南開大學
Nankai University

计算机学院

并行程序设计实验报告

Gauss 消去 pthread 并行优化研究

姓名：刘宇轩

学号：2012677

专业：计算机科学与技术

2022 年 5 月 1 日

目录

1 问题描述

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。

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

考虑在整个消去的过程中，如图1.1所示，主要包含两个过程:

1. 对于当前的消元行，应该全部除以行首元素，使得改行转化为首元素为 1 的一行
2. 对于之后的每一行，用改行减去当前的消元行，得到本次的消元结果

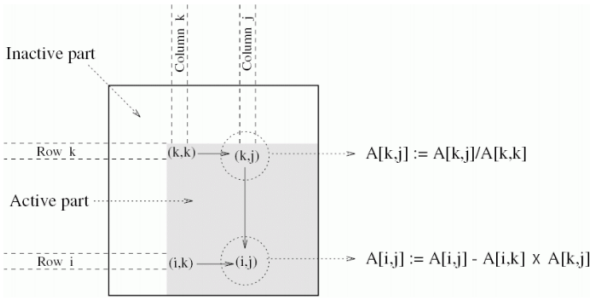

图 1.1: Gauss 消去算法逻辑图

而针对这两个过程，都适合采用并行的方式进行性能的优化。本次实验，采用 pthread 多线程编程，结合 SIMD 并行指令架构，针对上述两个过程进行并行优化，并考虑采用不同的任务划分方式，对比不同的任务划分方式的性能差异，此外还将考虑并行优化加速比随问题规模和线程数量的变化情况，本次实验还将同时设计 ARM 架构和 x86 架构两个平台的实验，对比在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 i7-11800H
CPU 主频	2.6GHz	2.3GHz
L1 Cash	64KB	48K
L2 Cash	512KB	1.25MB
L3 Cash	48MB	24MB
指令集	Neon	SSE、AVX、AVX512F
核心数	1	8
线程数	8	16

3 实验设计

考虑 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，一个是其余行减去消元行的过程。而就每个阶段而言，其所做的工作基本是一致的，只是在不同的消元轮次时，消元的起始位置不同。尤其是针对第二个阶段，即其余行依次减去消元行的过程，这个阶段每一行所做的工作是完全一致的，十分适合并行化处理，即将待消去的行平均分配给几个线程，由于这些数据之间不存在依赖性，因此每个线程只需要各自完成好自己的工作即可，不存在线程之间进行通信的额外开销。

而对于第一阶段，即消元行行内进行除法操作时，由于这个问题规模相对较小，如果将待操作的数据分配给不同的线程进行处理的话，线程挂起到唤醒这部分的时间开销相较于要处理的问题而言占比很高，因此不适合进行多线程并行处理，但是仍可以结合 SIMD 的向量化处理。同样在第二阶段，被消元行依次减去消元行的过程中，每一行内的减法运算同样也不适合进行多线程的并行处理，也可以采用 SIMD 进行向量化处理。

在本次实验中，将设计以下实验进行探究:

1. 采用信号量和 barrier 设计同步机制，基于 pthread 进行并行优化
2. 设计将数据按块划分的算法，从 cache 缓存的角度除法，对比与样例划分方式的性能差异
3. 设计动态的数据划分方式，从负载均衡的角度出发，对比不同任务划分方式的性能差异
4. 对比不同数据规模下和线程数下，并行算法的优化效果
5. 将 pthread 并行思想迁移到 x86 平台上，结合不同的 SIMD 指令级，测试其并行优化效果

以下是详细的实验设计方案

3.1 pthread 并行处理

对于 Gauss 消去的过程，在每一轮消去中主要包含两个阶段，首先是针对消元行做除法运算，然后是对于剩余的被消元行，依次减去消元行的某个倍数。在每一轮的过程中，除法操作和消元减法操作之间是有着严格的先后顺序的，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，即当 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。

在执行消去的时候，考虑对于数据采用分散的划分方式，即以线程数量为步长对于剩余的被消元行进行划分，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。

而由于不同的线程在执行消去操作时所需要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。在这一次同步控制中，出于方便考虑，使用了 barrier 进行同步控制。即只有当所有的线程都完成了消去任务之后，才会进入下一轮的消元。

3.2 数据块划分设计

在进行任务划分时，给出的样例中采用了等步长的划分方式，这种划分方式存在一定的弊端。即当数据规模比较大的时候，由于 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，这样就需要到 L2、L3 甚至内存中去读取数据。这将会造成额外的访存开销。

因此在进行数据划分的时候，考虑设计一种充分利用 cache 优化的数据划分方式，即将数据按块划分。每个线程负责连续的几行的消去任务。这样做的好处是，当线程正在处理当前行的时候，CPU 可能会提前预取下一行的数据到 cache 中，这就会使得下一次进行数据访问的时候，能够尽快在 cache 中命中，减少了不必要的访存开销。

3.3 数据动态划分设计

考虑在进行任务划分的时候，由于不同线程在执行任务的时候，所需要的时间可能不一致，甚至因为数据规模不是线程数量的整数倍，导致某些线程出现在个别轮次中处于空等待的状态。这是由于数据划分的时候，由于细粒度的数据划分导致的线程之间负载不均衡。

因此考虑在设计数据划分的时候采用动态的数据划分方式。即在对被消元行执行减法操作的过程中，并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的情况动态的进行数据划分。即通过一个全局的变量 index 来指示现阶段已经处理到哪一行。而当某一个线程完成了其被分配的任务的时候，会查看关于 index 的互斥量，如果这个互斥量并没有上锁，则说明当前处于可以进行任务划分的阶段。于是让这个线程对关于 index 的互斥量上锁，并将 index 所指的行分配给该线程，任务分配完成后，线程释放互斥量，然后去执行所分配的任务。

这样就可以保证每条线程都一直在执行被分配的任务，而不会出现个别线程由于负载不均衡出现空等待的现象，而其他线程还在执行任务。由于只有当所有线程的任务都执行完毕的时候才会进入下一轮迭代，因此那些进行空等待的线程就浪费了 CPU 的计算资源。这就是该实验设计选择进行优化的方向。

3.4 不同数据规模和线程数下的性能探究

考虑到线程的创建，调度，挂起和唤醒等操作相对于简单的计算操作而言，所需要的时间开销是非常大的。因此可以推测，当问题规模比较小的时候，由于线程调度导致的额外开销会抵消掉多线程优化效果，甚至还会表现出多线程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要的时间开销相对于线程完成任务所需要的时间而言已经占比很低，这样就能够正常反映出多线程并行优化的效果。因此，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。此外还将探究在所使用的线程数量不同的情况下，并行算法优化效果的变化情况。

3.5 x86 平台迁移

本次实验除了对 ARM 架构下采用 neon 指令集架构结合 pthread 多线程编程，对 Gauss 消去算法进行并行化处理，还将算法迁移到了 x86 平台上，采用 x86 中的 SSE、AVX 和 AVX512 指令集架构分别对算法进行重构，然后对比实验效果。

考察了本机所支持的指令集架构，情况如表1所示

指令集架构	版本
SSE	SSE/SSE2/SSE3/SSE4.1/SSE4.2
AVX	AVX/AVX2
AVX512	AVX512F

表 1: 本机支持指令集架构及版本

虽然本机只支持 AVX512F，但是 AVX512F 是 AVX512 的基本子集，能够支持基本的向量化计算，因此在本次实验中能够支持 AVX512 指令集架构的实验。

4 实验结果分析

4.1 ARM 平台

4.1.1 pthread 并行处理

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，pthread 并行算法中，同样融合了 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 1000 时采用步长为 100，而当问题规模大于 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4.2所示。

N	serial	neon	pthread
100	2.43	1.62	2.85
200	19.27	12.73	6.55
300	68.15	42.60	13.16
400	154.04	101.13	27.52
500	305.45	200.77	47.02
600	563.62	350.68	71.07
700	863.12	576.34	111.51
800	1271.78	830.52	144.63
900	1812.21	1184.45	221.38
1000	2461.83	1623.03	295.71
2000	19607.40	12781.80	2023.16

表 2: 不同数据规模下并行算法优化效果

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，而 pthread 多线程优化时，总共开启了 8 条线程，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。因此从时间表现情况来看，理论上 SIMD 优化算法所需要的时间应该是串行算法的 $\frac{1}{4}$，pthread 多线程所需要的时间应该是 SIMD 向量化的 $\frac{1}{7}$。而从实验数据来看，当问题规模较小的时候，pthrad 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要远远多余简单的运算操作。因此当问题规模较小时，由于运算操作在整个问题求解的过程中所占比例较低，因此线程额外开销的副作用就会显现出来。而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图4.2所示。

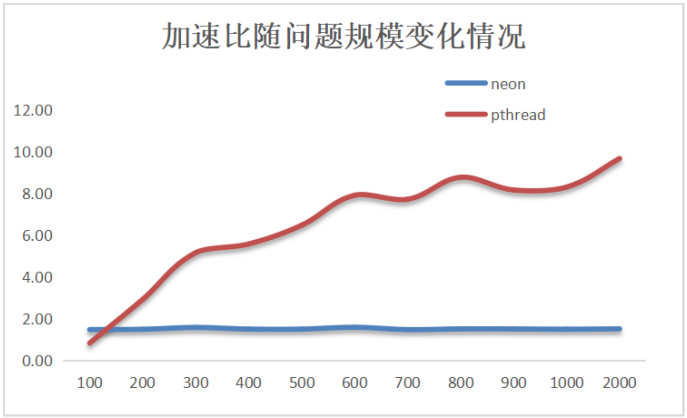

图 4.2: arm 平台加速比随问题规模变化情况

从图像中可以看出，SIMD 的加速比随着问题规模的增加基本保持稳定，由于算法中还涉及到其他的数据处理，因此其加速比只达到了 2 左右，并没有能够达到理论上的 4。而 pthread 优化的效果则随着问题规模的增加呈现出持续上升的趋势。这是因为，问题规模的增加，使得程序在运行的过程中，运算所占比例不断上升，这将会逐步抵消由于线程切换导致的额外开销。从数据中可以看出，当问题规模达到 2000 时，已经接近了其对 SIMD 的理论加速比。可以推测，当问题规模持续上升时，这个加速比将会接近 7。

4.1.2 数据划分方式对比

本次实验中，除了进行基础的 pthread 多线程优化尝试之外，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。结合前文实验设计，分别对比了循环划分，块划分和动态划分三种方式，在不同问题规模下的表现效果，并以 SIMD 算法为 baseline，其加速比变化情况如下图4.3所示。

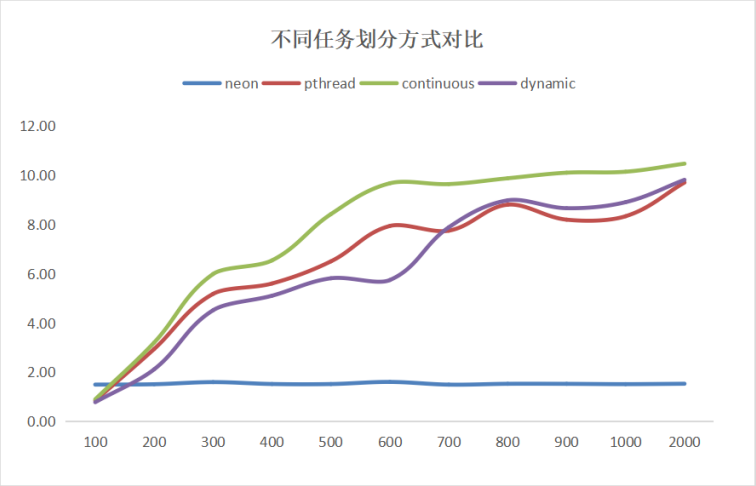

图 4.3: arm 平台不同任务划分方式对比

从 cache 优化的角度出发，循环划分和块划分的主要区别就在于能都利用到 cache 优化。就循环划分这种方式而言，线程在处理完当前行之后，接下来要处理的行距离当前间隔为 NUM_PHTREAD，因此当数据规模很大的时候，会因为 L1 cache 不能够容纳下足够的数据，或者由于 CPU 未能够及时的预取下一行数据，而导致 cache miss，因此需要额外的访存开销。而块划分的方式就能够很好的弥补这一点，其原因是对于每个线程而言，他所需要处理的数据之间在内存上是连续的，因此有很好的 cache 优势，因此能够减小由于 cache miss 导致的额外访存开销。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3所示。块划分的命中率能够达到 98%，而循环划分的方式只有 94% 左右，两者差异不大，因此在性能表现上的差异也不显著。

	discrete	**continuous**
L1 cache Hit	94.2%	98.1%

表 3: 循环划分和块划分 cache 命中对比

从负载均衡的角度出发，循环划分和动态数据划分的主要区别就在于能否充分利用各个线程的计算资源，尽可能减少同步等待所导致的额外开销。如果采用循环数据划分的方式，由于各个线程完成任务所需要的时间不尽相同，并且由于问题规模可能不是线程的整数倍，因此可能存在某些线程较早

完成任务进入同步等待状态，而其他线程还未完成任务，因此就浪费了一些计算资源。而动态数据划分就是从这个角度出发，尽可能充分利用每个线程的计算资源，使得任务能够在线程之间得到比较均匀的划分。从图4.3中也可以看出，当问题规模较小的时候，动态划分方式的表现不如循环划分，这是由于动态划分在保证负载均衡的前提下，牺牲了线程调度的开销，由于每个线程不清楚自己具体的工作，因此会存在比较大的线程同步和线程切换的开销，这种额外开销在问题规模比较小的时候会格外显著。而当问题规模提升的时候，可以发现，动态划分方式的表现已经能够超越循环划分，负载均衡带来的收益已经抵消了线程调度的额外开销。

4.1.3 线程数量对比

本次实验中，还探究了 pthread 多线程优化方法，在开启不同的线程数量时，优化效果的变化情况。为了能够显著体现 pthread 的优化效果，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所示。

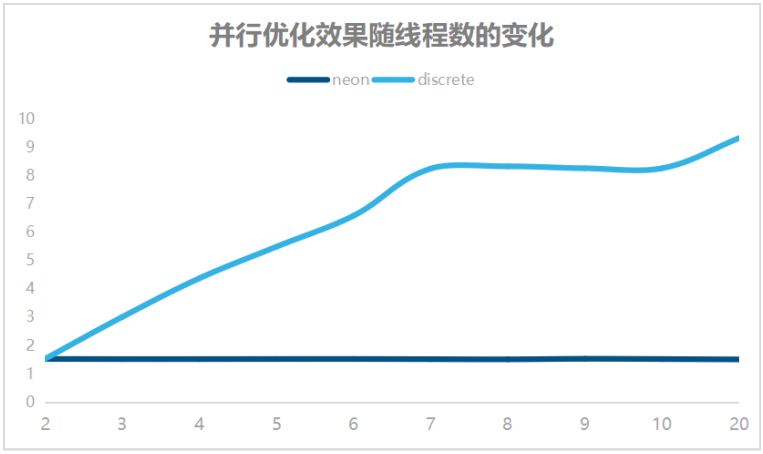

图 4.4: arm 平台不同线程数量效果对比

从图像中可以看出，随着线程数量的线性增加，pthread 多线程的优化效果也是呈现出线性提升的趋势。而当线程数量超过 8 个之后，其优化效果不再有显著的变化。这是由于实验使用的服务器单 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，就需要和服务器中的其他 CPU 核心借用线程，这之间会存在着额外的调度开销，因此抵消掉了性能的提升效果。

4.2 x86 平台迁移

4.2.1 多种 SIMD 指令集架构融合

基于前文在 ARM 平台上对于 pthread 多线程编程的探究，在本次实验中还将 pthread 多线程优化方法迁移到 x86 平台上，做同样的实验探究。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，而 pthread 多线程优化时，总共开启了 8 条线程，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 x86 平台上拥有更多的 SIMD 指令集架构，因此实验中分别探究了 SSE、AVX 和 AVX512 三种指令集架构配合 pthread 多线程的优化效果，测量在不同问题规模下的运行时间，如下表4.4所示。可以看出，pthread 多线程可以结合多种 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，并没有出现在某种指令集架构下不能够发挥很好的多线程优势的现象。

N	serial	SSE	pthread_SSE	pthread_AVX	pthread_AVX512
100	0.84	0.58	3.78	2.52	0.61
200	6.84	4.21	6.71	4.47	1.08
300	20.11	13.63	11.33	7.55	1.82
400	49.25	31.11	17.49	11.66	2.82
500	95.62	62.38	26.57	17.71	4.28
600	183.31	121.97	42.21	28.14	6.80
700	336.12	224.22	59.62	39.75	9.60
800	570.02	365.38	91.04	60.69	14.66
900	855.31	527.51	120.19	80.13	19.35
1000	1141.21	721.16	160.29	106.86	25.81
2000	10387.91	7048.52	1770.93	1180.62	285.17

表 4: x86 平台不同数据规模下并行优化效果

此外，实验还以 SSE 指令集架构为例，探究了随着问题规模的变化，不同 SSE 向量化处理和 pthread 多线程结合 SSE 向量化处理这两种方法的表现情况，变化趋势图如下图4.5所示。

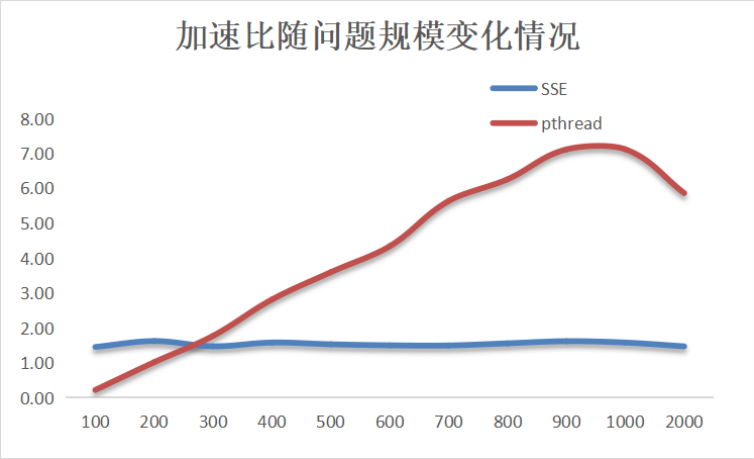

图 4.5: x86 平台加速比随问题规模变化

可以看出，在问题规模小于 1000 的时候，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，会发现加速比出现了一个下降的趋势。结合 VTune 性能分析工具分析的结果，如表5所示，分析其原因是因为，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，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。

N	L1 cache Hit	L2 cache Hit	L3 cache Hit
500	>99%	>99%	>99%
2000	92%	94%	90%

表 5: 不同数据规模下各级 cache 命中率对比

4.2.2 数据划分方式对比

本次实验中，还在 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，对于并行算法优化效果的影响。结合前文实验设计，分别对比了循环划分，块划分和动态划分三种方式，在不同

问题规模下的表现效果，并以 SIMD 算法为 baseline，其加速比变化情况如下图4.6所示。

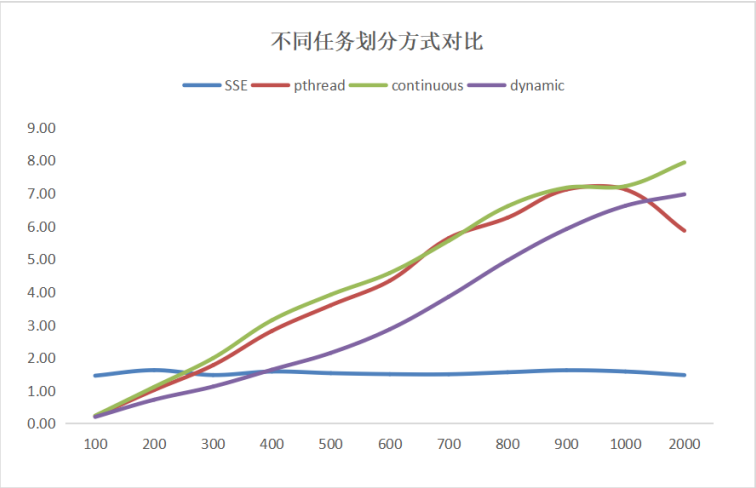

图 4.6: x86 平台数据划分方式对比

从 cache 优化的角度出发，循环划分和块划分的主要区别就在于能都利用到 cache 优化。就循环划分这种方式而言，线程在处理完当前行之后，接下来要处理的行距离当前间隔为 NUM_PHTREAD，因此当数据规模很大的时候，会因为 cache 不能够容纳下足够的数据，或者由于 CPU 未能够及时的预取下一行数据，而导致 cache miss，因此需要额外的访存开销。而块划分的方式就能够很好的弥补这一点，其原因是对于每个线程而言，他所需要处理的数据之间在内存上是连续的，因此有很好的 cache 优势，因此能够减小由于 cache miss 导致的额外访存开销。使用 VTune 工具对于这两种算法的 L1 cache 的命中率进行检测，如下表6所示。块划分的命中率能够达到 98.4%，而循环划分的方式只有 91.8% 左右，因此，对于块划分而言，由于其考虑到了 cache 特性，因此随着问题规模的增大，其性能并未明显受到访存开销的影响。而对于循环数据划分，则因为其划分方式会导致大量的 cache miss，因此访存开销会极大影响其性能表现。这也正符合图4.6的变化趋势。

	discrete	**continuous**
L1 cache Hit	91.8%	98.4%

表 6: 循环划分和块划分 cache 命中对比

从负载均衡的角度出发，循环划分和动态数据划分的主要区别就在于能否充分利用各个线程的计算资源，尽可能减少同步等待所导致的额外开销。从图4.6中也可以看出，当问题规模较小的时候，动态划分方式的表现不如循环划分，这是由于动态划分在保证负载均衡的前提下，牺牲了线程调度的开销，由于每个线程不清楚自己具体的工作，因此会存在比较大的线程同步和线程切换的开销，这种额外开销在问题规模比较小的时候会格外显著。而当问题规模提升的时候，可以发现，动态划分方式的表现已经能够超越循环划分，负载均衡带来的收益已经抵消了线程调度的额外开销。根据 VTune 性能分析工具，观察这三种任务划分方式的 CPU 占用率，可以得到如下对比图4.7。从途中可以看出，当动态数据划分的 CPU 占用率一直保持一个较高水平，并且相对比较均衡。而对比其余两种划分方式，由于其没有考虑负载均衡，因此在 CPU 占用率这个指标上，其波动十分明显，甚至会出现低于 20% 的占用率，这是对于计算资源的严重浪费。

图 4.7: 不同划分方式 CPU 占用率对比

4.2.3 线程数量对比

本次实验中，还探究了 pthread 多线程优化方法，在开启不同的线程数量时，优化效果的变化情况。为了能够显著体现 pthread 的优化效果，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4.8所示。但是对于图像中为何在线程数超过 8 之后会出现一个先上升在下降的趋势并未能够得出合理的分析。猜测是因为本机拥有八个 CPU 核心和 16 个逻辑核，因此当线程数小于 8 的时候，线程之间的调度相对比较简单，其所需要的开销较小。而当线程数超过 8 但是小于 16 时，线程调度会变得复杂。而由于本机只拥有 16 个逻辑核，因此当线程数达到 16 时也达到了一个峰值，随后便会开始下降。

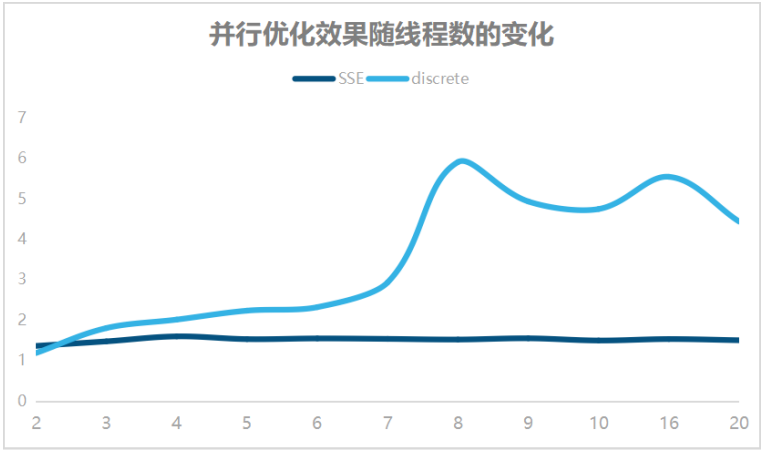

图 4.8: arm 平台不同线程数量效果对比

5 总结

在本次 pthread 多线程并行实验中,基于 Gauss 消元问题,对于消元过程中的减法操作采用 pthread 多线程优化，并在线程内进行除法或者减法运算时，仍然结合 SIMD 向量化处理，在 ARM 平台上采用 pthread+neon 的方式，在 x86 平台上采用 pthread+SSE/AVX/AVX512 的方式，探究了 pthread 多线程的并行优化效果。除此之外，还基于 cache 特性对比了循环划分和块划分的性能差异，基于负载均衡考虑对比了循环划分和动态划分的性能差异，可以验证，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均能够取得一定的性能提升。实验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，对于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。本次实验的相关代码和文档已经上传至GitHub。