南開大學
Nankai University

计算机学院

并行程序设计实验报告

Gauss 消元的多线程 Pthread 和 OpenMP 编程的研究

姓名：刘心源

学号：2112614

专业：计算机科学与技术

2023 年 5 月 13 日

目录

1 问题描述

在进行科学计算的过程中，经常会需要对多元线性方程组进行求解。一种有效的方法就是 Gauss 消去法。通过对线性方程组的线性矩阵进行自上而下的逐行消去，将其变为主对角线元素均为 1 的上三角矩阵。

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

图 1.1: Gauss 消去法的数学表示

考虑在整个消去过程中，如1.2所示，主要包含两个过程：

1. 对于当前的消元行，应该全部除以行首的元素，使得该行转化为行首元素为 1 的行
2. 对于之后的每一行，用该行减去当前的消元行，得到本次的消元结果

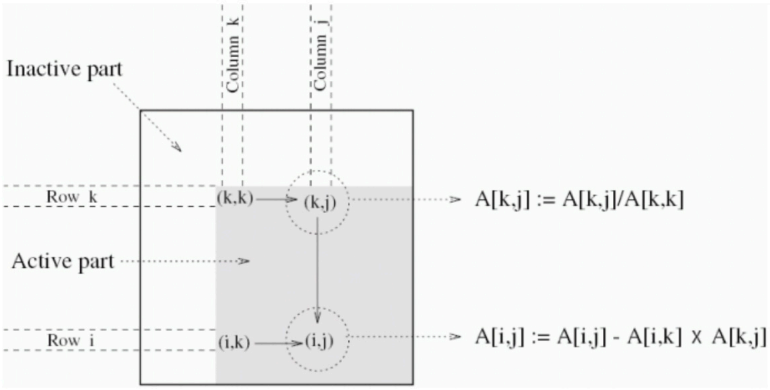


图 1.2: Gauss 消去法的逻辑表示

针对这两个部分，我们采用并行的方式进行性能的优化。本次实验，采用多线程 Pthread 和 OpenMP 编程，结合 SIMD 并行指令架构，针对上述两个过程进行优化。

- 多线程 Pthread

  对于多线程 Pthread 我们可以采用不同的任务划分方式并对比其性能差异，此外还可以考虑并行优化加速比随着问题规模和线程数量的变化情况，本次实验也在多个平台上进行测试来对比不同平台上多线程编程的性能差异；

- OpenMP

  对于多线程 OpenMP 我们可以考虑采用不同的任务划分方式并对比其性能差异，对比 OpenMP 的 SIMD 优化和手动的 SIMD 优化之间的差异，对比 OpenMP 多线程任务与手动 Pthread 多线程之间的性能差异。此外还将考虑并行优化的加速比与问题规模和线程数量的变化情况。本次实验也在多个平台上进行测试来对比不同平台上多线程编程的性能差异。

2 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 处理器	Intel Core i7-1165G7
CPU 主频	2.60GHz	2.80GHz
L1 Cache	64KB	48KB
L2 Cache	512KB	1.25MB
L3 Cache	48MB	12MB
指令集	Neon	SSE、AVX、AVX-512
核心数	1	8
线程数	8	16

表 1: 实验环境

3 Pthread 实验

3.1 Pthread 并行处理

Gauss 消去是一种常用的线性方程组求解方法。在该过程中，每一轮的消元过程主要包含两个阶段。首先，针对消元行进行除法运算，计算出除数，然后对于剩余的被消元行，依次减去消元行的某个倍数，以此消去变量。

然而，在每一轮的过程中，除法操作和消元减法操作之间存在着严格的先后顺序。即必须首先完成消元行的除法操作之后，才能够执行被消元行的减法操作。因此，需要引入信号量进行同步控制。当 0 号线程完成了对于消元行的除法操作之后，依次向其余挂起等待的线程发送信号，之后所有线程一起并行执行被消元行的减法操作。

为了提高效率，在执行消元操作时，可以采用分散的划分方式，将剩余的被消元行分配给不同的线程。不同的线程之间执行的工作是完全一致的，并且由于不同行之间并不存在数据依赖，因此可以避免线程之间的通信开销。

然而，由于不同的线程在执行消元操作时所需的时间可能并不相同，因此需要在所有线程完成本轮被分配的消元任务之后进行一次同步控制。这一次同步控制需要保证所有线程都已经完成了消元操作。为了实现这一目标，可以采用 barrier 进行同步控制。即只有当所有的线程都完成了消元任务之后，才会进入下一轮的消元。

在实际应用中，为了进一步提高效率，可以采用一些优化策略，如行交换、部分选主元、分块消元等。这些策略可以有效减少计算量，提高求解速度，从而在实际工程中得到广泛应用。

3.2 数据块的划分设计

在进行任务划分时，我们需要考虑数据规模对于计算效率的影响。在给出的样例中，采用等步长的数据划分方式，虽然简单易行，但会存在一定的缺陷。当数据规模比较大的时候，由于 CPU 的 L1 cache 大小有限，很有可能会导致在访问下一个间隔为线程数的行的时候出现 cache miss，这就会强制 CPU 从 L2、L3 甚至内存中去读取数据，造成额外的访存开销。

为了更好地利用 cache 优化，我们需要设计一种能够充分利用 cache 的数据划分方式。我们建议采用基于块的数据划分方式，块（block）划分是指将一个大任务划分为多个小块（block）的过程。这些小块可以在不同的线程中同时执行，从而加快整个任务的完成速度。通常情况下，一个大任务可以被划分为若干个小块，并将每个小块分配给不同的线程执行。在执行过程中，每个线程独立地处理自

己的小块，不会互相干扰。当所有线程都完成自己的任务后，整个大任务也就完成了。块划分可以提高程序的并发性和性能，但也需要合理的任务划分和同步措施，以确保不会出现死锁、竞争等问题。同时，块划分的效果也受到硬件、算法和数据结构等因素的影响。

这样做的好处是，当线程正在处理当前块时，CPU 可能会提前预取下一块的数据到 cache 中，使得下一次进行数据访问的时候，能够尽快在 cache 中命中数据项，减少了不必要的访存开销。通过采用基于块的数据划分方式，我们能够最大限度地利用 CPU 的 cache，减少不必要的访存开销，从而提高程序的运行效率。同时，我们也需要在块的大小和线程数之间进行权衡，以避免过度划分导致 cache 空间浪费或线程间通信开销增加。因此，在设计数据划分方案时，需要综合考虑各种因素，以实现最佳的性能优化效果。

3.3 数据动态划分设计

在任务划分过程中，由于不同线程所需的处理时间可能不同，甚至由于数据规模不是线程数量的整数倍，导致一些线程在某些轮次中处于空等待状态。这是由于数据划分过于细粒度导致的线程负载不均衡所致。

为了解决这个问题，可以采用动态数据划分方式。动态数据划分是指在程序运行时根据数据的实际情况动态划分数据。在执行消元行减法操作时，不明确指定某个线程对哪部分数据执行任务，而是根据各个线程任务完成情况动态划分数据。通过一个全局变量 index 指示当前处理到哪一行。当一个线程完成其任务时，会检查关于 index 的互斥量，如果该互斥量未被锁定，则说明可以进行任务划分。该线程会锁定关于 index 的互斥量，并将 index 所指的行分配给自己。任务分配完成后，线程会释放互斥量并执行所分配的任务。动态数据划分可以根据数据的大小、处理能力等因素来自适应地调整数据的划分方式，以提高程序的性能和并发性。动态数据划分需要在程序运行时动态分配和管理线程，因此会增加一些额外的开销和复杂度。这种动态数据划分方式可以确保每个线程都在执行分配给它的任务，避免线程负载不均衡导致的空等待现象。只有在所有线程完成任务后才会进入下一轮迭代，因此那些空等待的线程会浪费 CPU 计算资源。这是本实验设计进行优化的方向。

3.4 不同数据规模和线程数下的性能研究

线程的创建、调度、挂起和唤醒等操作相对于简单计算操作而言，需要更多的时间开销。因此，可以推测当问题规模较小时，线程调度的额外开销会抵消多线程优化的效果，甚至会导致多线程比串行算法更慢的情况。但随着问题规模的增加，线程之间调度切换所需时间相对于线程完成任务所需时间而言已经占比很低，从而能够展现出多线程并行优化的效果。因此，我们设计实验探究在不同数据规模下，多线程并行优化算法的优化效果。主要研究部分如下：

- 数据规模对性能的影响：

  随着数据规模的增大，程序的运行时间会增加，因为需要更多的计算和存储资源。因此，可以研究不同大小的数据集下，程序的运行时间和各种算法的加速比，以找到最优的数据规模。

- 线程数对性能的影响：

  线程数的增加可以提高程序的并发性和效率，但也会增加线程间通信和同步的开销，甚至可能导致性能的下降。因此，可以研究在不同线程数下，程序的运行时间和加速比，以找到最优的线程数。

- 数据规模和线程数的交互影响：

数据规模和线程数之间存在一定的交互影响。例如，在数据规模较小时，增加线程数可能无法提高程序的效率，反而会增加线程间通信的开销。因此，可以研究在不同数据规模和线程数下，程序的运行时间和效率，探究最优的组合方式。

4 Pthread 实验结果分析

4.1 ARM 平台

4.1.1 Pthread 并行实现及性能对比

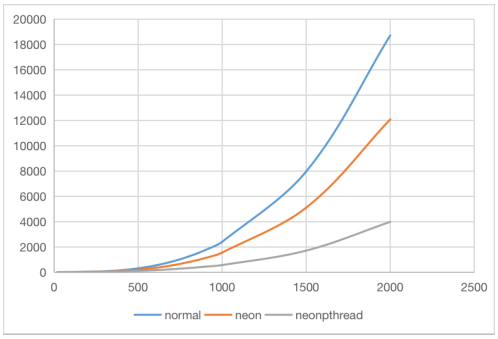
为了能够探究 pthread 并行算法的优化效果，考虑调整问题规模，测量在不同任务规模下，pthread 并行优化算法对于普通串行算法和 SIMD 向量化优化算法的加速比。在本次实验中，在 pthread 并行算法融合了 SIMD 的向量化处理。在 ARM 平台上，SIMD 的实现是基于 Neon 指令集架构的。为了能够比较全面的展现并行优化效果随问题规模的变化情况，在问题规模小于 100 时步长为 10，大于 100 并小于 1000 时步长变为 100，当问题规模大于 1000 时，步长调整为 1000. 三种算法的在不同问题规模下的表现如下表2所示。

n	normal	neon	neonpthread	n	normal	neon	neonpthread
20	0.020856	0.016752	1.33704	**300**	67.0514	42.8051	41.4113
30	0.0698238	0.05198	2.20288	**400**	147.802	94.5649	69.3711
40	0.1642	0.11458	3.29453	**500**	295.778	188.252	104.666
50	0.3212	0.21632	4.36195	**600**	515.451	330.389	151.049
60	0.553289	0.37287	5.1717	**700**	824.137	525.026	227.272
70	0.87642	0.568881	6.24966	**800**	1229.45	787.561	317.7
80	1.30326	0.85414	6.98589	**900**	1745.21	1112.19	427.442
90	1.85105	1.17255	7.62749	**1000**	2396.25	1542.22	552.661
100	2.53538	1.65095	8.31717	**1500**	7958.38	5095.92	1697.52
200	19.9153	12.8058	20.3088	**2000**	18715.6	12079.3	3972.67

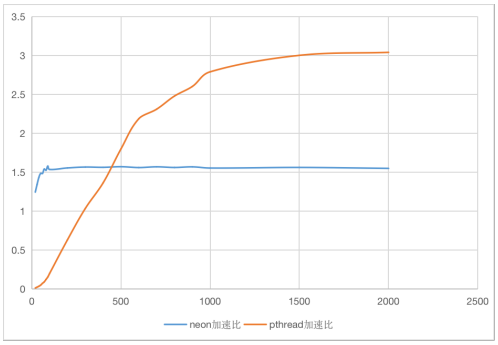
表 2: 不同数据规模下 SIMD 和 pthread 结合优化的效果

在实验设计中，我们采用了 SIMD 向量化处理和 pthread 多线程优化来提高算法的运行效率。我们采用四路向量化处理实现了 SIMD 优化算法，同时采用了 8 条线程的 pthread 多线程模型。其中一条线程负责除法操作，剩余的 7 条线程负责做消元操作。从理论上来看，SIMD 优化算法所需要的时间应该是串行算法的 1/4，而 pthread 多线程所需要的时间应该是 SIMD 向量化的 1/7。然而，实验数据表明，当问题规模较小的时候，pthread 多线程算法的时间性能甚至差于普通的串行算法。这是由于线程的创建、挂起、唤醒和切换等操作所需要消耗的时钟周期数远远多于简单的运算操作。因此，当问题规模较小时，线程额外开销的副作用就会显现出来。但随着问题规模的增大，pthread 多线程的优势就能够显现出来。两种并行优化算法的加速比变化如下4.3所示。

随着问题规模的增加，我们也可以看到 SIMD 加速比基本保持稳定，但由于算法涉及其他数据处理，因此加速比只达到了约 1.5，未能达到理论上的 4。与之相比，pthread 优化的效果呈持续上升趋势。这是因为随着问题规模的增加，程序运行过程中运算所占比例逐步增加，逐渐抵消了线程切换导致的额外开销。当问题规模达到 2000 时，pthread 的加速比已经接近 SIMD 的理论加速比。因此，可以推断，随着问题规模的不断增加，pthread 的加速比将接近 7。



(a) 不同算法执行随问题规模变化



(b) 不同算法的加速比随问题规模变化情况

图 4.3: 数据趋势图

4.1.2 数据划分方式对比

在本次实验中，我们不仅进行了基础的 pthread 多线程优化实验，还从数据划分的角度出发，探究了不同的数据划分方式。由于 Gauss 消去过程的计算集中在除法和消去部分，这对应着矩阵右下角 (n-k+1)*(n-k) 的子矩阵，因此任务划分可以被视为对该子矩阵的划分。对于除法部分，由于只涉及到一行，我们只能采用垂直划分（即列划分）。而对于消去部分，我们可以采用水平划分，将外层循环拆分为多个线程，每个线程分配若干列，也可以采用垂直划分，将内层循环拆分为多个线程，每个线程分配若干行。这两种划分策略在负载均衡、同步和 cache 利用率等方面可能会有微小的差异。结合前面的实验设计，我们对比了循环划分、块划分在不同问题规模下的表现效果，并以 Gauss 普通算法为 baseline，其实验结果如下表：

n	普通高斯	块划分	循环划分	n	普通高斯	块划分	循环划分
20	0.020894	1.76289	1.85421	**300**	67.2037	33.2836	40.7777
30	0.0697199	2.59195	2.51451	**400**	148.735	44.3437	62.6695
40	0.16482	3.36754	3.51432	**500**	297.605	71.5093	105.736
50	0.3211	4.08164	4.11315	**600**	519.541	102.59	163.738
60	0.55384	5.19836	4.79746	**700**	823.959	135.777	234.52
70	0.87732	5.44882	5.59414	**800**	1236.3	194.21	366.68
80	1.30629	6.5026	7.77671	**900**	1754.39	250.388	462.091
90	1.85463	7.90478	7.53331	**1000**	2396.23	335.169	645.321
100	2.54074	8.76527	8.50363	**1500**	7984.06	987.114	1820.57
200	19.9605	18.2012	20.2691	**2000**	18787.8	2210.97	4053.13

表 3: 不同划分方式在不同数据规模下的耗时

从 cache 优化的角度可以看出，循环划分和块划分都是常用的优化算法，它们的主要区别在于如何最大化利用缓存（cache）优化，从而提高程序的性能。在循环划分中，线程处理完当前行后，需要处理下一行时，需要跨越一个间隔（NUM_PHTREAD），因此当数据规模很大时，可能会出现以下两种情况：第一，L1 缓存的大小无法容纳足够多的数据，导致数据频繁地被换出，从而产生额外的访存开销；第二，CPU 不能及时预取下一行数据，也会导致缓存未命中，从而产生额外的访存开销。这些开销将显著影响程序的性能。

相比之下，块划分的方式能够更好地利用缓存，其原因是对于每个线程而言，它所需要处理的数据之间在内存上是连续的，因此有很好的缓存优势。每个线程处理完一个块后，下一个块所需的数据已经被预取到缓存中，从而避免了额外的访存开销。因此，块划分能够有效地减小由于缓存未命中导

致的额外访存开销，提高程序的性能。

使用 perf 工具对这两种算法的 L1 缓存命中率进行检测，结果如表5发现块划分的命中率可达到 98%，而循环划分的方式仅有 93%左右。虽然两者之间的差异不大，但是在处理大规模数据时，这种差异将会被放大，进而显著影响程序的性能表现。

	循环划分	块划分
L1 cache Hit	93.2%	98.7%

表 4: 循环划分和块划分的 cache 命中率对比

因此，在选择算法时，应该考虑到数据规模的大小和缓存优化的效果，以选择最优的算法来提高程序的性能。同时，还应该注意到不同计算机的缓存结构可能不同，因此应该针对特定的硬件环境进行优化，以获得最佳的性能表现。

4.1.3 线程数量对比

在本次实验中，我们还探究了 pthread 多线程优化方法，在开启不同的线程数量的时候，优化效果的变化情况是不一样的。为了探究什么数据规模下可以显著体现出 pthread 的优化效果，我们选取数据规模为 1000，调整线程数量，观测加速比的变化情况如下图4.4。

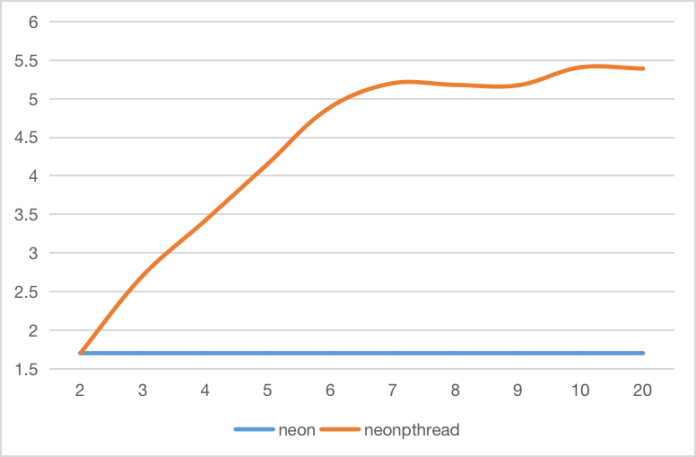


图 4.4: arm 平台不同线程数量效果对比

由图像中可以看出，随着线程数量的线性增加，pthread 多线程的优化效果也呈现出线性提升的趋势。而当线程数量超过 8 个之后，优化效果逐渐趋于稳定，不再有显著的变化。这是由于实验使用的服务器华为鲲鹏服务器单 CPU 核心能够提供 8 个线程，因此当线程数量小于 8 个的时候，CPU 核心能够使用自己的 8 个线程调度任务，而当所需要的线程数量超过 8 个之后，就需要服务器中的其他 CPU 核心借用线程，这会存在额外的调度开销，因此抵消掉了性能的提升效果。

4.2 x86 平台

4.2.1 Pthread 并行实现及多种 SIMD 指令集架构对比研究

在实验中，我们也在 x86 平台上进行了研究。在实验设计时，SIMD 进行向量化处理的时候，采用的是四路向量化处理，同时采用了 8 条线程的 pthread 多线程模型。其中一条线程负责除法操作，剩余的 7 条线程负责做消元操作。由于在 x86 平台上有更多的 SIMD 指令集架构，因此实验中分别探究了 SSE、AVX、AVX512 三种指令集配合 pthread 多线程的优化效果，同时也对不同问题规模下的运行时间进行测量，如下表5所示。

n	SSE	SSE-thread	AVX	AVX-pthread	AVX512	AVX512-pthread
20	0.015932	3.00524	0.01288	2.54034	0.01206	2.64855
40	0.06996	3.75631	0.08323	4.0169	0.0795	3.74919
60	0.25933	4.82458	0.1194	4.69117	0.19443	5.25172
80	0.56961	5.98599	0.38831	6.30742	0.52019	6.12442
100	1.38918	7.44256	0.81875	7.25671	0.54292	7.56552
200	8.44452	16.7328	4.53618	15.572	4.2204	16.1611
300	29.1553	33.1266	15.8265	28.2662	11.5475	28.9586
400	59.0385	56.8799	35.541	43.7855	25.0823	44.1242
500	122.565	92.3873	69.2648	70.6968	45.8336	68.433
600	206.065	135.332	126.007	103.407	92.3513	102.119
700	318.251	200.223	191.424	145.167	140.595	139.408
800	476.421	281.158	288.241	198.899	219.934	194.435
900	684.295	375.541	428.992	262.246	310.436	254.181
1000	1186.33	488.79	747.851	339.016	635.14	319.424
1500	3749.58	1392.8	2482.9	862.94	1978.14	794.004
2000	8202.23	3125.83	4641.85	1831.38	3652.14	1655.54

表 5: x86 平台不同算法不同数据规模的优化效果

由表5可以看出，pthread 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，没有出现在某种指令集架构下不能很好发挥多线程优势的现象。同时，我们根据表中数据，绘制出不同 SIMD 向量化处理与 pthread 多线程结合写的加速比，变化趋势如下图4.5。

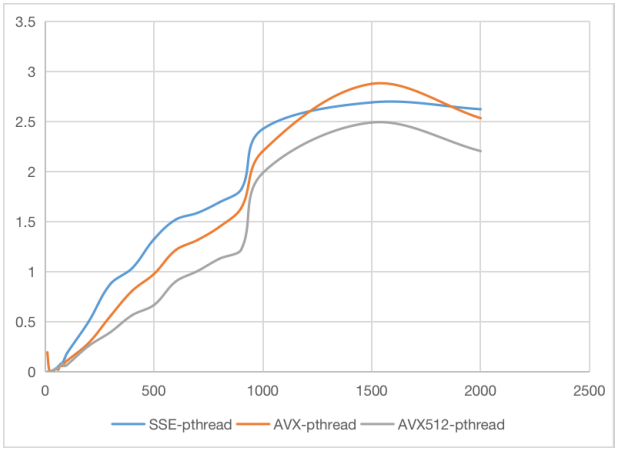


图 4.5: x86 平台加速比随问题规模的变化

可以看出当问题规模小于 1500 时，加速比随着问题规模的线性增长呈现出一个线性上升的趋势，而当问题规模超过 1000 的时候，我们发现加速比出现下降的趋势，在 x86 平台的 WSL 上使用 perf 进行性能分析，分析结果如下表6。

N	L1 cache Hit	L2 cache Hit	L3 cache Hit
500	>99%	>99%	>99%
1000	98%	98%	97%
2000	92%	93%	90%

表 6: 不同数据规模下各级 cache 命中率对比

由此我们可以析其原因，即当问题规模增加时，超过了线程 cache 的大小，导致出现了大量的 cache miss，这种额外的访存开销在一定程度上抵消了多线程的优化效果，使得加速比的变化出现拐点。

4.2.2 数据划分方式对比

本次实验也在 x86 平台上，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的效果的影响。结合前文的实验设计，分别对比了循环划分、块划分和动态划分三种划分方式在不同问题规模下的表现效果，测量其程序耗时并绘制加速比对比图。

n	NORMAL	循环划分	块划分	动态划分
10	0.00569	2.57893	2.40503	2.29929
20	0.051128	2.94802	2.6895	3.01194
30	0.10579	3.39335	3.4997	3.35492
40	0.26596	4.09753	4.16443	3.85485
50	0.4523	4.7454	4.71085	4.60424
60	0.84603	5.1484	5.31904	5.34159
70	1.95565	5.8503	6.07058	5.60124
80	2.02208	6.39443	7.21493	6.88899
90	2.54799	7.48112	7.64404	7.80924
100	3.68347	8.34839	8.18908	8.86324
200	28.0954	20.3433	17.2086	21.3732
300	97.7561	37.6455	34.727	41.191
400	213.302	63.3778	59.6118	65.6624
500	423.13	110.699	96.1449	107.552
600	720.375	169.181	162.068	168.777
700	1145.98	253.536	233.871	235.374
800	1752.25	344.073	328.316	326.326
900	2482.79	456.306	426.58	546.036
1000	3407.46	592.863	577.886	546.036
1500	11469.2	1697.52	1641.79	1521.25
2000	29307.7	5792.24	5741.95	4455.28

表 7: x86 平台不同数据划分方式耗时对比

对于循环划分和块划分进行分析对比，由图4.6可以看出，循环划分和块划分的主要区别在于块划分能够更好地利用缓存，而循环划分则在这方面存在缺陷。就循环划分而言，线程在处理完当前行之后，需要跨越一定的间隔才能处理下一行，这个间隔大小为线程总数 num_pthread。因此，在处理大规模数据时，可能会出现无法将足够的数据缓存在 cache 中，或者 CPU 无法预取下一行数据导致 cache 未命中的情况，从而增加了额外的访存开销。相比之下，块划分可以很好地解决这个问题。因为对于每个线程而言，它需要处理的数据在内存中是连续存放的，这样就可以充分利用缓存的优势，减少因为缓存未命中而导致的额外访存开销。

我们使用 perf 工具对于这两种算法的 L1 cache 的命中率进行检测，如下表8所示。可以看出循环划分的 cache misses 占比明显比块划分的 cache misses 占比要高，因此对于块划分来说，由于其考虑

到了 cache 的特性，随着问题规模的增大，其性能并没有明显受到访存开销的影响。而对于循环划分而言，由于其划分方式会造成大量的 cache misses，因此访存开销会极大地影响其性能表现。

	循环划分	块划分
L1-dcache-load-misses	15.56%	14.72%

表 8: 使用 perf 测量循环划分和块划分 cache misses 占总程序 cache miss 的百分比对比

从负载均衡的角度来看，不同的划分方式对负载均衡的影响也不同。静态循环划分和动态数据划分之间的主要区别在于能否充分利用各个线程的计算资源，以尽可能减少同步等待所导致的额外开销。静态循环划分将问题的计算任务按照循环次数进行分配，每个线程执行一定次数的循环，分配是在程序开始运行时完成的。这种方式的优点在于可以避免线程调度的额外开销，但是由于循环次数的分配是静态的，因此可能会导致负载不均衡的问题，一些线程的计算资源得不到充分利用，从而影响计算效率。根据图4.6所示，当问题规模较小时，静态循环划分的表现优于动态数据划分。这是因为动态划分这种方式可以避免负载不均衡的问题，但是需要进行线程调度，因此会增加一定的额外开销。当问题规模较小时，由于线程调度的额外开销占比较大，动态数据划分的表现不如静态循环划分。但当问题规模较大时，动态数据划分可以通过负载均衡的优化带来更高的计算效率，从而超过静态循环划分的表现。

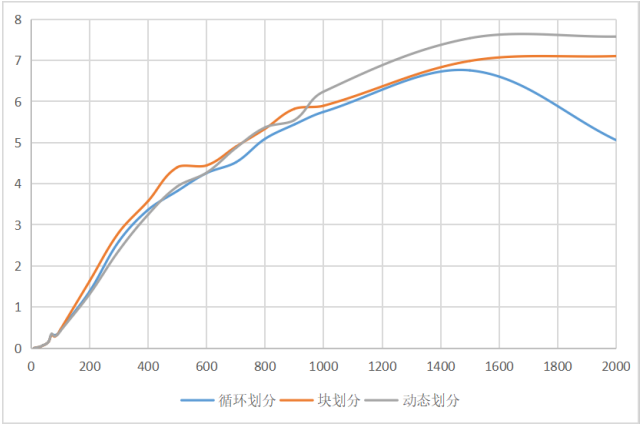


图 4.6: arm 平台不同线程数量效果对比

根据 VTune 性能分析工具的分析结果，我们对这三种任务划分方式的 CPU 占用率进行观察，可以得到如下图。从图中对比可知，动态数据划分的 CPU 占用率一直保持在一个较高水平，并且相对均衡，而对比其他的两种划分方式，由于没有考虑负载均衡，在 CPU 占用率这个指标上波动非常明显，这是对于计算资源的一种浪费。

4.2.3 线程数量对比

在本次实验中，我们还探究了 pthread 多线程优化方法，在开启不同的线程数量的时候，优化效果的变化情况是不一样的。为了探究什么数据规模下可以显著体现出 pthread 的优化效果，我们选取数据规模为 1000，调整线程数量，观测加速比的变化情况如下图4.7。

由图像中可以看出，随着线程数量的线性增加，pthread 多线程的优化效果也呈现出线性提升的趋势。但是在线程数超过 8 的时候图像有一个先上升再下降的趋势。对此进行推测是由于实验采用设备具有 8 个物理核心和 16 个逻辑核心，当线程数小于 8 的时候，线程之间的调度比较简单，所需开销较小；当线程数超过 8 个时，线程调度会变得复杂，但是由于实验采用电脑只有 16 个逻辑核心，所以

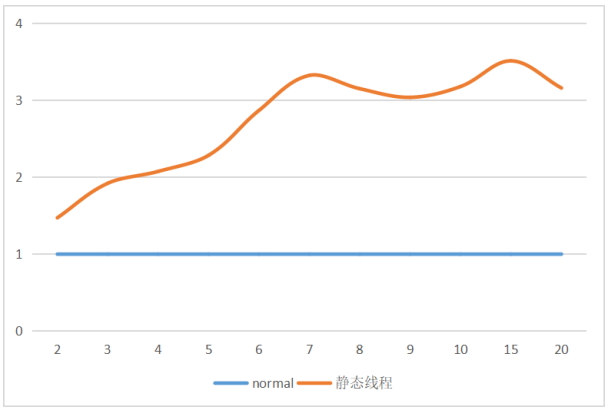


图 4.7: X86 平台不同线程数量效果对比

当线程数达到 16 时会达到一个峰值，随后开始下降。

5 OpenMP 实验

由于 Gauss 消去的整个过程中主要涉及到两个阶段，一个是在消元行行内除法过程，另一个是其余行减去消元行的过程，对于每一个阶段而言，所做的工作基本是一致的，只是在不同的消元轮次中消元的起始位置不同，特别是在第二个阶段中，即逐行减去消元行的过程，每行所执行的操作是完全相同的。这个阶段非常适合并行处理，可以将待消元的行均匀地分配给多个线程处理。由于这些数据之间不存在依赖关系，因此每个线程只需要处理自己分配到的行，无需进行额外的线程间通信，从而避免了额外的开销。

在第一阶段中，需要对消元行内的数据进行除法操作。由于此过程的规模较小，若将其分配给多个线程进行处理，线程间切换的时间会占用较大开销，因此多线程并行处理不适宜。然而，我们可以利用 SIMD 技术实现向量化处理。具体来说，将数据按照一定的块大小进行分组，然后对每个块内的数据进行一次除法操作，可以大大减小除法操作的次数。同时，SIMD 可以在同一时间内对多个数据进行计算，加快运算速度。

在第二阶段中，需要将其余行依次减去消元行，此过程每一行所做的工作是完全一致的，适合进行多线程并行处理。我们可以将待消元的行分配给多个线程处理，每个线程独立地完成自己所负责的行的操作。由于数据之间不存在依赖性，不需要线程间进行通信，可以减少额外的开销。不过，在每行内部执行减法运算并不适合进行多线程并行处理，但同样可以采用 SIMD 技术实现向量化处理。我们可以将每行数据按照一定的块大小进行分组，然后对每个块内的数据进行一次减法运算，可以大大加快运算速度。

在本次实验中，将会设计以下实验进行探究：

- 使用 Openmp 对 Gauss 消去算法进行多线程优化；
- 设计不同的任务划分方式，调整 schedule 的参数，对比在不同任务规模下的性能表现；
- 对比按行划分和按列划分之间的性能差异
- 对比动态线程创建和静态线程创建之间的性能差异
- 使用 Openmp 进行 SIMD 向量化处理并与之前手动的 SIMD 性能进行对比；
- 使用 Openmp 进行多线程处理并与之前手动 pthread 的性能进行对比；
- 对比不同线程数量下的 Openmp 性能表现情况；
- 在 devcloud 上尝试任务卸载，将 Gauss 消元的消元过程卸载到 GPU 上运算并测试其性能表现；

5.1 OpenMP 并行处理

在 Gauss 消去算法中，除法和消元两个阶段的执行顺序严格固定，必须先完成除法操作才能执行消元操作，这意味着需要进行同步。在使用 openmp 进行多线程实验时，可以先使用单个线程处理除法操作，然后将后续的消元过程分配给多个线程，使用线程私有变量进行循环迭代，将待消元矩阵声明为线程间共享。为了优化性能，可以使用 openmp 的 simd 预编译选项进行自动向量化。在实验中，还需要将 openmp 多线程程序和之前设计的 pthread 多线程程序进行性能对比，并将单线程条件下的 simd 优化算法的性能与两种多线程算法进行对比。

使用 openmp 和 SIMD 算法对 Gauss 消元进行优化可以获得较好的性能提升。理论上，采用 openmp 的优化算法的加速比应该与线程数量成正比，最优效果下能够达到 NUM_THREADS 倍的

性能提升。同时，结合 SIMD 算法之后，加速比将进一步提升，具体的加速比取决于向量化处理的宽度。例如，当采用四路向量化优化时，整体的最优加速比应该能够达到 4 * NUM_THREADS 倍。对比之前的手动设计的 SIMD 向量化算法，使用 openmp 和 simd 预编译选项能够更加方便地进行实验探究，并能够更好地利用多核计算机的并行计算能力。此外，还可以将使用 openmp 的多线程程序和使用 pthread 的多线程程序进行性能对比，对比不同算法设计的性能差异。

5.2 数据划分设计

OpenMP 提供了多种任务划分方式，其中 static、dynamic 和 guided 是常用的三种方式。对于不同的负载特性，我们可以选择适合的任务划分方式。

静态数据划分是最简单的方式，即在分配任务时就已经确定每个线程负责的任务范围。在 Gauss 消元的过程中，由于每个阶段都重新进行任务划分，因此负载不均的问题并不明显，直接采用静态数据划分也可获得不错的效果。

从负载均衡的角度出发，可以采用动态数据划分。在每一轮消元过程中，最终决定本轮消元时间的是运行时间最长的线程。可能存在个别线程提前完成任务而进入空闲等待，导致计算资源浪费。因此可以采用动态任务划分方式，但可能会产生较大的线程调度开销。

对于高斯消元这类任务规模逐渐减小的计算密集型任务，如果将任务范围固定分配给每个线程，就会出现个别线程在某些轮消元中没有被分配到任务，造成计算资源浪费。因此我们可以使用 guided 参数，随着任务推进，指数缩减到指定的步幅，这样可以尽可能地利用全部线程的计算资源。

5.3 行列的划分角度

为了优化高斯消元算法的性能，可以考虑从行列两种划分角度进行实验设计。通常情况下，采用按行划分的方式，因为在高斯消元的过程中，每一行所需要进行的操作都是相同的，且行内的数据是连续的，具有较好的空间局部性。但是，由于外层循环的迭代，每一轮各个线程所处理的行号可能不同，导致各个线程的 L1 和 L2 cache 中可能不存在所需数据，从而引起伪共享问题，这会增加线程之间访问 cache 的开销，因此按行划分的方式会受到时间局部性的制约。

另一方面，考虑按列划分的方式。在高斯消元的过程中，每一列所进行的操作也是完全相同的，即将该列中的值依次减到该列中的其余位置处，各列之间不存在数据依赖。因此也可以考虑按列划分的任务方式。但是，由于计算过程中每个线程中的数据访问跨行，因此可能会有较高的缓存失效率，这会受到空间局部性的制约。此外，由于高斯消元的过程是从左到右、从上到下逐渐收缩的，因此按列划分的方式也会受到时间局部性的制约。尽管如此，按列划分仍是一种值得尝试的优化方式。

5.4 不同数据规模和线程数下的性能探究

考虑到多线程程序相较于串行程序，线程的创建、调度、挂起和唤醒等操作所需的时间开销非常大，这些开销可能抵消了多线程带来的优化效果。因此，当问题规模较小时，多线程程序可能会表现出比串行算法更慢的情况。而随着问题规模的增加，线程之间调度切换所需的时间相对于线程完成任务所需时间而言占比会越来越低，从而能够更好地反映出多线程并行优化的效果。

为了探究多线程并行优化算法的优化效果，需要在不同数据规模下设计实验。在此过程中，需要考虑如何平衡线程数量和问题规模之间的关系。具体而言，对于较小的数据规模，需要谨慎选择线程数量，以避免线程创建和调度等开销过大导致程序效率下降。而对于较大的数据规模，应该适当增加线程数量，以充分发挥多线程并行计算的优势。

除了数据规模，还需要考虑所使用的线程数量对并行算法优化效果的影响。具体而言，可以通过实验探究不同线程数量下的优化效果，并寻找最优线程数量，以达到最好的算法优化效果。

总之，为了全面探究多线程并行优化算法的优化效果，需要从不同角度进行实验设计，包括数据规模和线程数量等方面，并综合考虑线程创建、调度和任务执行等各种因素。

6 OpenMP 实验结果分析

6.1 ARM 平台

6.1.1 OpenMP 并行实现及性能对比

为了探究 openmp 并行算法的优化效果，我们对问题规模进行了调整，并测量了不同任务规模下五种算法的时间性能表现。这五种算法分别为：串行算法、手动 SIMD 算法、手动 pthread 算法、openmp 版本的 SIMD 算法和 openmp 版本的多线程算法。其中，pthread 多线程算法和 openmp 多线程算法都采用了与 SIMD 算法的结合，并启用了 8 条线程，并采用了四路向量化处理。为了展现并行优化效果随问题规模的变化情况，我们在问题规模小于 1000 时采用步长为 100，而当问题规模大于 1000 时，步长调整为 500。下表9展示了五种算法在不同问题规模下的表现。

n	serial	SIMD	openmp_SIMD	pthread	openmp
100	0.34606	0.26414	0.1788	8.4406	1.29174
200	2.73712	2.10322	1.0636	18.503	3.01852
300	9.40638	7.02764	3.45278	31.3003	6.17042
400	23.4602	16.8215	8.53298	40.7963	10.7536
500	50.8499	35.6443	20.8996	58.1806	16.603
600	89.2829	61.4309	37.3442	78.8596	23.3091
700	141.157	92.7715	49.3492	104.583	34.3046
800	214.473	139.746	74.5927	140.419	48.7123
900	303.316	198.883	104.781	180.883	65.9972
1000	411.905	272.765	142.044	229.296	86.328
1500	1374.21	917.591	492.431	616.767	252.946
2000	3538.42	2304.48	1383.33	1371.52	543.87
2500	10096.8	4686.29	2877.92	2546.15	1025.44

表 9: 不同算法在 arm 平台上随问题规模的变化情况

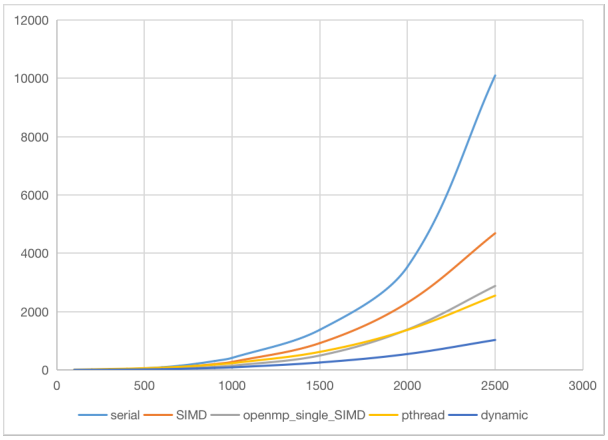


图 6.8: ARM 平台不同算法耗时

为了能够更加直观的观察算法的性能表现随问题规模的变化情况，利用测量的数据绘制出图6.8，同时计算了四种并行优化算法的加速比随时间的变化情况，如6.9所示。

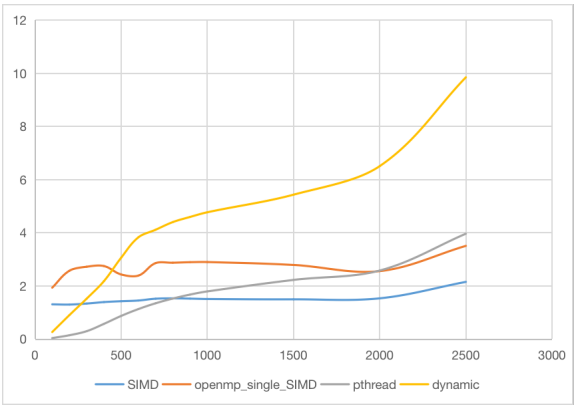


图 6.9: ARM 平台不同算法加速比效果对比

由图6.9可以看出，随着问题规模的增加，几种并行优化算法的加速比都呈现一种递增的趋势。在计算的过程中除了乘法操作和消元操作，还有很多其他不能进行向量化处理的运算，所以使用 SIMD 进行向量化优化的两种方法没有达到理论加速比 4 倍，保持在 1.5 左右的加速比。

而对于两种多线程的并行优化算法，其加速比随着问题规模的增加呈现出快速增长的趋势。因此我们可以发现，随着问题规模的增加两种并行优化算法都接近了理论加速比 8 倍，这说明实验是很适合进行多线程优化的，不同线程在分配任务的时候不存在严重的数据依赖问题，线程之间额外的通信开销很少。

同时我们比较了手动实现的 SIMD 算法和使用 OpenMP 库实现的 SIMD 算法的性能表现。通过对比两种算法的性能指标，我们发现手动实现的 SIMD 算法在性能上略优于 OpenMP 版本。通过 perf 分析，我们可以看到，OpenMP 版本所需的指令数 instructions 明显高于手动 SIMD 实现方法，同时在时钟周期 cycles 指标上也表现更差，因此性能表现更差。这一结果表明，在 SIMD 算法中，手动实现能够更好地利用硬件的优势，提高算法的效率和性能。

此外，实验还比较了手动实现的 pthread 多线程算法和 openmp 多线程算法的性能差异。从图像可以看出，手动实现的 pthread 多线程算法性能一直落后于 openmp 多线程算法。这是因为在手动实现 pthread 多线程算法时，Gauss 消元的各个轮次之间存在严格的先后关系，并且每个轮次内部的除法操作和消元操作都存在严格的依赖关系。因此，在手动实现多线程时，需要进行同步和线程间通信的开销会更多，而 openmp 则可以更加高效地管理多个线程之间的同步，因此其性能更优。

6.1.2 数据划分对比

本次实验主要探究不同任务划分方式和任务划分粒度对算法性能的影响，重点从负载均衡的角度进行分析。在 openmp 中，提供了三种不同的数据划分方式：static、dynamic 和 guided。其中，static 方法在线程创建完成后即明确划分了任务，dynamic 方法则在线程执行过程中动态划分任务，而 guided 方法则是随着任务推进逐步缩减任务划分粒度。在本次实验中，我们对比了这三种方法在不同任务规模下的性能表现，并绘制了加速比随问题规模变化的图像如图6.10。

结果表明，对于小规模任务，static 方法效果最佳，而在大规模任务中，guided 方法表现更好。此外，我们还分析了任务划分粒度的影响，发现对于不同任务规模和不同划分方式，最优划分粒度均有所不同。这些结果为进一步优化算法性能提供了重要参考。

从图像中可以看出，随着任务规模的增加，这三种划分方式的加速比都逐渐增加。其中，static 方

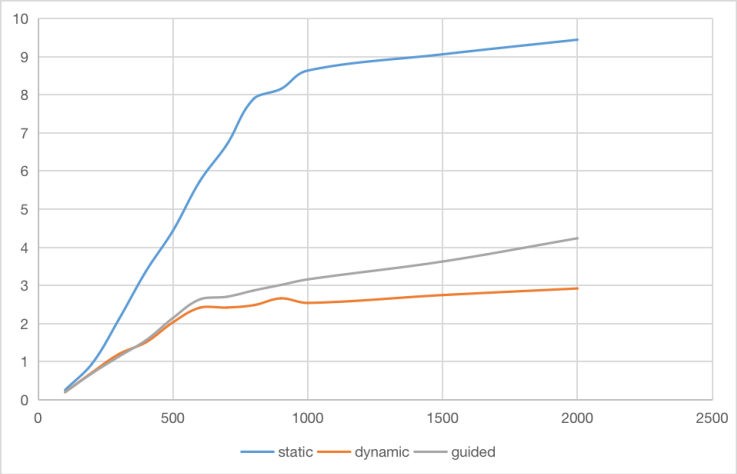


图 6.10: ARM 平台不同数据划分加速比效果对比

法在问题规模较小的情况下表现最好，但在达到一定规模后性能表现变化较小。这是因为 static 方法将任务静态地划分给线程，在消元的每一轮中，任务基本是均匀的，因此不存在严重的负载不均的问题。然而，随着问题规模的增大，矩阵的每个部分的计算量变得不均匀，导致一些线程负载过重，影响了整体性能。相反，guided 方法会逐步缩减任务划分的粒度，尽可能让所有线程都被分配任务，从而平衡负载不均，因此在问题规模较大时表现较佳。

另外，dynamic 方法在任务执行过程中动态地划分任务，但也存在个别线程不会被分配任务的情况，导致浪费计算资源。因此，相对于 dynamic 方法，guided 方法能够更好地平衡负载，从图像中也可以看出，在不同规模的问题下，采用 guided 方法的性能表现要优于 dynamic 方法。总之，在不同的任务规模和计算负载情况下，选择合适的数据划分方式和划分粒度，可以更好地平衡计算资源，提高算法性能。

6.1.3 行列划分对比

在本次实验中，我们不仅从负载均衡的角度出发，研究了不同粒度的任务划分方式的性能影响，还从缓存的角度出发，考虑了空间局部性和时间局部性，探究了行列划分两种方式的性能表现。我们比较了行划分和列划分两种方式在不同矩阵规模下的性能表现如表所示，并绘制了加速比随问题规模的变化曲线。

n	行划分	列划分	n	行划分	列划分
100	1.34732	2.68678	**700**	39.1122	289.8
200	2.81478	12.5281	**800**	56.361	431.076
300	6.12708	33.5659	**900**	78.6658	547.042
400	10.1849	80.4111	**1000**	107.051	692.154
500	17.0638	116.917	**1500**	337.142	1933.27
600	26.72312	208.847	**2000**	809.596	4091.74

表 10: 行划分与列划分在不同数据规模下耗时

从图6.11中可以看出，由于按列计算的方式，对于矩阵的访问来说是跨行访问的，因此随着任务规模的增加，由于空间局部性的限制，会产生比较严重的 cache miss，，因此在访存的时候需要到内存中或者其他线程的 cache 中去访问，这会导致很严重的访存开销。

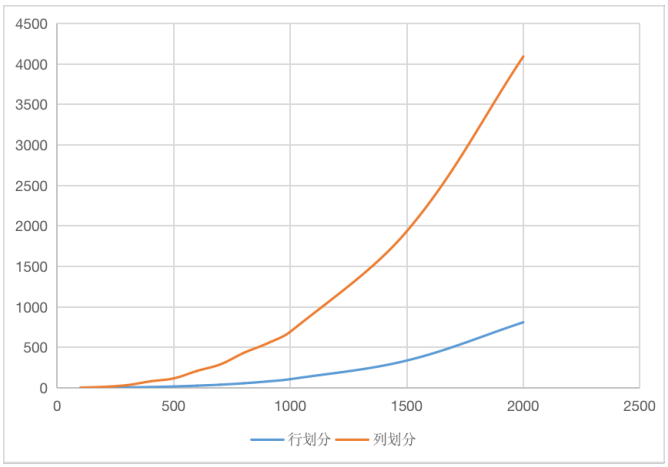


图 6.11: 行划分与列划分不同数据规模性能对比图

随着问题规模的增大，行划分方式的性能相对于列划分方式有所提升。这是因为行划分方式可以更好地利用时间局部性，即将矩阵的行存储在连续的内存块中，利用 CPU 的缓存机制，减少了数据访问的开销，提高了算法的性能。而列划分方式则可能会导致多次跳跃式访问内存，增加了算法的延迟。

此外，在较小规模的矩阵上，行列划分的性能表现差异不明显，但随着矩阵规模的增大，行划分方式的性能提升更加明显，而列划分方式的性能提升相对较小。这也说明了空间局部性的重要性，即将相邻的数据存储在一起，可以利用 CPU 的缓存机制，提高算法的性能。

当考虑到时间局部性时，行列划分两种方式都会受到限制。这是因为随着任务的推进，线程分配的任务行号或列号会逐渐减少，导致无法利用缓存中已经计算的数据。因此，可以通过进一步改进算法来优化时间局部性。例如，可以使用数据重用技术，将已经计算过的数据保存在缓存中，以便后续使用。此外，可以尝试对算法进行重组，以尽可能地利用缓存中的数据，从而提高算法的时间局部性。此外，还可以尝试对数据进行预取，提前将需要计算的数据加载到缓存中，以便后续使用。通过这些方法的应用，可以有效提高算法的时间局部性，从而提高算法的性能。

6.1.4 线程数量对比

本次实验探究了在不同线程数量下，openmp 多线程算法的性能表现，其测量结果如下表11, 绘制加速比对比图如图6.12。通过观察性能图像可以发现，随着线程数量的增加，三种任务划分方式的加速比呈现出一个线性增加的趋势，这表明 openmp 多线程算法在处理本问题时表现出良好的可扩展性，如图6.12所示。这种可扩展性使得我们能够利用更多的硬件资源，从而更快速地解决更大规模的问题。在本实验中，我们还探究了不同粒度的任务划分方式和行列划分方式对算法性能的影响，结果显示在不同情况下，不同的划分方式可能会产生不同的性能表现。

num_threads	static	dynamic	guided
2	66.0012	160.406	107.311
3	53.3839	134.197	141.55
4	39.3537	103.693	95.431
5	32.3386	124.749	106.525
6	28.4561	121.851	92.8757
7	23.7392	104.507	86.1887
8	23.0728	101.562	82.4122

表 11: 不同线程数下多种划分方式耗时测量

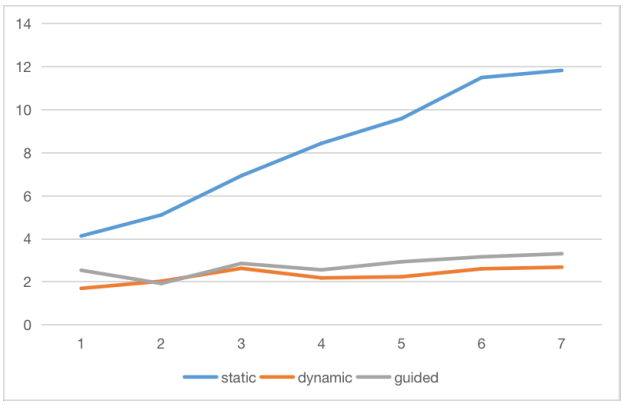


图 6.12: 不同线程数量加速比对比

6.1.5 线程管理方式对比

本次实验中，我们也对动态线程创建和静态线程创建这两种线程管理方式的性能对比。实验结果如下表所示。

n	静态线程	动态线程	n	静态线程	动态线程
100	1.34758	1.68458	700	38.7704	38.9526
200	2.92508	3.57122	800	56.3226	59.6495
300	6.00804	7.45082	900	78.8047	81.2005
400	10.1908	11.7409	1000	106.983	105.279
500	16.7999	18.0599	1500	333.838	352.95
600	25.9407	27.0638	2000	824.963	771.153

表 12: 动态 or 静态线程管理的测量耗时

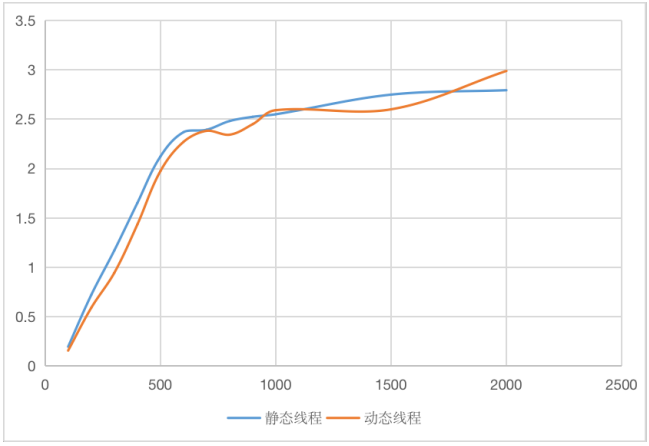


图 6.13: 动态 or 静态线程管理加速比对比

静态线程创建是指在编译时确定线程的数量，并在程序启动时创建这些线程。在静态线程创建中，程序员需要指定线程的数量，并使用编程语言中的线程库来创建这些线程。这种方法的优点是，线程的数量是固定的，这使得编写代码更加简单和容易。缺点是无法根据需要动态地创建或销毁线程，这可能会导致资源的浪费或效率降低。动态线程创建是指在运行时根据需要创建线程。这种方法允许程序动态地创建和销毁线程，从而更好地利用计算机资源。在动态线程创建中，程序员可以根据需要创

建线程，并使用编程语言中的线程库来实现这一点。优点是可以根据需要动态地创建和销毁线程，从而更好地利用计算机资源。缺点是程序员需要更多地关注线程的数量和状态，这可能会增加代码的复杂性和难度。

因为静态线程创建适用于需要固定数量的线程的情况。如果在本次实验中需要处理的任务是近似相同的，而且需要固定数量的线程来完成这些任务，那么静态线程创建是更合适的选择。相比之下，动态线程创建更适用于需要根据需要动态地创建和销毁线程的情况，例如处理不同类型的任务或处理可变数量的任务。因此在本次实验中，更适合采用静态线程创建的方式，来减小线程创建、初始化以及任务划分的额外开销，这种开销相对于简单的计算而言是比较大的。这也可以从实验数据中得到进行印证。

我们使用 perf 对于两种方式的线程管理时间对比，当 n=200 的时候，静态线程创建的时间开销为 3.21%，而动态线程创建的时间开销为 5.16%，动态线程创建开销稍大，但是在数据规模较大的时候，这种差距将变得更大，如下表13所示。

N	**静态**	**动态**
200	3.20%	5.16%
1000	<0.1%	3.67%

表 13: 动静态线程创建开销对比

6.2 x86 平台

6.2.1 OpenMP 在 x86 平台上的并行实现

基于前文在 ARM 平台上对于 openmp 多线程编程的探究，在本次实验中还将 openmp 多线程优化方法迁移到 x86 平台上，做同样的实验探究。结合 SSE 指令集架构配合 openmp 多线程测量优化效果。其结果如下表14。可以看出 openmp 多线程可以结合各种 simd 指令集架构，在多个平台上表现稳定，并没有出现在某个平台或是某个指令集架构下不能发挥很好的多线程优势效果。

n	serial	SIMD	openmp_single_SIMD	pthread	schedule_static
100	0.2009	0.1021	0.0353	2.0052	5.8125
200	1.3	0.701	0.4996	3.6926	11.799
300	4.2963	2.4858	1.513	7.2654	17.9809
400	9.1126	5.9044	3.1769	9.3236	23.4325
500	18.9042	9.332	5.5826	15.3693	27.8281
600	27.8963	17.1333	10.0592	17.6379	30.343
700	50.643	24.41	18.0049	22.0466	36.0395
800	77.0331	41.0953	24.7466	28.6115	43.1449
900	106.882	57.7034	35.7607	37.6328	48.8193
1000	153.785	95.1766	56.0649	56.0454	70.9115
1500	993.014	578.628	212.782	150.906	148.826
2000	1521.16	1026.39	693.545	354.257	367.453
2500	2722.09	1962.08	1593.93	1150.3	1076.11
3000	5742.5	3830.69	3189.87	2523	2485.94

表 14: x86 平台不同算法执行耗时

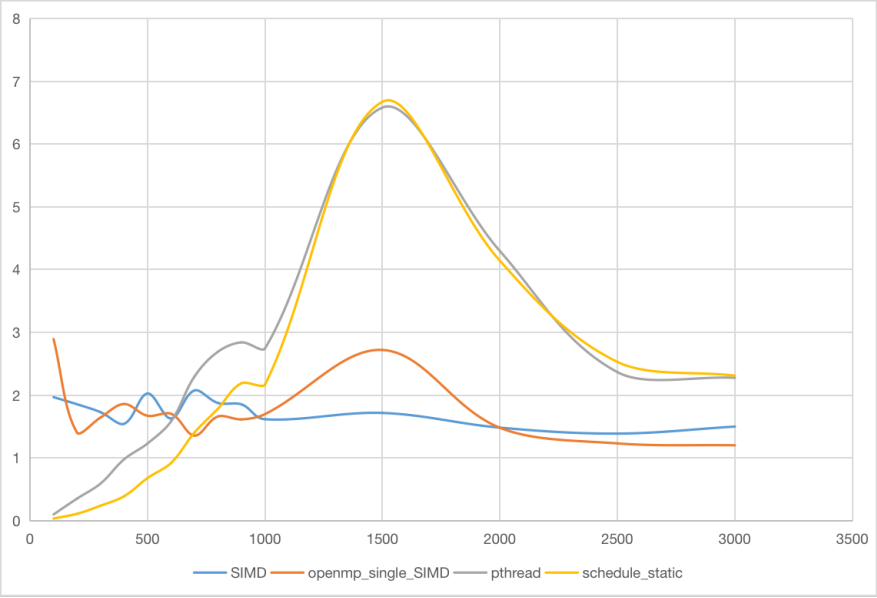


图 6.14: x86 平台上不同算法加速比

6.2.2 数据划分对比

openmp 为任务划分提供了三种选项：static、dynamic 和 guided。static 方法是在线程创建完成之后，就明确划分了任务，dynamic 方法则是在线程执行的过程中去动态的划分任务，而 guided 方法

则是随着任务的推进逐步缩减任务划分的粒度。在 x86 平台的实验中，我们测量结果如下图：

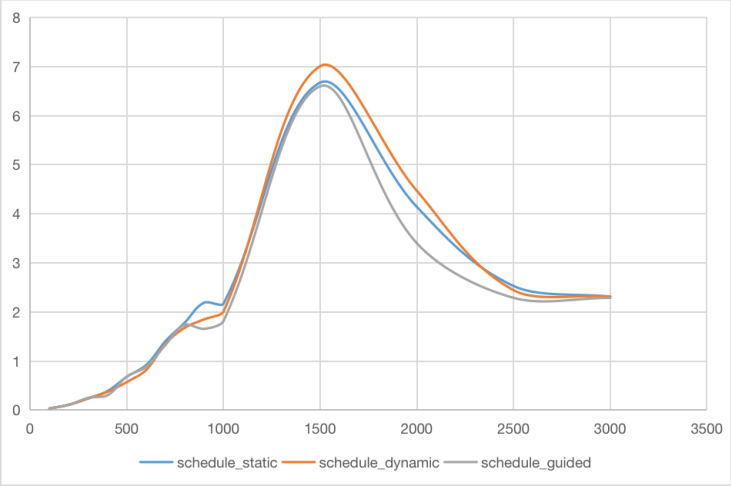


图 6.15: x86 平台上不同数据划分的加速比

可以看出在数据规模在 1500 内的时候，三种划分方式的加速比都随着数据规模增大而逐渐增长，但是在数据规模在 1500 之后，几种划分方式的加速比逐渐下降。这是因为当数据规模较小时，三种划分方式的任务负载是比较均衡的，缓存中的数据访问模式不会受到太大影响，因此三种划分方式的性能表现比较相似。随着数据规模的增大，任务的数量也增加了，可以更好地发挥多线程的并行性。但是当数据规模超过 1500 时，由于任务的数量增加到一定程度，线程之间的同步和划分开销增加，超过了并行化带来的收益。

同时，在三种划分方式中，guided 的耗时最少，dynamic 其次，static 最多，这是因为 guided 在任务划分时可以根据任务执行的进度自适应调整划分粒度，充分利用了多线程并行的优势。而 static 的任务划分方式在任务量不均衡时可能导致某些线程的负载过重，dynamic 的划分方式需要频繁地从任务池中获取任务，增加了同步开销。与此同时，static 划分方式在执行时会将任务静态地分配给不同的线程，因此每个线程访问的数据可能不连续，从而导致缓存命中率降低，性能下降。而 dynamic 划分方式是动态地将任务分配给线程，因此每个线程访问的数据会更加连续，从而可以提高缓存命中率，性能得到一定提升。guided 划分方式则是动态地调整任务粒度，这样可以避免过多的线程竞争导致的负载不均衡，从而可以提高缓存命中率，性能得到进一步提升。

7 实验总结与反思

本次实验探究了针对 Gauss 消元问题的多线程并行优化，使用 pthread 和 openmp 进行优化，并进行优化效果的对比以及融合实验。在消元过程中，通过多线程进行并行化操作，并利用 SIMD 向量化处理来进一步提高计算效率。在 ARM 平台上采用 pthread+neon 和 openmp+neon 的方式，在 x86 平台上采用 pthread+SSE/AVX/AVX512 和 openmp+SSE/AVX/AVX512 的方式，进行了多线程并行优化效果的对比实验。

在 pthread 实验中，我们通过考虑 cache 特性对比了循环划分和块划分的性能差异，通过考虑负载均衡对比了循环划分和动态划分的性能差异，验证了考虑了 cache 特性的块划分方式和考虑了负载均衡的动态划分方式都能够取得一定的性能提升。此外，我们还探究了开启线程的数量同优化性能之间的关系，并结合实验平台的硬件参数进行了合理假设和分析。在实验的过程中，利用 perf 和 VTune 等性能分析工具，对深层次的内核和硬件事件进行分析，从底层的角度解释了表面上性能差异的原因。

在 openmp 实验中，我们通过考虑 cache 的空间局部性和时间局部性对比了行划分和列划分的性能差异，通过考虑负载均衡对比了 openmp 提供的三种 schedule 方式的性能差异。实验结果表明，考虑 cache 特性的行划分方式和考虑负载均衡的动态划分方式能够取得一定的性能提升。此外，本次实验还探究了线程数量和优化性能之间的关系，并结合实验平台的硬件参数进行了分析。为了深入理解性能差异，本次实验还利用 perf 和 VTune 等性能分析工具对深层次的内核和硬件事件进行了分析，解释了性能差异的原因。

在这次实验中，我也对课上所学到的东西有了进一步的认识，将 PPT 上的理论知识在自己花时间实践一遍的时候，对于这些线程管理，优化方式等都有了更多的认识。本次实验的相关代码和文档已经上传刘大圆的 Github。