南開大學
Nankai University

计算机学院

并行程序设计实验报告

高斯消去算法和 LU 分解算法的 MPI 并行化研究

姓名：蔡沅宏 黄明洲

学号：2213897 2211804

专业：计算机科学与技术

2024 年 6 月 9 日

目录

1 任务描述

1.1 研究任务及报告说明

本研究旨在分别对 gauss 方法和 LU 算法进行并行化优化，运用 mpi 方法进行性能提升。通过实验数据分析，同时结合前几次实验中使用的多线程和 simd 方法以寻求更好的算法性能。最后，根据实验结果对两种算法进行横向对比，分析它们适用的情况。

本报告主要展示小组成员蔡沅宏针对 LU 算法展开的一系列研究与分析，报告将在结尾模块展示小组成员完成各自任务后针对这两种算法应用的合作分析结果。

1.2 研究问题说明

1.2.1 Guass 消去

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。Gauss 消去可用来进行线性方程组求解，矩阵求秩，以及逆矩阵的计算。

高斯消元法计算公式：

- 将方程组表示为增广矩阵 [A|b]，通过消元操作将系数矩阵 A 化为上三角矩阵；
- 利用回代求解出方程组的解 x；

Gauss 消去示意图如1.1所示：

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

图 1.1: Gauss 消去示意图

1.2.2 LU 算法

LU 分解是一个重要的数值分析方法，它将矩阵分解为下三角矩阵 L 和上三角矩阵 U 。在科学计算、工程技术等领域中具有广泛应用。通过 LU 分解将原始线性方程组转化为两个简单的三角形方程组，从而可以更容易地求解线性方程组。同时可以用 L 和 U 的对角线元素相乘得到原矩阵的行列式。此外，通过 LU 分解后，可以快速求解矩阵的逆矩阵，从而可以方便地进行矩阵运算。

LU 分解算法计算公式：

- 将系数矩阵 A 分解为两个矩阵 L 和 U 的乘积，其中 L 为下三角矩阵，U 为上三角矩阵；
- 利用 LU 分解可以简化方程组求解的过程，提高计算效率；

LU 算法示意图如1.2所示：

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}=\begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & 1 \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix}$$

图 1.2: LU 算法示意图

2 实验环境

此实验将对 LU 算法在 x86 和 ARM 两个平台下进行算法并行优化及分析，下表是对于实验环境详细展示，如表1所示：

	ARM	**x86**
CPU 型号	华为鲲鹏 920 服务器	12th Gen Intel Core i7-12700H
CPU 主频	2.6Ghz	2.3Ghz
一级缓存	64KB	1.2MB
二级缓存	512KB	11.5MB
三级缓存	48MB	24.0MB
指令集	NEON	SSE、AVX、AVX512

表 1: 实验环境

3 实验设计

在 LU 分解中，可以并行化的部分主要在于对不同行和列的独立处理。MPI（Message Passing Interface）非常适合处理需要分布式计算的任务，特别是矩阵运算中：

1. 行或列的分配：将矩阵的行或列分配给不同的计算节点，每个节点负责计算自己的块。

2. 通信开销：在某些步拆解时，需要进行行/列的通信来共享计算中间值（需要广播已计算好的行或列值）。

MPI 程序设计步骤如下：

1. 矩阵分块：将矩阵划分为多个块，每个块分配给不同的 MPI 进程。例如，可以行分解（行块分配）或者行列分解形式。

2. 局部计算：每个进程在其本地块上执行 LU 分解的相关计算。

3. 数据共享：需要在某些步通过 MPI 通信原语（比如 MPI_Bcast, MPI_Scatter, MPI_Gather）来同步数据。这部分的同步和传递是能否高效进行并行计算的关键。

4. 合并结果：最终将所有计算块合并，得到完整的 (L) 和 (U) 矩阵。

事实上，在上一章对于 LU 算法的多线程编程的思想可以用在 mpi 编程上面，只需要在加上有关进程间通信的内容即可。

以下展示出简单的 LU 分解算法的 MPI 实现的程序伪代码：

Algorithm 1 LU Decomposition using MPI

Input: A matrix m of size $N \times N$, number of processes $size$, rank of the current process $rank$
Output: L and U matrices such that $m = L \times U$

function OWNERPROCESS($row, size$)
  $rowsPerProcess \leftarrow \lfloor \frac{N}{size} \rfloor$
  $extraRows \leftarrow N \mod size$
  if $row < (rowsPerProcess + 1) \times extraRows$ **then**
    return $\lfloor \frac{row}{rowsPerProcess+1} \rfloor$
  else
    return $\lfloor \frac{row - extraRows}{rowsPerProcess} \rfloor$
  end if
end function
function BLOCKSIZE($rank, size$)
  $rowsPerProcess \leftarrow \lfloor \frac{N}{size} \rfloor$
  $extraRows \leftarrow N \mod size$
  if $rank < extraRows$ **then**
    return $rowsPerProcess + 1$
  else
    return $rowsPerProcess$
  end if
end function
function LUDECOMPOSITION($matrix, L, U, rank, size$)
  $n \leftarrow$ length of $matrix$
  Initialize L and U as $n \times n$ zero matrices
  for $i \leftarrow 0$ to $n - 1$ **do**
    if $rank ==$ OWNERPROCESS($i, size$) **then**
      $L[i][i] \leftarrow 1.0$
      for $j \leftarrow i$ to $n - 1$ **do**
        $sum \leftarrow 0.0$
        for $k \leftarrow 0$ to $i - 1$ **do**
          $sum \leftarrow sum + L[i][k] \times U[k][j]$
        end for
        $U[i][j] \leftarrow matrix[i][j] - sum$
      end for
      for $j \leftarrow i$ to $n - 1$ **do**
        if $j \geq i$ **then**
          $value \leftarrow U[i][j]$
          Broadcast $value$ to all processes
        end if
      end for
    else
      Initialize row as a vector of size n
      if $rank >$ OWNERPROCESS($i, size$) **then**

```
            if rank == ownerProcess(i, size) + 1 then
                Copy L[i] to row
            else
                Broadcast row from owner process
                for j ← i + 1 to n − 1 do
                    sum ← 0.0
                    for k ← 0 to i − 1 do
                        sum ← sum + L[j][k] × row[k]
                    end for
                    L[j][i] ← (matrix[j][i] − sum)/row[i]
                end for
            end if
        else
            Copy U[i] to row
        end if
    end if
end for
end function
```

根据以上伪代码将 mpi 代码成功实现后，首先应该对它进行进一步的深入探索，使其性能最优。

其次，在之前几次实验的基础上，我们可以尝试着将 MPI、多线程、simd 方法进行结合以获得更高的算法优化收益。在理论分析上，MPI 与多线程这种结合应该可以很好的进行矩阵运算的并行。MPI 处理进线程间的通信，多线程加速每个块的局部运算，从而达到高效地利用各个计算资源的效果。MPI 与 SIMD 的结合也应该适合处理相关的 LU 分解问题。MPI 处理大块通信，SIMD 加速单一操作中的多数据并行，提高效率。那么，如果能实现综合利用 MPI 、多线程和 SIMD，我希望能通过这三个技术的综合应用在 LU 分解算法中达到最优效果。

在本次实验中，为能全面深入的了解和探究 MPI 等并行化方法的性能，我将在实验中进行多方面的探究：

1. MPI 方法研究：
   - 尝试不同任务划分策略
   - 尝试不同的工作方式
2. MPI 性能探究
   - 改变矩阵的大小，观测算法运行时间的变化
   - 改变进程数，观测算法运行时间的变化
3. 使用 MPI 与 simd 结合的方式，探究性能优化
4. 使用 MPI 与多线程 OpenMP 结合的方式探究性能优化
5. 将 MPI、simd、OpenMP 结合进行多进程多线程向量化优化，对比其性能提升

本实验涉及的所有代码都已上传至Github。

4 实验过程及结果分析

为方便测算与分析，我在实验中使用的矩阵维度均为 2^n，以下报告中涉及维度时所述均为 n（表示是 2 的 n 次方）。

我们通过逐步改变矩阵大小，测算不同算法完成 LU 分解所需时间，来对不同算法的性能进行比较。同时，对不同规模下，相同算法的优化比的变化趋势进行分析；对相同规模下，不同算法的实际与理论优化比的不同进行分析，以进一步分析深层原因。

同时需要注明的是，除对于使用不同进程数的效果的分析外，其余模块的分析均建立在使用线程数为 8 的并行算法基础上。

4.1 MPI 方法探索

在按照初级代码进行实验后，我发现在我设置的默认进程数为 4 的情况下，实验的优化效果并不是很好，甚至与串行算法相比还有所倒退，分析原因，这可能是因为，按照上诉所说的分割方法，每个进程负责的元素所依赖的数据都不在本进程中，所以需要在每轮计算开始前。拿到自己所需要的数据，这会带来极大的通信开销，进程数增大带来并行化效果不足以抵消的通信额外开销，为得到更好的 MPI 性能，我们需要对 MPI 进行进一步的研究。

4.1.1 任务划分

首先，我们要先明白为什么在上述划分方法下算法的效果不佳。分析发现，其最根本原因，还是因为数据结构与并行划分方式不太契合，比如：本来同一行内的元素，在地址空间上是放在一个块一起的，整个通信则可以传输一个块的数据。但是按照这里采用的列划分方式，导致了传送受阻。所以这种划分方式在 MPI 实现下难以产生较好的并行效果。

并且，这样的划分方式虽然简单，但是也会带来可能比较严重的负载不均问题，这进一步导致了最后实验的效果较差。

根据实验指导书上给出的多种数据划分方法，为了能更好的针对数据依赖问题，我在实验中进行了多种尝试，如行、列块划分和列循环划分的方式，对 MPI 性能进行研究。在不同数据规模下，我以效果最差的列划分为 baseline，算出其余两种方法的相对优化比，得到以下折线图4.3：

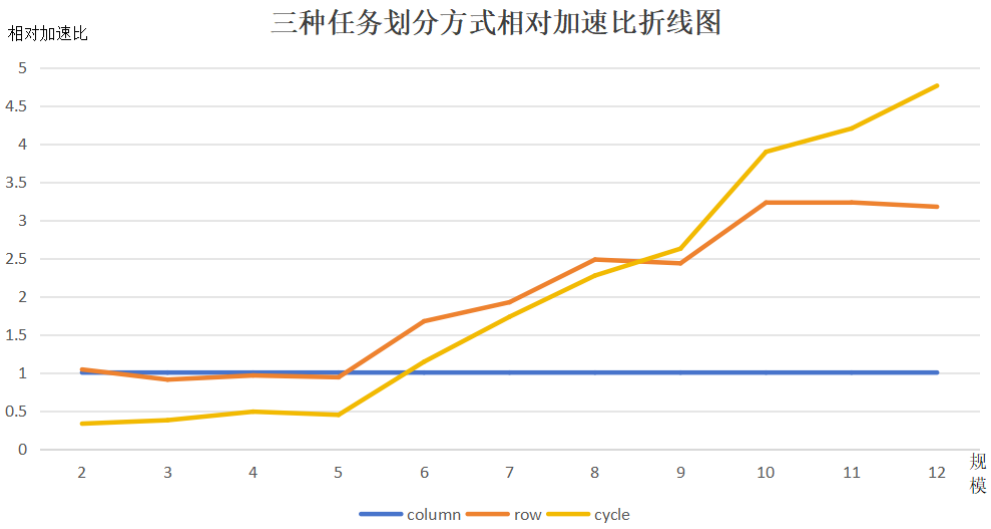

图 4.3: 不同任务划分方法在 x86 架构上运行的相对加速比折线图

从实验结果中，我们可以看出行划分的加速效果远好于列划分的加速效果，这是由于在内存中，矩阵的数据是按行存储的，因此行划分更符合数据访问模式，可以更好地利用缓存，减少缓存未命中的情况，从而提高内存访问效率。同时，在 LU 分解算法中，存在数据依赖性，后续计算需要依赖前面的计算结果。使用行划分的方法，使得每一整行均由某一个进程处理，更好地利用了这种数据依赖性，减少了进程间的通信开销。

而循环划分的方式又优于行划分，这是由于块划分的方式，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。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。所以当矩阵规模越大，循环划分的优势越明显。

4.1.2 工作方式

实验指导书上（以 Guass 消去为例）说到：流水线算法和普通的块划分的区别在于一个进程负责行的除法运算完成之后，并不是将除法结果一对多广播给所有后续进程，而是（点对点）转发给下一个进程；当一个进程接收到前一个进程转发过来的除法结果时，首先将其继续转发给下一个进程，然后再对自己所负责的行进行消去操作；当一个进程对第 k 行完成了第 $k-1$ 个消去步骤的消去运算之后，它即可对第 k 行进行第 k 个消去步骤的除法操作，然后将除法结果进行转发，如此重复下去，直至第 $n-1$ 个消去步骤完成。

对于 LU 分解算法而言，亦可以使用流水线思想进行代码的优化。考虑到在 LU 分解中，后续步骤依赖于前一步完成的结果。MPI 在接收数据的时候会处于阻塞状态，这会带来不小的时间开销，但是实际上，某些独立计算可以在数据准备好之前就提早开始。所以，可以设置多个阶段的流水线，每个阶段处理一部分任务。当一个任务完成时，立即将数据传递到下一个阶段，减少空闲等待时间。实验结果如下图4.4所示：

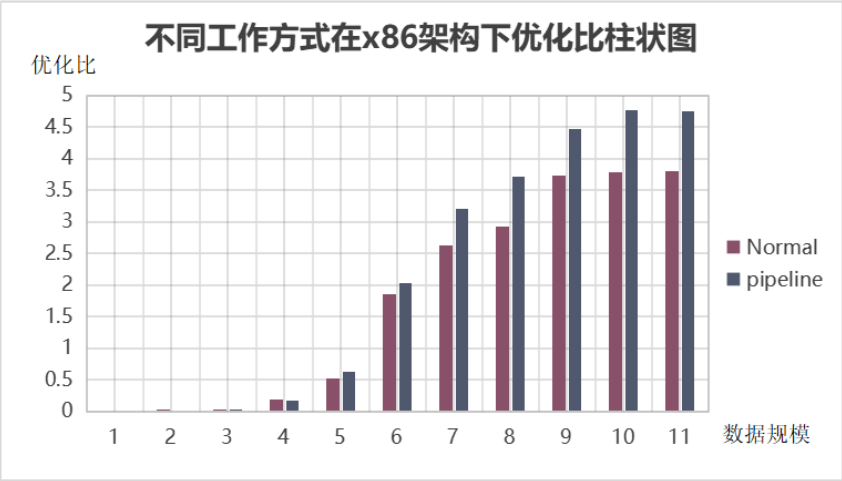

图 4.4: 不同工作方式在 x86 架构下效果展示

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，普通方法加速比只能够达到 4 左右，而采用了 pipeline 的加速比可以达到 5 左右，效果优于普通方法。

4.2 MPI 性能探究

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。数据规模与以往几次实验一样，仍选择以 2 的次方的形式选择。

以下实验把代码迁移到华为鲲鹏服务器上进行测试，实验分析基于在 ARM 平台上的运行结果。

以下是不同程序在 ARM 架构下的运行结果：

数据规模	ARM 架构下程序运行时间（单位：ms）				
	serial	mpi	mpi_neon	mpi_omp	mpi_omp_neon
2	0.00097413	1.8239	2.00434	103.542	107.363
3	0.0016665	4.00329	4.25349	180.345	190.692
4	0.00420614	5.18584	4.80437	160.772	180.374
5	0.0190396	10.7548	7.35855	50.865	71.3749
6	0.12656	19.1593	9.26033	57.864	66.8127
7	1.01232	30.9336	15.896	54.4081	62.015
8	8.46296	58.4822	27.0783	61.1737	47.143
9	69.7063	133.449	57.7468	68.1163	50.1718
10	619.605	321.126	129.6045	132.244	83.7138
11	5763.13	1215.49	464.947	311.176	217.012

表 2: ARM 平台下实验数据

4.2.1 矩阵大小对 MPI 性能的影响

首先，我们先对实验可能出现的情况进行分析，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。

根据表2的实验数据，我们可以得到以下表3：

数据规模	2	3	4	5	6
MPI 加速比	0.000534092	0.000416283	0.000811082	0.001770335	0.006605669
数据规模	7	8	9	10	11
MPI 加速比	0.03272558	0.144710014	0.522344116	1.92947628	4.741404701

表 3: ARM 平台下 MPI 加速比

从表中，我们可以看到，当数据规模较小时，MPI 的加速比确实很低，而随着数据规模的渐渐增大，加速比也逐渐显现出来，这与之前的预测结果相同。

而同时，我们也发现，MPI 的加速比也无法达到我们所设的进程数，而是会停留在 5 左右，这是因为采用并行优化计算的同时，多进程也会带来进程间通信的开销，此外，进程间仍存在负载不均衡等小问题，会影响优化效果。因此，优化比无法达到 8，在实验中停留在 5 左右。

此外，我们还发现，与上一章的多线程编程比较起来，多进程一开始的优化比的会比多线程的低很多，并且，从负加速效果到正加速效果所需要的数据规模也更大，这可能与进程间通信的开销比线

程创建和销毁的开销来得更大，事实上，也确实如此，多线程编程是基于同一个进程内的，所以多线程编程是更加细粒度的编程方法，在以上单独使用的情形下，确实应该会有着比 MPI 更好的效果。

规模	ARM 运行时间（ms）		加速比
	未优化	openmp	
2	0.00097413	0.083433	0.01167
3	0.0016665	0.114141	0.0146
4	0.00420614	0.121343	0.03466
5	0.0190396	0.15133	0.12582
6	0.12656	0.214552	0.58988
7	1.01232	0.413532	2.44798
8	8.46296	2.1353	3.96336
9	69.7063	12.4637	5.59275
10	619.605	107.213	5.7792
11	5763.13	1034.32	5.5719

表 4: 上一章 OpenMP 编程的实验数据

4.2.2 进程数对 MPI 性能的影响

我们仍然先对实验可能出现的结果进行分析。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。

在本次实验中，为了能够很好地展现出 MPI 的优化效果，问题规模选定为 11（即矩阵维数为 2048），由于华为鲲鹏服务器的限制，因此进程数量尝试了 2-8。实验结果如图4.5所示。

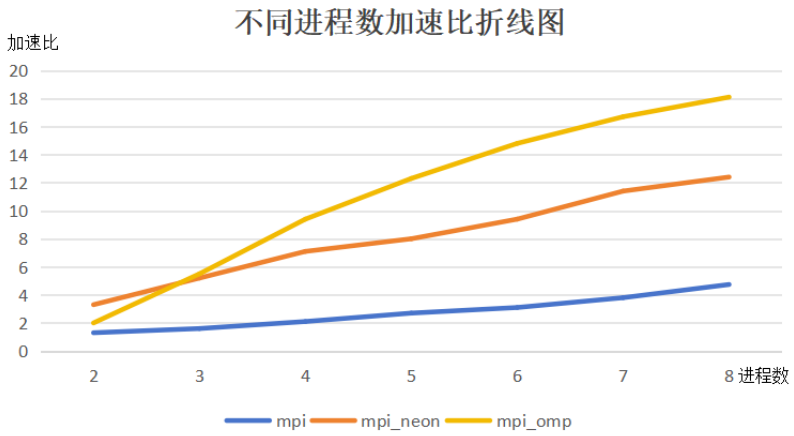

图 4.5: 不同任务划分方法在 x86 架构上运行的相对加速比折线图

随着进程数量的增加，可以看到只使用了 MPI 的方法呈现出随着进程数线性递增的趋势。但是加入了多线程和指令集并行之后，可以发现这种线性的增长关系被破坏。但是我们也可以看待，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仍基本呈现出随着进程数增加线性递增的趋势。

而 MPI 与多线程的并行就产生了不是线性增加的现象，当进程数超过 6 之后，性能提升速度放缓，造成这种现象的原因可能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以及线程创建和销毁的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使得优化比不再明显增加。

4.3 MPI 与 simd、OpenMP 结合

根据以上实验数据，我们可以得到以下表5：

数据规模	优化比			
	mpi	mpi_neon	mpi_omp	mpi_omp_neon
2	0.000534092	0.00048601	9.40807E-06	9.07324E-06
3	0.000416283	0.000391796	9.24062E-06	8.73922E-06
4	0.000811082	0.000875482	2.61621E-05	2.3319E-05
5	0.001770335	0.002587412	0.000374316	0.000266755
6	0.006605669	0.0136669	0.002187198	0.001894251
7	0.03272558	0.063683946	0.018606053	0.016323793
8	0.144710014	0.312536607	0.138343112	0.179516789
9	0.522344116	1.207102385	1.023342431	1.389352186
10	1.92947628	4.78073678	4.685316536	7.401467858
11	4.741404701	12.39524075	18.52048358	26.55673419

表 5: ARM 平台下各方法加速比

从表中可以看出，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。

对于单独采用 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 4.7 左右，而实际开启了 8 个进程，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。

此外，我们还可以发现一开始的时候，4 种方法的加速比体现一个越来越小的趋势，这是由于单独使用这几个方法时，当数据规模较小时，都会体现负优化的效果，那么当它们合在一起时，负优化的效果也会叠加，所以出现这样的现象也不足为奇。

为了进一步将 NEON 和 OMP 在于 MPI 结合后的性能与单独使用时的性能进行比较，我以 MPI 的性能为 baseline，得到以下数据：

数据规模	相对于 MPI 的优化比		
	mpi_neon	mpi_omp	mpi_omp_neon
2	0.909975353	0.017615074	0.016988162
3	0.941177715	0.022197954	0.020993487
4	1.079400629	0.032255865	0.028750485
5	1.461537939	0.211438121	0.150680421
6	2.068965145	0.331109152	0.286761349
7	1.945998993	0.568547698	0.498808353
8	2.159744149	0.956002334	1.240527756
9	2.310933247	1.959134598	2.659840787
10	2.477738042	2.428284081	3.835998366
11	2.614254958	3.906117438	5.601026671

表 6: ARM 平台下各方法相对于 MPI 的加速比

在以上数据的基础上，我们可以得到以下折线图4.6，进而更好地分析基于 MPI 并与 SIMD、OMP 结合后的方法的性能提升。

由于 NEON 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，NEON 实际的加速比只达到了 2.0 之间，如图4.6所示。这也和之前 SIMD 的实验结果契合，证明在 LU 分解问题上，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

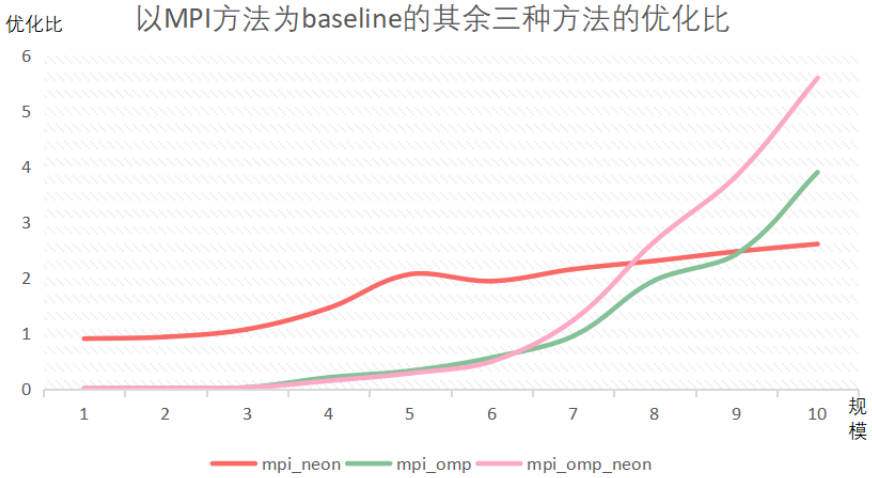

图 4.6: 不同任务划分方法在 x86 架构上运行的相对加速比折线图

达到这个数值。

而在实验中，OMP 采用了 8 线程的算法，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 11 的时候，这个性能提升也只达到了 4 倍左右，如图4.6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 1/8，因此，对于每个进程而言，即使问题规模达到了 11，分配给每个进程的任务也只有 8 左右，而通过之前的实验可以表明，在矩阵大小为 8 左右的任务规模下，其加速比仍然还较低，这个实验结果符合上一章针对多线程的研究结果。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。

最后，我们在实验中尝试了将 MPI 和 NEON、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5所示。在问题规模达到 11 的时候，取得了 26.5 倍的性能提升。这和单独采用各个优化方式后取得的性能提升的优化比的乘积基本一致。

5 小组实验总体分析

本次实验中，我与小组成员黄明洲共同进行了高斯消去算法和 LU 分解算法的 MPI 编程研究，其中，由我负责 LU 分解算法的 MPI 编程研究，他负责高斯消去算法的 MPI 编程研究。

根据我们在实验中得到的数据和分析结果，我们对于 Guass 消去算法和 LU 分解算法在多进程 MPI（Message Passing Interface）编程中表现出不同的性质和特性的共同分析得到了以下结果：

首先，Guass 消去算法的并行化存在明显的困难，其每一步操作都依赖于前一步的结果，导致串行特性较多。此外，每一步消去之后需要同步数据到其他进程以继续处理下一步，频繁的通信和同步增加了开销，同时矩阵元素的更新依赖于前一行的相应元素，使得数据依赖性非常强，这增加了并行环境下的数据传输复杂性和通信量。

相较之下，LU 分解算法更适合并行化处理。各进程可以在局部进行大量计算，减少了通信频率。LU 分解支持更有效的分块策略，每个矩阵块内的计算可并行进行，而块之间的依赖较少，配合适当的调度，可将数据传输和计算重叠，提高并行效率。实际的 MPI 实现中，通过分块和优化通信模式，可以减少通信开销，同时更均匀地分布计算负载。

对于任务与数据划分方面，理论上来说 Guass 消去算法一般采用行划分策略，将矩阵的不同行分

配给不同的进程，由于后续运算需要前面结果，因此频繁的通信和同步是必须的。而 LU 分解则采用块划分策略，每个进程处理一个或多个完整的子矩阵块，局部计算大大减少了整体通信次数和数据依赖性。两者在此次 MPI 实验中，使用循环划分的方法都有着非常好的效果。

在数值稳定性方面，Guass 消去算法和 LU 分解都需要引入部分选主元策略，但在并行化过程中，LU 分解通过局部块内的选主元，复杂性和通信需求会更低。

综合来看，由于 Guass 消去算法的串行特性和高同步需求，使得其在大规模并行计算中的适应性较差，只适用于中小规模问题和较少进程数量的场景。而 LU 分解算法由于其块划分和通信优化策略，更适合大规模矩阵的高性能计算场景，尽管实现较复杂，但明显提高了并行计算的效率和扩展性。因此，**LU 分解算法更适合于 MPI 编程中的高性能计算**，尤其是在需要处理大规模线性系统的情况下。

6 个人任务完成总结

本次实验中，我们小组完成了对 Guass 和 LU 算法在 MPI 编程中的并行加速，并对两种算法的特点和性质进行了分析。

在我的任务部分，我完成了对 LU 算法的 MPI 并行优化实验，在 x86 和 ARM 平台上进行了实验，并对算法性能进行了全面分析。

在实验中，我通过对比串行和并行算法的运行时间和性能表现，深入分析了 MPI 并行优化对 LU 算法性能的影响。探索发现并行算法的性能相关性质，以及充分利用串行与并行方法，提高算法的运行速度。同时总结各种方法在不同平台下的性能表现，比较不同 MPI 实现的优缺点，并讨论多进程通信对算法性能的影响。

此外，我还将 MPI 方法与 SIMD、多线程方法进行结合，以获得更好的算法性能，并对结合后产生的现象进行了深刻剖析。

通过本次实验，我进一步加深了对 MPI 并行编程工具的理解，并且了解了它们在不同平台下的性能特点。同时，在实验中我不断地尝试和调整代码，涉及到任务划分策略、工作方式的探索，以找到最优的实现方式也让我对 MPI 并行编程有了更深刻的理解。

总的来说，本次实验任务完成度较高，成功实现了对 LU 算法的 MPI 并行优化，并且对算法性能进行了全面分析。通过实验，我不仅提高了对并行计算和优化技术的认识，对 MPI 并行编程工具的使用和优化有更深入的理解，也为进一步的算法优化和并行计算研究打下了基础。