南開大學
Nankai University

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

姓名：刘宇轩

学号：2012677

专业：计算机科学与技术

2022 年 6 月 16 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

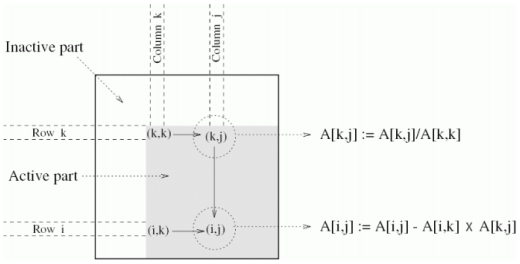

图 1.1: Gauss 消去算法逻辑图

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 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

2 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 处理器	Intel Xeon Processor
CPU 主频	2.6GHz	2.6GHz
L1 Cash	64KB	32K
L2 Cash	512KB	1MB
L3 Cash	48MB	24MB
指令集	Neon	SSE、AVX
核心数	1	1
线程数	1	1

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不同的进程，由于这些数据之间不存在依赖性，因此每个进程只需要各自完成好自己的工作即可，不存在进程之间进行通信的额外开销，只需要在各个进程完成了分配的工作之后进行一次同步。

而当任务下发给不同的进程之后，在各个进程内又可以开启多条线程来处理任务，由于人物之间是不存在数据依赖的，因此不需要进行数据的通信，因此考虑在同一个进程内开启多条线程来处理任务是十分合理并且有效的。

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 MPI 对于 Gauss 消去的算法进行多线程优化，分析在不同任务规模下的性能表现
2. 设计不同的任务划分方式，即循环划分和块划分，对比在不同任务规模下的性能表现
3. 对比不同的消息发送方式，即采用广播和 pipeline，对比在消息发送方式下的性能表现
4. 使用 MPI 配合 SIMD 向量化处理，对比其性能提升
5. 使用 MPI 配合 OMP 进行多线程优化，对比其性能提升
6. 使用 MPI 配合 SIMD 和 OMP 同时进行多进程多线程向量化优化，对比其性能提升
7. 尝试将 openmp 优化方法从 x86 平台迁移到 arm 平台上，分析性能表现
8. 对比采用不同进程数量，MPI 的性能表现情况

以下是详细的实验设计方案

3.1 MPI 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。因此在除法运算和消元之间需要进行以及每一轮次消元结束之后需要进行一次同步。

在考虑使用 MPI 进行多进程的实验设计时，考虑使用单个进程首先来分发任务。然后在外层循环中，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，需要将除法的结果广播给其他所有进程，而其他进程接收除法结果的过程就自然的完成了除法后进程间的同步。

在这本次实验中，我们没有采用主从模式，即所有的进程都会去执行消元的任务，因此可以发现，消元的结果即使每次除法完成后的结果的累计，而除法做完之后都会进行一次广播，因此当最后一行完成之后，所有的除法结果都已经被广播到了各个进程中。因此可以省去最后一次的结果同步的过程。

由于 MPI 是进程级的并行，因此在各个进程内还可以考虑使用线程级和指令集的并行优化，即与之前的 OpenMP 以及 SIMD 进行结合，使得并行的优化效果得到叠加提升。综合考虑所有的优化方式，最后的提升效果应该与进程数量，线程数量和向量化路数的乘积成正比。

3.2 数据划分设计

考虑到 MPI 属于进程级的并行手段，因此如果存在负载不均的情况，其浪费的资源要远远高于线程级的负载不均。因此在本次实验中也从负载均衡的角度出发，设计实验对比了不同数据划分方式的性能差异。主要对比了循环划分和块划分之间的差异。

最朴素的想法就是采用块划分的方式，即给各个进程分配连续的几行数据。但是考虑到 Gauss 消元的过程中，随着消元的进行，前面完成了除法的行将不会再参与后续的运算，也就是从整体的角度来考虑的话，各行之间的计算量是不同的，因此这种划分方式就会导致在后续的消元过程中，负责前面几行的进程处于闲置状态，因此是不能够接近理论加速比的。

而从负载均衡的角度出发，可以使用循环划分的方式。即将数据按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。

3.3 不同数据收发方式的实验对比

考虑到 MPI 是进程级的并行，因此消息通信是在不同进程之间完成的，而进程间的通信开销是远比线程级的通信开销要大的，因此合理的设计进程间的通信方式和限制通信次数对于提高性能优化会有比较明显的效果。

采用最朴素的方式就是当负责除法的进程完成了除法工作之后，将除法的结果依次发放给所有的进程，这个通信的开销是和进程的数量成正比的。而由于通信是需要时间的，因此当有些线程还没有接收到除法的结果的时候，是不能够开展后续的消元工作的。因此这里会存在着比较长的空闲等待。

而 pipeline 的方式就是利用流水线的思想去减少了这种空闲等待的性能损失，即当一个进程收到了除法的结果后，他需要将这个结果转发给下一个进程，之后就可以开始自己的工作了。这样就可以减少了负责除法工作进程的阻塞时间。

综合分析以上两种方法，采用广播的方式的时间开销为 $O(N^3 logN)$，采用 pipeline 的方式的时间开销为 $O(N^3)$。

3.4 问题规模和进程数量的对比

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。因此，设计实验探究在不同数据规模下，多进程并行优化算法的优化效果。此外还将探究在所使用的进程数量不同的情况下，并行算法优化效果的变化情况。由于进程数量的增加，会导致进程间的通信开销同时增加，因此可以预见，并不一定是进程越多性能越好。

4 实验结果分析

4.1 x86 平台

4.1.1 MPI 并行实现及性能对比

为了能够探究 MPI 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，串行算法，MPI 优化，MPI+SIMD 优化，MPI+OMP 优化以及 MPI+SIMD+OMP 优化的时间性能表现。其中 MPI 采用了 8 个进程进行多进程并行，OMP 采用了 8 条线程进行多线程并行，SIMD 并采用了四路向量化处理。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 500。五种算法在不同问题规模下的表现如表1所示。

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
1500	5227.93	1248.94	587.47	587.94	400.51
2000	12547.30	2855.66	1231.45	1220.32	775.82
2500	24559.00	5482.94	2339.07	2357.57	1335.25
3000	42296.90	9959.27	3978.63	3689.22	2177.82

表 1: x86 平台性能随问题规模变化情况

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，特意利用测量的数据计算了四种并行优化算法的加速比随时间的变化情况，如图4.2所示。

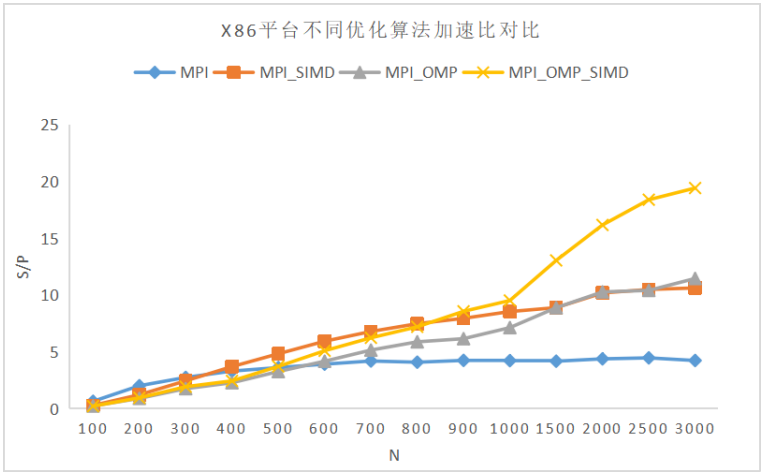

图 4.2: x86 平台各种并行算法加速比随问题规模变化情况

从图像中可以看出，随着问题规模的增加，四种并行优化算法的加速比都呈现一个递增的趋势。就单独采用 MPI 进行多进程并行的优化方式而言，可以看到其随着任务规模的增加，加速比逐渐上升直至平稳在 4.5 左右，而实际开启了 8 条线程，其加速比并没有能够达到理论加速比，其原因在于通信开销以及其余没有进行多进程并行的部分。

在实验中还对比了基于 MPI 和 SIMD、OMP 的结合，并以 MPI 的性能为 baseline，分析增加了其他优化方式的性能提升。

由于 SIMD 采用了 4 路向量化的手段，因此其相对于 baseline 的理论加速比应该能达到四倍。但是实验表明，SIMD 实际的加速比只达到了 2.0-2.5 之间，如图4.3所示。这也和之前的实验向契合，证明在 Gauss 消元问题上，由于其他未能够进行 SIMD 向量化运算部分的影响，其理论加速比只能达到这个数值。

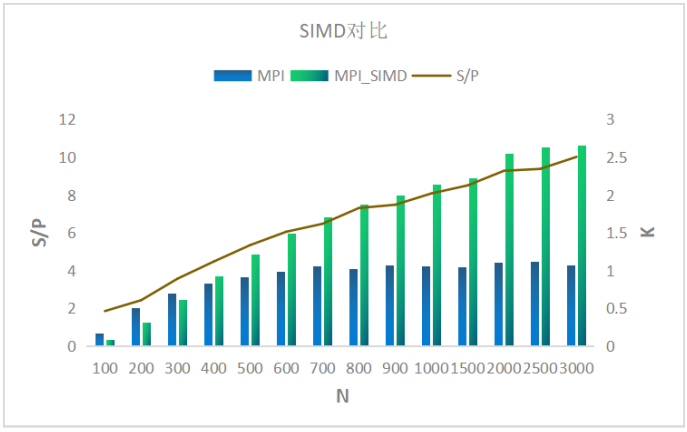

图 4.3: x86 平台 SIMD 优化对比

由于 OMP 拉起了 8 条线程，因此其相对于 baseline 的理论加速比应该能够达到 8 倍。但是实验表明，当问题规模达到 3000 的时候，这个性能提升也只达到了 2.7 倍左右，如图4.4所示。但是可以看出，随着问题规模的增加，这个加速比还有提升的趋势。分析原因，由于 MPI 多进程分配任务，每个进程上的任务只有总任务的 $\frac{1}{8}$，因此，对于每个进程而言，即使问题规模达到了 3000，分配给每个进程的任务也只有 400 左右，而通过之前的实验可以表明，在 400 左右的任务规模下，其加速比很难达到 OpenMP 的理论加速比。并且其趋势也能够表明，随着问题规模的增加，这个加速比应该是会继续上升的。

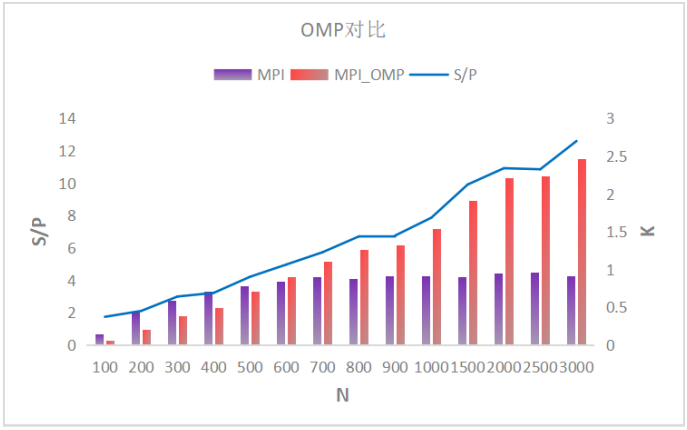

图 4.4: x86 平台 OMP 优化对比

最后在实验中尝试了将 MPI 和 SIMD、OpenMP 融合到一起，同时启用多进程多线程向量化优化. 实验结果如表2所示。在问题规模达到 3000 的时候，取得了将近 20 倍的性能提升。这和单独采用某一种优化方式取得的性能提升的乘积基本一致。

N	200	300	400	500	600	700	800	900	1000	2000	3000
S/P	0.964	1.943	2.454	3.726	5.105	6.246	7.211	8.574	9.527	16.173	19.421

表 2: MPI+OMP+SIMD 优化加速比

4.1.2 数据划分对比

在本次实验中，从负载均衡的角度触发，探究不同的任务划分方式对于算法性能的影响。由于 MPI 是进程级的并行方式，因此进程间的通信开销是很大的，所以要尽可能地利用每一个进程，并且减少进程之间的通信次数。实验对比了两种不同的任务划分方式，块划分和循环划分，实验结果如图4.5所示。

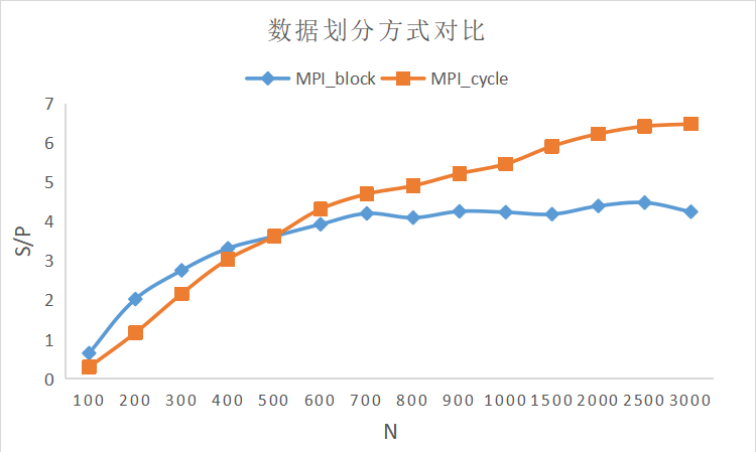

图 4.5: x86 平台数据划分方式对比

从图像中可以看出，随着任务规模的增加，这两种划分方式的加速比都逐渐增加，但是随着任务规模的增加，循环划分的方式逐渐反超了块划分的方式。并且块划分方式率先达到了性能瓶颈，并且只达到了理论加速比的一半。这很符合预期，从整个过程来看，采用块划分的方式，随着消元的推进，负责前面行的进程会逐渐空闲下来，因此在后续的计算过程中，实际工作的进程会越来越少。平均下来每个进程只有一半的时间在有效工作，因此其加速比只达到了 4 左右。而对于循环数据划分的方式而言，由于其从计算量的角度去划分任务，因此整体来看每个线程的有效工作时间几乎相同，达到了比较好的负载均衡，充分利用了每个线程的计算资源，因此其加速比逐步逼近理论加速比 8。

4.1.3 数据收发方式对比

考虑到 MPI 在接收数据的时候处于阻塞状态，因此对于进行除法操作的进程而言，如果按照顺序向各个进程发送除法结果的话，那么该进程就需要等待所有的数据全部发送完成后才能开始处理自己的消元任务，而这个是完全浪费的时间。因此可以考虑借用流水线的思想，即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，并将这个结果转发给下一个进程，然后就可以开始自己的消元任务。这样可以有效减少进程等待而浪费的时间。实验结果如图4.6所示。

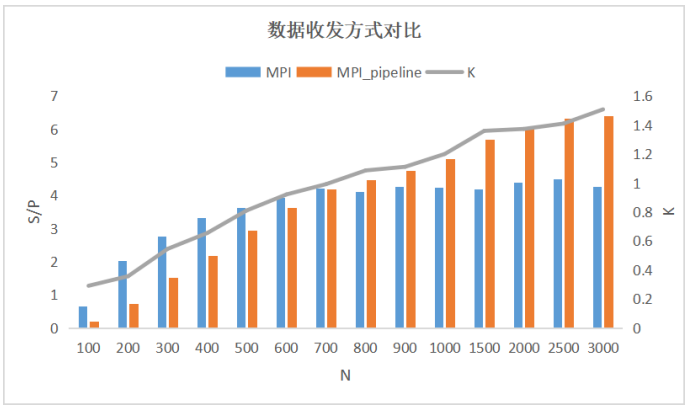

图 4.6: x86 平台数据收发方式对比

从图像中可以看出，随着问题规模的增加，采用 pipeline 方式的数据收发可以提高性能瓶颈，普通的收发方式加速比只能够达到 4 左右，而采用了 pipeline 的加速比可以达到 6 左右，已经能够接近理论加速比了。

4.1.4 进程数量对比

在本次实验中，还探究了在不同进程数量下的性能优化效果。为了能够很好地展现出 MPI 的优化效果，问题规模选定为 1000，线程数为 8，向量化路数为 4。由于实验资源有限，因此对于进程数量的尝试只尝试了 2-8，为了能够尽可能还原真实的应用场景，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。由于在同一个节点内，完全可以采用进程内线程级的并行，而采用进程间进程级的并行无疑会导致不必要的性能损失。因此在实验的过程中，我们在不同节点之间采用进程级并行，在同一个节点内采用线程级并行，在同一个线程内采用指令集并行。实验结果如图4.7所示。

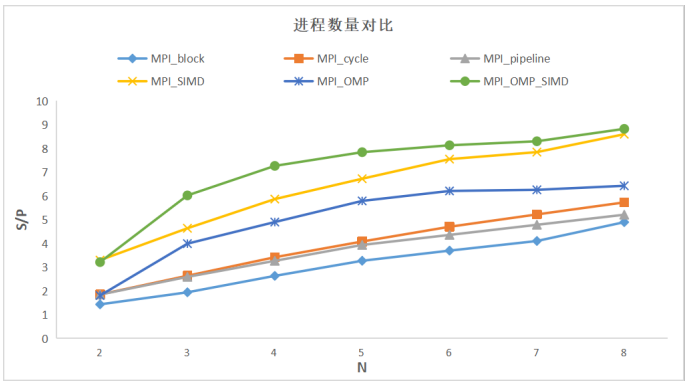

图 4.7: x86 平台进程数量对比

随着进程数量的增加，可以看到单纯进行进程级并行的三种方法呈现出随着进程数线性递增的趋势。但是加入了线程级并行和指令集并行之后，可以发现这种线性的增长关系被破坏。进程数量对于指令集并行的影响并不大，指令集并行的优化方式也随着进程数呈现线性递增趋势。但是线程级并行就遇到了一定的瓶颈，当进程数超过 6 之后，线程级并行的 OMP 便不再有性能提升了，因此这也大大限制了综合线程级并行和指令集并行的优化方法的性能提升。

造成这种现象的原因应该是各个进程分配的任务同进程数量成反比，由于每个进程分配的任务变少，因此线程级并行产生的额外线程同步和调度开销相对于计算占比会变得更加明显，额外开销同优化之间相抵消就呈现了一个瓶颈状态，或许随着线程数量得增加，还可能出现下降的趋势。

4.2 arm 平台迁移

4.2.1 MPI 并行实现及性能对比

为了能够探究 MPI 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，串行算法，MPI 优化，MPI+SIMD 优化，MPI+OMP 优化以及 MPI+SIMD+OMP 优化的时间性能表现。由于 ARM 平台只有两个节点，因此我们只开启两个进程，其中 MPI 采用了 2 个进程进行多进程并行，OMP 采用了 8 条线程进行多线程并行，SIMD 并采用了四路向量化处理。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 500。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，特意利用测量的数据计算了四种并行优化算法的加速比随时间的变化情况，如图4.8所示。

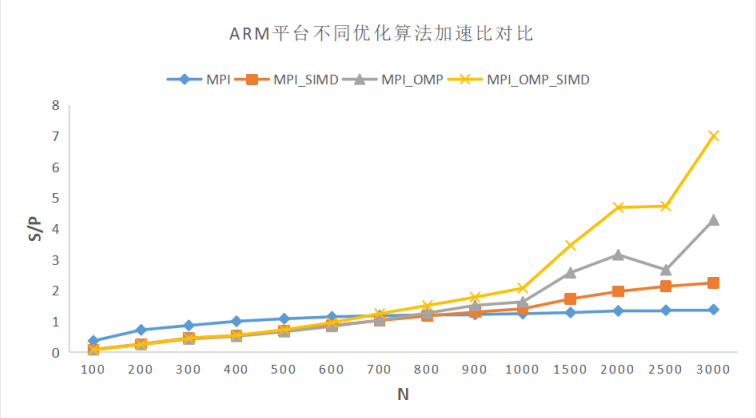

图 4.8: ARM 平台各种并行算法加速比随问题规模变化情况

从图像中可以看出，随着问题规模的增加，四种并行优化算法的加速比都呈现一个递增的趋势。就单独采用 MPI 进行多进程并行的优化方式而言，可以看到其随着任务规模的增加，加速比逐渐上升直至平稳在 2 左右，基本达到理论加速比。

在实验中还对比了基于 MPI 和 SIMD、OMP 的结合，并以 MPI 的性能为 baseline，分析增加了其他优化方式的性能提升。

由于 SIMD 采用了 4 路向量化的手段，因此其相对于 baseline 的理论加速比应该能达到四倍。但是实验表明，SIMD 实际的加速比只达到了 1.6，如图4.9所示。

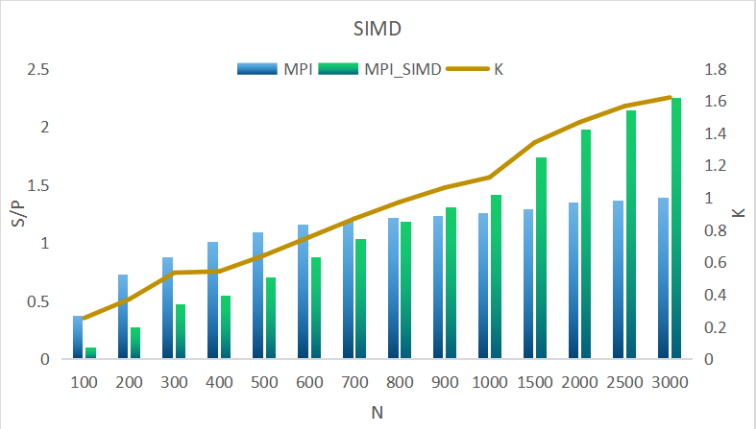

图 4.9: ARM 平台 SIMD 优化对比

由于 OMP 拉起了 8 条线程，因此其相对于 baseline 的理论加速比应该能够达到 8 倍。但是实验表明，当问题规模达到 3000 的时候，这个性能提升也只达到了 3 倍左右，如图4.10所示。但是可以看出，随着问题规模的增加，这个加速比还有提升的趋势。并且其趋势也能够表明，随着问题规模的增加，这个加速比应该是会继续上升的。

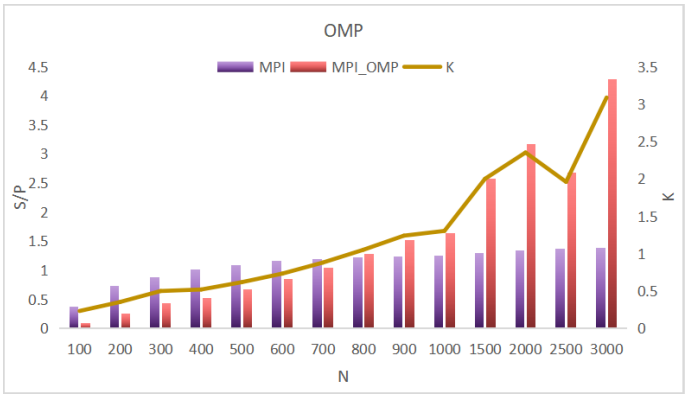

图 4.10: ARM 平台 OMP 优化对比

4.2.2 数据划分对比

在本次实验中，从负载均衡的角度触发，探究不同的任务划分方式对于算法性能的影响。由于 MPI 是进程级的并行方式，因此进程间的通信开销是很大的，所以要尽可能地利用每一个进程，并且减少进程之间的通信次数。实验对比了两种不同的任务划分方式，块划分和循环划分，实验结果如图4.11所示。

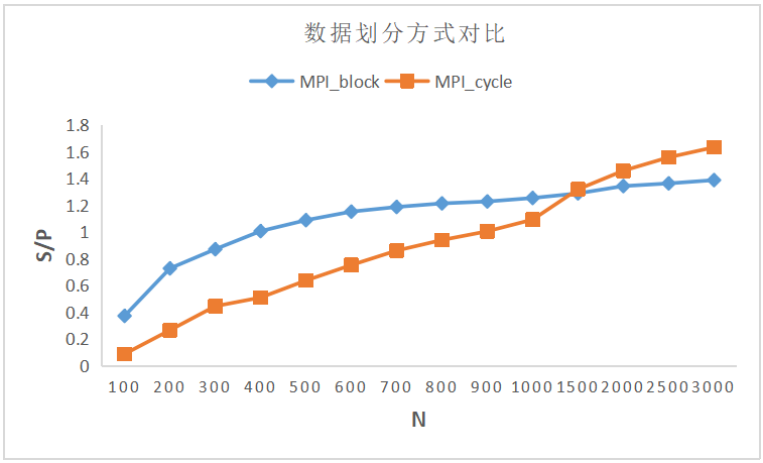

图 4.11: ARM 平台数据划分方式对比

从图像中可以看出，随着任务规模的增加，块划分方式率先达到了性能瓶颈，并且只达到了理论加速比的一半左右。这很符合预期，从整个过程来看，采用块划分的方式，随着消元的推进，负责前面行的进程会逐渐空闲下来，因此在后续的计算过程中，实际工作的进程会越来越少。平均下来每个进程只有一半的时间在有效工作。而对于循环数据划分的方式而言，由于其从计算量的角度去划分任务，因此整体来看每个线程的有效工作时间几乎相同，达到了比较好的负载均衡，充分利用了每个线程的计算资源，因此其加速比逐步逼近理论加速比 2。

4.2.3 数据收发方式对比

本次实验中同样在 ARM 平台上对比了不同数据收发方式的性能差异，实验结果如图4.12所示。

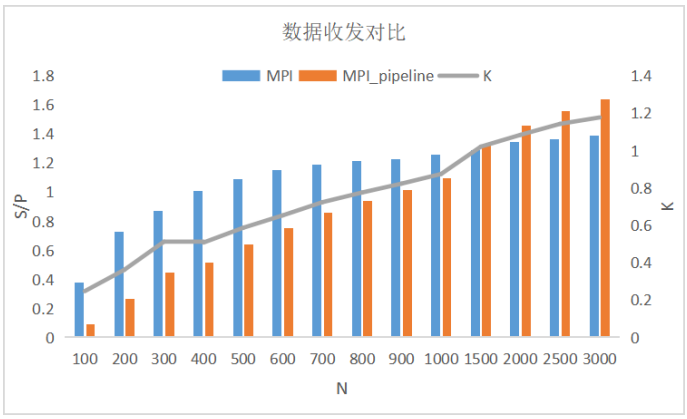

图 4.12: ARM 平台数据收发方式对比

从图像中可以看出，随着问题规模的增加，采用 pipeline 方式的数据收发可以提高性能瓶颈，普通的收发方式加速比只能够达到 1.2 左右，而采用了 pipeline 的加速比可以达到 1.6 左右。同时也注意到了，由于 ARM 平台上只启用了两个进程，因此不同的数据收发方式在本次实验中的效果并不明显，而且当数据规模较小的时候，由于 pipeline 的方式增加的数据收发的次数，反倒是表现出了更差的性能。

5 总结

在本次 MPI 多进程并行实验中，基于 Gauss 消元问题，对于过程中的消元操作采用 MPI 多进程优化，并在进程内增加了线程级的 OMP 并行，在线程级又增加了指令集的 SIMD 并行，在 ARM 平台上采用 neon 的方式，在 x86 平台上采用 SSE 的方式，探究了其并行优化效果。最终综合了进程级并行、线程级并行和指令集并行，取得了十分显著的优化效果。除此之外，还基于负载均衡考虑对比了块划分和循环划分的性能差异，通过实验可以验证，考虑了负载均衡循环划分方式能够取得显著的性能提升。在本次实验中，还针对进程级的 MPI 通信方式进行了探究，对比了在数据收发时采用广播和流水线两种方式的性能差异，实验证明流水线的方式能够很好的降低等待时延，提高性能。本次实验还探究了不同进程数量对于性能的影响，通过实验发现当结合多线程优化时，一味增加进程数量可能会导致线程调度的额外开销影响显著。本次实验的相关代码和文档已经上传至GitHub。