南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

Gauss 消去 MPI 并行优化研究

曹嘉悦

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 6 月 1 日

摘要

MPI 是用于分布式内存并行程序设计的函数库。本工作在 SIMD 和 openMP 并行化的基础上，利用 x86 和 ARM 两种平台，探索了块划分、循环块划分、流水线算法等多种方法的高斯消去并行化效果。作者在不同规模、不同进程数下，测试了高斯消去 MPI 并行化性能，并对结果正确性进行验证，必要时辅以 cache 优化等辅助优化手段。

关键字：MPI 流水线 高斯消去 数据划分 并行优化

目录

一、 实验背景

(一) 实验目的

1. 学习使用 MPI 进行高斯消去的并行化，并与 SIMD、OpenMP 等方法结合优化性能；2. 比较不同平台、规模、进程数下，任务调度和划分等算法策略的优劣；

(二) 实验环境

本实验代码在 Microsoft Visual Studio 2019 环境下编写和初步编译，使用 mpiexec 调试后，在金山云服务器上进行 x86 平台性能测试。通过修改后，适配 arm 平台的代码首先在本地 WSL2 Ubuntu 上的 qemu 模拟编译、运行，之后使用 Xshell 和 Xftp 提交至华为鲲鹏服务器，使用 TMD-GCC 64bit Release 的 mpic++ 编译器进行无优化编译，并进行性能测试。x86 平台配置参考金山云服务器配置，arm 平台配置参考鲲鹏服务器配置。

(三) 实验描述

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。考虑在整个消去的过程中，、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。

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

$$\begin{cases} l_{1,1}x_1 = b_1 \\ l_{2,1}x_1 + l_{2,2}x_2 = b_2 \\ \vdots \quad \vdots \quad \ddots \quad \vdots \\ l_{m,1}x_1 + l_{m,2}x_2 + \cdots + l_{m,m}x_m = b_m \end{cases} \qquad \begin{cases} x_1 = \dfrac{b_1}{l_{1,1}}, \\ x_2 = \dfrac{b_2 - l_{2,1}x_1}{l_{2,2}}, \\ \vdots \\ x_m = \dfrac{b_m - \sum_{i=1}^{m-1} l_{m,i}x_i}{l_{m,m}}. \end{cases}$$

图 1: 高斯消去过程图

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 oenmp 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，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。此外本次实验还尝试使用 GPU 进行。

(四) 朴素算法

将整个矩阵转化为上三角矩阵以便实现矩阵求解——

朴素串行算法

```
void plain() {
        for (int i = 0; i < ROW - 1; i++) {
                for (int j = i + 1; j < ROW; j++) {
                        matrix[i][j] = matrix[i][j] / matrix[i][i];
                }
                matrix[i][i] = 1;
                for (int k = i + 1; k < ROW; k++) {
                        for (int j = i + 1; j < ROW; j++) {
                                matrix[k][j] = matrix[k][j] - matrix[i][j] *
                                    matrix[k][i];
                        }
                        matrix[k][i] = 0;
                }}}
```

（五） 矩阵数据初始化

为避免因矩阵非满秩，导致测试过程中出现 Naf 和 Nan，本文采用如下方式生成测试数据：首先生成一个上三角元素为 1-1001 范围内随机浮点数的满秩矩阵，然后用-2-2 内的随机浮点数乘矩阵的某行，并随机加至另一行，并重复此操作 1000 次。经实验，此法生成的矩阵内元素值适中，不会发生溢出。重复使用该方法生成 10000 个 1024*1024 矩阵，均未检测到 Naf 或 Nan。

二、 实验设计

(一) 数据划分

常见的数据划分有以下几种，其可以根据问题的性质和计算资源的分布选择合适的划分策略。

- 块划分（Block Decomposition）：将矩阵按行或列划分为块，并将每个块分配给不同的进程。每个进程负责处理一个块的计算。块划分通常在矩阵行数或列数能够被进程数整除时使用。
- 循环划分（Cyclic Decomposition）：将矩阵的行或列循环地分配给不同的进程。每个进程负责处理一部分循环中的行或列。循环划分可以更好地平衡负载，但可能会导致通信开销增加。
- 斜划分（Skew Decomposition）：将矩阵按对角线方向划分为斜块，并将每个斜块分配给不同的进程。这种划分方式适用于具有特殊结构的矩阵，如三角矩阵或带状矩阵。
- 高斯-赛德尔划分（Gauss-Seidel Decomposition）：将矩阵按行或列划分为两个部分，其中一个部分由当前进程更新，另一个部分由其他进程更新。该划分方式用于高斯-赛德尔迭代方法，其中每个进程需要与其他进程交换数据。

本次实验我们重点采用块划分方式和循环划分的方式优化高斯消去。

块划分通过将待处理的矩阵依据系统资源和算法需求划分为多个块，然后对每个块进行局部的高斯消去操作。在块内部的消元操作完成后，处理器之间需要进行通信以保持消元结果的一致性。继续进行后续的高斯消去步骤，如回代和求解解向量。同样，这些步骤也可以进行块划分和

并行处理。我们再把连续几行划分为一个块后，考虑随消元的进行，已经完成除法的行将不再参与后续的运算，从而导致可能出现进程闲置的情况，这其实是有点负载不均在的。

而循环划分会将数据按照照进程数量等步长分配给每个进程，这样从整体来看，每个进程负责的任务的计算量大致是相同的，不会导致严重的负载不均现象。因此在整个过程中，所有的进程基本保持满负荷，整体的加速比就会接近理论的加速比。

(二) 数据收发方式

- 广播（Broadcast）方式：最朴素的方式是将除法的结果依次发送给所有进程。这种方式的通信开销与进程数量成正比，但存在较长的空闲等待时间。采用广播方式的时间开销为 $O(N^3 \log N)$。
- 流水线（Pipeline）方式：采用流水线思想，当一个进程接收到除法的结果后，立即将结果转发给下一个进程，从而减少负责除法工作的进程的阻塞时间。这种方式的时间开销为 $O(N^3)$。

MPI_AllReduce 函数 其工作方式是每个进程将自己的数据与其他进程的数据进行归约操作，并将结果存放在接收缓冲区中。最后，归约结果将广播给所有进程，使得每个进程都能访问到相同的归约结果。

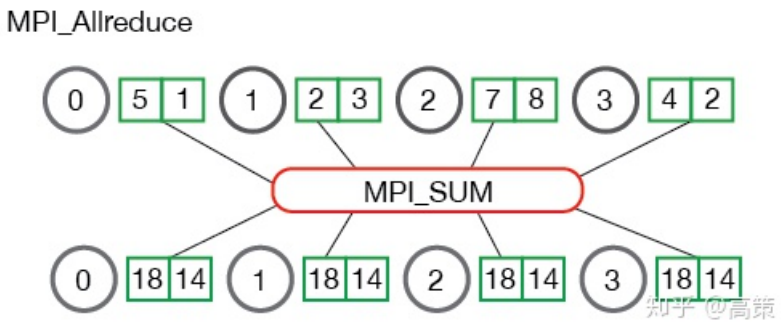

图 2: MPI_AllReduce 函数原理示意图

实验中，我们会对两者的优化效果进行对比。理论上，流水线方式的加速效果会更好，我们需要实验验证这一点。

(三) 问题规模和进程数量

通常情况下，随着问题规模的增加，高斯消去的计算复杂度也会增加。较大的问题规模可能需要更多的计算资源和更长的计算时间。在这种情况下，MPI 可以提供并行计算的能力，将计算任务分配给多个进程来加速计算过程。通过合理划分数据和任务，可以充分利用并行计算资源，减少计算时间。

进程数量是指参与 MPI 并行计算的进程的数量。增加进程数量可以增加并行计算的规模和速度。然而，进程数量的增加也会带来一些问题。首先，进程间的通信开销会增加，因为需要进行更多的消息传递和同步操作。这可能会导致额外的延迟和通信开销，从而影响性能。其次，如果进程数量过多，可能会导致负载不均衡的问题，即某些进程的计算任务较重，而其他进程空闲等待。因此，选择合适的进程数量是很重要的，需要考虑问题规模、计算资源和通信开销等因素，以获得最佳的性能提升。

而进程数量和问题规模也会存在相互的关联。考虑到进程间的通信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，由于进程通信阻塞导致的额外开销会抵消掉多进程优化效果，甚至还会表现出多进程比串行算法更慢的情况。而随着问题规模的增加，进程通信阻塞所需要的时间开销相对于每个进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进程并行优化的效果。

这些都需要后续实验的验证。

(四) 并行处理

在 Gauss 消去算法中，每一轮消去包含两个主要阶段：除法运算和消元减法操作。这两个阶段之间必须严格按顺序执行，即必须先完成消元行的除法运算，然后才能执行被消元行的减法操作。此外，在每一轮结束后还需要进行一次同步操作。

在使用 MPI 进行多进程实验设计时，可以考虑使用单个进程来分发任务。然后，在外层循环中，每个进程都需要判断当前进行除法运算的行是否是自己分配的任务。当除法运算完成后，需要将结果广播给其他所有进程，这样其他进程就可以接收到除法结果，实现进程间的同步。在这个实验中，没有采用主从模式，即所有进程都会执行消元的任务。因此，可以注意到除法结果在每次完成后累计，并且每次完成除法后都进行一次广播，因此当最后一行完成时，所有的除法结果已经广播到各个进程中。因此，可以省略最后一次结果同步的步骤。

由于 MPI 是进程级别的并行，还可以考虑在各个进程内部使用线程级别和指令集级别的并行优化，结合之前提到的 OpenMP 和 SIMD（Single Instruction, Multiple Data）技术，以实现并行优化的叠加效果。综合考虑所有优化方法，最终的性能提升效果应该与进程数量、线程数量和向量化程度成正比。

三、 实验结果

(一) ARM

1. 数据划分对比

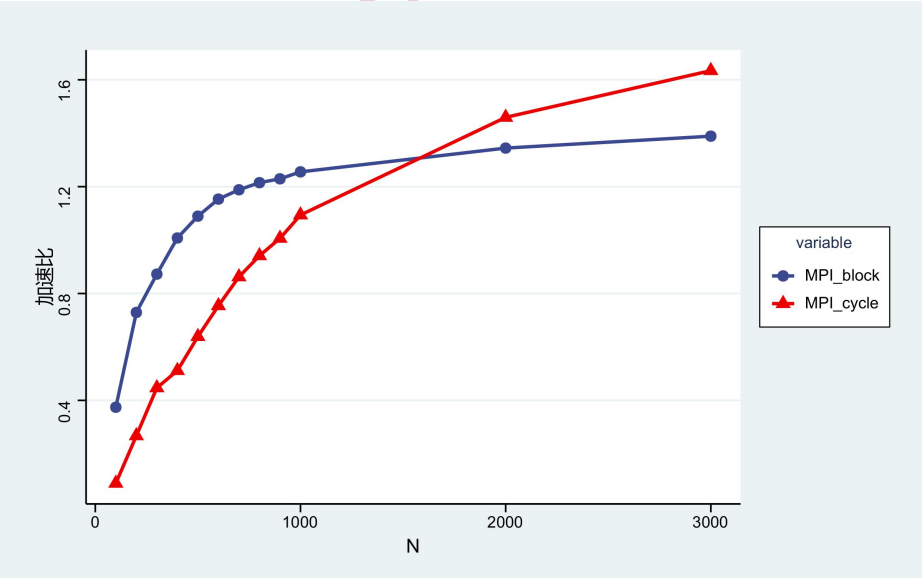

图 3: 块划分和循环划分加速比

由图表可知，在较小规模下，先不比较块划分和循环划分的数据划分方式的效果。我们首先可以发现这两种数据划分方式效果都不如平凡算法效果好。考虑有两个方面的原因：

MPI 是一种进程间通信的机制，在小规模问题中，通信开销可能会比计算开销更加显著。循环划分/块划分优化的高斯消去算法涉及进程间的通信和数据交换，这会引入一定的额外开销。当问题规模较小时，通信开销可能占据了较大的比例，从而影响了算法的性能。

循环划分/块划分优化高斯消去算法将问题分割为多个子问题，并由不同的进程处理各自的子问题。对于小规模问题，划分的粒度可能会较大，导致每个进程处理的任务较少。这样一来，进程间的负载均衡可能不够好，一些进程可能会处于空闲状态，从而导致算法的效率降低。

而比较两种划分方式，在问题较小规模下，块划分加速比更好，而随着问题规模的增大，块划分首先达到了性能的瓶颈——即随着规模的变更，加速比变化幅度小。考虑可能是因为随着消元的推进，负责前面行的进程会逐渐空闲下来，从而让实际的工作进程慢慢减少。而此时循环划分的方式能够更好的实现负载均衡，充分利用线程的计算资源。

2. 数据收发方式

考虑借用流水线的思想，从执行除法操作的进程开始，每个进程都只是接收上一个进程发来的除法结果，并将这些结果转发给下一个进程，然后开始自己的消元任务。理论上，这种方式可以节省大量的时间。

N	serial	MPI_pipeline
100	0.33862	3.65523
200	2.7185	10.3002
300	9.10204	20.3559
400	21.4773	41.9716
500	42.0603	65.8022
600	72.9207	96.792
700	116.331	135.279
800	174.761	185.523
900	250.14	247.211
1000	346.754	315.751
2000	2833.69	1944.59

表 1: 性能测试结果 1(单位:ms)

由于随着规模的改变，时间性能前后量级相差较大，所以我们这里用图表展示测试结果，不难看出数据规模较小的时候，流水线收发效果不是很好，考虑还是由于相比较问题规模，此时使用流水线收发通信开销过大，此时通信和计算之间并不能达到很好的平衡。而随着数据规模的增大，流水线的优势得以体现，作者的实验中，流水线优化性能此时应该还并没有达到性能瓶颈。

3. 问题规模

N	serial	MPI_block	MPI_cycle	MPI_pipeline
100	0.33862	0.90476	3.77945	3.65523
200	2.7185	3.72827	10.1691	10.3002
300	9.10204	10.4317	20.3652	20.3559
400	21.4773	21.3106	42.0001	41.9716
500	42.0603	38.6061	65.8516	65.8022
600	72.9207	63.2224	96.6892	96.792
700	116.331	97.8989	134.915	135.279
800	174.761	143.845	185.646	185.523
900	250.14	203.481	248.521	247.211
1000	346.754	276.217	316.976	315.751
2000	2833.69	2107.88	1941.73	1944.59
3000	9726.77	7002.86	5951.89	5948.32

表 2: 性能测试结果 2(单位:ms)

我们把上述实验的数据统计成一张表格进行展示，我们总结出的一般规律是：优化效果在数据规模太小时，都不能很好的体现，考虑通信开销的缘故导致性价比不高。随着数据规模的不断增大，使得循环划分的优势逐渐体现，并且在实验中作者觉得已经差不多达到了性能瓶颈，而对于流水线划分的方式优化，作者认为此时还没有达到性能瓶颈，猜想如果继续增大问题的规模，性能还会继续增加。

4. 进程数量

我们设置问题规模在 3000 的前提下，分别令进程数等于 2/4/6/8 进行对比。得到的是实验结果加速比对比图如图所示。

我们不难看出随着进程数的增加，各种优化方式的加速比都得到了提升——考虑是计算资源得到了更好的利用造成的这一结果。

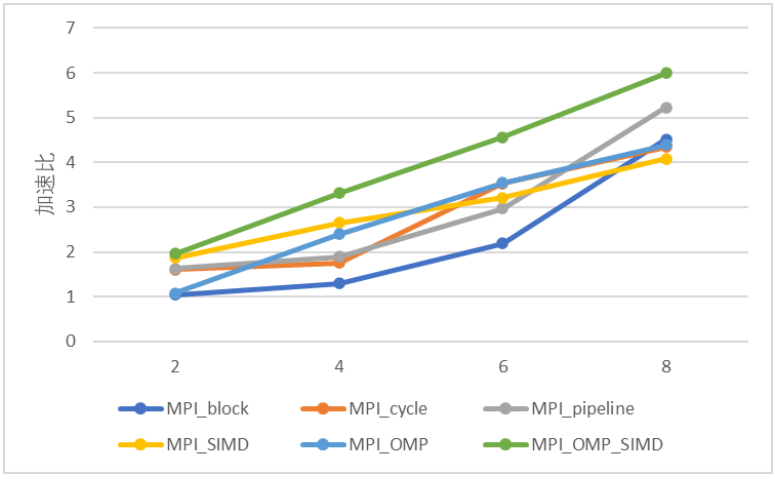

图 4: 块划分和循环划分加速比

5. 并行优化

在考虑和并行优化结合的实验中，我们设计了三种基本思想。

- 和 SIMD 结合
- 和 OMP 结合
- 和 SIMD+OMP 结合

其中，我们令 MPI 的进程数为 8，OMP 使用 8 条线程进行多线程并行，SIMD 采用了四路向量化进行处理。我们将得到的结果绘制图表如下：

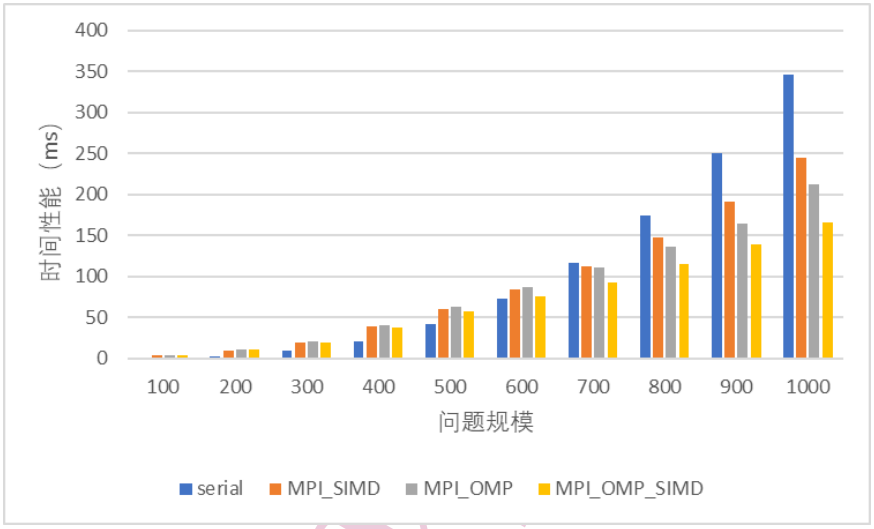

图 5: 并行优化性能测试结果

我们由图表不难得到，在达到一定规模后，优化效果出现了一定的规律——即相对于朴素算法而言，效果呈现为 OMP+SIMD>OMP>SIMD，我们发现随着数据规模的增大，优化效果更加明显，我们下面展示出规模为 2000 和 3000 时对应的加速比。

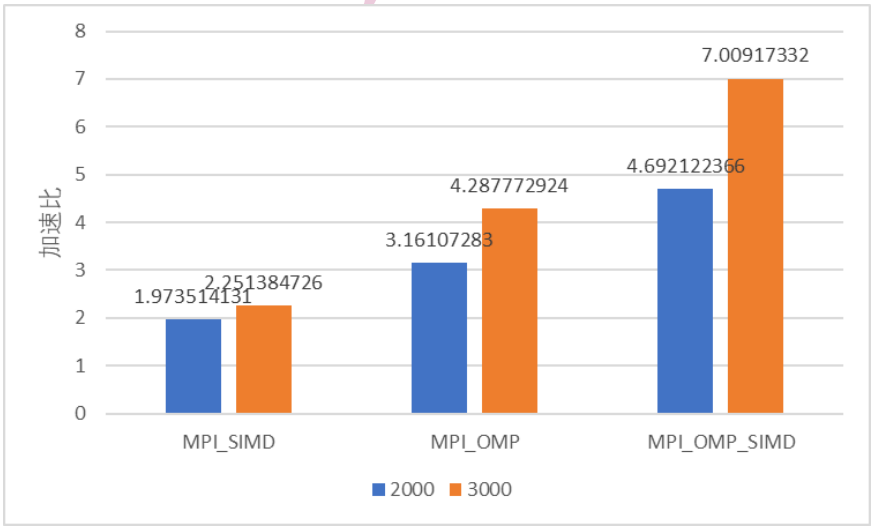

图 6: 并行优化性能测试结果

随着问题规模的扩大加速比确实也会增大，且当我们将 OMP 和 SIMD 方法结合时，会存在加速比约等于两种方法单独优化加速比的乘积的现象，这一点并不难解释，符合我们的认知。

(二) X86

1. 数据划分对比

在本次实验中，考虑负载均衡的问题，探究不同任务的划分方式对于算法性能的影响。由于 MPI 是涉及进程的通信开销，所以我们应该尽可能地利用每一个进程，实验涉及对比了循环划分和块划分这样两种数据划分方式。

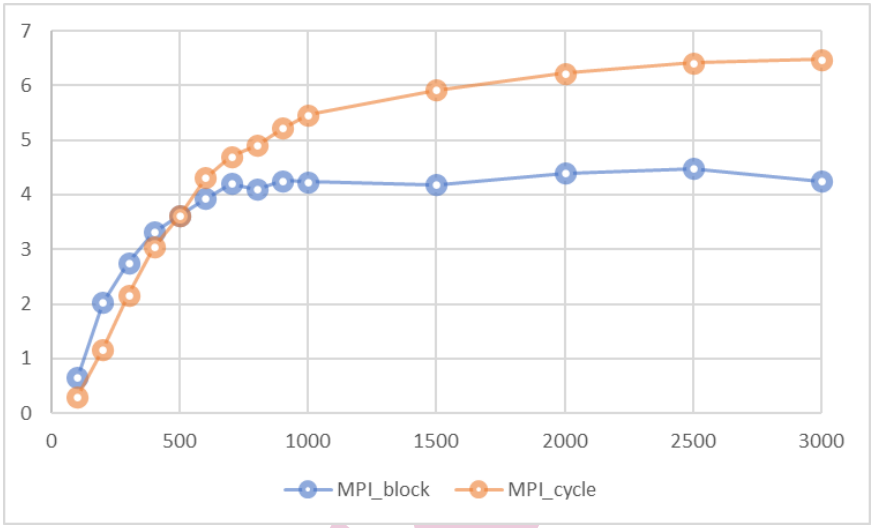

图 7: 优化性能测试结果

我们不难发现，图像在 X86 上的趋势和 ARM 上基本一致，块循环方式会在规模达到 1000 左右达到性能瓶颈，总体来说，当规模随着增大，通信开销占比不再太大时，循环划分这种数据划分方式的优良性能得以体现，其加速比比块划分方式高不少。

2. 数据收发方式

我们把同样的实验迁移到 X86 架构上，由于前后数据规模的变化太大，画图时统一单位会导致效果不明显，所以我们这里采用图表来展示结果。

N	serial	MPI_pipeline
100	1.56398	8.20069
200	12.4596	17.1908
300	41.8384	27.7777
400	98.9403	45.513
500	193.401	65.7848
600	336.782	93.0645
700	534.298	128.134
800	792.936	178.092
900	1130.92	238.955
1000	1549.75	304.846
2000	12547.3	2080
3000	42296.9	6605.21

表 3: 流水线优化性能测试结果 (单位:ms)

由表中可以看出，随着规模的增大，流水线的优势越来越明显。将高斯消去算法与 MPI 流水线相结合，可以将不同的行变换阶段分配给不同的进程或处理节点并行执行。每个进程或节点负责处理一部分矩阵的行，并通过消息传递与其他进程进行数据交换和通信。

3. 问题规模

N	serial	MPI_block	MPI_SIMD	MPI_OMP	MPI_OMP_SIMD
100	1.56	2.39	5.16	6.38	6.70
200	12.46	6.14	10.11	13.58	12.92
300	41.84	15.16	17.01	23.60	21.53
400	98.94	29.82	26.66	43.15	40.31
500	193.40	53.36	39.96	59.04	51.90
600	336.78	85.72	56.66	80.44	65.97
700	534.30	127.10	78.56	103.44	85.54
800	792.94	193.51	105.94	134.60	109.95
900	1130.92	265.65	142.08	183.35	131.90
1000	1549.75	365.86	181.19	217.02	162.68
2000	12547.30	2855.66	1231.45	1220.32	775.82
3000	42296.90	9959.27	3978.63	3689.22	2177.82

表 4: 性能测试结果 3(单位:ms)

结果趋势和 ARM 架构完全一致，我们仍然可以发现，数据规模较小的时候，各种优化方式无论是进程级并行、线程级并行和指令级并行，都不能表现出自己很好的性能。因为可能相比较任务分离所带来的优势，此时的通信开销占比过大，计算资源利用并不充分，反而使得效果更加差劲，甚至不如朴素算法。而当规模变大的过程中，并行优化效果得以体现。

4. 进程数量

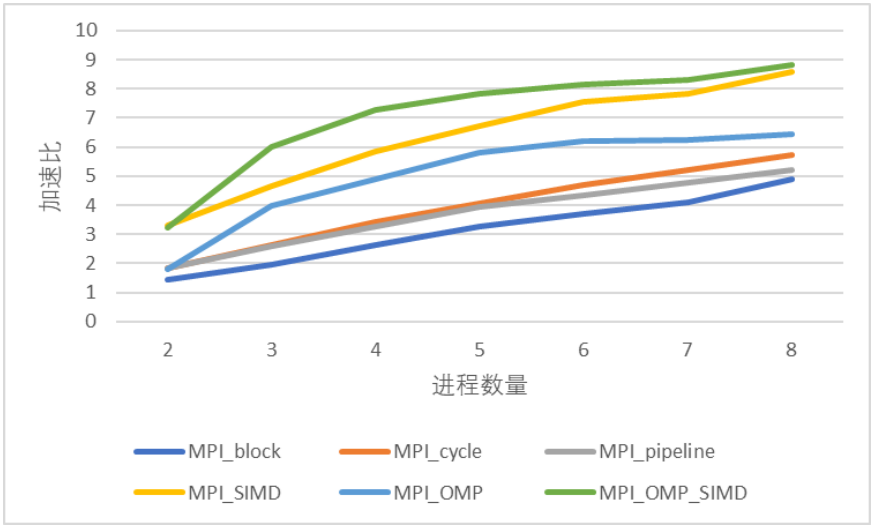

图 8: 进程数量测试结果对比

在进程数量从 2 变化到 8 的过程中，加速比不断增大。考虑增加进程数有以下优势：

- 加速计算：增加进程数可以将计算负载分配给更多的处理单元，从而并行地执行高斯消去算法的各个阶段。这可以提高整体计算速度，加快解决大规模线性方程组的速度。
- 提高可扩展性：通过增加进程数，MPI 并行算法在处理更大的问题规模时能够更好地扩展。这是因为并行计算可以更有效地利用集群或多核系统中的资源，从而实现更高的计算吞吐量。
- 并行通信：增加进程数可以增加通信带宽，进而提高进程间的数据传输速度。这对于高斯消去算法中的数据交换和通信操作非常重要，因为在消元过程中需要进行跨进程的数据传递。

但是，我也猜想过多的进程数量也可能会导致额外的线程调度开销，影响性能。

5. 并行优化

我们仍然设置了三组实验，结合 SIMD，结合 OMP，结合 SIMD+OMP，我们将三组实验结果展示如下：

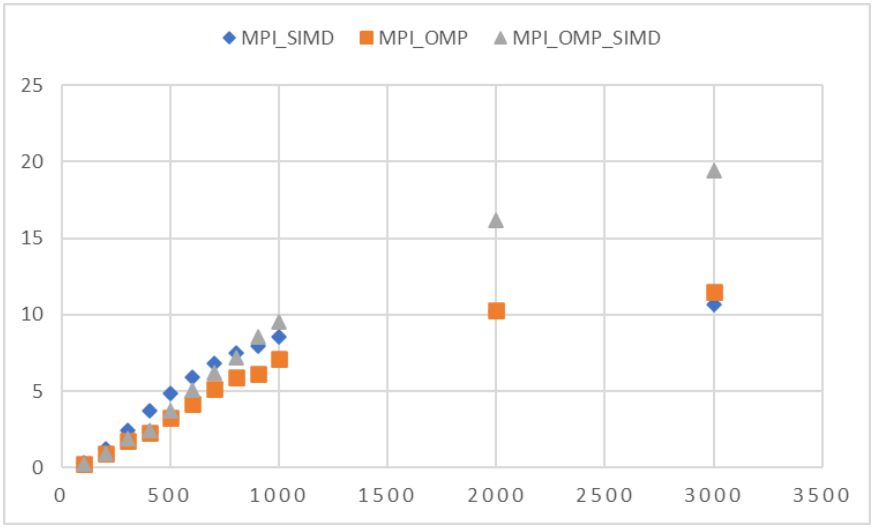

图 9: 优化性能测试结果

我们可以看到综合使用进程级并行、线程级并行和指令级并行，可以得到更好的优化性能，且随着数据规模的增大，加速比也增大，说明并行效果得到了更好的呈现，在数据规模较小的时候，OMP 的加速效果相比较是最差的。

四、 总结

在这个实验中，我针对 Gauss 消元问题考虑了负载均衡的问题，对比了块划分和循环划分两种方式的性能差异。通过实验证明，采用循环划分方式可以显著提高性能，考虑了负载均衡的优化方法在并行计算中起到了关键作用。

在实验中还探究了进程级 MPI 通信方式的影响，并对比了广播和流水线两种方式的性能差异。实验结果表明，流水线方式可以降低等待时延，从而提高性能。

同时，实验还研究了不同进程数量对性能的影响。通过实验发现，当结合多线程优化时，线程数量的增加确实可以提高性能，但是过多的进程数量可能会导致额外的线程调度开销，影响性能。

最后，我进行了多层次的并行优化。首先采用 MPI 多进程优化，将消元操作划分为多个进程并行执行。然后在每个进程内部引入了 OMP 线程级并行，利用多线程并发处理消元操作。此外，我还通过 SIMD 指令集的并行化，根据 ARM 平台使用 NEON 方式，在 x86 平台使用 SSE 方式进行指令级并行。

综合使用进程级并行、线程级并行和指令级并行，结果取得了显著的优化效果。这种综合优化方法可以充分利用多个层次的并行性，加速高斯消元问题的求解过程。整个实验的代码和文档已上传至**超链接**https://github.com/JIAYUE11111/MPI.git。