，当数据规模较小时，使用线程数少的反而运行速度比线程多的速度快。这是由于当数据规模较小时，使用多线程产生的优化不如创建和销毁线程以及线程间通信所需开销来得大，那么此时，线程数多反而成了浪费。所以我们可以看到，在数据规模小的时候，线程数少，运行速度更快，而当数据规模逐渐增大，这时多线程的优势就体现出来了，线程数多产生的优化效果逐渐超过了线程数少的。

4.3 openmp 性能探究

4.3.1 针对不同数据规模的分析

在第一个模块确定使用 openmp 编程的相应的一些方法之后，我们就可以测算出算法相应的效益，我在华为鲲鹏服务器（ARM 平台）上，在不同数据规模的情况下对算法性能进行了测试，实验数据如下表5所示:

规模	ARM 运行时间（ms）		加速比
	未优化	openmp	
2	0.00097413	0.083433	0.01167
3	0.0016665	0.114141	0.0146
4	0.00420614	0.121343	0.03466
5	0.0190396	0.15133	0.12582
6	0.12656	0.214552	0.58988
7	1.01232	0.413532	2.44798
8	8.46296	2.1353	3.96336
9	69.7063	12.4637	5.59275
10	619.605	107.213	5.7792
11	5763.13	1034.32	5.5719
12	93122.6	18032.28	5.15422

表 5: 实验数据

根据以上实验数据，我们可以看到当数据规模较小时，openmp 的优化后的效果也是不如原本 LU 算法的效果的，而随着数据规模增大，优化效果逐渐明显。这个现象产生的原因与 pthread 情况下产生这种现象的原因基本一样，分析可以见前一个部分，这里就不再赘述。

同时为了探究优化比的规律，我们做出相应的优化比折线图，如图4.8所示：

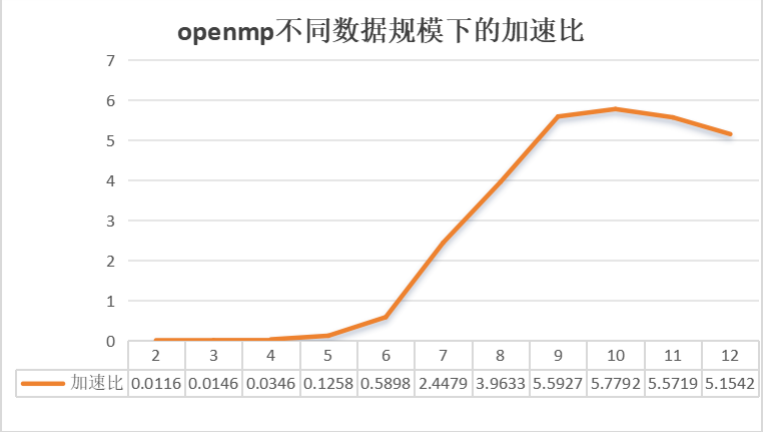


图 4.8: openmp 并行算法在 ARM 平台上运行的加速比折线图

根据折线图，我们可以很直观的看出，openmp 算法带来的优化比也会出现一开始很低，之后逐渐上升，但是会低于我们所使用的线程数（8），在这次实验中优化比低于 6，当数据规模过大时，优化比会略微有下降，这些现象都与 pthread 算法中出现的现象极其相似，所以出现这些现象的原因这里就不再赘述，详见上一个部分。

4.3.2 针对使用不同线程数的分析

类似的，我们在 ARM 平台下可以测试出不同线程数的 openmp 方法优化效果的差别。由以上 pthread 分析结果，我们可知，当数据规模较小时，多线程编程的优化效果并不明显，为了能之间体现出多线程编程的优势，同时减少 cache miss 对优化效果的影响，我选定矩阵规模为 n=10，在这样的矩阵规模下对不同线程数的 openmp 方法进行测试，得到如下表6的结果：

线程数	2	3	4	5	6	7	8
ARM 下优化比	1.3863	2.0854	3.1342	3.57697	4.2167	5.1075	5.7792

表 6: 实验数据

根据上表做出折线图，如图4.9所示：

根据上图，我们发现随着线程数的增加，openmp 的加速比基本呈现出一个线性增加的趋势，这种线性增加的趋势表明程序能够很好地将工作负载分配给不同的线程，同时也说明了 openmp 线程之间的通信和同步开销不会很大，所以才能使得加速比能够近似于线性增长。

4.4 pthread 方法与 simd 方法结合的性能探究

为进一步提升算法的性能，尝试将多线程方法与上一章的 simd 方法进行结合以获得更大的并行优化收益。

接下来先尝试将 pthread 方法与 simd 方法结合。本节内容以 pthread 方法与 neon 结合为例，分析“pthread+neon”的并行优化在 ARM 平台下的运行效果。

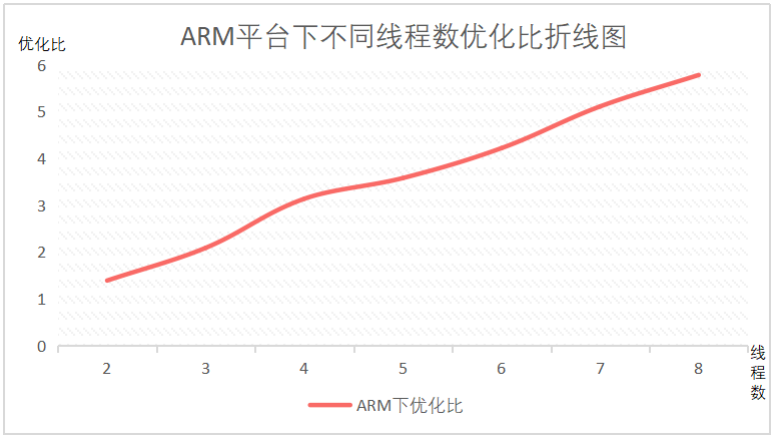


图 4.9: 在矩阵规模 n=10 时 openmp 在 ARM 平台下不同线程数的优化比折线图

实验数据如下表7所示：

规模	ARM 运行时间（ms）			NEON 产生的加速比
	未优化	pthread	pthread+neon	
2	0.00097413	0.319809	0.341008	0.937834
3	0.0016665	0.326855	0.352119	0.928252
4	0.00420614	0.375484	0.399657	0.939516
5	0.0190396	0.3722	0.368411	1.010285
6	0.12656	0.371093	0.352413	1.053006
7	1.01232	0.607588	0.504621	1.20405
8	8.46296	2.5744	1.15116	2.23635
9	69.7063	15.7661	6.88185	2.29097
10	619.605	114.888	41.8633	2.74436
11	5763.13	1105.2	333.922	3.30975

表 7: 实验数据

由上表数据所示，当数据规模较小时，neon 产生的加速比并不明显。这是因为串行算法虽然能够同时并行多条数据，但是每一条指令所消耗的时间比更长。而指令消耗带来的额外开销甚至会比并行处理所能带来的优化时间更高，因而在矩阵规模比较小时，优化比也并不明显，甚至会慢于原来的串行代码；当矩阵规模变大，这种额外的时间开销与并行优化的时间相比，相对比例不断减小，因此优化比不断提高。这部分在上一章 simd 相关编程报告中已有详细分析。

为了进一步研究 neon 与 pthread 之间的关系，我从上一章 simd 相关编程的研究结果中找出了只使用 neon 时的优化比，并将其与和 pthread 结合后的 neon 的优化比进行对比分析，数据对比如下表8所示：

数据规模	2	3	4	5	6	7	8	9	10	11
与 pthread 结合后 neon 产生的加速比	0.937834	0.928252	0.939516	1.010285	1.053006	1.20405	2.23635	2.29097	2.74436	3.30975
单独 neon 产生的加速比	0.9607	0.9598	1.08237	1.45379	2.4623	2.8528	3.5452	3.7134	3.6713	3.8676

表 8: 实验数据

根据以上数据做出相应的对比折线图，如图4.10。从图像中，我们可以看到，与 pthread 结合后，neon 所带来的优化不如单独使用 neon 带来的优化多，我认为可能有以下两点原因：

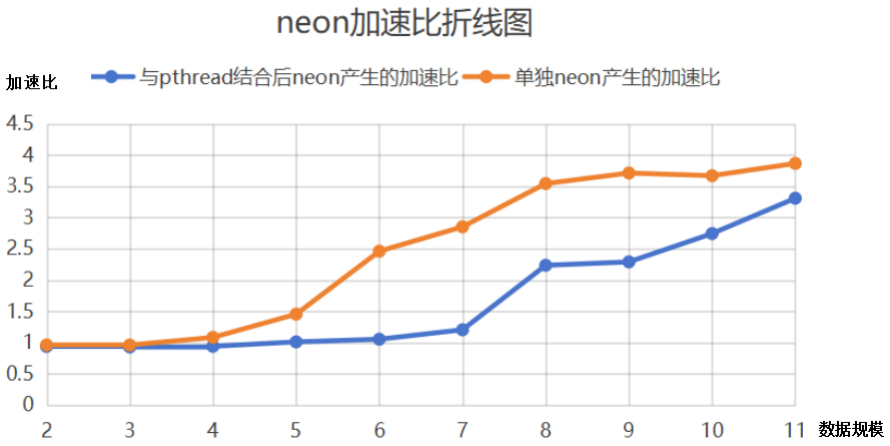


图 4.10: neon 优化比折线图

- pthread 分割数据会导致 NEON 处理的数据规模变小。NEON 在处理大规模数据时能够发挥更好的性能，因为它可以通过 SIMD 指令在单个线程中同时处理多个数据。当数据规模被分割成较小的块时，正如上面的分析，每个线程所处理的数据量减少了，指令消耗带来的额外开销在并行处理所能带来的优化时间中的占比更大，因而优化比有所降低，无法完全发挥 NEON 的优势。
- 线程间的通信和同步开销可能也是导致 NEON 优化效果变差的原因之一。当使用 pthread 创建多个线程时，这些线程可能需要在共享内存中进行数据交换和同步操作，这会引入一定的开销。尽管 NEON 技术可以在单个线程中实现 SIMD（单指令多数据）并行计算，但在多线程情况下，需要额外的同步操作来确保数据的一致性和正确性，这可能会对性能产生一定影响，导致加速比略低于单独使用 NEON 时。

4.5 openmp 方法与 simd 方法结合的性能探究

上面的讨论中研究和分析了 pthread 与 neon 在 ARM 平台下结合产生的相关现象，在这部分，我将以 openmp 与 SSE 和 AVX 在 x86 架构下的结合为例，探索 openmp 与 simd 方法结合所产生的现象。

首先，在实验中，我选择了 sse4 路向量编程和 AVX8 路向量编程，希望能通过多组数据探寻规律。此外，我还找到上次实验中获取到的信息，得到单独实现 sse 和 avx 编程时算法的优化效果。

openmp 与 simd 结合后的优化效果以及单独使用 simd 的优化效果汇总如下表9所示：

从图中我们可以看到，与 pthread_neon 相似，当数据规模较小时，无论是 omp_sse 还是 omp_avx 优化效果都是负效果。随着数据规模增加，优化效果逐渐变好。但是与单独使用 SSE 或 AVX 相比，结合后的 SIMD 方法产生的优化并不如单独使用 SIMD 来得高，换句话说，结合后产生的优化效果会略低于单独使用 simd 产生的优化与单独使用 openmp 方法产生优化的乘积，这与 pthread_neon 在 ARM 下运行产生的现象也极其相似，具体原因分析上一节已经由详细分析，这里就不再赘述。

然而同时，我们也要注意到，无论是 pthread 与 simd 结合还是 omp 与 simd 结合后，simd 产生的优化虽略低于单独使用 simd 产生的优化，但是优化效果还是十分明显的，这说明 pthread 和 openmp 多线程方法可以结合多种 SIMD 指令集架构，并且在各种指令集架构上的表现基本保持稳定，能够产生不错的结合优化效果，并没有出现某种指令集架构与多线程方法结合后不能够发挥很好的多线程优势的现象。

规模	SSE4 路向量优化比	AVX8 路向量优化比	omp_sse	omp_avx
2	0.981	0.75552	0.010483	0.008134
3	0.9931	1.2672	0.013523	0.009123
4	1.3068	1.49261	0.03487	0.041345
5	1.9449	1.94496	0.09673	0.143456
6	2.6105	3.1407	1.35246	2.01487
7	2.3955	4.39553	5.24564	6.88153
8	3.8688	5.5024	9.14776	14.3457
9	3.9285	6.53679	15.4355	26.1345
10	3.9683	6.94736	17.7674	28.7588
11	3.9792	7.39234	19.4576	36.3763

表 9: 实验数据

4.6 countLU 的 pthread 实现及 simd 实现

Count_LU 算法是优化的 LU 算法，主要针对 LU 分解中的计算过程进行优化。相比传统的 LU 分解算法，Count_LU 算法在计算下三角矩阵和上三角矩阵时采用了一种更高效的计算方式，可以减少乘法和加法的次数，从而提高计算效率。

以下是 Count_LU 算法重点代码:

```
void croutLuDecomposition(vector<vector<double>>& mat, vector<vector<double>>& lower,
vector<vector<double>>& upper) {
    int n = mat.size();

    for (int i = 0; i < n; i++) {
        lower[i][i] = 1.0;

        for (int j = i; j < n; j++) {
            double sum = 0.0;
            for (int k = 0; k < i; k++) {
                sum += lower[i][k] * upper[k][j];
            }
            upper[i][j] = mat[i][j] - sum;
        }

        for (int j = i + 1; j < n; j++) {
            double sum = 0.0;
            for (int k = 0; k < i; k++) {
                sum += lower[j][k] * upper[k][i];
            }
            lower[j][i] = (mat[j][i] - sum) / upper[i][i];
        }
    }
```

```
24 }
```

Count_LU 算法通过优化计算下三角矩阵和上三角矩阵时的乘法和加法操作，减少了计算量，提高了计算效率。同时重新设计了计算流程，减少了数据之间的依赖关系，提高了并行度，从而加速计算过程。

根据上一次实验的测试数据，我们发现，Count_LU 算法对于较小规模的矩阵，由于数据量较少，优化效果可能相对较小，因为内存访问和计算开销相对较低。而对于较大规模的矩阵，Count_LU 算法的优化效果可能更为明显，因为传统的 LU 分解算法在大规模数据下容易出现计算瓶颈，而 Count_LU 算法的优化方法能够更有效地提高计算效率，加速计算过程，总体而言 Count_LU 算法可以带来优化。

以上只是对算法进行串行的优化，而如果将 Count_LU 算法进行并行化处理，是否可以获得比直接并行化更好的效果。所以在此次实验中，为探究此问题，我尝试使用 simd 和 pthread 方法对 count_LU 算法进行优化。

实验数据如下表10所示：

数据规模	优化比			
	Count_LU	LU_pthread	Clu_pthread	Clu_pthread_sse
2	1.00143	0.00506	0.005013	0.005215
3	1.01247	0.0101	0.0118	0.00923
4	1.06324	0.032747	0.04251	0.032123
5	1.08465	0.16333	0.14253	0.15873
6	1.14766	1.0043	0.93569	1.135
7	1.09727	2.4897	2.5246	3.45347
8	1.12475	3.37767	3.53751	5.2576
9	1.23625	3.44624	3.52883	6.4567
10	1.20357	4.42459	4.48645	9.2623
11	1.25627	4.66411	4.86527	12.86
12	1.24462	4.26736	4.4264	14.371

表 10: 实验数据

从以上实验数据可以看出，当串行优化与并行优化结合后产生的优化效果确实会略高于单独使用并行优化的效果。这说明算法的优化需要综合考虑串行和并行两个方面，才能够实现最佳的效果。在串行方面，通过算法设计和代码优化来减少计算和内存访问的开销，提高效率；在并行方面，利用 SIMD 和多线程编程来并行加速算法的执行。综合考虑这些方面，才能够使算法达到更高的性能和效率，实现最佳的优化效果。

同时，我们发现在优化算法 count_LU 的并行优化中同样存在很多与 LU 算法并行优化相似的现象，比如数据规模小时优化比较低、优化比最终低于使用的线程数、与 pthread 结合使用的 sse 优化效果不如单独使用时来得好等等，这些现象的原因与前面分析过的 LU 算法中的相同现象的原因一样，这里就不再赘述。

4.7 pthread 方法与 openmp 方法的比较

在以上对于 pthread 和 openmp 方法进行多方面探索之后，接下来，我们根据实验数据对这两种方法的异同进行分析。

pthread 和 openmp 完整的实验数据如下表11所示。

从表中我们可以看到，当数据规模较小时，无论在哪个平台下，pthread 和 openmp 优化的算法都会起到副作用，运行时间都远大于普通的串行算法，这是由于无论是 pthread 还是 openmp 都需要付

数据规模	x86 平台下运行时间			ARM 平台下运行时间		
	未优化	pthread	openmp	未优化	pthread	openmp
2	0.001623	0.320776	0.101242	0.00097413	0.319809	0.083433
3	0.003654	0.361904	0.158245	0.0016665	0.326855	0.114141
4	0.012284	0.375121	0.142387	0.00420614	0.375484	0.121343
5	0.062131	0.380398	0.172576	0.0190396	0.3722	0.15133
6	0.408137	0.406388	0.336724	0.12656	0.371093	0.214552
7	2.82359	1.13409	0.913476	1.01232	0.607588	0.413532
8	21.8062	6.45598	5.23563	8.46296	2.5744	2.1353
9	173.723	50.4093	44.2485	69.7063	15.7661	12.4637
10	1441.2	325.725	321.245	619.605	114.888	107.213
11	12126.7	2647.35	2334.87	5763.13	1105.2	1034.32
12	143345	33591.1	34365.4	93122.6	18055.308	18032.28

表 11: 实验数据

出创建线程、销毁线程和线程间通信的代价，而这个代价就会导致当数据规模较小时，所付出的代价远超过优化带来的效益，就会使得优化产生副作用，这一点算是两种方法的共同点。

同时，我们也注意到，openmp 所付出的代价似乎比 pthread 小一点，在数据规模小时，openmp 所带来的副作用明显小于 pthread。这一点从两者编程特点上也能窥见一二。

在编程过程中，涉及到线程创建和销毁时，pthread 需要显式地调用 pthread 库中的函数来创建和销毁线程，通常需要手动管理线程的生命周期。而 OpenMP 是通过编译器的指令或者 API 调用来指定并行区域，编译器会负责生成并管理线程，无需手动创建和销毁线程。同时，pthread 方法需要使用共享内存或者消息传递等机制来实现线程间的通信和数据共享，需要手动管理线程间的同步和互斥。而 OpenMP 方法在并行区域内的变量通常是共享的，OpenMP 会自动处理线程间的同步和数据共享，无需显式地编写同步和互斥的代码。

在这样的编程区别下，openmp 会自动选择出比较好的线程创建、销毁和同步的方式，由编译器来管理这些事务，可以更好地优化线程的调度和资源利用，减少了一些与线程管理相关的开销。相比之下，pthread 由个人选择的方式效果就可能不尽人意，引入一些额外的开销，导致带来的副作用大于 openmp。

随着数据规模的增大，两种方法都开始对算法运行产生了明显的优化，而 openmp 基本上始终有着更好的优化效果，知道当数据规模到达 n=12 时，二者才基本持平。这一点也符合前面的分析结果，openmp 有着比 pthread 更好的线程管理性能，线程相关开销比较小，所以带来的优化效果更好，而当数据规模够大时，多线程带来的优化远远领先于线程开销，就使得两种方法的线程开销带来的副作用可以基本认为在同一个量级，这样就使得两种方法在数据规模大的运算上效果逐渐接近。

总的来说，OpenMP 使用指令预处理器和运行时库来管理线程并行化，Pthread 通过创建和控制线程来实现并行化，开发者需要显式地创建和管理线程。openmp 相对于 pthread 提供了更高层次的抽象和自动化，简化了并行程序的编写和管理，同时也减少了一些线程管理的开销。但是 Pthread 相对于 OpenMP 更加灵活，pthread 可以更精细地控制线程的创建、同步和销毁，实现更复杂的并行模式。

5 小组实验总体分析

本次实验中，我与小组成员黄明洲共同进行了高斯消去算法和 LU 分解算法的多线程编程研究，其中，由我负责 LU 分解算法的多线程编程研究，他负责高斯消去算法的多线程编程研究。

根据我们在实验中得到的数据和分析结果，我们对于两种算法的共同分析得到了以下结果：

高斯消去算法和 LU 分解算法都适合被并行化处理，但是在多线程环境中，LU 分解算法更容易并行化，因为 LU 分解算法的主要计算包括矩阵的分解和前/后向替换，这些操作可以很容易地并行化处理。相比之下，高斯消去算法的计算过程是逐步的，每一步都依赖于前一步的结果，这会增加并行化的复杂性。因而在实验数据中，我们可以看到，使用相同线程数，在同样的数据规模下，LU 分解算法得到的加速比更高，而高斯消去算法得到的加速比稍低一点。

同时，在性能方面，由于 LU 分解算法更容易实现多线程并行化，通常会比高斯消去算法具有更好的性能。通过使用多线程并行计算，LU 分解算法能够更有效地利用多线程的优势，提高计算效率，加快求解过程。

从多线程分解的角度来看，高斯消去算法更适用于小规模问题，特别是当问题规模不大时，而 LU 分解算法更适用于大规模问题，特别是当问题规模较大且需要高效处理时。高斯消去算法在解决线性方程组时效果较好，而 LU 分解算法则在解决大型稠密矩阵的求解和矩阵求逆等问题时表现更出色。

6 个人任务完成总结

本次实验中，我们小组完成了对于 Guass 和 LU 算法的 pthread & openmp 多线程加速，并对两种算法的特点与用途进行分析。

在我的任务部分，我完成了对 LU 算法的 pthread 和 openmp 并行优化实验，在 x86 和 ARM 平台上进行了实验，并对算法性能进行了全面分析。

在实验中，我通过对比串行和并行算法的运行时间和性能表现，深入分析了多线程并行优化对 LU 算法性能的影响。探索发现并行算法的性能相关性质，以及充分利用串行与并行方法，提高算法的运行速度。同时总结各种方法在不同平台下的性能表现，比较 pthread 和 OpenMP 的优缺点，并讨论 simd 加速对算法性能的影响。

通过本次实验，我进一步加深了对并行编程工具 pthread 和 OpenMP 的理解，并且了解了它们在不同平台下的性能特点。同时，在实验中我不断地尝试和调整代码，涉及到对线程管理方式、数据划分策略的探索，以找到最优的实现方式也让我对多线程编程有了更深刻的理解。

总的来说，本次实验任务完成度较高，成功实现了对 LU 算法的多线程并行优化，并且对算法性能进行了全面分析。通过实验，我不仅提高了对并行计算和优化技术的认识，对并行编程工具的使用和优化有更深入的理解，也为进一步的算法优化和并行计算研究打下了基础。