



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

并行程序设计

MPI 编程——高斯消去算法

孟启轩 2212452

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2025 年 6 月 23 日

摘要

结合课堂内容以及实验指导书，我采用多种不同的 MPI 并行策略对高斯消元算法进行优化，在 x86 平台上对比分析了不同任务划分方式、不同 MPI 编程方法对程序性能的影响，还与多线程和 SIMD 进行结合。

关键字：MPI 多进程高斯消元 x86 OpenMP SIMD 并行

目录

一、 问题概述	1
(一) 高斯消去算法概述	1
(二) 串行算法原理分析	1
二、 实验环境	2
三、 实验介绍	2
四、 MPI 设计实现	2
(一) MPI 版本的普通高斯消元	3
(二) 不同任务划分方式	3
(三) 通信优化：阻塞 vs. 非阻塞 vs. 单边通信	3
五、 对比分析方向	4
六、 实验结果分析	4
(一) 串行算法与 MPI 并行算法对比以及不同进程数下算法的性能	4
(二) 不同任务划分方式对比分析	5
1. 行划分方式分析	6
2. 列划分方式分析	6
3. 流水线算法分析	7
4. 二维划分方式分析	8
(三) 不同 MPI 编程方法	8
1. 4 进程下不同通信方法的性能表现	9
2. 8 进程下不同通信方法的性能表现	10
3. 进程数对不同通信方式的影响分析	10
(四) MPI 结合多线程 (OpenMP) 和 SIMD (SSE)	11
1. MPI 结合 OpenMP	11
2. MPI 与多线程 (OpenMP) 和 SIMD (SSE) 的结合	12
(五) 其他分析与现象	13
1. cache 优化	13
2. 第一次执行时间高很多	14
七、 遇到的问题以及说明	14
八、 总结	15

一、 问题概述

(一) 高斯消去算法概述

高斯消去算法是求解线性方程组 $Ax = b$ 的经典算法，其核心分为两个阶段，消去过程和回代过程。

消去过程：将系数矩阵 A 转换为上三角矩阵

1. **归一化**：对第 k 行进行除法操作，使主元 $A[k, k]$ 变为 1。

$$A[k, j] \leftarrow \frac{A[k, j]}{A[k, k]} \quad \text{对所有列 } j \geq k.$$

2. **消去**：利用第 k 行对后续行 ($k+1$ 至 N 行) 执行减法操作，消除这些行在第 k 列的元素。

$$A[i, j] \leftarrow A[i, j] - A[i, k] \cdot A[k, j] \quad \text{对所有行 } i > k \text{ 和列 } j \geq k.$$

回代过程：求解未知数 x_i

1. 从最后一行开始，逆向逐行求解未知数 x_i 。

$$x_i \leftarrow \frac{b_i - \sum_{j=i+1}^n A[i, j] \cdot x_j}{A[i, i]} \quad \text{对 } i = N, N-1, \dots, 1.$$

(二) 串行算法原理分析

串行算法通过消去过程将系数矩阵转化为上三角形式，利用消元因子逐步消除下方变量，确保每行仅保留当前主元及右侧变量。回代过程利用上三角特性，从末行开始逐层代入已知变量，最终求出所有解。此方法时间复杂度为 $O(n^3)$ ，其中消去过程时间复杂度为 $O(n^3)$ ，主要进行行消元，有三重循环，回代过程时间复杂度为 $O(n^2)$ ，主要进行逆向求和，有两重循环。

串行算法核心代码

```

1 // 消去过程
2 for (int k = 0; k < n; k++) {
3     for (int i = k + 1; i < n; i++) {
4         double factor = A[i][k] / A[k][k];
5         for (int j = k + 1; j < n; ++j) {
6             A[i][j] -= factor * A[k][j];
7         }
8         b[i] -= factor * b[k];
9     }
10 }
11 // 回代过程
12 x[n - 1] = b[n - 1] / A[n - 1][n - 1];
13 for (int i = n - 2; i >= 0; i--) {
14     double sum = b[i];
15     for (int j = i + 1; j < n; j++) {
16         sum -= A[i][j] * x[j];
17     }
18     x[i] = sum / A[i][i];
19 }

```

二、 实验环境

实验环境主要硬件信息如下：

属性	AMD Ryzen 7 5800H
架构	x86_64
指令位宽	64-bit
核心数	8
线程数	16 (2 线程/核, 支持 SMT)
主频 (Base)	3.2 GHz
一级缓存 / L1 Cache	64 KB
二级缓存 / L2 Cache	512 KB
三级缓存 / L3 Cache	16 MB
厂商	AMD
型号	Ryzen 7 5800H
缓存/内存带宽支持	高速缓存优化, 适合高性能计算
支持指令集扩展	SSE, AVX2, FMA 等
用途定位	笔记本/移动平台

三、 实验介绍

高斯消元作为一种经典的线性方程组求解方法，其核心包括标准化主元行、逐行消元与回代等步骤，计算量大、数据访问密集，极易成为程序的性能瓶颈。为提升其运行效率，实验将利用 MPI 并行编程模型对高斯消元算法进行加速优化。MPI (Message Passing Interface, 消息传递接口) 是一种用于在分布式内存系统中进行并行计算的通信协议和编程模型。它定义了一套标准化的函数接口，使不同节点间能够高效地进行数据交换，是高性能计算中最广泛采用的通信机制之一。目前常见的 MPI 实现包括 OpenMPI、MPICH、Intel MPI 等，广泛应用于科学计算、数值模拟、人工智能训练等领域。

MPI 支持多种通信方式，包括点对点通信 (如 `MPI_Send` 和 `MPI_Recv`)、集体通信 (如 `MPI_Bcast`、`MPI_Reduce`) 以及同步与异步通信操作。开发者可根据应用需求灵活组织进程间的数据流和同步行为。MPI 的主要特点包括：

- **可移植性**：可在不同操作系统和平台上运行，兼容性强；
- **高性能**：设计上支持底层网络优化，实现低延迟高带宽通信；
- **扩展性**：能支持从数十到数万个并发进程；
- **灵活性**：提供丰富的通信原语和用户自定义数据类型；

四、 MPI 设计实现

本实验围绕高斯消元算法，结合课程所学内容，设计并实现了多种基于 MPI 的并行优化方案。通过不同的任务划分、通信策略和调度机制，并结合 OpenMP、SIMD 以及 cache 等优化策略，有效提升了算法的执行效率与可扩展性。以下对各方案的设计与实现进行简要说明。

(一) MPI 版本的普通高斯消元

本程序实现了基于 MPI 的高斯消元算法，采用一维行块划分策略将 $n \times n$ 的矩阵按行划分给 m 个进程，尽量保持负载均衡。矩阵的初始化与划分由 0 号进程完成，并通过 MPI_Scatterv 将对应行数据与向量 b 分发至各进程。在主消元阶段，每轮由持有主元的进程完成除法操作，并通过点对点通信 (MPI_Send/MPI_Recv) 向后续进程广播更新行；接收进程据此进行局部消元。所有轮次完成后，通过 MPI_Gatherv 收集各进程的结果，最终在 0 号进程进行回代求解。

(二) 不同任务划分方式

一维行块划分

该策略将系数矩阵的行按块分配至不同进程，每个进程负责连续的若干行。对于第 i 个进程，其负责的行范围为 $[r_1, r_2]$ ，根据 $\lfloor n/m \rfloor$ 的行数均分，同时将余数行平均分配至前 $n \bmod m$ 个进程。消元过程按轮次进行，在第 k 轮中，拥有第 k 行的进程负责进行除法操作并将该行广播给所有后续进程，后续进程根据该行完成局部的消去操作。回代阶段可选择在 0 号进程进行集中处理，或在各节点并行完成。

一维列块划分

与行划分不同，列块划分方案中每个进程负责若干列。在消元阶段，需先广播第 k 行第 k 列的主元值给所有进程，由各进程在本地完成除法与消去。由于每一列在同一进程中具有良好的局部性，该方式适用于列为主的数据访问模式。为了避免广播过程中的阻塞等待，程序中还引入了非阻塞通信手段，提升整体通信与计算的重叠度。

二维块划分

二维划分方案将整个矩阵划分为 $p \times q$ 的网格，每个进程负责一个子块区域。该方案实现更细粒度的任务划分，并结合行列方向的广播机制：主元所在进程需在行方向广播除法结果，同时在列方向广播主元值。由于涉及双向通信与同步，二维划分需要精确安排数据依赖与通信流程，但其数据局部性与负载均衡性较好，适合大规模并行环境。

流水线算法

流水线算法在每一轮消元中采用点对点通信的方式代替一对多广播。当某进程完成第 k 行的除法操作后，将结果仅发送给下一个进程；接收进程在转发结果的同时开始局部消去，形成链式传递与处理的流水线结构。该方法有效降低了广播带来的通信瓶颈，实现通信与计算的高度重叠，尤其适合通信代价较高的集群环境。

(三) 通信优化：阻塞 vs. 非阻塞 vs. 单边通信

在不同的算法实现中，分别采用了阻塞通信、非阻塞通信（双边通信）、以及基于 MPI_Put 和 MPI_Get 的单边通信。阻塞通信结构简单，适用于结构清晰的同步场景；非阻塞通信允许计算与通信重叠，提高并行效率；单边通信进一步简化了同步流程，使数据交换更具灵活性。通过对比实验，本项目评估了各类通信方式在不同任务划分策略中的性能表现与适用场景。

五、 对比分析方向

通过不同问题规模（2 的整数次方）下的程序执行时间进行不同优化策略的对比分析，为了提高准确度，采用 MPI 计时方式并取 10 次执行时间的平均值。

- 串行算法与 MPI 并行算法对比以及不同进程数下算法的性能
- 不同任务划分方式对比分析
- 不同 MPI 编程方法（阻塞 vs 非阻塞 vs 单边通信）
- MPI 结合多线程（OpenMP）和 SIMD（SSE）

六、 实验结果分析

（一） 串行算法与 MPI 并行算法对比以及不同进程数下算法的性能

对 MPI 版本的高斯消元不同进程数（2、4、8）在不同问题规模下的运行时间与加速比结果进行分析。

表 1: MPI 不同进程数下的执行时间（单位：ms）

问题规模	串行	2 进程	4 进程	8 进程
32	0.040	0.02875	0.04560	0.13587
64	0.325	0.05703	0.07717	0.22695
128	2.188	0.26067	0.21366	0.39541
256	15.568	1.67832	1.18161	1.25733
512	140.059	16.5675	13.0117	23.9866
1024	1139.062	129.127	80.6078	89.5096
2048	9196.672	1837.26	1644.52	1497.68

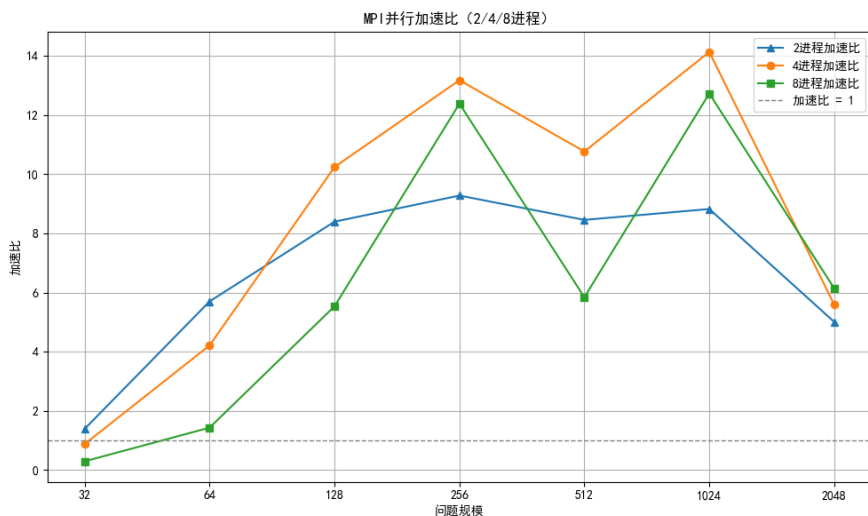


图 1: MPI 并行加速比 (2/4/8 进程)

从实验数据和加速比折线图可以看出，随着问题规模的增大，MPI 并行算法在大多数情况下都显著优于串行算法，通过进程间协作，显著缩短了计算时间，特别是在计算规模较大时效果更为明显，体现了并行计算的优势，并且 4 进程方案在综合性能上表现较好，既能获得较高的加速比，又避免了过多的通信开销。当问题规模较小时（如 $n \leq 32$ ），串行算法运行极快，消耗时间仅为数十微秒，此时 MPI 引入的进程启动、数据划分、通信与同步等固定开销占比较大，反而造成了整体效率下降。这种情况称为“并行计算负效益区间”，是并行计算中常见现象，并非所有问题都适合并行处理。随着问题规模增加，数据量变大，计算量快速增长，而 MPI 通信和启动等固定开销相对稳定，因此并行算法能够逐步展现出性能优势。

- **2 到 4 进程加速效果显著**：由串行向并行转换时，加速比提升明显，这是由于计算负载被合理分担至多个核心上，并行计算部分大大减少了总执行时间。此阶段通信开销尚未成为瓶颈。
- **4 到 8 进程收益递减**：继续增加进程数时，尽管计算子任务进一步被细化，但通信量和同步频率也随之上升，进程间的等待和冲突变得更频繁，导致计算资源不能被完全利用，甚至可能出现部分进程空闲等待。这种现象符合并行计算中的 **Amdahl 定律**，即串行部分限制了并行效率的上限。
- **通信与缓存瓶颈影响**：当进程数增多时，每个进程处理的数据块变小，导致缓存命中率下降，同时引入更多内存访问开销。再加上进程间通信的数据量增长，特别是广播主元行时，容易造成总线拥堵或网络延迟，进一步限制了并行扩展性。
- **负载不均带来的性能波动**：在某些问题规模（如 $n = 512$ ）下，加速比出现下降，这可能是由于任务划分不均，部分进程计算负载明显重于其他进程，导致整体被“最慢进程”拖慢。

（二）不同任务划分方式对比分析

为评估任务划分策略对 MPI 并行高斯消元程序性能的影响，实验分别实现了四种典型任务划分方式：**行划分**、**列划分**、**二维划分**与**流水线算法**，并在 4 与 8 进程两种并发度下测试多个问题规模的运行时间。图2展示了各配置下的对数尺度耗时热力图，颜色越深表示耗时越长。

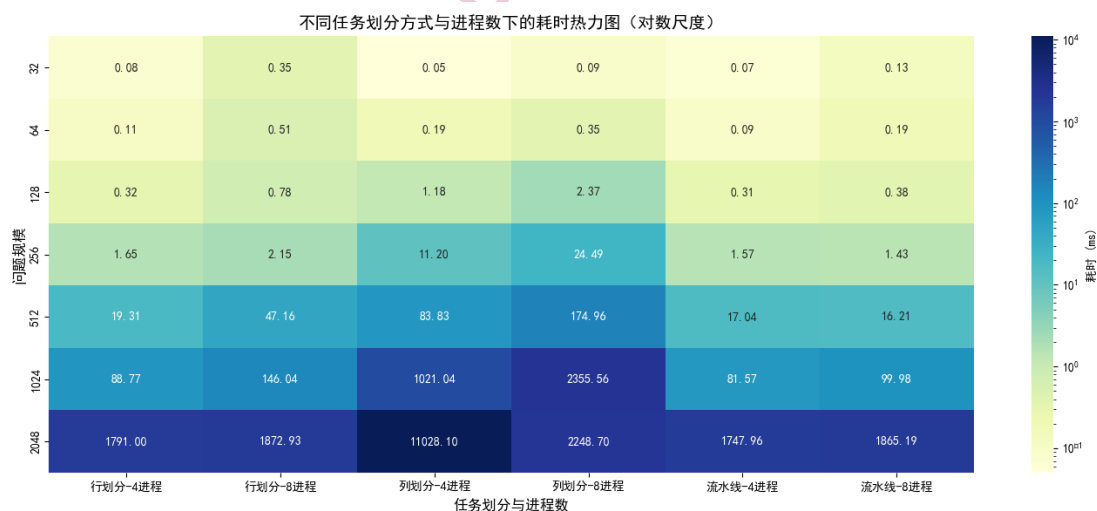


图 2: 不同任务划分方式与进程数下的耗时热力图

1. 行划分方式分析

- 行划分是最常见的任务划分方式，即将矩阵的每一行视作最小任务单元，按行均匀划分给各个进程，主元所在行由其所在进程进行归一化并广播至其余进程以完成消元。
- 在实验中，行划分方案在问题规模 $n = 128 \sim 512$ 时展现出良好的稳定性，尤其在 4 进程配置下，其缓存访问结构紧凑、通信频次适中，能够充分利用多核计算资源。
- 然而，随着进程数增加，主元广播操作变得更加频繁且同步代价更高，导致通信瓶颈加剧。在 $n = 1024$ 和 $n = 2048$ 时，8 进程相较 4 进程配置，加速比提升幅度明显减弱，说明其扩展性逐渐受限。
- 鉴于本实验平台为 AMD Ryzen 7 5800H，其采用共享 L3 缓存（16MB）和单 NUMA 域结构，行划分方案能更好地利用线程间高速缓存一致性机制，体现出较强的缓存局部性优势。
- 行划分通信逻辑简单、调试友好，适合高斯消元任务，是一种通用且高效的划分策略。

2. 列划分方式分析

- 列划分将矩阵的每一列分配给不同进程进行操作，在高斯消元过程中需要频繁访问整行数据并广播对角元素，导致通信模式复杂，缓存命中率低。
- 实验数据显示，列划分在绝大多数问题规模下表现最差，耗时远高于其他任务划分方式。这种现象的根本原因在于：
 - 每一轮都涉及多个进程并发访问不连续内存行，破坏了 L1/L2 缓存的空间局部性；
 - 广播对角元需要所有进程等待最慢者同步完成，形成“短板效应”，在多核 CPU 上表现为大量 CPU 时间浪费；
 - 实验平台不具备 RDMA 等硬件支持，列划分的通信复杂性在无专用互连优化的环境下表现尤为低效。
- 此外，AMD Ryzen 5800H 虽支持 SMT 和共享缓存，但并不针对复杂的数据搬移操作进行优化，列划分带来的访存跨线程冲突、缓存污染等问题放大。

列划分方式再优化分析

针对原始列划分方式通信代价高、缓存局部性差等问题，我尝试再引入两项优化策略以提升性能：

- **非阻塞广播**：采用 `MPI_Ibcast` 等非阻塞通信原语，在主元行广播阶段避免全进程阻塞；
- **通信与计算重叠**：在完成主元广播任务的同时，提前对当前数据块执行局部消元操作，从而减少通信等待时间。

表 2: 列划分 MPI 与再优化后的执行时间 (单位: ms)

问题规模	列划分		非阻塞广播 + 通信计算重叠	
	4 进程	8 进程	4 进程	8 进程
32	0.05035	0.09182	0.07125	0.11466
64	0.18688	0.35129	0.22490	0.40274
128	1.17546	2.37314	1.49608	2.40363
256	11.2041	24.4945	14.5375	18.8463
512	83.8299	174.963	93.3766	161.478
1024	1021.04	2355.56	953.412	2369.55
2048	11028.1	2248.7	10711.1	22562.7

尽管上述优化在理论上可减轻通信瓶颈,但实验结果表明其性能提升极为有限,甚至在大部分规模下反而出现性能下降。进一步分析其原因:

- **通信逻辑依赖未消除**: 列划分方案本质上仍然需要依赖对角元广播,其通信结构为“全进程-全同步”,即便引入非阻塞操作,进程间的数据依赖依旧存在,阻塞只是被延后,无法从根本上打破同步瓶颈。
- **通信与计算重叠难以实现**: 在列划分中,主元行广播必须完成之后才能继续当前列的消元,数据依赖链条较长,计算启动滞后,导致所谓“重叠”实质上并未带来明显并行性提升。
- **调度和开销反增**: 引入非阻塞通信及请求管理机制会增加调度复杂性和缓存压力,尤其在多进程高并发情况下更易引发线程间通信拥塞与上下文切换频繁,形成负优化。
- **平台支持不足**: AMD Ryzen 5800H 的内存子系统更偏向于多核共享计算优化而非通信密集场景,缺乏针对异步消息机制的硬件级调度器,导致非阻塞通信在执行层面缺乏并行调度保障;
- **数据访问依旧不规整**: 列划分方案核心问题在于跨行访问和全矩阵依赖,并未通过优化得到缓解,因此缓存失效率高,访存效率差仍然成为性能瓶颈。

3. 流水线算法分析

- 流水线算法采用点对点异步通信的方式,将主元行以链式方式在进程间逐步转发,每个进程在接收数据后立即进行局部消元,并将主元继续传递给下一个进程,形成计算与通信重叠的“流水线结构”。
- 实验表明,流水线算法在中大规模问题中表现出极强的扩展性,在绝大多数问题规模下性能要优于最常见的行划分。
- 相比集中广播,流水线方案显著减少了通信阻塞与全局同步,尤其在本平台的多核心 SMT 结构下,点对点异步消息机制充分利用了 CPU 的并行通信能力。
- 此外,由于流水线仅在进程链中传递数据,避免了缓存穿透和广播冲突,特别适合共享缓存架构。
- 流水线方案结构设计合理、通信重叠度高、并行粒度自然,是适配现代多核 CPU 的高效并行策略。

4. 二维划分方式分析

表 3: 二维划分 MPI 的执行时间 (单位: ms)

问题规模	4 进程	16 进程
32	0.0748	0.7304
64	0.209	0.5936
128	0.5811	1.082
256	2.1731	3.5389
512	22.8967	42.1566
1024	184.537	710.437
2048	4448.07	11928.3

- 二维划分策略将整个矩阵在行和列两个维度上均匀划分, 形成 $p \times q$ 的进程网格结构, 各进程负责处理局部子块, 并协作完成主元选取、广播与消元操作。该方式具有更好的负载均衡性和并行性, 理论上通信量最小, 适合大规模进程集群。然而在本实验平台上, 二维划分策略的性能表现并不十分理想。
- 分析其根本原因在于: 二维划分引入了更多的进程间边界, 需要更频繁地进行横向纵向两个方向的通信与同步, **在共享内存平台上模拟网络通信代价高昂**, 并且进程数量增多导致资源竞争严重, 最终掩盖了算法本身的通信重叠与并行优势。
- 此外, 本平台并未配备高效的网络拓扑或低延迟互联设备, 进程间通信仅依赖本地内存结构与操作系统调度, 进一步限制了二维划分策略的潜力发挥。

结合热力图和实验数据的综合结论

1. **行划分方式**性能优异, 适合通信结构稳定、资源紧凑的通用并行计算平台。
2. **列划分方式**通信代价高、访问模式不规整, 在无分布式缓存机制支持的共享内存平台上表现极差。
3. **二维划分方式**具有较好的负载均衡性和并行性, 适合部署在支持硬件级通信加速的分布式并行平台上, 在本实验所用共享内存多核平台中, 反而因其高通信复杂度与多进程调度负担而不如流水线与行划分方式表现优异。
4. **流水线算法**在任务规模较大、多进程数的场景下具备良好的伸缩性与通信重叠能力, 是高性能并行消元任务的最优选择。
5. **平台特性决定通信策略适应性**: 本实验平台为 AMD Ryzen 消费级多核结构, 具备较强计算密度与共享缓存, 但不具备通信加速硬件, 列划分效果更差, 而流水线、行划分更贴合本平台特点。

(三) 不同 MPI 编程方法

为研究 MPI 中不同 MPI 编程方法 (不同通信方式) 对并行性能的影响, 我在 4 进程与 8 进程两个并发度下, 分别测试了阻塞通信、非阻塞通信与单边通信三种方法在不同问题规模下的

执行时间与加速比。我将从整体趋势、具体进程规模下的比较，以及进程数对通信方法的影响等角度展开分析。

表 4: 不同通信方式与进程数下的 MPI 执行时间 (单位: ms)

问题规模	串行	阻塞通信		非阻塞通信		单边通信	
		4 进程	8 进程	4 进程	8 进程	4 进程	8 进程
32	0.040	0.2771	0.13024	0.2757	0.06359	0.23209	0.61467
64	0.325	0.5358	0.1815	0.5418	0.08335	0.37538	1.00327
128	2.188	0.22534	0.27678	0.21872	0.24171	0.76672	1.85688
256	15.568	1.23964	1.44167	0.90217	1.16907	2.23544	3.78825
512	140.059	13.4963	17.2328	13.733	11.5401	15.0099	16.1673
1024	1139.062	91.4767	77.4909	84.2777	63.9211	87.6803	92.4931
2048	9196.672	1667.25	1412.93	1244.76	972.468	1542.64	1488.27

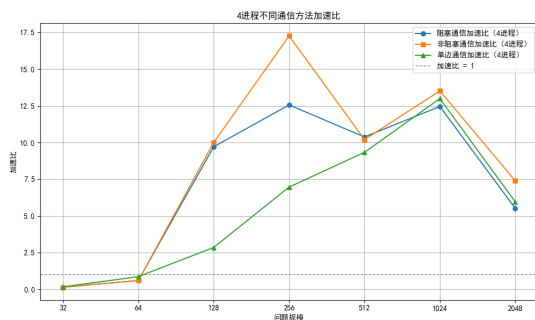


图 3: 4 进程不同通信方法加速比

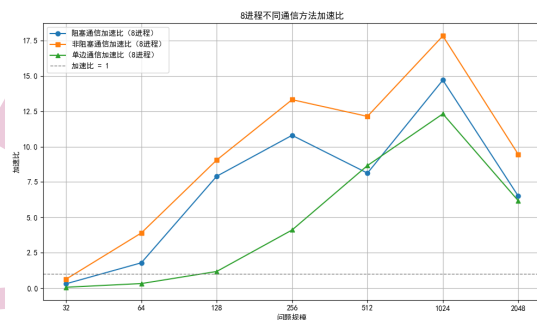


图 4: 8 进程不同通信方法加速比

整体分析：通信方式与性能的关系

整体来看，不同通信方式的性能差异随着问题规模的扩大而逐渐显现。在小规模问题（如 $n \leq 32$ ）下，由于计算量非常小，所有通信方式的固定开销都显得相对“昂贵”，导致整体运行时间甚至超过串行实现，加速比低于 1 的现象正是这种通信主导的负效益的体现。

随着问题规模增大，通信与初始化等固定开销被大量计算所“稀释”，加速比随之显著上升。特别是在 $n = 256 \sim 1024$ 区间内，加速最为明显，这是因为此区间内通信和计算开销达到相对平衡点，能最大化利用多进程资源。其中，非阻塞通信方案因其能同时重叠通信与计算，减少等待时间，性能表现最为优越。

1. 4 进程下不同通信方法的性能表现

在 4 个进程的配置下，三种通信方式均能获得显著加速，但具体表现存在差异，背后的原因与代码实现密切相关：

- **非阻塞通信最优:** `nonblocking_gauss` 使用了典型的流水线转发机制,主元行使用 `MPI_Isend` 异步传给下一个进程,并在下一个进程通过 `MPI_Irecv` 接收后继续转发,这种点对点链式传递减少了广播同步压力。同时,通信与后续计算可并发执行,在 4 进程结构中非常适配。
- **阻塞通信表现稳定:** `blocking_gauss` 使用的是集中广播,每轮主元所在进程通过多次 `MPI_Send` 将行同步发送至所有高编号进程,后者通过 `MPI_Recv` 等待数据。这种结构简单

但同步严重,导致在多进程间会出现通信等待链,制约了加速比提升。但在 $n = 128 \sim 1024$ 区间内仍能保持 8 至 12 倍的加速比,说明通信与计算仍有较好配合。

- **单边通信低于预期:** `onesided_gauss` 虽然理论上可以实现进程主动拉取主元行,减少发送方压力,但实际中需依赖 `MPI_Win_fence` 进行同步保护。由于 MPI RMA 的 `fence` 同步是全局屏障,在进程数较少时额外同步代价难以摊销,因此整体表现反而不如前两者。

2. 8 进程下不同通信方法的性能表现

当进程数扩大至 8 时,通信的复杂性与同步代价进一步上升,不同通信方式的性能差异更为显著:

- **非阻塞通信依然表现最优:** 得益于流水线转发结构, `nonblocking_gauss` 的每个进程在接收前一进程传递的主元行后立即转发,形成通信“流水线”,有效利用了更多的通信通道与计算资源。实验中在 $n = 1024$ 达到最高加速比 17.66,充分展示其扩展性。
- **阻塞通信瓶颈明显:** 由于进程数翻倍,集中广播中主元发送方需执行更多次 `MPI_Send`,后续进程同步等待的链更长,延迟积累严重。例如在 $n = 512$ 时出现加速比下降,是阻塞式同步在高并发下的典型性能瓶颈。
- **单边通信稍有改善但仍不足:** 8 进程下, `onesided_gauss` 的表现略优于 4 进程,说明 RMA 模型在一定并发下具备潜力。但由于仍需依赖 `MPI_Win_fence` 的同步,以及多个进程并发访问同一内存窗口可能带来冲突,性能整体仍低于非阻塞通信。

3. 进程数对不同通信方式的影响分析

- **非阻塞通信最具扩展性:** 程序设计充分利用了 `MPI_Isend / MPI_Irecv` 的异步性,加之链式结构天然适合高并发场景,因此即使进程数增加,也不会增加主元行的广播成本,反而能更充分并发,表现出线性甚至超线性加速。
- **阻塞通信扩展性有限:** 广播由一个发送者对多个接收者实现,随进程数增加通信量线性上升,同时接收方需同步等待数据,导致通信占比上升明显,不能很好支撑高并发。
- **单边通信扩展依赖实现与平台支持:** 虽然理论上能实现任意进程对共享内存的高效访问,但整体表现仍不稳定,实际中取决于 MPI 实现的优化程度与网络硬件,我笔记本的 AMD Ryzen 7 5800H 就不支持 RDMA,也不具备优化的 RMA 支持,因此扩展性不如预期。

综合实验数据与实现可得如下结论:

1. 非阻塞通信方式结合了异步机制与链式转发结构,在不同规模与并发数下始终具有最优性能,在实际系统中的通用性和扩展性最强;
2. 阻塞通信结构简单,调试便利,但不具备良好扩展性,在进程数较少、对性能要求不高的场景具备一定优势;
3. 单边通信具备理论潜力,但在主流 CPU 平台上受限于 MPI RMA 实现和同步机制;

表 5: 三种通信方式性能对比总结

通信方式	通信重叠	同步特性	适用场景
阻塞通信	否	显式点对点同步	小规模并发、调试方便
非阻塞通信	是	异步、链式传输	通用平台、高性能计算
单边通信	取决于平台	显式窗口 +fence 同步	高速网络支持 RMA

(四) MPI 结合多线程 (OpenMP) 和 SIMD (SSE)

1. MPI 结合 OpenMP

为深入探究进程数与线程数在混合并行策略中的协同关系，在保持总逻辑核数为 8 不变的前提下，比较了不同进程/线程组合策略对高斯消元程序性能的影响，测试方案包括：

- 2 进程 8 线程 进程较少，每进程占用全部逻辑核
- 4 进程 4 线程 进程与线程均匀分配，典型混合模式
- 8 进程 2 线程 高并发进程，线程数较少
- 8 进程 8 线程 超出物理核上限，线程资源超发

表 6: MPI+OpenMP 不同进程数/线程数下的执行时间 (单位: ms)

问题规模	2 进程 8 线程	4 进程 4 线程	8 进程 2 线程	8 进程 8 线程
32	2.72	2.27	2.02	3.90
64	5.16	4.46	4.15	9.55
128	10.75	9.25	8.32	16.28
256	24.69	20.69	17.01	39.69
512	63.64	59.56	57.73	130.32
1024	139.50	165.07	135.34	278.88
2048	1649.82	1653.09	1532.71	2058.96

- **线程数过多会引发资源竞争与调度开销：**在 8 进程 8 线程的策略中，系统需调度 64 个线程（远超 AMD Ryzen 5800H 的 16 个逻辑核），导致频繁的线程切换、缓存失效与上下文切换开销，从而严重拖慢整体性能，表现为“超线程负优化”。
- **进程数增多有助于通信并行化：**从 2 进程到 8 进程，数据划分更加细粒度，主元传播与消元过程可并发展开，通信开销在一定程度上被掩盖。但过多进程也会引发通信拥堵与 MPI 内部调度竞争。
- **4 进程 4 线程是性能与资源使用的平衡点：**在各类组合中，4 进程 4 线程为典型的“对称混合”模式，能较好地协调 MPI 通信与 OpenMP 计算之间的资源分配，在多数问题规模下展现出较为稳定的性能。
- **多线程策略更依赖缓存结构与亲和性：**当线程数增加时，尽管每个进程可充分利用线程并行，但也极易引发 L1/L2 缓存竞争，如果没有显式绑定线程亲和性，则线程漂移可能进一步拖慢执行速度。

- **任务粒度不匹配将抵消并行优势：**对于较小问题规模 ($n \leq 128$)，由于任务粒度过小，线程调度与 MPI 通信的开销远大于并行计算收益，导致加速比下降或负优化。而在 $n \geq 1024$ 时，较大的任务粒度能更好地摊薄调度与通信开销，从而发挥并行优势。

2. MPI 与多线程 (OpenMP) 和 SIMD (SSE) 的结合

以 4 进程 MPI 为基准,对比分析 MPI+SSE、MPI+OpenMP(4 进程 4 线程) 以及 MPI+OpenMP+SSE(4 进程 4 线程), 测的实验数据并绘制加速比曲线如图 5 所示。

表 7: 不同优化方式下的执行时间 (单位: ms)

问题规模	MPI	MPI+SSE	MPI+OMP	MPI+OMP+SSE
32	0.0456	0.04207	2.26576	2.21323
64	0.07717	0.07737	4.45805	4.34707
128	0.21366	0.17390	9.24762	9.09241
256	1.18161	0.83729	20.6880	20.4695
512	13.0117	10.1631	59.5632	59.2489
1024	80.6078	63.8206	165.072	147.553
2048	1644.52	1381.85	1653.09	1533.25

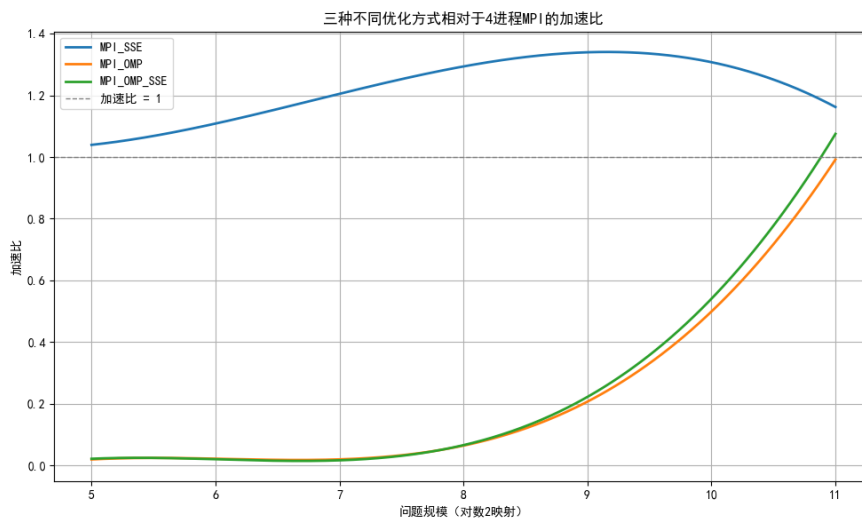


图 5: 三种优化方式相对于 4 进程 MPI 的加速比曲线

从图中可以看出，**仅使用 SSE 向量化的 MPI + SSE 优化策略**在所有问题规模上均表现为稳定的性能提升，优化效果最好。这说明 SIMD 能有效利用 CPU 的向量指令集，在数据并行场景中显著减少浮点运算延迟。

然而，**引入 OpenMP 的两种方案** (MPI + OpenMP、MPI + OpenMP + SSE) 在绝大多数问题规模中加速比明显低于 1，出现“负优化”现象，直至问题规模 $n \geq 2048$ 才接近 MPI 性能。

出现负优化的原因分析如下：

- **线程创建和调度开销：**OpenMP 引入的线程池、线程调度和同步机制在小规模任务中带来显著额外开销，掩盖了其本身的并行加速潜力。

- **数据划分粒度过细**: 在每个进程内仅使用 4 线程进行共享内存并行时, 若任务规模不大, 则每线程工作量极小, 导致线程间切换和负载不均衡问题加剧, 效率下降。
- **缓存争用与 NUMA 影响**: 多线程共享缓存结构容易导致缓存污染和伪共享, 尤其在数据访问不连续或多个线程频繁访问相同内存区域时, 性能不升反降。
- **SSE 与 OpenMP 的调度不一致**: 向量化指令本质上依赖于数据对齐和顺序性, 当与 OpenMP 动态调度并用时, 可能破坏数据连续访问模式, 影响 SIMD 效果。

MPI+SSE 是效果最好的轻量优化方案, 其实现成本低、性能提升稳定。而引入 OpenMP 的并行化方案仅在大规模数据量下才能发挥其优势, 否则反而因同步与调度成本产生负效应。并且多层优化应注重任务划分粒度与线程调度策略, 避免引入过多并行“嵌套”而导致负优化, 必要时应结合静态调度和线程亲和性策略进一步控制并行开销。

(五) 其他分析与现象

1. cache 优化

除上述优化策略外, 我还对 MPI 进行了 cache 优化, 在 4 进程和 8 进程下进行对比分析。

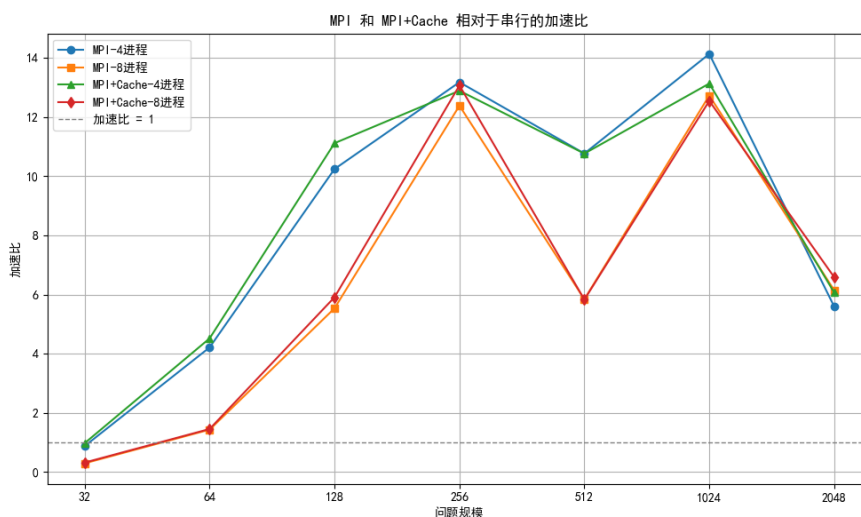


图 6: MPI 和 MPI+Cache 相对于串行的加速比

从加速比折线图和实验数据中可以观察到, 问题规模 $n = 128 \sim 1024$ 时, **MPI+Cache 优化方案相较于原始 MPI 有明显性能提升**。例如在 $n = 512$ 的情况下, 4 进程版本的 MPI+Cache 与 MPI 在加速比上几乎持平, 而在 $n = 1024$ 时, MPI+Cache-4 的加速比仍维持较高水平, 而纯 MPI 已出现波动下降的趋势。这说明**适当利用缓存特性可以提高数据局部性、降低主存访问延迟**, 进而优化性能。然而, 在大规模问题 $n = 2048$ 时, MPI+Cache 的性能略低于 MPI 原始方案, **主要原因是缓存容量不足以承载完整数据块**, 导致缓存命中率下降、数据回写频繁, 加剧总线 and 内存压力。同时, Cache 优化未能减少通信成本, 对于大规模问题而言, 其边际效益不如通信结构优化明显。

综合来看, **MPI+Cache 具有良好的加速效果**, 但在过小或过大的问题规模下, 其优势会被启动开销与缓存容量瓶颈所抵消。

2. 第一次执行时间高很多

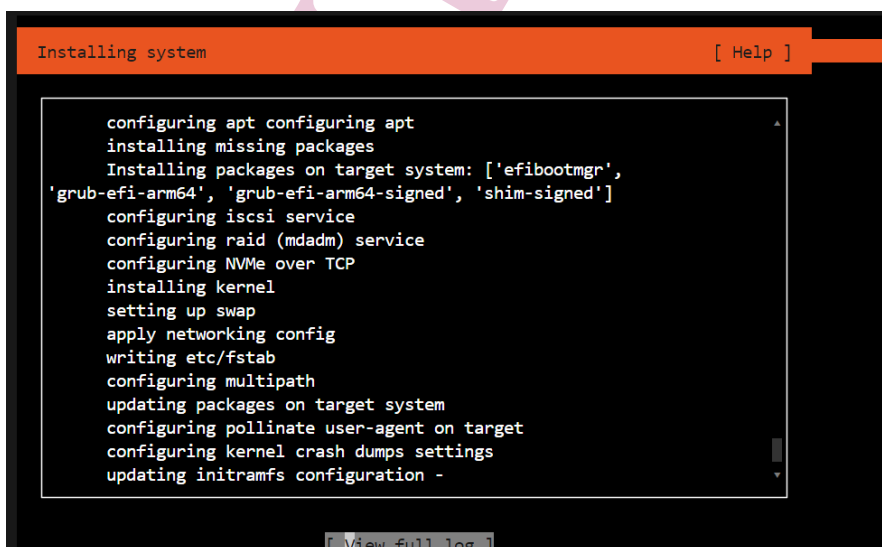
在本次 MPI 实验中, 我对每个问题规模进行了 10 次测量并取平均值作为最终耗时数据。观察发现, 当问题规模小于等于 256 时, 第一次执行的耗时明显高于后续执行。我推测这种现象是由启动开销造成的, 即程序在首次运行时涉及多个初始化步骤, 包括 MPI 环境的启动、进程间通信缓冲区的分配、内存页的映射与热启动、动态库加载以及操作系统对线程/进程调度的初始优化调整。这些初始化操作在首次执行时尤为明显, 而在随后的多次运行中, 系统缓存、调度策略与内存映射等机制已被激活, 从而显著减少了额外开销。因此, 对于小规模任务, 启动成本在总耗时中占比更大, 进而导致第一次执行时间偏高; 而在问题规模增大后, 计算与通信负载主导耗时表现, 初始开销的相对影响逐渐降低。

七、 遇到的问题以及说明

本次实验最初在 x86 平台上进行地较为顺利, 想要将程序迁移到鲲鹏服务器上进行分析, 但是编译程序并通过 qsub 脚本执行后, qstat 看到任务始终在等待状态 (Q), 真的是一直在等待状态, 无论如何都无法进行实验。作业延期以后, 我隔一两天就用鲲鹏服务器尝试一下, 但还是无法进行 MPI 实验。

于是我换了个思路, 想要在本机通过 WSL 和 QEMU 模拟 ARM 架构, 但是 MPI 依赖的是运行时环境, 而 QEMU 只支持模拟单个进程, 无法在 qemu-user 模式下模拟多个 MPI 进程并进行完整的 MPI 实验。所以我打算用 QEMU System 启动完整 ARM Ubuntu 镜像 + 原生 apt 安装 OpenMPI 来进行实验, 安装一个真正的 ARM 虚拟机。

然后, 我就在本机下载 Ubuntu ARM64 镜像, 用 QEMU 启动虚拟机, 但是下载安装过程十分缓慢, 我花了 6 个小时进行安装与配置, 好不容易运行起了 ARM 虚拟机, 进入到了真正的 ARM 架构, 然后我只需要安装 MPI 进行实验就可以了。



```
Installing system [ Help ]

configuring apt configuring apt
installing missing packages
Installing packages on target system: ['efibootmgr',
'grub-efi-arm64', 'grub-efi-arm64-signed', 'shim-signed']
configuring iscsi service
configuring raid (mdadm) service
configuring NVMe over TCP
installing kernel
setting up swap
apply networking config
writing etc/fstab
configuring multipath
updating packages on target system
configuring pollinate user-agent on target
configuring kernel crash dumps settings
updating initramfs configuration -

[ View full log ]
```

图 7: 安装 Ubuntu ARM64


```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIG1eFONfG5zXbSA5Z/b3kh8CJwnohM7GK
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQgQDXeOz7pa1rOKMVH/CXtN0zkFoADTyxh
IXbn+gouW+1xag888xBCrpCpVZVQ203KLJ8pdGHayYUmYa9Xy8NjECwV11jZDe50olq/
83qaVgL0FNqF+QwGpukSPf4W+vjd2rya0SAz0sWPBsn7ppsgYqO2PUkmQ7Gsi5LIaiqS1
WpkIh/D95VfV4G7HWG20bTuDyY+FDW68g4E7DuX9OyzNUEMUq2dc13u0a1hGPJpGqrBjh
-----END SSH HOST KEY KEYS-----
[ 245.288565] cloud-init[1520]: Cloud-init v. 24.2-0ubuntu1~22.04.1
[ OK ] Finished Cloud-init: Final Stage.
[ OK ] Reached target Cloud-init target.

mqx@armvm:~$ uname -m
aarch64
mqx@armvm:~$
```

图 8: 命令行

但是不知道为什么，它又自动进行了一些初始化工作之后，我输入任何命令都没有反应了。然后我就退出尝试重新进入，想要通过 VScode 进行 SSH 连接，这样也方便一些，但是退出重新登录后还是输入命令没有反应。前前后后搞了半天，本以为是胜利的曙光，没成想是“压垮骆驼的最后一根稻草”。最后，无奈我只能在 x86 上进行本次实验了。

八、 总结

通过本次并行编程实验，我深入掌握了 MPI 并行计算技术，系统探索了在高斯消元算法中 MPI 的不同划分策略和编程策略对性能的影响。此外，我还认识到通信方式、线程划分与内存访问模式对最终性能有显著影响，并将 MPI 与多线程和 SIMD 结合起来。实验过程不仅锻炼了我对并行程序调试与优化方面的实践能力，也加深了我对并行计算原理和高性能计算系统架构的理解。

源代码链接 [Github](#)