



南開大學
Nankai University

计算机学院
并行程序设计期末报告

高斯消元法

姓名：王旭尧黄天昊
学号：2012527 2011763
专业：计算机科学与技术

2023 年 7 月 2 日

摘要

计算机的发展速度和摩尔定律非常相符。但在近几年，随着芯片集成规模极限的逼近，以及能耗和成本的因素，具有多核结构的产品逐渐成为市场的主流。并行程序的开发，也越来越成为计算机发展不可或缺的一部分。通过本学期的课程学习，作者尝试对高斯消元法进行并行程序设计。本文讨论了普通高斯消元法与消元子模式的特殊高斯消元法的并行优化算法，主要尝试从 Cache 优化、超标量优化、SIMD 优化、Pthread 优化、OPenMP 优化、MPI 优化等方面入手，探讨在消元时的合适优化策略。在文中也探讨了不同的平台、指令集、划分方式等各种因素对于并行程序性能的影响，侧重于通过算法探究来提升个人并行程序的研讨能力，为将来更好地运用并行程序做铺垫。

关键词：多核 并行 高斯消元法

Abstract

The speed of computer development is very consistent with Moore's Law. However, in recent years, with the approaching limit of chip integration scale, as well as the factors of energy consumption and cost, products with multi-core structure gradually become the mainstream of the market. The development of parallel programs is becoming an indispensable part of computer development. Through the course study of this semester, the author tries to design the parallel program for Gaussian elimination method. In this paper, the parallel optimization algorithms of ordinary Gaussian elimination method and special Gaussian elimination submode are discussed, and the appropriate optimization strategies are discussed from the aspects of Cache optimization, superscalar optimization, SIMD optimization, Pthread optimization, OPenMP optimization, MPI optimization and so on. The paper also discusses the influence of different platforms, instruction sets, partitioning methods and other factors on the performance of parallel programs, focusing on improving the research ability of individual parallel programs through algorithm exploration, paving the way for better use of parallel programs in the future.

Key Words: multicore parallel Gaussian elimination

目录

1 引言	5
2 前期准备	5
2.1 研究运用的主要环境	5
2.2 研究运用的主要软件	5
2.2.1 Code::Blocks	5
2.2.2 MobaXterm	6
2.2.3 Intel VTune Profiler 2022.2	6
3 高斯消元问题分析	6
3.1 问题背景	6
3.2 普通高斯消元	6
3.3 特殊高斯消元	7
3.3.1 问题引出	7
3.3.2 特点分析	8
3.3.3 算法思路	8
3.3.4 并行优化方案	9
4 普通高斯消元问题探究	10
4.1 Cache 优化	10
4.1.1 X86 平台下的 Cache 优化	10
4.1.2 ARM 平台下的 Cache 优化	11
4.2 超标量优化	11
4.2.1 X86 平台下的超标量优化	12
4.2.2 ARM 平台下的超标量优化	12
4.3 SIMD 优化	13
4.3.1 X86 平台下的 SIMD 优化	13
4.3.2 ARM 平台下的 SIMD 优化	14
4.4 Pthread 优化	15
4.4.1 X86 平台下的 Pthread 优化	15
4.4.2 ARM 平台下的 Pthread 优化	18
4.5 OPenMP 优化	19
4.5.1 X86 平台下的 OPenMP 优化	19
4.5.2 ARM 平台下的 OPenMP 优化	22
4.6 MPI 优化	23
4.6.1 金山云 (X86) 平台下的 MPI 优化	24
4.6.2 ARM 平台下的 MPI 优化	27
5 消元子模式的特殊高斯消元	28
5.1 SIMD 优化	28
5.1.1 X86 平台下的 SIMD 优化	29
5.1.2 ARM 平台下的 SIMD 优化	29

5.2	Pthread 优化	30
5.2.1	线程版本	31
5.2.2	线程数量	32
5.3	OpenMP 优化	33
5.3.1	X86 平台下的 OpenMP 优化	33
5.3.2	ARM 平台下的 OpenMP 优化	33
5.3.3	chunk_size 与 thread	34
5.4	MPI 优化	35
5.4.1	MPI 多进程优化	35
5.4.2	MPI 与 OpenMP 的组合优化	36
5.5	CUDA 优化	37
5.6	细节规划	38
5.6.1	数据读入	38
5.6.2	存储方式	38
5.6.3	多平台多指令集	38
5.7	各种优化方法总结	38
6	总结与展望	39
6.1	总结	39
6.2	展望	39

1 引言

在王刚老师的带领以及助教们的帮助下，本学期的《并行程序设计》课程，让我们感觉收获颇丰。课上课下，我们不仅仅了解了并行程序开发的基础知识，更是掌握了如 Latex 排版、Linux 命令行操作、VTune 性能分析等一系列基本技能。当今计算机多核并行的发展趋势，也越来越说明了开设这门课程的远见性与必要性。课程内容充实足量，可能我们未能彻底地掌握所有并行程序设计的知识点，但是已经学到的东西已足以让我们看到并行程序设计的精妙。并行程序的开发，无疑是将来一个及其重要的开发领域，是人们追求计算机更快、更强的必经之路。

通过这学期的不断学习，我们也尝试了运用并行程序设计的思想解决实际问题。对于选题，我们采取了老师给的默认选题高斯消元，在接下来的内容中，我们将介绍运用 SIMD 并行化、Cache 优化、Pthread 并行化等多种并行优化方法对于普通高斯消元、特殊高斯消元算法的性能提升，同时，我们还将对在不同实验平台、不同指令集、多线程的不同算法策略等方面进行比较分析，力求运用所学，尽最大能力完成高斯消元的并行优化探索。

2 前期准备

2.1 研究运用的主要环境

我们在实现高斯消元时使用的主要环境分别是本地的 X86 平台、华为鲲鹏服务器上的 ARM 平台、金山云上的 X86 平台。由于金山云 X86 平台使用不多，网上搜索也未能找到相关 CPU 信息，因此以下仅给出鲲鹏服务器以及本地的 CPU 相关信息。

\	ARM	X86
CPU 型号	华为鲲鹏处理器	Intel Core i7-10510H
CPU 主频	2.6GHz	2.3GHz
L1 Cash	64KB	256KB
L2 Cash	512KB	1.0MB
L3 Cash	48MB	8.0MB
内核	4	4

2.2 研究运用的主要软件

2.2.1 Code::Blocks

对于代码的编写和调试，我们这里主要运用 Code::Blocks 这款软件。Code::Blocks 是一个开放源码的全功能的跨平台 C/C++ 集成开发环境。Code::Blocks 由纯粹的 C++ 语言开发完成，它使用了著名的图形界面库 wxWidgets(3.x) 版。由于其优越的跨平台性以及开放源码的特点，它不但可以支持 Windows 和 GNU/Linux，而且 Windows 用户可以不依赖于 VS. NET，编写跨平台 C++ 应用。这就为我们在不同平台上进行高斯消元的测试提供了方便。Code::Blocks 具有灵活而强大的配置功能，支持代码完成，支持工程管理、项目构建、调试。因此，我们大多数情况下选择 Code::Blocks 进行编写和调试，然后将代码放置于不同平台上去运行。

2.2.2 MobaXterm

MobaXterm 又名 MobaXVT，是一款增强型终端、X 服务器和 Unix 命令集 (GNU/Cygwin) 工具箱。我们大多数的在鲲鹏服务器上的连接、运行操作，都在该软件上完成。

MobaXterm 可以开启多个终端视窗，而且 MobaXterm 还有很强的扩展能力，可以集成插件来运行 Gcc, Perl, Curl, Tcl / Tk / Expect 等程序。我们这里一般用 MobaXterm 来支持各种连接如: SSH, X11, RDP, VNC, FTP, MOSH 等。然后，运用相关平台时，可以通过 Unix 命令来完成一系列操作。传输文件时，也只需要连接 SSH 终端，来支持 SFTP 传输文件。当然，MobaXterm 也能安装各种插件，为我们使用提供了进一步的便利。

2.2.3 Intel VTune Profiler 2022.2

Intel VTune Profiler 是一个全平台的性能分析工具，可以支持分析本地或远程的 Windows, Linux 及 Android 应用，这些应用可以部署在 CPU, GPU, FPGA 等硬件平台上。支持分析的语言包括: DPC++, C, C++, C#, Fortran, OpenCL, Python, Go, Java, 汇编等。

我们在高斯消元的性能分析中，也采取了这款工具进行性能分析，同时，我们也可运用 Perf 生成报告，然后运用 VTune 再做分析。Perf 的相关用法我们可以在[这里](#)找到。VTune 借助 Perf 的分析，则可参考这篇[博客](#)。

3 高斯消元问题分析

3.1 问题背景

求解线性方程组，作为众多学科与工程计算的核心，在实际的工程应用中大量存在。而高斯消元法是求解线性方程组的一种最基本和最简单的算法。由于它的结果较为稳定并且精确性高，因而在许多领域中得到运用。

高斯消元法，主要通过对方程组进行增广矩阵初等行变换，达到消除未知量的目的。其构成包括消元和回代，基本思想是：

- (1) 消元过程：将方程组逐列逐行消去变量，转化为等价的上三角形方程组。
- (2) 回代过程：按照方程组的相反顺序求解上三角形方程组，得到原方程组的解。

随着时代的发展，人们运算的线性方程组维数日益增大，运算复杂度不断提升。如何加速数据处理的过程，缩短数据处理时间，提升系统整体的实时性已成为关键问题。如果我们对于高斯消元法使用并行化的优化策略，相信对数据处理效率的提高大有帮助。我们将借助着这个背景探索普通高斯消元以及消元子模式下的特殊高斯消元。

3.2 普通高斯消元

考虑在整个消去的过程中，如图所示，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，使得该行转化为首元素为 1 的一行
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

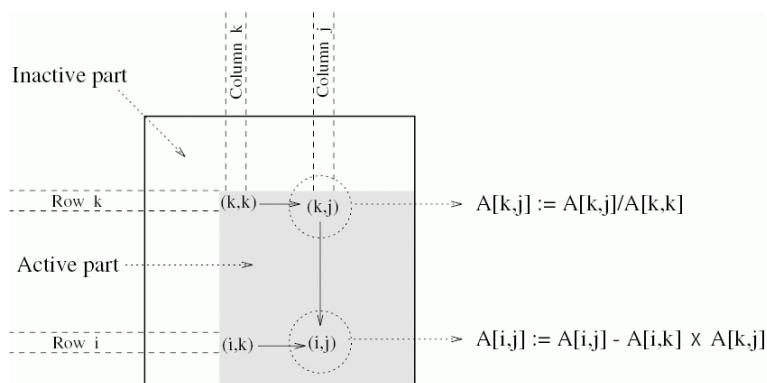


图 3.1: 高斯消去法逻辑示意图

我们结合助教提供的资料，得到高斯消去的串行算法伪代码如下，也就是我们之前所说的消元、回代过程。

Algorithm 1 高斯消元 LU 分解

Input: 系数矩阵 $A^{(1)}$

Output: LU 分解后的矩阵 U

```

1: for  $k = 1 \rightarrow n$  do
2:   for  $j = k + 1 \rightarrow n$  do
3:      $A[k][j] \leftarrow A[k,j]/A[k,k]$ 
4:   end for
5:    $A[k][k] \leftarrow 1.0$ 
6:   for  $i = k + 1 \rightarrow n$  do
7:     for  $j = k + 1 \rightarrow n$  do
8:        $A[i][j] \leftarrow A[i,j] - A[i,k] \times A[k,j]$ 
9:     end for
10:     $A[i][k] \leftarrow 0$ 
11:  end for
12: end for
  
```

观察高斯消去算法的算法逻辑示意图和伪代码，针对上述提到的两个过程，我们发现都适合采用并行的方式进行性能的优化。我们在这两个过程中可以使用 cache 优化、超标量优化（指令级并行）、采用循环展开技术降低循环操作、SIMD 并行化、Pthread 并行化（结合多线程并行化的不同算法策略优化）、OpenMP 并优化、MPI 并行化。再通过对于高斯消去法的不同优化方案在不同平台、不同指令集（SSE、AVX、AVX-512 等）的情况下，我们可以比较分析出最佳的优化策略。同时，由于优化策略的可叠加性，我们可以采取将多种优化方案结合，以此对于一定规模的高斯消元得到最大化的性能提升。

3.3 特殊高斯消元

3.3.1 问题引出

多项式方程组求解是计算机代数中的一个基础问题，经调查发现 Gröbner 基方法是最有效的方法之一，也是研究代数与代数几何时不可或缺的强有力工具。

1965 年,奥地利数学家 W. Gröbner 的学生 B. Buchberger 提出了 Gröbner 基这一概念并介绍了一种算法——Buchberger 算法,其用于计算多项式环理想的一组具有良好性质的基——Gröbner 基或标准基,且其算法现已被诸多计算机代数系统改进,诸如: Maple、Mathematica、Magma、Sage、Singular、Macaulay 2、CoCoA 等。

Buchberger 提出的原始 Gröbner 基算法,主要存在三方面的改进:

1、冗余计算、无用准则对存在于 S-多项式约简到 0 的过程中,这将消耗大量的时间,降低 Gröbner 基算法的计算效率,因此,选择合适的准则,删除冗余计算、无用准则对是必要的。

2、在计算 Gröbner 基时,消耗时间最多的是最基本的多项式约简运算过程——多项式的加法及单项式和多项式之间的乘法,因此,提高多项式约简的效率对加速整个算法具有重要意义。

3、在计算 Gröbner 基时,不同单项式序的选择对算法的效率具有很大的影响,同时选择策略对算法效率的提升中也扮演着重要的角色,因此,选择恰当的单项式序和策略对提高算法的性能也是必不可少的。^[9]

基于这些改进方向,无数学者在寻找更高效的算法方面做了深入研究。1979 年, Buchberger 提出了两条准则、一条选择策略用于改进 Gröbner 基算法,查阅资料我们可以了解到其中两条准则是用于删除冗余的 S-多项式的计算^[2],提高原始 Buchberger 算法的计算效率。而在 1983 年, Lazard 介绍了一种基于线性代数理论计算 Gröbner 基的方法^[7]。1985 年, Buchberger 在文献^[3]中详细介绍了 Gröbner 基理论与算法。1991 年, Giovini、Mora、Niesi 等人则基于 ≤ 2 -非齐次多项式理想提出了“suger degree”选择策略,用于选择准则对或 S-多项式对^[1]。1992 年, Möller、Mora 等人首次在文献^[4]中提出利用合冲模方法删除冗余 S-多项式计算,思想虽然值得借鉴,但其算法计算效率低。于是, Faugère (1999)、Courtois (2000)、Ding (2008) 等人基于 Lazard 的方法又提出了著名的 F4 算法^[5]、XL 算法^[8]、基于 XL 算法改进的 MutantXL 算法^[6]。其中, F4 算法可以同时高效的删除多个冗余的 S-多项式,其计算效率高,且 F4 算法已在计算机代数系统 Maple 和 Magma 中改进。2002 年, Faugère 基于 Möller、Mora 等人的合冲方法,介绍了签名的概念以及重写准则,用于删除冗余 S-多项式,并提出了 F5 算法,其计算效率比 F4 算法更高效,且在当时被认为是最高效的算法,并成功解决了著名的密码学问题——隐藏域方程组 (Hidden Field Equation HFE) 问题,但 F4、F5 算法在计算 Gröbner 基的过程中将会消耗大量的计算机内存。近几年,无数科学家更是持续在这个领域辛勤耕耘,收获颇丰,比如布尔多项式环上的 Gröbner 基算法、利用主合冲以及在计算 Gröbner 基的过程中得到的非主合冲提出了新的准则以找出更多的冗余计算。

3.3.2 特点分析

算法源自布尔 Gröbner 基计算。并且在 HFE80 的 Gröbner 基计算过程中,高斯消元时间占比可以达到 90% 以上,得到广泛运用,因此我们对其进行并行优化设计,以此获得运算速度的提升。

由于应用的特殊性,运算均为有限域 $GF(2)$ 上的运算,即矩阵元素的值只可能是 0 或 1。这也导致其加减乘除运算的简便性,即加减法均为异或运算,乘除法变为与运算。高斯消去过程中实际上只有异或运算——从一行中消去另一行的运算退化为减法。正是对于矩阵这个特性,我们提出了“消元子”模式的高斯消元,希望通过并行优化的方法取得更快的运算速度。

3.3.3 算法思路

对于特殊的高斯消元,我们将矩阵分为“消元子”和“被消元行”两类,在输入时即给定。

我们可以这样理解两者关系。“消元子”因为首个非零元素的位置不同而相互区分,因而首个非零元素的位置可以认为是“消元子”区分的“标签”。“消元子”可以理解作为一种化简矩阵的特定行存在,

即为普通高斯消元法结束后上三角矩阵（或下三角矩阵）的某个特定行。

如图中所示，我们对于各行消元子，给出的“标签”分别为 1,2,3,4,5,7（也就是首个非零元素的位置）。

首项 列号	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
25	1	0	1	0	1	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0
24	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
23	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
22	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
21	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
20	NULL																									
19	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

图 3.2: “标签” 阐述

而“被消元行”则是相当于未处理的“消元子”。由于矩阵中只存在 0 和 1 两个数字，我们只需要观察“被消元行”的每个 1，是否都有对应的“消元子”，若全部存在，说明“被消元行”最终必能全部化为 0 的一行，我们可将其丢弃。若存在无法消去的 1，则将其首个无法消去的 1 作为“标签”，再将“被消元行”作为“消元子”存在。如此循环往复，直到处理完所有“被消元行”，那么也就完成了高斯消元。

接下来我们给出特殊高斯消元法的伪代码：

符号说明：

R: 所有消元子构成的集合

R[i]: 首项为 i 的消元子

E: 所有被消元行构成的数组

E[i]: 第 i 个被消元行

lp(E[i]): 被消元行第 i 行的首项

Algorithm 2 Gröbner 基计算中的高斯消元

Input: 所有消元子构成的集合 R

所有被消元行构成的数组 E

被消元行数 m

Output: E

```

1: for  $i = 0 \rightarrow m - 1$  do
2:   while  $E[i] \neq 0$  do
3:     if  $R[\text{lp}(E[i])] \neq \text{NULL}$  then
4:        $E[i] \leftarrow E[i] - R[\text{lp}(E[i])]$ 
5:     else
6:        $R[\text{lp}(E[i])] \leftarrow E[i]$ 
7:       break
8:     end if
9:   end while
10: end for

```

分析上述伪代码，我们不难发现在“被消元行”被“消元子”时，我们可以进行并行优化操作。

3.3.4 并行优化方案

1、SIMD

我们将尝试采用多层级的 SIMD 优化策略。由于多核处理的体系存在，我们会将多个“被消元行”用现有的“消元子”集合进行同时处理，再对处理后的所有“被消元行”进行“标签”的标记。接着，我们再将处理后的“被消元行”进行内部“消元子”“被消元行”划分，如此循环往复处理，最后得到“标签”都为唯一的一组处理后的“被消元行”，将其全部加入到原有的“消元子”中去，即可完成多个“被消元行”的处理。不过由于“被消元行”的重复自我处理，有可能存在冗余，我们需要根据具体的情况进行优化设计。

2、Pthread

我们考虑在“被消元行”被“消元子”处理时，尝试交叉计算、同时运行的优化效果，同时快速排除舍弃全为 0 的行。另外，可以利用 pthread 技术尝试实现多线程的局部“标签”唯一化处理，以分治的思想，进行问题的划分后，再综合。

3、OpenMP 和 MPI 编程

OpenMP 和 MPI 编程技术可以设计在多处理器上的并行计算方案，因此我们需要对于特殊的高斯消元的优化考虑此方案是否也可行。我们将尝试使用将矩阵横向分块处理，进行多处理器的计算划分，然后再进行全局归并。

4 普通高斯消元问题探究

4.1 Cache 优化

在对一个 $n \times n$ 矩阵，计算每一列与给定向量的内积这个问题上。我们设计两种算法计算内积，一种算法为平行算法，即逐列访问矩阵元素，一步外层循环；另一种为 cache 优化算法，即逐行访问矩阵元素，一步外层循环。由于后者访存更符合空间局部性以及 cache 的命中机制，因此通过两个算法时间消耗的比较，突出 cache 优化算法的好处及计算机的底层运行机制。

我们还实现了分平台优化比较。在本地 X86 平台上，我们实现通过 Code::Blocks 运用 QueryPerformance 系列函数进行程序时间测算，通过时间来比较性能。另外，我们在 VTune 上运行两种算法的程序，分析运行时间的长短、cache 命中率、超标量等指标，比较得出两种算法的优劣。在 ARM 平台上，我们采取 perf 显示各项程序性能，比较分析各级缓存的命中率，由此比较得出两种算法的优劣。

4.1.1 X86 平台下的 Cache 优化

1. 在 CodeBlocks 运用 QueryPerformance 系列函数进行时间的测算

我们通过用 QueryPerformance 系列函数对于平行算法、优先算法两者进行计时，得到如下实验结果：

Algorithm	Problem scale	Total time(ms)	Number of cycles	Single cycle time(ms)
Parallel algorithm	n=512	99.0	100	0.990
	n=1024	575.9	100	5.759
	n=2048	7671.2	100	76.712
	n=4096	36160.8	100	361.608
Optimization algorithm	n=512	63.1	100	0.631
	n=1024	234.7	100	2.347
	n=2048	998.8	100	9.988
	n=4096	3982.5	100	39.825

分析上述表格我们可以发现：优化算法始终快于平行算法，而且当问题规模越大时，优化算法所凸显的优势越大。由此我们可以猜测，当问题规模较小时，由于优化算法循环次数多，其符合空间局

部性（访存连续）以及 cache 的命中机制的优势不显著；当问题规模较大时，程序运行的时间主要花费在数据提取上，导致优势异常显著。

2. 运用 VTune 进行各项指标的分析

我们从 VTune 的分析中可以看出各级缓存的命中情况，以及两种算法的运行速度。由于数据量庞大，我们下面以图片形式展现。

Algorithm	Parallel algorithm				Optimization algorithm			
Problem scale	n=512	n=1024	n=2048	n=4096	n=512	n=1024	n=2048	n=4096
Number of cycles	100	100	100	10	100	100	100	10
L1_HIT	387,043,950	1,666,396,745	6,269,685,975	2,300,945,455	338,462,390	1,365,605,795	5,586,807,960	2,450,992,400
L1_MISS	9,700,460	7,956,810	31,830,915	3,504,475	4,706,720	1,532,410	6,268,755	2,503,145
L2_HIT	352,020	835,905	3,784,850	0	216,505	970,880	5,153,305	2,00,735
L2_MISS	9,349,855	7,124,245	28,068,210	3,253,990	4,491,045	562,105	1,119,140	250,585
L3_HIT	9,285,870	6,492,980	15,506,215	500,715	4,461,360	503,340	606,310	0
L3_MISS	18,375	721,505	11,406,690	1,500,150	14,410	48,435	500,960	0

分析表格我们不难发现，优化算法的各级缓存命中率都远高于平行算法，由此可见优化算法通过对空间局部性以及 cache 的命中机制的适应，大幅度提升了计算机的运行效率。特别是当问题规模比较大时，优化算法的各级缓存未命中率几乎只有平行算法的五分之一。对于命中次数和未命中次数优化算法都比平行算法小，我想是因为当未命中时，会继续尝试命中，导致命中数增多。

4.1.2 ARM 平台下的 Cache 优化

我们通过脚本在鲲鹏服务器上通过输入各项命令行，运行 perf 显示各项程序性能，比较分析各级缓存的命中率，得到如下结果：

n	L1-dcache-load-misses	L1-dcache-loads	L1-dcache-stores
512	3,488,144	449,002,409	449,004,358
1024	26,944,787	1,792,804,196	1,792,812,115
2048	168,078,268	7,166,620,139	7,66,586,053

表 1: 平行算法性能测试

n	L1-dcache-load-misses	L1-dcache-loads	L1-dcache-stores
512	1,010,003	449,165,005	449,161,734
1024	4,024,358	1,792,912,798	1,792,916,665
2048	16,030,759	7,165,562,569	7,165,548,616

表 2: cache 优化算法性能测试

分析量表数据我们不难发现，无论数据量的大小，两种算法的一级数据缓存读取次数几乎相等，但 cache 优化算法的 L1 未命中数都远低于平行算法。由此可以得出结论：优化算法通过对 cache 的命中机制的适应，大幅度提升了计算机的运行效率。

4.2 超标量优化

在计算 n 个数的和，我们考虑两种算法设计思路：逐个累加的平凡算法（链式）；适合超标量架构的指令级并行算法（相邻指令无依赖），如最简单的两路链式累加，再如递归算法——两两相加、中间

结果再两两相加，依次类推，直至只剩下最终结果。

设计四种算法计算累加和，其中分为一种平行算法，三种优化算法。平行算法采用直接链式相加求和的方法，优化算法分别采用多链路径、递归、二重算法三个方法进行优化。我们通过使用高精度计时测试程序执行时间，比较算法的性能。

由于两路链式算法能更好地利用 CPU 超标量架构，两条求和的链可令两条流水线充分地并发运行指令，因此两者在各项指标的评价上应该理论上更优于平行算法。我们将在实验结果中的总体执行的周期数 (Clockticks)，执行指令数 (Instructions Retired) 以及 CPI (IPC 的倒数，每条指令执行的周期数) 等评价指标中验证我们的猜想。

4.2.1 X86 平台下的超标量优化

我们选择本地 X86 作为实验环境，运用 VTune 进行分析。在 VTune 中，我们分析上文提到的总体执行的周期数 (Clockticks)，执行指令数 (Instructions Retired) 以及 CPI (IPC 的倒数，每条指令执行的周期数) 等指标，比较得出两种算法的优劣。然后，改变测试数据量的大小，反复多次实验。

我们选择 Microarchitecture Exploration 类型，接下来我们进入 Bottom-up 数据栏，在这一页面中我们可以看到具体每个函数执行的 CPU 时间，周期数，执行指令数以及 CPI。下图是我们设置问题规模为 2^{15} 个元素求和 100 次和 2^{22} 个元素求和 100 次的实验结果。

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
► DoubleCycle	30,600,000	55,800,000	0.548
► chain	19,800,000	30,600,000	0.647
► recursion	19,800,000	70,200,000	0.282
► Multipath	12,600,000	19,800,000	0.636

图 4.3: 2^{15} 个元素求和 100 次

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate
► DoubleCycle	3,587,400,000	7,547,400,000	0.475
► recursion	3,385,800,000	7,137,000,000	0.474
► chain	2,606,400,000	4,615,200,000	0.565
► Multipath	1,834,200,000	3,159,000,000	0.581

图 4.4: 2^{22} 个元素求和 100 次

根据分析结果，如上述两图所示，我们可以看到在问题规模为 2^{15} 个元素求和 100 次时，chain 函数其执行指令数为 19,800,000，执行周期数为 30,600,000，其 CPI 为 0.647。明显不及多链路径、递归、二重算法三个优化算法的 CPI，这与我们之前的分析相同。同理，当扩大问题规模时，我们发现算法的优劣规则仍然使用，但 CPI 之间的区别缩小，我们这里猜测可能原因是与计算机本身的工作负载有关。

4.2.2 ARM 平台下的超标量优化

我们通过脚本在鲲鹏服务器上通过输入各项命令行，在 ARM 平台上运行 perf 显示各项程序性能，下面我们得到的是各种算法运行的数据结果。

n	instructions	cycles	isn per cycle
2^{15}	63,782,222	25,409,507	2.51
2^{22}	8,028,962,154	3,208,400,156	2.50

表 3: chain 算法

n	instructions	cycles	isn per cycle
2^{15}	40,845,394	17,982,892	2.27
2^{22}	5,092,950,126	2,246,591,721	2.27

表 4: Multipath 算法

n	instructions	cycles	isn per cycle
2^{15}	96,593,758	32,985,261	2.93
2^{22}	12,223,331,362	4,297,865,582	2.84

表 5: 递归算法

n	instructions	cycles	isn per cycle
2^{15}	99,851,636	34,638,168	2.88
2^{22}	5, 137,119,500	12,642,733,754	2.46

表 6: 二重算法

从上述四张表中的数据，我们不难发现当数据量较小时，的确是优化算法比平行算法 IPC 更大，性能更好，不同算法分别利用了 CPU 超标量架构，两条求和的链、两条流水线充分地并发运行等方法实现算法的快速运算。比如递归算法、二重算法的 IPC 在 2^{15} 运算量级，IPC 分别为 2.93、2.88，高于 chain 算法的 IPC。

但为什么会出现当数据量上升，各种优化算法 IPC 下降的情况呢？甚至有些优化算法甚至低于 chain 算法呢？特别是 Multipath 的 IPC，尤其引人深思。我想，一个主要的原因就是当数据量大到一定程度，我们访问数据不仅要使用缓存，还需要用到内存，这时由于优化算法本身代码的长度较长，便会导致程序运行缓慢。

4.3 SIMD 优化

4.3.1 X86 平台下的 SIMD 优化

同样的，我们这次在本地 X86 平台下，也采用 VTune 进行性能分析。在 VTune 中，我们对于串行、SSE、AVX 获取 CPI 等数据后，进行比较分析。接下来，我们在下表给出 VTune 对于上述三种算法运行的结果总代码的结果，如下表所示：

	Clockticks	Instructions Retired	CPI Rate
普通	26,219,905	18,262,865	1.436
SSE (4 路)	22,322,635	19,292,099	1.157
AVX (8 路)	26,313,174	19,142,454	1.375

通过分析，我们不难发现，在 VTune 中运行时，SSE 和 AVX 的 CPI 小于串行的高斯消元算法，但是 AVX 的 CPI 值大于 SSE 的 CPI 值，由于运行代码时，每次结果存在随机性，所以我们这里考虑可能是数据量偏小、随机误差或是数据读取时产生误差。

同时，我们在 Code::Blocks 中，运用 QueryPerformance 进行对齐后的三种算法在不同数据规模下的时间测算，获得相应的数值，记录，如下表。

数据规模 $n \times n$	8	64	128	256
循环 100 次				
串行	0.0978ms	35.2388ms	326.849ms	2009.52ms
SSE (4 路)	0.086ms	25.1488ms	172.215ms	846.773ms
AVX (8 路)	0.1044ms	17.6971ms	152.033ms	565.541ms

分析数据，我们可以发现当数据规模较小时，三者循环运行 100 次差距不大。比如说当 $n=8$ 时，串行算法一度比 AVX 运行时间还短。说明当数据规模比较小时，由于运行时间主要取决于计算机本身性能以及代码的长度以及复杂度。而随着数据规模的逐步扩大，我们就可以发现多路向量化的并行算法展现出越来越大的优势。AVX (8 路) 相较于 SSE (4 路) 更优，表现得也越来越显著。特别是当数据规模 $n = 256$ 时，也就是形成一个 256×256 的矩阵时，AVX 进行高斯消元仅需 0.5s 左右，而串行算法需要 2s，这就可以说明当数据规模十分大的时候，多路向量化的并行优化策略将变得十分显著。

4.3.2 ARM 平台下的 SIMD 优化

我们在 ARM 平台上用 Neon 算法，实现编程，在考虑是否对齐、算法影响以及一系列其他因素下对最后结果的影响。下表是我们在 ARM 平台上运用 perf 测算出的 IPC 以及循环 100 次高斯消元所消耗的时间。

数据规模 $n \times n$	8	64	128	256
循环 100 次				
instructions	1,572,042	151,742,795	1,153,548,704	9,013,541,418
cycles	1,245,164	113,920,220	870,619,102	6,887,785,177
insn per cycle	1.26	1.33	1.32	1.31
time	0.00133230s	0.43418470s	0.334184630s	2.668303600s

分析所得数据，我们可以发现 IPC 值在 Neon 指令中随着数据规模的增大，基本不变，时间随着数据规模的增大，大致成上升趋势。但我们发现当数据规模 n 由 64 变为 128 时，时间反而缩短了，多次测试都是这个结果。我怀疑，是否是因为数据量过小，因为两者缓存与寄存器之间存储关系不同的影响，而最终导致 $n=128$ 时比 $n=64$ 时运行更快。

接下来，我们再来看看在 ARM 平台上常规的串行算法所产生的结果：

数据规模 $n \times n$	8	64	128	256
循环 100 次				
instructions	1,693,842	301,206,795	2,389,897,505	19,063,224,613
cycles	1,139,513	136,430,056	1,060,162,981	8,433,661,804
insn per cycle	1.49	2.21	2.25	2.26
time	0.00110610s	0.52113560s	0.407390830s	3.24539860s

通过与串行算法的比较，我们还是能够从时间上清楚的发现，Neon 指令的 SIMD 算法还是总体上快于串行算法，这也从另一方面说明了 SIMD 的优化是有效的。

4.4 Pthread 优化

4.4.1 X86 平台下的 Pthread 优化

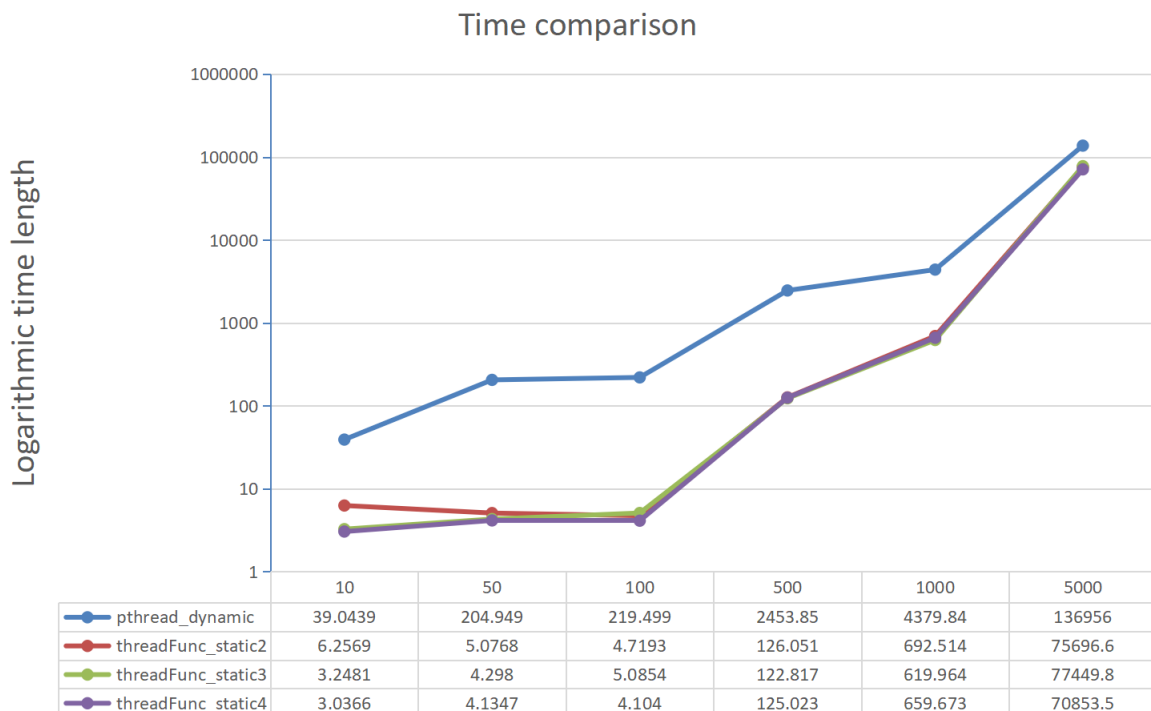
1、线程版本

我们在 X86 平台上运行四个版本的 Pthread 实验。根据指令集的不同，我们额外增加 SSE、AVX 的串行实验作为对比实验，与 SSE、AVX 的静态线程 + 信号量同步 Pthread 实验进行比较分析，下面是我们通过 QueryPerformance 系列函数进行计时所得到的结果。在实验时，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，我们总共开启了 4 条线程，其中一条线程负责除法操作，剩余的 3 条线程负责做减法操作。

NUM	10	50	100	500	1000	5000
Serial	NUM_THREAD = 4					
C++	0.0171	0.2437	1.0041	319.222	1752.42	157312
SSE	0.0026	0.2015	1.0509	262.647	1359.68	165085
AVX	0.0019	0.1121	0.3112	67.6693	358.329	95232
Pthread	NUM_THREAD = 4					
pthread_dynamic	39.0439	204.949	219.499	2453.85	4379.84	136956
threadFunc_static2	6.2569	5.0768	4.7193	126.051	692.514	75696.6
threadFunc_static3	3.2481	4.298	5.0854	122.817	619.964	77449.8
threadFunc_static4	3.0366	4.1347	4.104	125.023	659.673	70853.5
threadFunc_static2_SSE	3.1872	6.6205	4.2083	71.4814	292.912	35695.1
threadFunc_static2_AVX	3.9459	7.6563	6.2592	75.1557	307.225	36927.9

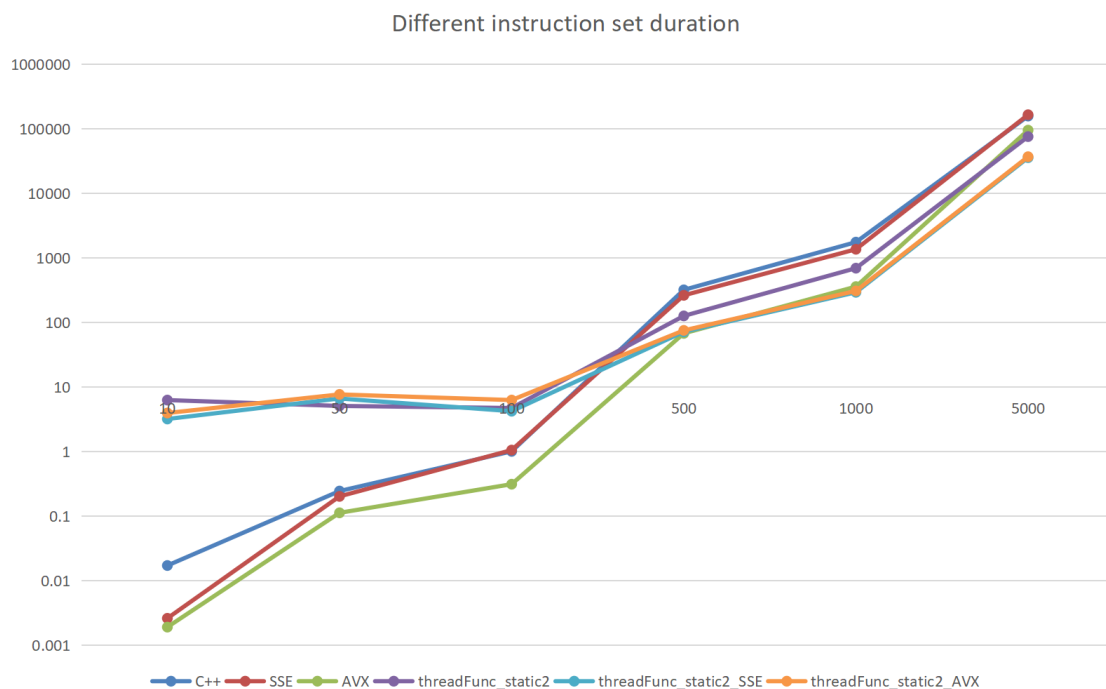
单位 (ms)

分析数据，我们不难发现，当指令集相同时，动态线程，每轮除法完成后创建线程，该轮消去完成后销毁线程。这样的话创建和销毁线程的次数过多，而线程创建、销毁的代价是比较大的。因此在任何数据规模，动态线程的时间都远大于静态线程所消耗的时间。我们根据表中数据不难绘制出如下的图：



观察表中数据结合图像，我们也可发现数据量较小时，使用动态线程，创建销毁线程耗时占比更大，当数据量规模增大后，占比减小。

对于不同的指令集，我们发现当数据规模小的时候，不使用多线程可能运行速度更快，说明多线程完成高斯消元节约的时间不及创建线程所浪费的时间。而当数据规模不断增大时，线程额外开销的副作用不断减小，多线程的优势得以彰显。



2、线程数量

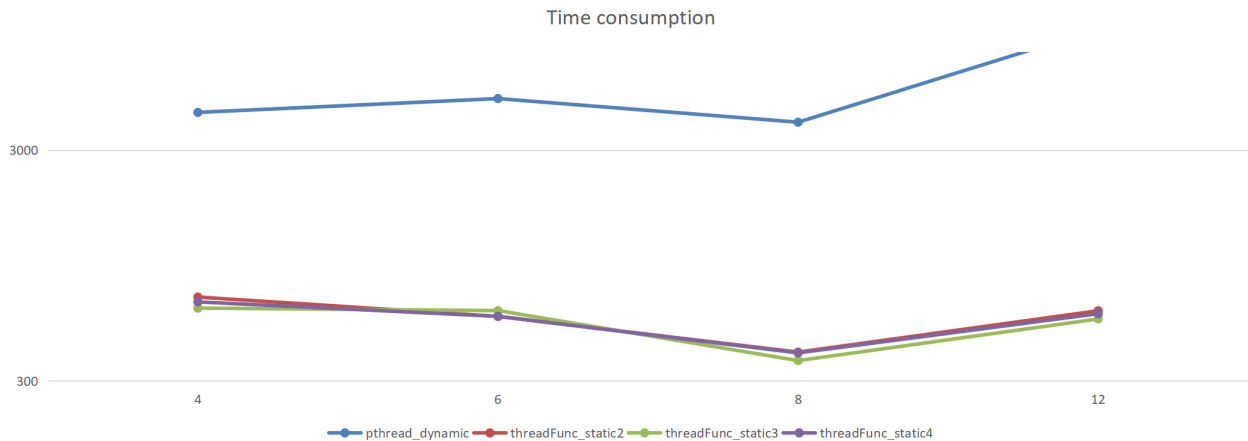
下面我们探讨线程数量对于运行效率产生的影响。我们分别设立 4、6、8、12 条线程，探究在不同指令集下、不同版本的运行情况。经过实验，得下表：

NUM_THREAD	4		6		8		12	
N	1000	5000	1000	5000	1000	5000	1000	5000
Serial								
C++	1752.42	157312	1650.2	155762	1252.77	150836	1623.7	NULL
SSE	1359.68	165085	1346.03	166231	1039.35	164209	1426.15	NULL
AVX	358.329	95232	430.747	94966	294.754	93425	356.873	NULL
Pthread								
pthread_dynamic	4379.84	136956	5028.64	108959	3971.9	91590.1	10081.9	NULL
threadFunc_static2	692.514	75696.6	570.398	67126.5	399.343	62179	603.775	NULL
threadFunc_static3	619.964	77449.8	604.785	65568.2	367.009	63674.1	556.345	NULL
threadFunc_static4	659.673	70853.5	571.151	65874.3	396.344	64675.7	586.388	NULL
threadFunc_static2_SSE	292.912	35695.1	298.33	32338.1	245.531	38305.7	297.98	NULL
threadFunc_static2_AVX	307.225	36927.9	289.107	32383.8	268.504	42047.9	298.182	NULL

单位 (ms)

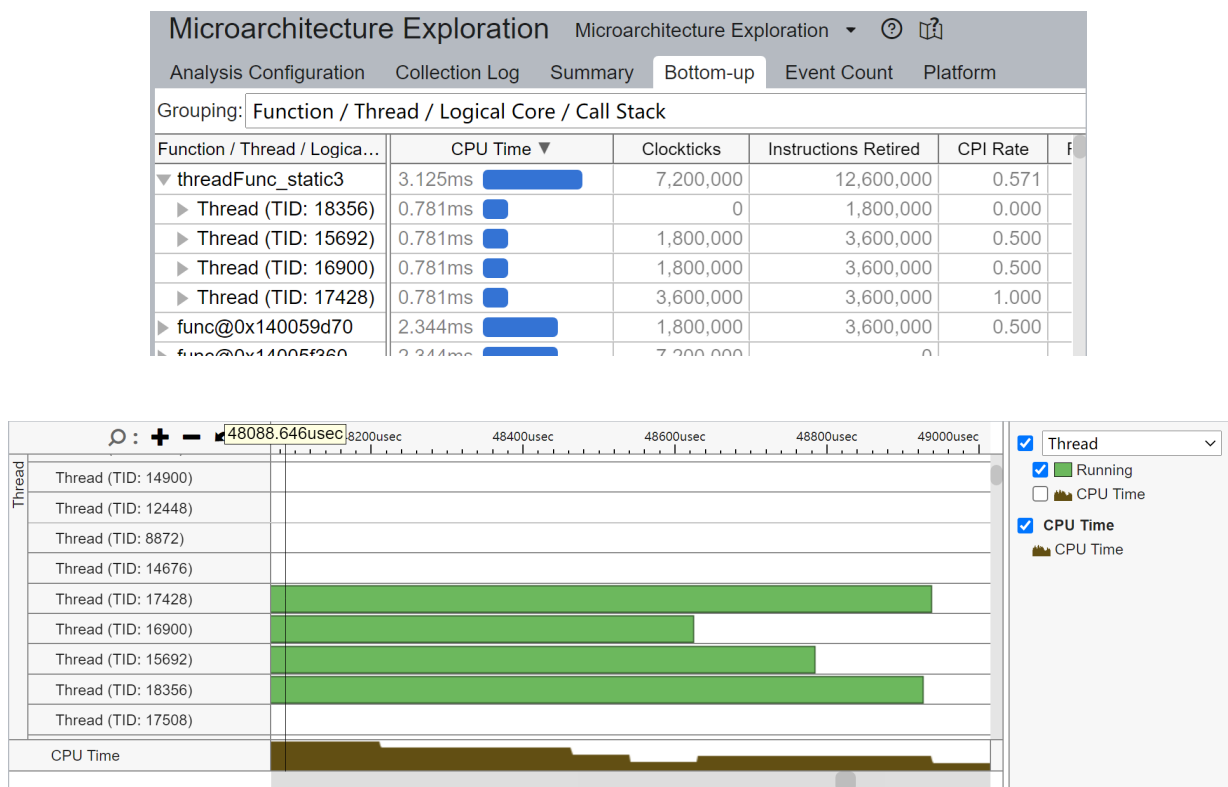
在 N 为 1000 与 5000 的数据规模下，我们比较所有的方案可以发现，对于单线程，创建线程数目大小丝毫不会影响运行时间的长短，而对于多线程，当线程数目小于 8 时，随着线程数量的增加，各版多线程时长大抵呈现下降趋势，运行效率提升，但下降时长有限。当线程数目大于 8 时，比如线程

数目为 12 时，反而时长上升，这里我们考虑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 线程调度任务，而当所需要的线程数量超过 8 个之后，就需要借用线程，这之间会存在着额外的调度开销，因此抵消了多线程对于高斯消元的优化。



3、VTune

接下来，我们探究 Pthread 在 VTune 上的运行效果，我们在此仅以静态线程 + 信号量同步 + 三重循环版本做分析。通过选择 Microarchitecture Exploration，我们选择含有 thread 的 Grouping，比如 Function/thread/Logical Core/Call Stack。这样就可以看到每个线程的 id 及其运行信息。然后我们根据线程 id 找到各线程的运行时间以及对应事件状态。



由于前端各线程十分接近，我们对于尾部进行放大。我们可以看出 16900 线程最先执行结束，为

代码中的 0 号线程, 15692 线程第二个结束为 1 号线程; 18356 线程第三个线程为 2 号线程, 17428 线程最后一个结束为 3 号线程。仔细分析我们发现 3 号线程结束时间与 2 号线程结束时间十分相近, 猜测可能是由于循环划分, 且数据量较小导致划分末端划分不均, 最后线程获得数量较少, 产生误差。

4.4.2 ARM 平台下的 Pthread 优化

1、线程版本

同样, 我们在 ARM 平台上运行四个版本的 Pthread 实验。根据指令集的不同, 我们额外增加 Neon 的串行实验作为对比实验, 与 SSE、AVX 的静态线程 + 信号量同步 Pthread 实验进行比较分析, 下面是我们通过 `<sys/time.h>` 头文件中 `gettimeofday(&start, NULL)` 与 `gettimeofday(&end, NULL)` 测量时间所得到的结果其中, SIMD 向量化处理与 X86 平台相同, 采用的是四路向量化处理, 启用 8 个线程。

下面是我们实验得到的结果:

NUM_THREAD = 8/NUM	10	50	100	500	1000	2000	5000
Serial							
C++	0.003	0.33	2.547	327.682	2687.66	21856	400418
NEON	0.004	0.226	1.701	211.479	1712.28	13517.7	219093
Pthread							
pthread_dynamic	2.715	14.068	27.713	170.384	614.302	3627.31	54473
threadFunc_static2	0.655	1.894	4.184	64.918	440.033	3375.92	59076.1
threadFunc_static3	0.647	1.8	3.788	65.936	432.709	3284.55	55034.4
threadFunc_static4	0.52	1.382	2.634	62.064	422.262	3258.21	59597.2
threadFunc_static2_NEON	0.632	1.851	4.245	52.425	282.634	1951.41	32406.3

单位 (ms)

分析数据, 我们不难发现, 与 X86 平台类似当指令集相同时, 动态线程, 由于创建和销毁线程的次数过多, 而线程创建、销毁的代价是比较大的, 动态线程的时间都远大于静态线程所消耗的时间。

对于不同的指令集, 我们发现当数据规模小的时候, 不使用多线程可能运行速度更快, 说明多线程完成高斯消元节约的时间不及创建线程所浪费的时间。而当数据规模不断增大时, 线程额外开销的副作用不断减小, 多线程的优势得以彰显。而且 Neon 指令在数据量小的时候, 消耗时间反而更长。而数据量大时, Neon 指令的优势得以凸显, 几乎只需消耗一半的时间。这说明指令集对于运行时间长度有较大影响。

2、线程数量

下面我们探讨线程数量对于运行效率产生的影响。我们分别设立 4、8、12 条线程, 探究在不同指令集下、不同版本的运行情况。经过实验, 得下表:

NUM_THREAD	4		8		12	
N	500	3000	500	3000	500	3000
Serial						
C++	320.126	77546.5	327.682	78490.5	327.984	77009.8
NEON	208.217	45590.6	211.479	45975	208.84	48759.9
Pthread						
pthread_dynamic	149.036	20366.5	170.384	11373.3	234.921	8134.33
threadFunc_static2	114.62	19763.3	64.918	10459.2	63.315	7178.82
threadFunc_static3	99.094	19200.1	65.936	10529.4	55.358	7183.74
threadFunc_static4	97.355	19456.2	62.064	10640.8	51.507	7160.48
threadFunc_static2_NEON	69.199	11565.8	52.425	7542.82	46.992	4246.29

单位 (ms)

我们在这里会发现一个有趣的现象，当数据量较少时，随着线程数的增加，消耗时间也增加，当数据量大时，线程数增加，消耗时间反而减小。这次结果也非常形象地说明了建立线程消耗时间与数据量处理时间之间的关系。但我们在 ARM 平台上没有发现，当线程数为 12 时，消耗时间反而增多，也就是当 CPU 所需要的线程数量超过 8 个之后，就需要借用线程，这之间产生额外的调度开销对实验结果的大幅度影响，我们在这里猜测可能是调度开销在 ARM 平台上较小，未能影响最终结果。

4.5 OpenMP 优化

我们使用 OpenMP 这一套支持跨平台共享内存方式的多线程并发的编程 API，实现适合的任务分配算法。在面对负载均衡问题时，我们考虑不同的线程任务划分方式，此处我们分别探讨 static、dynamic、guided 三种方式进行划分，并结合不同的指令集进行探讨对比。

下面我们简单说明不同划分方式的运用场景及其一些基本特点：

1、静态调度 (static)：每次哪些循环由那个线程执行时固定的，编译调试。由于每个线程的任务是固定的，但是可能有的循环任务执行快，有的慢，不能达到最优。假设有 n 次循环迭代， t 个线程，那么给每个线程静态分配大约 n/t 次迭代计算。

2、动态调度 (dynamic)：根据线程的执行快慢，已经完成任务的线程会自动请求新的任务或者任务块，每次领取的任务块是固定的。

3、启发式调度 (guided)：每个任务分配的任务是先大后小，指数下降。当有大量任务需要循环时，刚开始为线程分配大量任务，最后任务不多时，给每个线程少量任务，可以达到线程任务均衡。

通过对于不同平台上 (ARM 平台、x86 平台) 的高斯消元 OpenMP 并行化算法与串行算法比较，以及其他因素的影响 (如：数据规模、编程策略等)，观察高斯消元的性能影响情况。

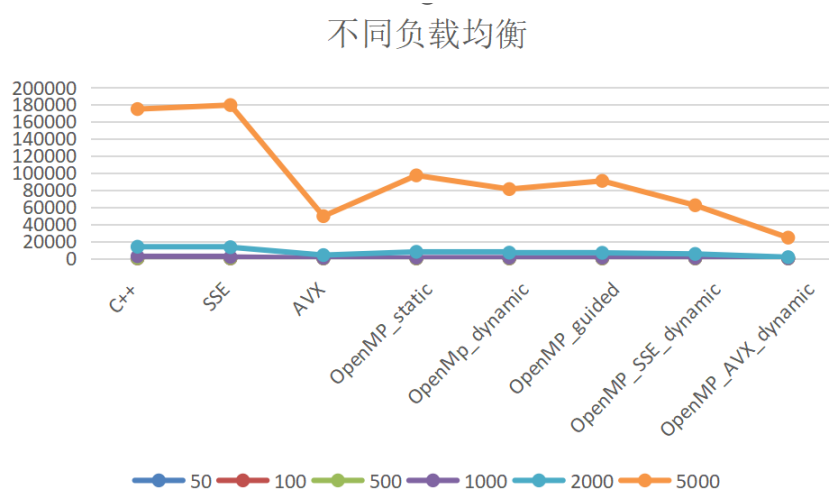
4.5.1 X86 平台下的 OpenMP 优化

1、负载均衡

我们在 X86 平台上运行不同版本的 OpenMP 实验。根据指令集的不同，我们额外增加 SSE、AVX 的 SIMD 实验作为对比实验，与 SSE、AVX 的 OpenMP 实验进行比较分析，下面是我们通过 QueryPerformance 系列函数进行计时所得到的结果。在实验时，SIMD 进行向量化处理的时候，采用的是四路向量化处理，而 OpenMP 多线程优化时，我们总共开启了 4 条线程，其中一条线程负责除法操作，剩余的 3 条线程负责做减法操作，我们此处先设定 chunk_size 为 24，观察面对负载均衡问题时，不同的线程任务划分方式 (static、dynamic、guided) 的性能。

NUM	50	100	500	1000	2000	5000
Serial	NUM_THREAD=4 chunk_size=24					
C++	0.1926	1.221	299.586	2419.14	13601.7	174611
SSE	0.2244	1.6218	237.459	1648.87	13073.8	179286
AVX	0.0793	0.3965	88.8363	483.315	3827.3	49357.4
OpenMP	NUM_THREAD=4 chunk_size=24					
OpenMP_static	1.4819	0.8878	118.7	830.258	7645.89	97050.1
OpenMp_dynamic	0.531	1.8998	129.898	844.899	6705.4	81019.4
OpenMP_guided	0.4573	1.672	142.846	859.161	6495.59	90614.2
OpenMP_SSE_dynamic	0.3371	1.1737	136.819	579.152	5195.32	62112.1
OpenMP_AVX_dynamic	0.2765	0.8499	75.0711	197.74	1427.94	24243.6

单位 (ms)



分析数据, 我们不难发现, 当指令集相同时, 数据量较大时, OpenMp 的并行多线程方法均快于串行算法和简单的 SIMD 并行算法。但是当数据量小时, OpenMP 性能反而不如串行算法, 这也再一次说明了当数据规模不断增大时, 线程额外开销的副作用不断减小, 多线程的优势得以彰显。通过上述图表, 我们也能发现, 看出当数据规模不同时, 各种方案的性能差异。

同时, 对于不同的负载均衡方式, 我们发现三者差别不是特别大。但还是能够看到 OpenMp_dynamic 相较于 OpenMp_static 在数据量较大时更快, 在数据量较小时反而比 OpenMp_static 慢。可以侧面反应动态线程与静态线程之间的关系。

对于不同的指令集, 我们发现这仍然是一个独立的影响因子, 其中 AVX 指令集最快, 然后是 SSE 指令集次之。

2、线程数量

下面我们探讨线程数量对于运行效率产生的影响。我们分别设立 4、8、12 条线程, 同时固定 chunk_size 为 24, 探究在不同指令集下、不同版本的运行情况。经过实验, 得下表:

chunk_size	24					
NUM_THREAD	NUM_THREAD=4		NUM_THREAD=8		NUM_THREAD=12	
NUM	500	5000	500	5000	500	5000
Serial						
C++	299.586	174611	359.931	173122	286.161	173134
SSE	237.459	179286	373.896	180612	243.912	179178
AVX	88.8363	49357.4	179.661	53665.2	77.6085	55495.2
OpenMP						
OpenMP_static	118.7	97050.1	742.372	86125.5	167.455	87073
OpenMP_dynamic	129.898	81019.4	1284.45	62436	498.97	62330.8
OpenMP_guided	142.846	90614.2	186.866	74802.8	489.681	72423.5
OpenMP_SSE_dynamic	136.819	62112.1	99.5103	49448.7	154.236	39535.7
OpenMP_AVX_dynamic	75.0711	24243.6	55.3975	20979.2	55.197	17174.8

单位 (ms)

在 N 为 500 与 5000 的数据规模下, 我们比较所有的方案可以发现, 对于单线程, 创建线程数目大小基本不会影响运行时间的长短。

而对于多线程, 当线程数目小于 8 时, 随着线程数量的增加, 各版多线程时长大抵呈现下降趋势, 运行效率提升, 但下降时长有限。当线程数目大于 8 时, 比如线程数目为 12 时, 有部分 OpenMP 方案 (如: OpenMP_static), 时间不降反升。这里我们考虑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 线程调度任务, 而当所需要的线程数量超过 8 个之后, 就需要借用线程, 这之间会存在着额外的调度开销, 因此抵消了多线程对于高斯消元的优化。但同时我们也观察到其他几种 OpenMP 仍然得到了效率提升, 这里我们考虑是由于 OpenMP 借用线程额外的调度开销较小, 使多线程并行的优点仍然得以显现。

观察表, 我们也不难发现, SSE、AVX 指令集结合 OpenMP 能发挥出良好的效果。使线程增加时, 运行效率得到极大提升。

3、chunk_size

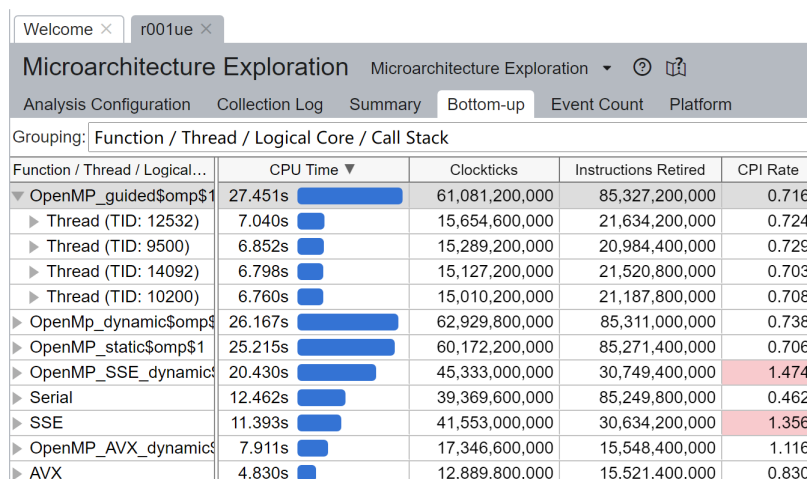
在为线程划分任务的时候, 可以通过设置 chunk_size, 使用循环划分或者块循环划分来均衡负载。我们这里探讨不同 chunk_size, 对最终实验结果的影响。经过实验得到如下结果:

NUM_THREAD	NUM_THREAD=4					
chunk_size	6		12		24	
NUM	500	5000	500	5000	500	5000
Serial						
C++	242.514	168616	259.333	179155	299.586	174611
SSE	221.663	170816	256.859	181770	237.459	179286
AVX	65.8237	50604.3	65.5122	52475.9	88.8363	49357.4
OpenMP						
OpenMP_static	132.788	95233.3	126.066	101491	118.7	97050.1
OpenMP_dynamic	113.248	82261.2	109.881	81375.8	129.898	81019.4
OpenMP_guided	124.471	96529.6	140.167	93422.5	142.846	90614.2
OpenMP_SSE_dynamic	84.0608	61212.5	77.3432	64777.8	136.819	62112.1
OpenMP_AVX_dynamic	36.0869	22555.6	31.9465	25630.9	75.0711	24243.6

分析数据, 我们可以看到 chunk_size 对于实验结果的影响不大, 经计算 chunk_size 波动范围为 6, 相差误差最大在 4% 至 5% 左右。

4、VTune

接下来,我们探究 OpenMP 在 VTune 上的运行效果,我们在此仅以 4 线程, chunk_size 为 24 做分析。通过选择 Microarchitecture Exploration, 我们选择含有 thread 的 Grouping, 比如 Function/thread/Logical Core/Call Stack。这样就可以看到每个线程的 id 及其运行信息。然后我们根据线程 id 找到各线程的运行时间以及对应事件状态。



Function / Thread / Logical...	CPU Time	Clockticks	Instructions Retired	CPI Rate
OpenMP_guided\$omp\$1	27.451s	61,081,200,000	85,327,200,000	0.716
Thread (TID: 12532)	7.040s	15,654,600,000	21,634,200,000	0.724
Thread (TID: 9500)	6.852s	15,289,200,000	20,984,400,000	0.729
Thread (TID: 14092)	6.798s	15,127,200,000	21,520,800,000	0.703
Thread (TID: 10200)	6.760s	15,010,200,000	21,187,800,000	0.708
OpenMp_dynamic\$omp\$1	26.167s	62,929,800,000	85,311,000,000	0.738
OpenMP_static\$omp\$1	25.215s	60,172,200,000	85,271,400,000	0.706
OpenMP_SSE_dynamic	20.430s	45,333,000,000	30,749,400,000	1.474
Serial	12.462s	39,369,600,000	85,249,800,000	0.462
SSE	11.393s	41,553,000,000	30,634,200,000	1.356
OpenMP_AVX_dynamic	7.911s	17,346,600,000	15,548,400,000	1.116
AVX	4.830s	12,889,800,000	15,521,400,000	0.830

比较分析各个方法的 CPI, 发现 OpenMP 的不同划分方式 CPI 相接近, 侧面也论证了我们之前运用 QueryPerformance 测量时间所得的数据相接近, 同时 OpenMP 的多线程执行相较于 SSE、AVX 的 CPI 更小, 说明多线程运行速度更快, 效率更高。

4.5.2 ARM 平台下的 OPenMP 优化

1、负载均衡

同样,我们在 ARM 平台上运行不同版本的 OpenMP 实验。根据指令集的不同,我们额外增加 SSE、AVX 的 SIMD 实验作为对比实验,与 SSE、AVX 的 OpenMP 实验进行比较分析,下面是我们通过 <sys/time.h> 头文件中 gettimeofday(&start,NULL) 与 gettimeofday(&end,NULL) 所得到的结果。在实验时, SIMD 进行向量化处理的时候,采用的是四路向量化处理,而 OpenMP 多线程优化时,我们总共开启了 4 条线程,此处设定 chunk_size 为 24,观察面对负载均衡问题时,不同的线程任务划分方式 (static、dynamic、guided) 的性能。

下面是我们实验得到的结果:

NUM	50	200	1000	4000
Serial	NUM_THREAD=4		chunk_size=24	
C++	0.682	42.965	2899.48	229468
Neon	0.467	27.914	1787.05	151209
OpenMP	NUM_THREAD=4		chunk_size=24	
OpenMP_static	0.684	43.016	2906.33	229071
OpenMp_dynamic	0.681	42.957	2896.69	217868
OpenMP_guided	0.68	42.975	2945.37	225651
OpenMP_NEON_dynamic	0.461	27.917	1808.13	141890

单位 (ms)

分析数据,我们不难发现,与 X86 平台有点不同,在 ARM 平台上时, OpenMP 几乎没有怎么提升,和直接串行的几乎没有差异。但运用 Neon 指令,我们可以看到其还是得到了大幅度的性能提升,我们猜测 ARM 平台底层可能对于 OpenMP 没有很好的支持。

2、线程数量

下面我们探讨线程数量对于运行效率产生的影响。我们分别设立 4、8 条线程，探究在不同指令集下、不同版本的运行情况。经过实验，得下表：

chunk_size	24			
NUM_THREAD	NUM_THREAD=4		NUM_THREAD=8	
NUM	200	4000	200	4000
Serial				
C++	42.965	229468	42.475	224378
Neon	27.914	151209	27.9	119008
OpenMP				
OpenMP_static	43.016	229071	42.511	213667
OpenMp_dynamic	42.957	217868	42.473	215805
OpenMP_guided	42.975	225651	42.449	214593
OpenMP_NEON_dynamic	27.917	141890	27.895	132048

单位 (ms)

我们在这里会发现一个有趣的现象，在 N 为 200 与 4000 的数据规模下，我们比较所有的方案可以发现，对于单线程，创建线程数目大小基本不会影响运行时间的长短。OpenMP 与串行实验结果相差不大，这说明了 OpenMP 作为一套支持跨平台共享内存方式的多线程并发的编程 API，其效果在不同平台上运行所得结果类似但仍存在不同。

3、chunk_size

下面我们再来看看不同 chunk_size 中

NUM_THREAD	NUM_THREAD=4			
chunk_size	12		24	
NUM	200	4000	200	4000
Serial				
C++	42.965	229468	42.442	217722
Neon	27.914	151209	27.863	128396
OpenMP				
OpenMP_static	43.016	229071	42.529	226936
OpenMp_dynamic	42.957	217868	42.435	220900
OpenMP_guided	42.975	225651	42.444	225032
OpenMP_NEON_dynamic	27.917	141890	27.885	125905

分析数据，我们可以看到 chunk_size 对于实验结果的影响不大。

4.6 MPI 优化

我们在进行 MPI 优化的时候，选用金山云作为 X86 平台进行实验。

在设计实现适合的任务分配算法，分析其性能后。我们在金山云 (X86) 平台上编程实现、进行实验、测试不同问题的规模、不同节点数/线程数下的算法性能（串行和并行对比），讨论一些基本的算法/编程策略对性能的影响。

在此基础上，我们额外对多线程 (OpenMP) 单独实现结果、SIMD 单独实现结果以及两者结合的实验结果进行了分析比较。同时，我们探讨了 MPI 并行化的不同算法策略（如块划分、循环划分等不同任务划分方法）及其复杂度的分析，同时实现了将对应方法迁移到 ARM 平台上并观察其不同平台的最终结果。

在进行 MPI 多进程的实验设计时，我们采取类似多线程的设计思想，使用单个进程来分发任务，再在外层循环中，每个线程都要判断当前做除法的行是否是自己分配的任务，当这一行的除法操作完成之后，需要将除法的结果告知给其他所有进程。

因为 MPI 是从进程方面考虑并行，因此在后续优化过程中，我们还可以考虑在各个单个进程内使用其他并行优化措施，比如运用之前的 OpenMP、SIMD，可使并行的优化效果得到叠加提升。

4.6.1 金山云 (X86) 平台下的 MPI 优化

1、MPI 多进程优化

我们根据实验目的结合实验设计思想，探讨在不同问题规模下，串行算法与 MPI 多进程两者的实际处理效率。其中 MPI 多进程任务划分方式我们这里先采取块划分。我们可以得到不同问题规模下的表现如下所示。

N	100	500	1000	1500	2000	3000
Serial	1.69382	205.401	1678.79	5330.93	13852.4	43486.8
MPI	2.49907	57.6582	395.744	1385.96	2995.76	10023.2

分析表中数据，我们不难看出单独采取 MPI，相较于串行算法速度最多优化到约为原先的 4.3 倍后，便逐渐趋于收敛，明显难以达到由于 MPI 开启 8 进程，速度变为原来 8 倍，时间缩短为原来的 1/8 的理论效果。我们猜测可能是由于存在通信开销、一些无法进行并行的代码以及未能充分利用开启的 8 个多进程，导致最后难以到达理论值。

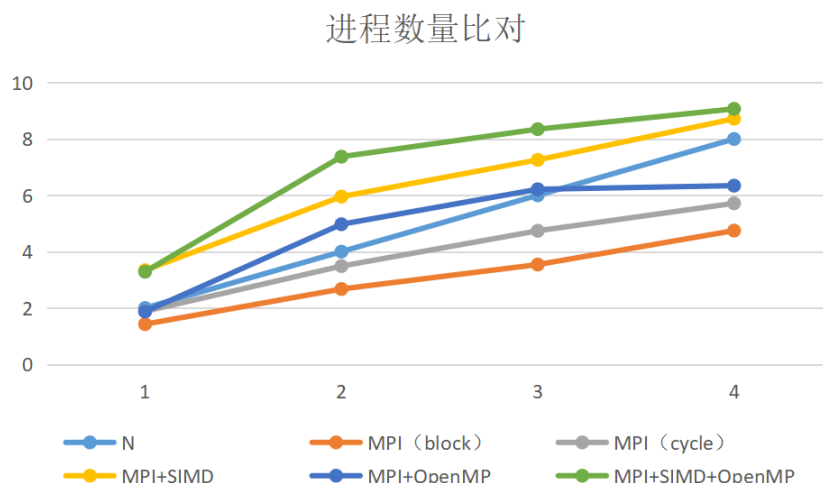
2、不同节点数/进程数

我们继续探究不同进程数量的 MPI 优化效果。此处我们选定问题规模为 1000，线程数采取常用的 8 线程，向量化路数也采取常用的四路向量化处理。我们申请与进程数相同的节点，并且在每个节点只开启一个进程。分别运用 2、4、6、8 个进程测量串行算法、MPI 多进程、MPI+SIMD、MPI+OPenMP、MPI+SIMD+OPenMP 等各类优化算法，实验结果下图所示。

下表为实际结果得到的时间：

N=processes	Serial	MPI (block)	MPI (cycle)	MPI+SIMD	MPI+OpenMP	MPI+SIMD+OpenMP
2	1391.21	976.245	739.012	416.492	743.624	423.254
4	1392.34	520.790	399.301	233.87	280.03	188.880
6	1390.97	392.845	293.478	191.654	223.832	166.623
8	1391.61	293.178	243.443	159.635	219.424	153.452

下图为各类算法相较于串行算法所得比例比较所得



分析图中数据我们不难看出，随着进程数量的提升，各种算法大体成增长趋势，说明进程数量越多，运行效率越高效。

3、MPI、SIMD、OPenMP 三者的组合优化

接下来根据实验设计思想，考虑在不同问题规模下，对如下优化算法的耗时长短测量。

测量 MPI 多进程、MPI+SIMD、MPI+OPenMP、MPI+SIMD+OPenMP

我们对于 MPI 多进程优化时，采用 8 个进程；OPenMP 多线程优化时，开启了 8 条线程；SIMD 进行向量化处理时，采用四路向量化处理，其中一条线程负责除法操作，剩余的 3 条线程负责做减法操作。此处以 MPI 作为基准进行比较分析。最后我们可以得到不同问题规模下的表现如下所示。

N	100	500	1000	1500	2000	3000
MPI	2.49907	57.6582	395.744	1385.96	2995.76	10023.2
MPI+SIMD	6.35532	51.0937	216.018	597.581	1331.45	4078.69
MPI+OpenMP	6.28232	40.8236	181.436	595.953	1322.96	3889.21
MPI+SIMD+OpenMP	6.19821	52.9893	176.865	410.736	865.764	2478.11

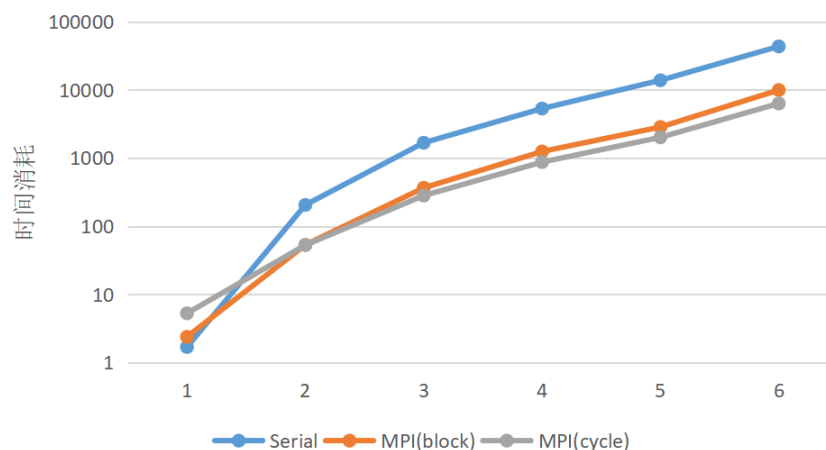
分析表中数据，我们可以看到当同时使用 MPI+SIMD 时，我们仍然难以做到理论中 SIMD 采用四路向量化，从而提升四倍速度的结果，实验结果显示可能只能达到 MPI 速度的 2.5 倍左右。这与我们之前实验结果类似，也就是存在代码使得 SIMD 无法进行向量化。同时使用 MPI+OPenMP 时，结果也类似，理论上由于存在 8 条线程，会提升 8 倍的速度，但是实际上却只有提升了 2.57 倍左右，说明多线程分配仍然存在代码可能难以实现多线程的分配。我们最后分析当同时采取三种优化方式时，也就是将 MPI+SIMD+OPenMP 融于一体，可以显著发现速度得到大幅度提升，但是当数据规模为当前最大时，其相较于 MPI 的提升速度约为 4.04，绝对不是简单的两种分开提速的实验加速倍速的乘积，因此我们猜测这里可能存在 SIMD 与 OPenMP 之间的相互牵连或者也可能由于实验数据量不够大而导致的较大实验误差。

4、不同任务划分

下面我们进行 MPI 并行化下的不同算法策略，这里我们主要探讨循环划分和块划分之间的差异。考虑到高斯消元，完成前面行的消元较之于完成后面行的消元更加容易，我们推测块划分容易导致最后计算量划分不均，而循环划分就能很好的规避这个问题，让所有的进程都得到几乎相同的计算量，从而取得更优的效果。实验结果也可以证实这一点。

N	100	500	1000	1500	2000	3000
Serial	1.69382	205.401	1678.79	5330.93	13852.4	43486.8
MPI(block)	2.38807	53.3581	365.856	1248.94	2855.66	9959.27
MPI(cycle)	5.2821	52.9547	281.696	874.172	2015.98	6329.16

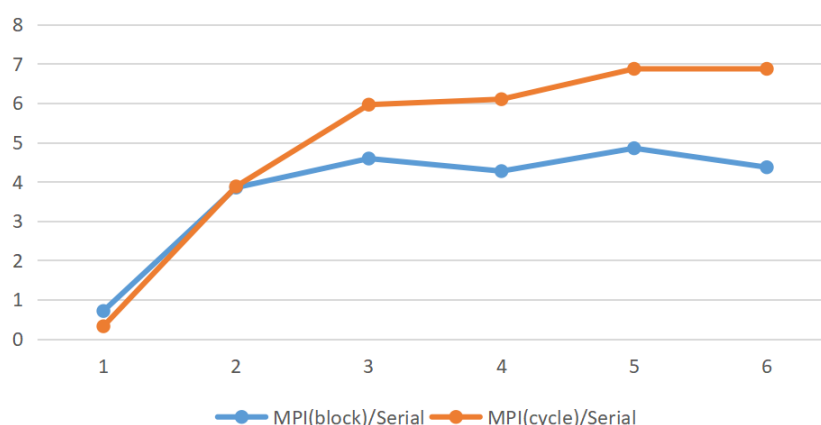
不同任务划分方式



从表与图像中可以看出，随着任务规模的增加，块划分由原先快于循环划分而最终不及循环划分，说明我们之前推测的正确性。再用 MPI(block) 的消耗时间与 MPI(cycle) 的消耗时间分别除 Serial 的消耗时间

N	100	500	1000	1500	2000	3000
MPI(block)/Serial	0.709	3.849	4.589	4.268	4.851	4.366
MPI(cycle)/Serial	0.321	3.879	5.960	6.098	6.871	6.871

MPI较Serial的效率提升



同时我们也可以看到相较于串行算法，MPI 采用循环划分方式已经提升了速率将近 6.8 倍，逼近 8 倍的理论值。

4.6.2 ARM 平台下的 MPI 优化

ARM 平台下所得结果大体与 X86 所得实验结果类似。但由于 ARM 平台只有两个节点，因此我们只开启两个进程。

1、MPI 多进程优化

我们根据实验目的结合实验设计思想，探讨在不同问题规模下，串行算法与 MPI 多进程两者的实际处理效率。我们可以得到不同问题规模下的表现如下所示。

N	100	500	1000	1500	2000	3000
Serial	0.40952	43.1703	348.694	1178.96	2931.79	9927.63
MPI	0.95486	39.6271	278.517	1012.16	2001.01	7001.18

分析表中数据，我们不难看出单独采取 MPI，相较于串行算法时间最多优化到约为原先的 1.4 倍后便逐渐趋于收敛，明显远远难以达到由于 MPI 开启 2 进程，速度变为原来 2 倍，时间缩短为原来的 1/2 的理论效果。我们猜测可能是由于存在通信开销以及一些无法进行并行的代码，导致难以到达理论值。

2、MPI、SIMD、OPenMP 三者的组合优化

接下来根据实验设计思想，考虑在不同问题规模下，对如下优化算法的耗时长短测量。

测量 MPI 多进程、MPI+SIMD、MPI+OPenMP、MPI+SIMD+OPenMP

我们对于 MPI 多进程优化时，采用 8 个进程；OPenMP 多线程优化时，开启了 8 条线程；SIMD 进行向量化处理时，采用四路向量化处理，其中一条线程负责除法操作，剩余的 3 条线程负责做减法操作。此处以 MPI 作为基准进行比较分析。最后我们可以得到不同问题规模下的表现如下所示。

N	100	500	1000	1500	2000	3000
MPI	0.95486	39.6271	278.517	1012.16	2001.01	7001.18
MPI_SIMD	3.5786	61.5325	259.29	697.563	1437.86	4820.32
MPI_OMP	3.9834	65.8972	219.95	478.394	892.433	2768.49
MPI_OMP_SIMD	4.06478	57.0161	163.813	334.992	603.52	1379.62

分析表中数据，我们可以看到当同时使用 MPI+SIMD 时，我们仍然难以做到理论中 SIMD 采用四路向量化，从而提升四倍速度的结果，实验结果显示可能只能达到 MPI 速度的 1.45 倍左右。这与我们之前实验结果类似，也就是存在代码使得 SIMD 无法进行向量化。同时使用 MPI+OPenMP 时，结果也类似，理论上由于存在 8 条线程，会提升 8 倍的速度，但是实际上却只有提升了 2.53 倍左右，说明多线程分配仍然存在代码可能难以实现多线程的分配。

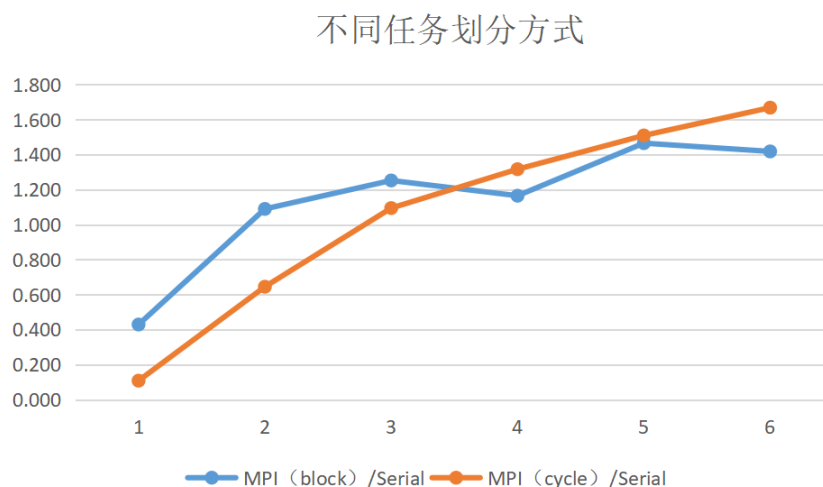
3、不同任务划分

下面我们进行 MPI 并行化下的不同算法策略，这里我们主要探讨循环划分和块划分之间的差异。考虑到高斯消元，完成前面行的消元较之于完成后面行的消元更加容易，我们推测块划分容易导致最后计算量划分不均，而循环划分就能很好的规避这个问题，让所有的进程都得到几乎相同的计算量，从而取得更优的效果。实验结果也可以证实这一点。

N	100	500	1000	1500	2000	3000
Serial	0.40952	43.1703	348.694	1178.96	2931.79	9927.63
MPI (block)	0.95486	39.6271	278.517	1012.16	2001.01	7001.18
MPI (cycle)	3.78965	66.9517	318.686	895.633	1942.73	5951.89

下面是与串行算法相比较分析后的数值结果。

N	100	500	1000	1500	2000	3000
MPI (block) /Serial	0.429	1.089	1.252	1.165	1.465	1.418
MPI (cycle) /Serial	0.108	0.645	1.094	1.316	1.509	1.668



从表与图像中可以看出，随着任务规模的增加，块划分由原先快于循环划分而最终不及循环划分，说明我们之前推测的正确性。同时我们也可以看到相较于串行算法，MPI 采用循环划分方式已经提升了速率将近 1.67 倍，逼近进程为 2 而产生 2 倍的理论值。

5 消元子模式的特殊高斯消元

我们根据特殊高斯消元的问题分析，可以整理出特殊高斯消元算法的一般流程为：

1. 将文件中的消元子和被消元行读入内存
2. 对每个被消元行，检查其首项，如有对应消元子，则将其减去（异或）对应消元子，重复此过程直至其变为空行（全 0 向量）或首项在范围内但无对应消元子或该行，若为情况 2 则该行计算完成
3. 如果某个被消元行变为空行，则将其丢弃，不再参与后续消去计算；如其首项被覆盖，但没有对应消元子，则将它“升格”为消元子，在后续消去计算中将以消元子身份而不再以被消元行的身份参与
4. 重复第二步和第三步的过程，直至所有被消元行都处理完毕，此时消元子和被消元行共同组成结果矩阵——可能存在很多空行。

根据问题分析时给出的特殊高斯消元算法思路中的伪代码，我们尝试采用不同的优化策略如：SIMD、Pthread、OpenMP、MPI 等，对于问题进行优化。

对于特殊高斯消元，前文我们已经提到运用位向量的思想，这样我们可以将消元操作变为位向量的异或操作，算法实现简单。

5.1 SIMD 优化

在 SIMD 优化方面，我们对于特殊高斯消元问题的分析与普通高斯消元类似。我们观察不同问题规模与 cache 大小不同关系时的程序性能变化，通过分析 cache 对性能的影响，揭示问题的内在本质。同时，本次实验也给出了是否对齐等因素，使实验结果更加全面。

由于 cache 大小已知，我们可以根据给出的列数建立消元子、按照给出的被消元行数建立消元行，并可以大致估算出每一级缓存大致可填充的数量，采用 `aligned_alloc()` 函数来完成地址是否对齐。此外，由于之前分析，可实现并行优化的最有可能的是异或消元部分。因此，我们对该部分按之前所说进行向量化处理。由此，即可编写得到代码，在经过实验测试，即可得到相应的实验结果。

5.1.1 X86 平台下的 SIMD 优化

1、非对齐

我们这里给出 SSE、AVX 下的特殊高斯所消耗的时间，并与串行算法做分析比较，使每个实例内的数组未完成字节地址对齐，也就是不采用 `aligned_alloc()` 函数。我们可以得到前 7 组数据的实验数据，如下表所示数据，计时单位为毫秒：

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Serial	0.0623	0.703	1.23	40	323	7801	107638
SSE	0.0602	0.697	1.12	39.6	302	7681	105572
AVX	0.0628	0.725	1.15	41.7	368	8192	108251

观察表数据可以看到，当数据规模小时，在 SIMD 优化下，SSE、AVX 指令集能够对于特殊高斯做适当优化，但是优化程度极其有限，不明显。

2、对齐

我们这里给出 SSE、AVX 下的特殊高斯所消耗的时间，并与串行算法做分析比较，使每个实例内的数组完成字节地址对齐，也就是采用 `aligned_alloc()` 函数。我们可以得到前 7 组数据的实验数据，如下表所示数据，计时单位为毫秒：

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Serial	0.0623	0.693	1.22	40	329	7803	108668
SSE	0.08	0.728	1.19	35.9	317	7469	103453
AVX	0.0647	0.739	1.03	41.8	355	8346	112411

观察对齐后的实验结果，我们发现这与对齐前相差的实验结果并不大。甚至一度出现 AVX 指令集的消耗时间大于并行所耗费的时间，这说明我们刚刚在非对齐得到的结论仍然成立，也就是说高斯消元不使用 SIMD 并行优化策略。至于出现 AVX 耗时长于串行算法，我们猜测可能是由于使用了对齐函数，导致运行时消耗时长，最终导致上述实验结果的产生。

5.1.2 ARM 平台下的 SIMD 优化

ARM 平台下的实验与 X86 平台相类似，我们运用 `perf` 对 NEON 指令集所消耗的时间进行测算，比较其与串行算法所消耗的时间长短，并探究是否对齐等因素对实验结果的影响。

1、非对齐

运用 NEON 指令集在 ARM 平台下对特殊高斯进行非对齐的 SIMD 优化操作，发现两者相差不大。说明非对齐情况下，SIMD 对于特殊高斯消元几乎没有优化效果。猜测很有可能这和 ARM 平台下的 CPU 读取效率有关，毕竟 CPU 不是一字节一字节读取的，而实际上是按照块来读取的，块的

大小可以为 2, 4, 8, 16。块的大小也称为内存读取粒度。举个例子, 假设 CPU 没有内存对齐, 要读取一个 4 字节的数据到一个寄存器中, (假设读取粒度为 4), 则会出现两种情况

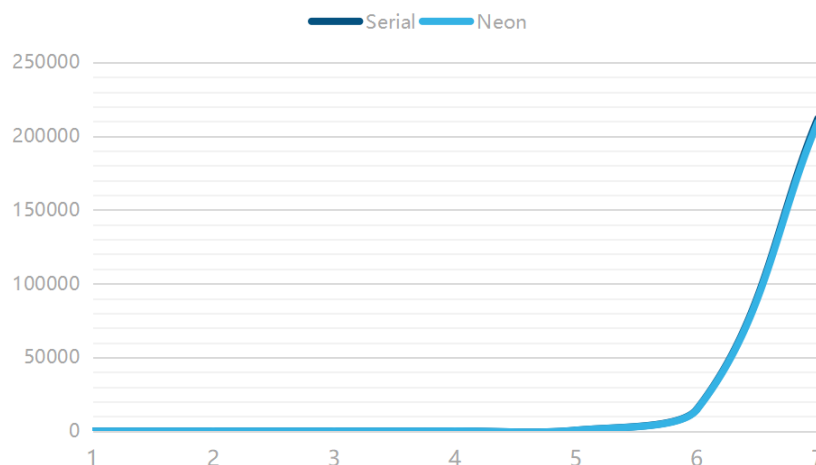
- 1、数据的开始在 CPU 读取的 0 字节处, 这刚 CPU 一次就你能够读取完毕
- 2、数据的开始没在 0 字节处, 假设在 1 字节处吧, CPU 要先将 0 3 字节读取出来, 在读取 4 7 字节的内容。然后将 0 3 字节里的 0 字节丢弃, 将 4 7 字节里的 5, 6, 7 字节的数据丢弃。然后组合 1,2,3,4 的数据。

由此可以看出, CPU 读取的效率不是很高, 可以说比较繁琐。而由于数据量可能仍然不是很大, 很难区分 SIMD 优化对于整体的微弱提升, 因此在非对齐状态下 CPU 因为非对齐而消耗的时间占了绝大部分, 导致 SIMD 优化聊胜于无。但如果有内存对齐的话: 由于每一个数据都是对齐好的, CPU 可以一次就能够将数据读取完成, 虽然会有一些内存碎片, 但从整个内存的大小来说, 都不算什么, 可以说是用空间换取时间的做法。而降低总体时间后, SIMD 的轻微的优化效果便将得以显现。我们可以从下面内存对齐的时候的分析得到理论结果。

2、对齐

我们在 ARM 平台下同样进行对齐操作, 来测算 Neon 指令集与串行算法之间的比较。我们可以得到前 7 组数据的实验数据, 如下表所示数据, 计时单位为毫秒:

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Serial	0.0349	0.582	1.02	69.2	603	14864	211897
Neon	0.0363	0.581	1.04	68.9	595	14590	209403



通过对实验结果的图与表的分析, 表中数据几乎相同, 而图中显示的两条线也高度重合, 我们可以看到 SIMD 几乎对于特殊高斯消元没有优化提升, 与 X86 平台的结果类似。Neon 指令集的使用对于程序整体性能几乎只有及其微弱的提升。

5.2 Pthread 优化

同 SIMD 优化模式相近, 我们尝试实现并行优化的最有可能的部分——异或消元部分。我们对于特殊高斯消元进行多线程处理, 与未处理的特殊高斯消元算法进行性能比较。由此通过分析实验结果来判断 Pthread 优化是否适用于特殊高斯消元。

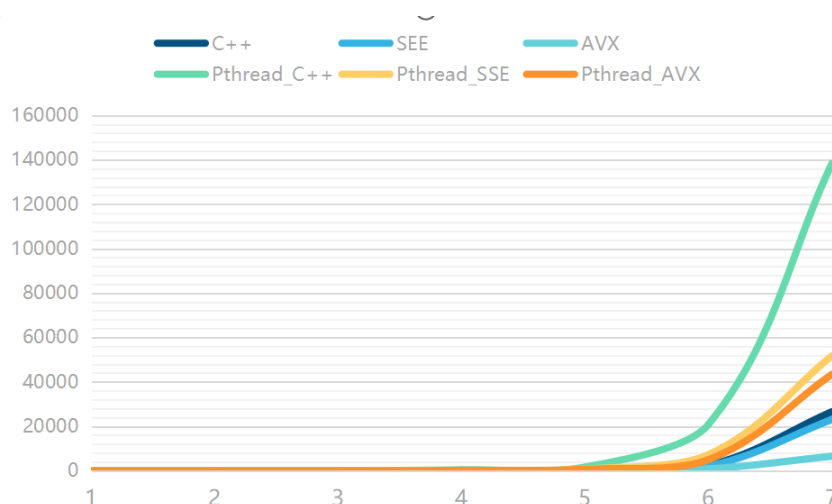
实验平台在这里我们仍然采取的是 X86 平台，运行不同版本的 Pthread 实验。根据指令集的不同，我们额外增加 SSE、AVX 的串行实验作为对比实验，与 SSE、AVX 的静态线程 + 信号量同步 Pthread 实验进行比较分析。以此来得出较为全面的实验结果。并且我们分别开设 2、4、8 条线程，来探究多线程对于特殊高斯消元产生的效果。

5.2.1 线程版本

我们在 X86 平台上运行两个版本的特殊高斯 Pthread 实验。根据指令集的不同，我们额外增加 SSE、AVX 的串行实验作为对比实验，与 SSE、AVX 的静态线程 + 信号量同步 Pthread 实验进行比较分析，下面是我们通过 QueryPerformance 系列函数进行计时所得到的结果。在之前实验时，发现 SIMD 未能达到预期的效果，因此我们决定先不加入 SIMD 优化，若多线程优化效果显著再决定是否使用 SIMD 做进一步的优化。

我们在用 Pthread 多线程优化时，总共先开启了 4 条线程，并且使用内存对齐，与一般的串行算法作比较。最终实验得到的结果如下：

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Serial							
C++	0.01229	0.69126	0.83245	23.5973	146.973	2473.52	26589.9
SEE	0.00283	0.52943	0.67585	15.8325	101.305	2107.94	23051.7
AVX	0.00135	0.13968	0.27853	5.6893	29.8321	869.364	6548.92
Pthread	THREAD=4						
Pthread_C++	0.63482	13.8974	15.1438	423.547	1649.83	20868.8	138333
Pthread_SSE	0.29664	7.07601	7.38721	162.902	772.894	7271.35	51726.4
Pthread_AVX	0.32334	6.93455	7.11788	156.095	648.505	4995.45	43410.8



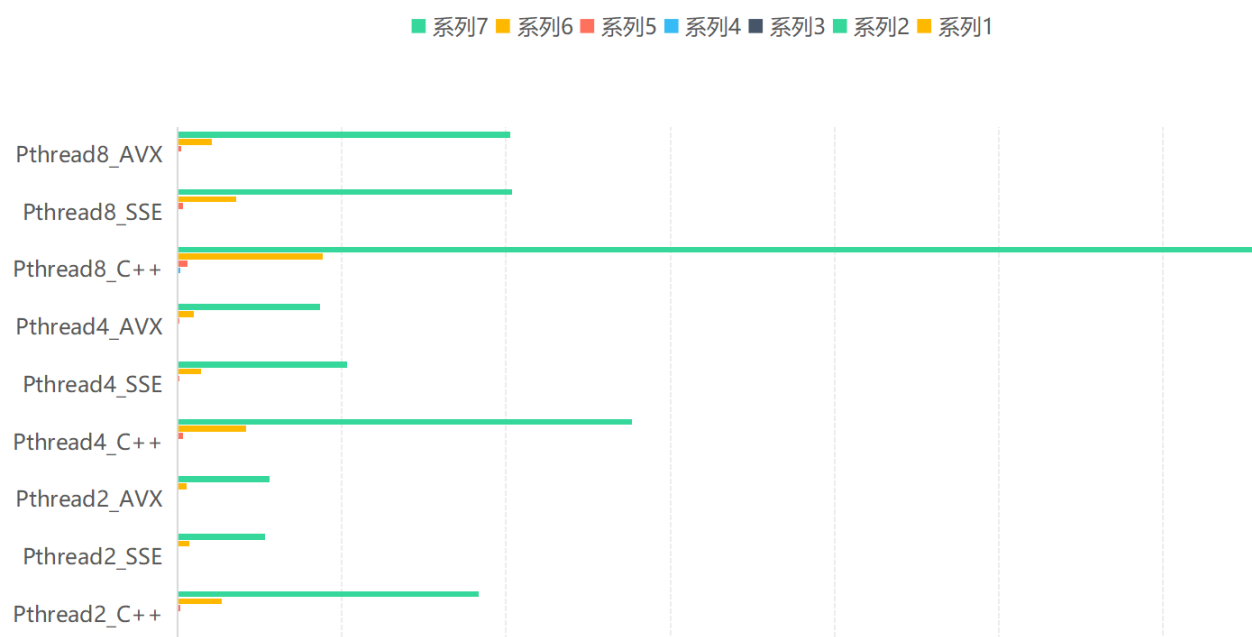
根据上述得到的实验结果数据。可以看到，在各个数据集下，多线程算法的运行时间均远大于串行算法的运行时间，可见线程间同步开销非常大。虽然 SSE、AVX 仍然保留着指令集优化的优势，但是从不同指令集方面我们也更能从中证明：多线程不适合特殊高斯的性能优化，甚至还可能消耗更多的时间。我们接下来再尝试使用不同的线程数量，以此估算是否可能是线程数量选取不当所导致多线程无法对特殊高斯进行优化。

5.2.2 线程数量

下面我们探讨线程数量对于运行效率产生的影响。我们分别设立 2、4、8 条线程，探究在不同指令集下、不同版本的运行情况。经过实验，得到下面的数据：

Pthread	THREAD=2						
Pthread2_C++	0.42345	9.58554	9.23284	274.253	1090.25	10676.6	99780.7
Pthread2_SSE	0.18168	4.83693	4.20596	92.072	517.683	3998.30	33568.3
Pthread2_AVX	0.17531	3.94468	4.13773	99.439	413.233	3640.39	27229.1
Pthread	THREAD=4						
Pthread4_C++	0.63482	13.8974	15.1438	423.547	1649.83	20868.8	138333
Pthread4_SSE	0.29664	7.07601	7.38721	162.902	772.894	7271.35	51726.4
Pthread4_AVX	0.32334	6.93455	7.11788	156.095	648.505	4995.45	43410.8
Pthread	THREAD=8						
Pthread8_C++	1.56164	29.11930	37.5122	987.029	3471.99	49281.3	276942.6
Pthread8_SSE	0.59138	16.00959	17.9835	313.055	1846.19	13936.2	109588.8
Pthread8_AVX	0.59502	17.20582	12.9314	294.100	1344.48	11709.39	89736.9

我们根据表中数据，绘制出相应柱形图以便更易比较大小，其中图中不同系列即分别代表不同的测试的数据集。



观察多线程算法不同线程数目下程序运行时间，我们发现，随着线程数目的增多，程序运行时间并没有像我们期望的那样减少，反而是随着线程数目的增加而增长，我们猜测其中可能的原因是线程数目越多同步开销越大，但是对于代码的优化加速效果却没有如此显著。这里说明多线程并行并不能有效地提升特殊高斯消元的运行效率。

不过反过来我们思考分析一下为什么多线程无法优化高斯消元的性能呢？有一种很大的可能就是未采取合适的多线程划分方式，也就是矩阵水平划分与垂直划分划分方式不当。我们开了许多线程，但是当真正开始工作时，部分线程只执行了极少部分的任务，最终导致许多线程未完成最终消元

时，即已经处于闲置状态。最终导致线程之间的同步开销，远大于多线程实际对于特殊高斯消元算法的性能提升。当然，也不排除是由于数据规模太小所导致线程优化效果不明显。

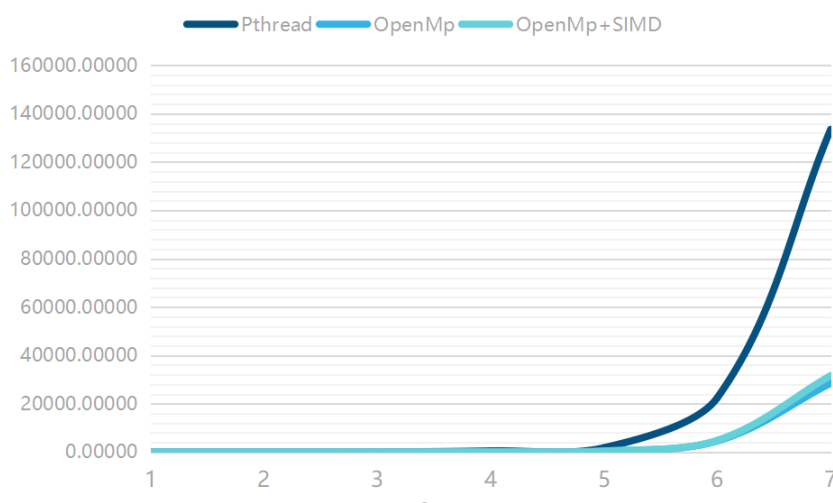
5.3 OpenMP 优化

5.3.1 X86 平台下的 OpenMP 优化

使用的 X86 平台并未发生改变，仍然是与 Pthread 相同的平台。这次，我们在 X86 平台下探究 OpenMP、OpenMP+SIMD 对于特殊高斯消元所产生的优化效果。对于线程的选取，我们选用 4 个线程，基准选取为相同四个线程的普通 Pthread 算法。

经过实验，我们可以得到以下实验结果：

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Pthread	1.34444	28.5286	31.9185	519.639	1937.77	21002.7	128761
OpenMp	0.09046	2.98015	4.27779	104.893	410.600	4814.24	27465.7
OpenMp+SIMD	0.04582	4.13583	3.85492	104.396	410.109	4959.77	29697.9



分析上述实验结果的数据，我们可以清晰地发现采用 OpenMP 在相同的线程数量下所消耗的时间更少，这就直接说明了 OpenMP 对于实验特殊高斯消元具有很好的优化作用。继续仔细观察表中数据与图显示的曲线，我们可以发现无论是在何种数据规模下，这种优化是始终存在的，而且即使 OpenMP 添加了 SIMD 优化策略，其所得到的消耗时间仍然与未添加前相差不大。也印证了我们之前 SIMD 中所得到的结论，就是 SIMD 对于特殊高斯消元算法的优化性能及其微弱，可以忽略不计。

5.3.2 ARM 平台下的 OpenMP 优化

同样，我们在 ARM 平台下进行 OpenMP 优化的探究。采用的方法依然是使用 OpenMP、OpenMP+SIMD 对于特殊高斯消元进行优化。对于线程的选取，我们选用 4 个线程，基准选取为相同四个线程的普通 Pthread 算法。

经过实验，我们可以得到以下实验结果：

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Pthread	0.64336	16.0582	14.2171	386.081	1473.48	20013.2	133481
OpenMp	0.08728	3.27836	3.33976	83.3949	331.409	3742.45	49034.3
OpenMp+SIMD	0.08549	3.32858	3.34704	89.4574	339.818	3845.77	52993.2

分析实验结果所得图与表，我们可以发现与 X86 平台相同的情况，这也更进一步证明了我们在 X86 平台上所得到的结论，也就是 OPenMP 对于特殊高斯消元确实有一定的优化作用，而 SIMD 的加持对于优化作用并不显著。

5.3.3 chunk_size 与 thread

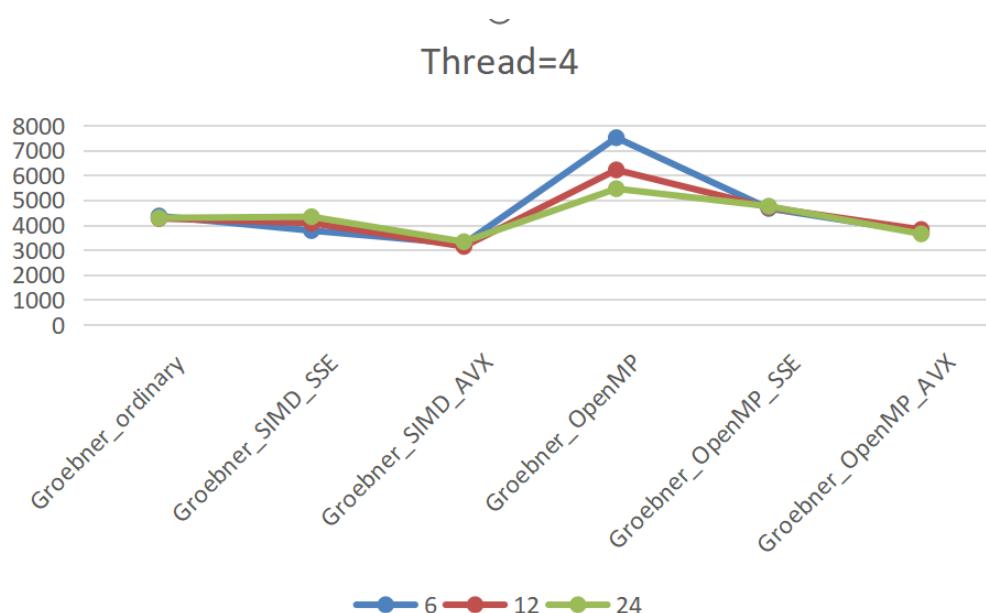
后续由于鲲鹏服务器连接不稳定等诸多因素，我们这里只进行 X86 平台的不同版本 OpenMP 实验分析。此处主要观察 thread 数量与 chunk_size 对于特殊高斯消元的影响。为了使实验更具有普遍意义，我们在这里采取了不同指令集进行分析比较。

根据指令集的不同，我们额外增加 SSE、AVX 的 SIMD 实验作为对比实验，与 SSE、AVX 的 OpenMP 实验进行比较分析，下面是我们通过 QueryPerformance 系列函数进行计时所得到的结果。在实验时，SIMD 进行向量化处理的时候，采用的是四路向量化处理，而 OpenMP 多线程优化时，我们这里观察在不同 chunk_size 和不同数量的 thread 下，dynamic 线程任务划分方式的性能。

经过实验测试，我们得到如下实验结果：

NUM_THREAD	NUM_THREAD=4			NUM_THREAD=8		
chunk_size	6	12	24	6	12	24
Groebner_ordinary	4358.93	4263.68	4283.75	4219.6	4309.19	4994.06
Groebner_SIMD_SSE	3774.31	4076.5	4330.34	3884.33	3779.74	3911.82
Groebner_SIMD_AVX	3238.42	3138.75	3326.24	3280.83	3157.41	3260.68
OpenMP						
Groebner_OpenMP	7504.7	6211.51	5456.04	7254.96	6306.14	5313.01
Groebner_OpenMP_SSE	4663.96	4702.73	4752.93	4969.92	4305.93	4412.17
Groebner_OpenMP_AVX	3707.1	3816.9	3643.76	4015.49	4074.75	3493.44

分析上述表格，我们可以看到运用 OpenMP 方式反而比不用 OpenMP 花费更多的时间，说明对于我们给出的例子，不太适用于 OpenMP 进行并行，但是我们不能完全否定 OpenMP，也就是当消元子数量更大、被消元行更多的时候，采取 OpenMP 可能更加有意义。而对于这一点我们之前的实验就能很好地证明这个道理。



再仔细观察表格，我们也能发现在特殊高斯消元中，随着当线程数量小于 8 时，线程数量增多，性能提升，且随着 `chunk_size` 的提高，无论是否采用了 OpenMP，其所花费时间都普遍缩短。观察图，我们能发现当线程相同时，不同方案所消耗的时长，由此可以分析得出，在一定范围内，`chunk_size` 越大，对特殊高斯消元的提升越显著，特殊高斯消元的性能提升与块划分有较大的关联。同时，我们也可以从侧面说明我们在 Pthread 优化中所说的猜想，也就是对于线程特殊高斯并没有显著的性能优化，这与划分方式有很大的关系。

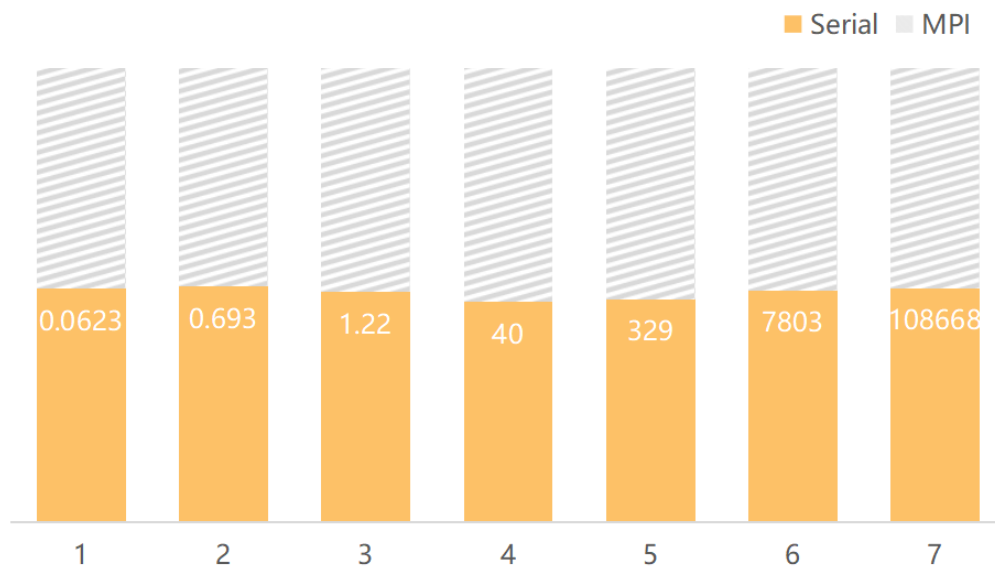
5.4 MPI 优化

5.4.1 MPI 多进程优化

我们对于特殊高斯消元的 MPI 优化策略探索，也全部在本地 X86 平台上完成。我们根据实验目的结合实验设计思想，探讨在不同问题规模下，串行算法与 MPI 多进程两者的实际处理效率。其中 MPI 多进程任务划分方式我们这里先采取块划分。而主要进行优化提升的部分，仍然是消元子消去被消元行时进行的运算。我们可以得到不同问题规模下的表现如下所示，表中时间单位均为毫秒。

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Serial	0.0623	0.693	1.22	40	329	7803	108668
MPI	0.0592	0.645	1.19	42.7287	341.394	7508.15	102564.3

根据表中数据，我们对于每种数据集进行比较分析，可以绘制下图以比较时间相差情况。



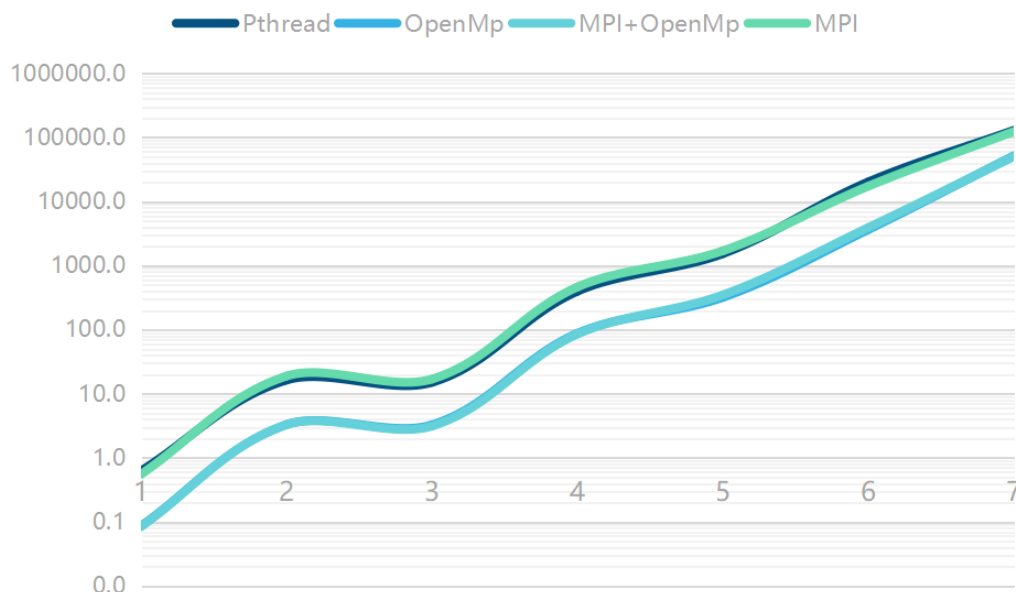
观察上图可以发现，图中分界线基本维持在中间，说明在各个数据集规模大小下，两者时间几乎一致，这也从侧面说明了 MPI 对于特殊高斯消元的算法性能提升不大。不过这也不排除可能未彻底发挥 MPI 并行化的优势，关于 MPI 优化我们在本次实验中未能看到有明显的作用。

5.4.2 MPI 与 OpenMP 的组合优化

接下来，我们再探讨 MPI 与 OpenMP 共同的优化效果。我们使用的 X86 平台并未发生改变，仍然是与 Pthread 相同的平台。在此平台下，我们探究 MPI、MPI+OpenMP 对于特殊高斯消元所产生的优化效果。对于线程的选取，我们选用 4 个线程，基准与之前相同，选取为相同四个线程的普通 Pthread 算法。

Column	130	254	562	1011	2362	3799	8399
消元子	22	106	170	539	1226	2759	6375
被消元行	8	53	53	263	453	1953	4535
Pthread	0.60261	15.6370	14.0567	403.283	1498.67	19149.7	134963
OpenMp	0.08557	3.19881	3.12976	86.8002	345.166	3716.94	49443.2
MPI	0.63972	17.6905	15.1201	404.708	1519.89	20121.6	128430
MPI+OpenMp	0.08703	3.45133	3.09597	86.2907	328.341	3657.26	51079.9

对上述表中数据处理，我们可以做出如下图，其中为了更好的显示数据量较小时的值，我们已经对纵坐标进行对数处理。



分析上述实验结果所得图与表，我们也能得到相似结论，也就是 OpenMP 确实促进了特殊高斯消元算法的性能优化。但是 MPI 对于特殊高斯消元算法的优化极其有限，基本上无影响，这也与我们在 MPI 多进程优化处得出的结果类似，OpenMP 与 MPI 共同优化，只发挥了 OpenMP 原有提升特殊高斯消去算法的优化性能，并未促使 MPI 的理论优化性能得到发挥。不过，这也可能与数据量大小的选取有关，因此只能说 MPI 在本次实验中未能展现出其对特殊高斯算法的优化表现。

5.5 CUDA 优化

由于无法使用更好的平台进行 CUDA 优化的调试与运行，而本地计算机显卡性能不高，为 NVIDIA GeForce MX250，仅为显卡中的入门级别。MX250 采用的是帕斯卡架构，基本上就是在上一代的 MX150 的基础上进行了小幅度的升级；如果是桌面级显卡阵营中，性能大概相当于 GTX1030。因此，个人觉得实现意义不大，进行相关探究也无法说明 CUDA 优化是否真的对于特殊高斯消元有性能上的提升，所以此处便不做过多的分析，相关代码已经在后续的连接中给出。下图是本地计算机的显卡显示图。



图 5.5: 本地显卡示意图

5.6 细节规划

5.6.1 数据读入

对于编程实验，若矩阵规模不大，我们就全部读入内存，不分批次。若矩阵规模较大，我们就考虑在读入“消元子”后，对于“被消元行”进行分批读入，当处理完读入后的“被消元行”后，抛弃无用行，然后再读入一批“被消元行”，如此循环读入，直到完成所有“被消元行”的处理。

5.6.2 存储方式

如按稠密矩阵存储，可采用位存储方式，即用整型值的一个比特保存矩阵的一个元素。这样，一对整数的异或运算即可完成很长的子向量的异或。SIMD 并行化也很方便。当然，矩阵很稀疏，也可采用稀疏矩阵存储方式，如 SIMD 并行化相对复杂。我们还可以考虑是否可以采取链表的存储方式。

5.6.3 多平台多指令集

在研究过程中，我们将尽力在不同平台、不同指令集指令集（SSE、AVX、AVX-512 等）的情况下，对于算法的性能进行综合的考量、评价、分析。对于每一种不同的情况，给出最合适的算法规划。

5.7 各种优化方法总结

通过对于在不同平台、不同指令集上 SIMD 优化、Pthread 优化等优化方法的探索，我们最终可以发现只有 OpenMP 对于特殊高斯算法具有较大的优化性能提升，其他的如多线程，使用不恰当反而会导致特殊高斯的程序性能降低。SIMD 优化与 MPI 优化则对于特殊高斯消元的影响不是很大。其中对于某类具体的优化方法，其影响因子也十分多，比如说划分方式，是否使用 profiling 及相关体系优化，每一种都会在特殊高斯内部得到微小的优化提升，此处就不在做过多展开。

6 总结与展望

6.1 总结

通过本次并行程序设计课程的学习，真的是让我学到了许多，知识难以言盖。精准到这门课的具体知识，如并行的各类优化算法的具体编写；发散到各类工具的使用、前沿技术的了解，如运用 VTune 进行程序性能分析、英伟达 OneAPI 的专业导师讲解。可以说，对于课程的每一部分都可以看出王刚老师和助教们的认真尽责。许多课程都没有十分具体的文档对讲解内容以及所需实现的实验进行说明，但是老师课程群里发的资料对于课下学习帮助真的蛮大。每次可以先在助教们的引导下完成普通高斯消元的内容，然后自己再尝试进行一些特殊高斯消元的拓展内容，这对于并行的知识点掌握还是挺有帮助的。课程所包含的知识点容量确实很大，不过通过这种学习方式，我想也是可以使每个同学尽自己最大的努力掌握到了自己可以掌握的知识。

本实验相关代码详见<https://github.com/Skyyyy0920/Parallel-Programming>

6.2 展望

若有时间，我相信我将进一步加深对于高斯消元两个问题的探讨，争取完全实现所有运用 SIMD 优化、Pthread 优化等优化方式对于特殊高斯消元做进一步的探究，如进行多线程并行化的不同算法策略（如块划分、循环划分等不同任务划分方法、流水线算法等）及其复杂性分析、不同编程方法（阻塞通信、非阻塞通信、双边通信、单边通信等）、profiling 及体系结构相关优化（如 cache 优化）等。使整个项目在现有基础上更加的全面、完善。

最后也希望王刚老师的《并行程序课程》能不断优化，使知识点讲解透彻清晰，得到越来越多人的认可。

参考文献

- [1] G. Niesi et al. A. Giovini, T. Mora. *One sugar cube, please, or Selection strategies in the Buchberger Algorithm*. 1991 Internat. Symp. on Symbolic and Algebraic Computation, ISSAC' 91, ACM, New York, 1991.
- [2] B. Buchberger. *A criterion for detecting unnecessary reductions in the construction of Gröbner-bases*. In EUROSAM 79, Lecture Notes in Comp. Sci., Springer, Berlin-New York, 1979.
- [3] B. Buchberger. *Gröbner-bases: An algorithmic method in polynomial ideal theory*. Multidimensional Systems Theory, 1985.
- [4] T. Mora H. M. Möller and C. Traverso. *Gröbner bases computation using syzygies*. Proc. ISSAC' 92, ACM Press, 1992.
- [5] J.-C.Faugère. *A new efficient algorithm for computing Gröbner bases (F_4)*. Vol. 139(1-3), 1999.
- [6] M. S. E. Mohamed et al J. Ding, J. Buchmann. *MutantXL*. In First International Conference on Symbolic Computation and Cryptography, Springer-Verlag, 2008.
- [7] D. Lazard. *Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations*. European Conference on Computer Algebra. Springer, Berlin, Heidelberg, 1983.
- [8] J. Patarin et al. N. Courtois, A. Klimov. *Efficient algorithms for solving overdefined systems of multivariate polynomial equations*. Proc. of EUROCRYPT' 00, Lect. Notes in Comp. Sci., 2000, vol. 1807: 392-407, 2000.
- [9] 赵向东. 一种 Gröbner 基的改进算法. MA thesis, 2020.