南開大學
Nankai University

计算机学院

并行程序设计实验报告

MPI 编程

姓名：王旭尧黄天昊

学号：2012527 2011763

专业：计算机科学与技术

2023 年 6 月 4 日

目录

1 引言

1.1 总体介绍

1.1.1 实验平台

我们首先在金山云 x86 平台上进行了高斯消去的基础 MPI 并行化实验。实验中我们测试了不同问题规模和不同节点数/线程数下算法的性能。接下来，我们研究了不同任务划分策略（包括块划分、块循环划分、列循环划分、二维划分和流水线划分）对性能的影响，并分析了其复杂度。然后，我们将 MPI 与 SIMD、Pthread 和 OpenMP 相结合，探讨它们之间的性能表现。我们还研究了不同的 MPI 编程方法，包括阻塞通信、非阻塞通信、单边通信以及 MPI 自身的多线程支持。最后，我们比较了不同平台之间的性能差异。在这些实验探究过程中，我们结合了 MPI 知识和性能分析工具来解释底层原因。

1.2 实验环境

金山云 本次实验主要在金山云上实现，以下是服务器所采用的相关软硬件信息：

- 内核：x86_64
- CPU 核心数：2
- CPU 主频：2.593GHz
- L1d cache：32K
- L1i cache：32K
- L2 cache：1024K
- L3 cache：25344K

本次实验在 4 个节点、每个节点 2 个核心上计算，即在 8 个核心上计算。

鲲鹏 此外，本次实验还在华为鲲鹏服务器提供的 ARM 平台上进行，以对比不同平台性能差异。鲲鹏服务器的信息如下：

- 内核：Linux master 4.14.0-115.el7a.0.1.aarch64
- 编译器：华为毕升编译器 (clang 12.0.0)
- CPU 型号：鲲鹏 920 服务器版
- CPU 核心数：96 核 96 线程
- CPU 主频：2.6GHz
- L1d cache：96*64KB
- L1i cache：96*64KB
- L2 cache：48MB
- L3 cache：20GB

1.2.1 github 链接

github 链接为https://github.com/Skyyyy0920/Parallel-Programming

1.3 任务分配

本次实验由王旭尧（2012527）和黄天昊（2011763）共同完成。其中基本实验部分的各个小实验，即使用不同组合的节点、线程等参数的实验由我们共同完成；对于探索部分，王旭尧完成不同任务划分以及不同平台的对比，黄天昊完成 MPI 编程方法和不同并行方法结合的探索

2 基本实验

在金山云 X86 平台进行高斯消去实验，主要探究不同问题规模、不同节点数/线程数下的算法性能（串行和并行对比）。

2.1 算法设计

基础版本的代码中，我们使用块划分实现，并设置进程数为 8。基本思路是：程序中的问题规模为 N，进程数为 num_threads，则给每个进程分配 N/num_threads 行的数据，对于第 i 个进程，分配的范围为 $[i*(N-N\%num\_threads)/num\_threads, (i+1)*(N-N\%num\_threads)/num\_threads-1]$。需要注意的是，上述范围公式在处理余数的最后一个进程时不适用，为了防止越界问题，需要进行特殊处理。

每个消去步被视为主要目标的算法设计，在每个消去步内并行执行，而消去步之间则按顺序执行。每个消去步可以分为” 除法” 和” 消去” 两个步骤。在” 除法” 步骤中，处理第 k 行的数据，而在” 消去” 步骤中，处理从 k+1 行和 k+1 列开始的子矩阵。由于” 消去” 步骤需要使用上一个” 除法” 步骤得到的第 k 行数据，存在数据依赖性，因此必须确保在每个进程执行它们的” 消去” 步骤时，它们都已经获得了” 除法” 步骤的结果。在块划分的情况下，为每个负责该块的进程设计消去操作，并将结果发送给其他进程。以下是相应的代码实现：

```
if(k>=myid*(N-N%n_threads)/n_threads&&k<=(myid+1)*(N-N%n_threads)/n_threads-1){
    //做除法...
    for(int j = 0;j < n_threads; j++){
        if(j == myid) continue;//避免死锁
        MPI_Send(&A[k][0],N,MPI_FLOAT,j,100 - myid,MPI_COMM_WORLD);
    }
}
else
    MPI_Recv(&A[k][0],N,MPI_FLOAT,
    k/(N-N%n_threads)/n_threads,
    100-k/(N-N%n_threads)/n_threads,
    MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

在这段内容中，需要注意 send 和 recv 函数的两个方面。首先，在调用 send 函数时，需要遍历所有进程并确保不会将消息发送给自己，否则由于没有对应的接收函数，可能会导致死锁的问题。其次，可

以将 tag 设置为一个与 myid（进程 ID）相关的值，例如使用 100-myid，然后通过一定的规则将 recv 函数中的 source 参数反解为相应的进程和块划分。

在消除步骤中，保持进程的划分策略不变，并将相应的行分配给负责该块的进程进行消除。在这里需要注意边界的判断，对于最后一个线程，应使用 N 作为右边界。

最终结果可以在不需要通信的情况下进行消除。对于进程 0 而言，在每一次的除法操作中，它会接收到一行已处理好的数据。经过 k 步操作后，进程 0 会得到最终的结果。程序正确性验证如图2.1所示。

图 2.1: 高斯消去正确性验证

2.2 复杂度分析

对于第 k 个消元步而言，除法需要 $O(n-k-1)$ 复杂度，消去为并发计算，需要 $O(n-k-1)$ 次乘法和减法，因此计算的总复杂度为

$$O(3n(n-1)/2)$$

对于通信，广播共需要

$$\sum_{k=0}^{n-1}(t_s + t_w(n-k-1))logn = t_s nlogn + t_w(n(n-1)/2)logn$$

将计算和通信的时间开销相加，再乘上处理器数量 n，可得到总代价为

$$O(n^3 logn)$$

事实上，这并不是一个代价最优的算法，因为各消去步之间是串行执行，存在大量等待开销，总的进程空闲时间达到 $O(n^3)$。更多算法将在下一节进阶内容中探索。

2.3 串并行（不同问题规模）

在金山云平台上进行了问题规模为 512、1024 和 2048 的测试，并得到了与图2.2中所示数据相符的结果。观察数据可知，相较于串行计算，使用 MPI 并行计算可以获得最高 4.5 倍的加速比，且随着问题规模的增加，加速比也呈现增长趋势。MPI 加速比与进程数量之间存在差距的原因在于通信和等待的开销。

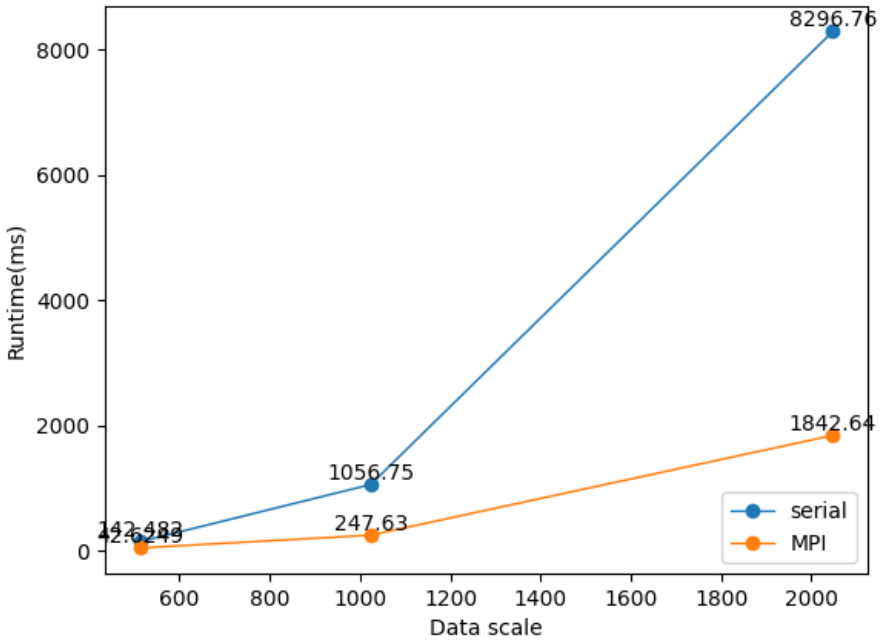

图 2.2: 高斯消去串并行对比

使用 Vtune 做 profiling，得到如图3.13所示的结果。

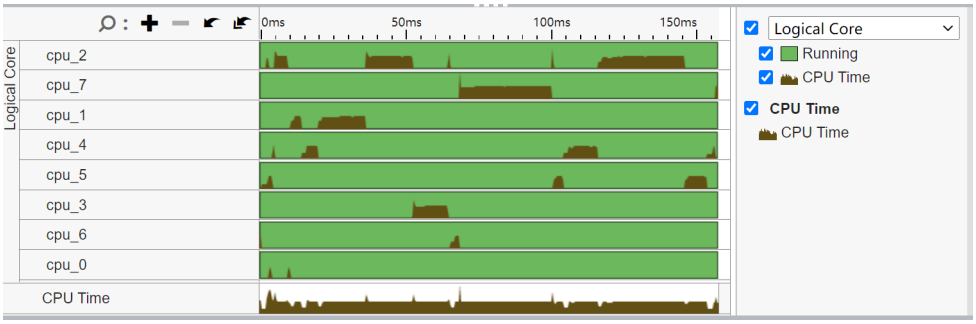

图 2.3: MPI8 进程 profiling

可以看到 2 点信息：1. 的确有 8 个 CPU 核心在执行程序；2. 各 CPU 核心的工作并发度较低，存在大量的等待开销。这与我们分析的结果相符合。

2.4 不同进程数

分别尝试串行、MPI8 进程、MPI4 进程、MPI2 进程、MPI1 进程，结果如图2.4所示。

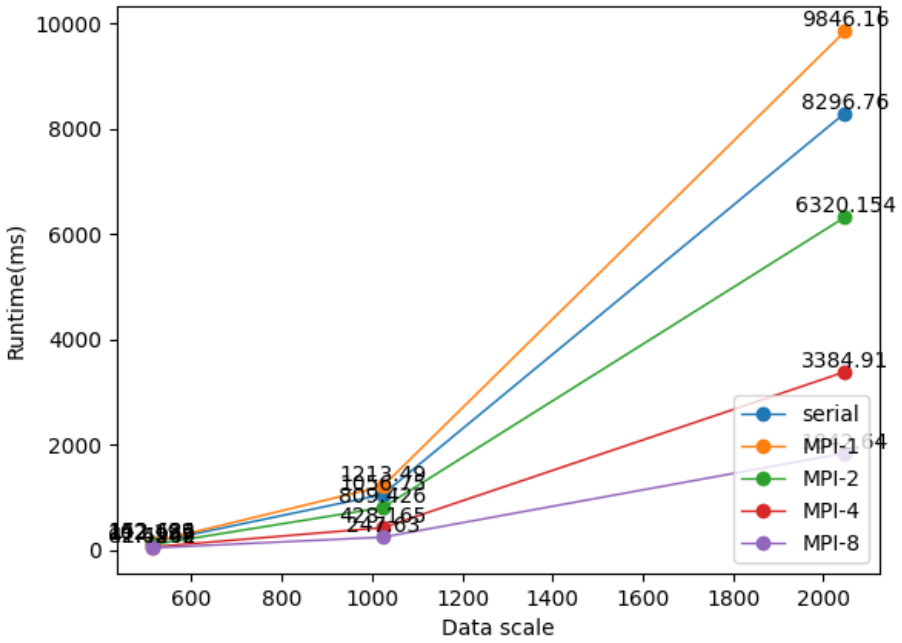

图 2.4: 不同进程数比较

分析数据，可以总结出 MPI 与进程数有关的的 2 个性质：

1. MPI1 进程的效率低于串行，这是因为通信和等待的开销；
2. 串行执行相比，并行执行 MPI2、4、8 进程的效率均更高。加速比指标显示了相对于串行执行，使用 2、4 和 8 个进程的加速情况。具体而言，相对于串行执行，2 个进程的加速比达到了 1.3 倍，4 个进程的加速比达到了 2.4 倍，而 8 个进程的加速比达到了 4.5 倍。这些结果表明，随着进程数的增加，加速比也会增加，即进程数与加速比之间存在正相关关系。然而，加速比和进程数之间并不是线性关系，随着进程数的增加，加速比的提高速度会减缓。这是因为进程数的增加会导致更大的通信开销，从而限制了加速比的提升速度。

2.5 不同节点和线程数组合

探究在相同并发度 (总线程数) 下,不同节点数与每节点线程数的组合时程序性能。分别设置 nodes=2:ppn=4 和 nodes=1:ppn=8 和 nodes=4:ppn=2，在数据规模 512、1024 和 2048 下测试，得到数据如图2.5所示。

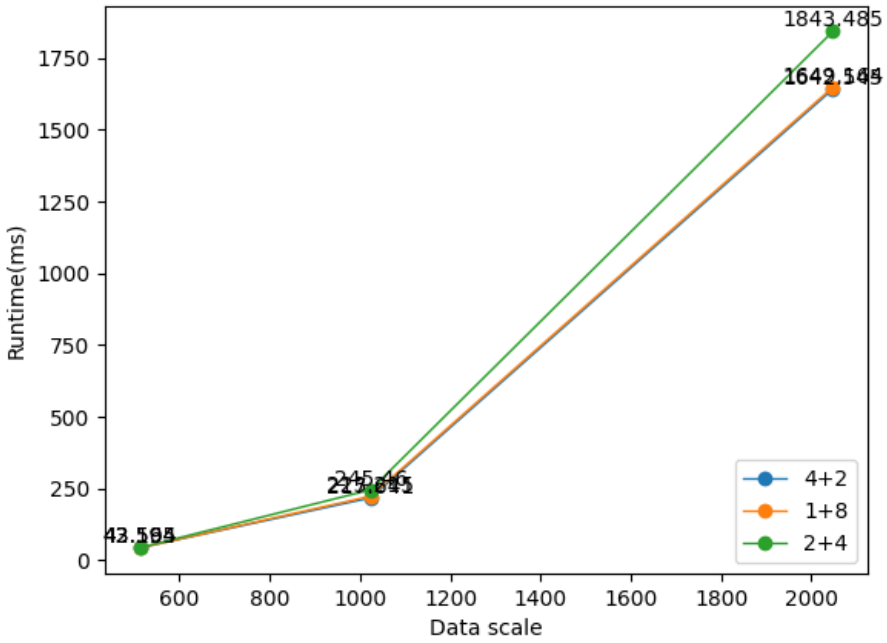

图 2.5: 不同组合对比

可以发现，组合 nodes=4:ppn=2(图中为 4+2) 性能提升最高，其次是 nodes=8:ppn=1，最后是 nodes=2:ppn=4。

3 更多探索

3.1 不同任务划分

块划分的算法见前一节：基本实验，复杂度经分析为 $O(n^3logn)$，这里不再赘述。下面探究更多划分方法：

3.1.1 循环块划分

为了解决负载不均匀的问题，特别是余数部分，我们尝试采用更细粒度的循环划分方法。具体而言，我们将每个消元步骤中的每一行分配给一个线程，线程之间轮流执行对各行进行除法操作。下面是实现的方法：

```
1  if(k % n_threads == myid){
2      //做除法...
3      for(int j = 0;j < n_threads; j++){
4          if(j == myid) continue;//避免死锁
5          MPI_Send(&A[k][0],N,MPI_FLOAT,j,100-myid,MPI_COMM_WORLD);
6      }
7  }
```

```
8   else
9       MPI_Recv(&A[k][0],N,MPI_FLOAT,k%n_threads,100-(k%n_threads),MPI_COMM_WORLD,MPI_STATUS_IGNO
```

复杂度分析 对于第 k 个消元步而言，除法需要 $O(n-k-1)$ 复杂度，消去为并发计算，需要 $O(n-k-1)$ 次乘法和减法，因此计算的总复杂度为

$$O(3n(n-1)/2)$$

对于通信，广播共需要

$$\sum_{k=0}^{n-1}(t_s + t_w(n-k-1))logn = t_s nlogn + t_w(n(n-1)/2)logn$$

由于循环划分的进程间负载差距最多为一行元素，在 n 个消除步后，将空闲时间由块划分时的 $O(n^3)$ 加速为了 $O(n^2p)$。

实验结果 在金山云平台、8 进程数下，分别测得规模 512、1024、2048 的程序运行时间如表1所示。和单纯块划分相比，循环块划分负载更均匀，执行时间的确更少。

数据规模	512	1024	2048
时间 (ms)	47.1645	238.16	1843.19

表 1: 循环块划分

3.1.2 循环列划分

算法设计 我们使用循环划分的方法来划分每个循环步中的列元素。在除法阶段，拥有对角线上元素的进程负责将该元素广播给其他进程。然后，所有进程都对自己负责的列元素进行除法操作。在此之后，无需广播除法结果，因为需要使用除法结果的后续行都由同一个进程负责，可以直接在本地进行消去操作。下面是实现流程的概述：

```
1  MPI_Bcast(A[k], N, MPI_FLOAT, k % n_threads, MPI_COMM_WORLD);
2  //做除法...
3  //做消去...
```

复杂度分析 循环列划分和循环行划分的复杂度分析只有一个区别，那就是 cache 的连续性利用。循环行划分由于利用了 cache 的空间局部性，因此效率会高于列划分，可以从实验结果中得到证实。

实验结果 在金山云平台、8 进程数下，分别测得规模 512、1024、2048 的程序运行时间如表2所示。和循环行划分相比，循环列划分未利用空间局部性，执行时间的确较多。

数据规模	512	1024	2048
时间 (ms)	55.1944	264.596	1678.96

表 2: 循环列划分

3.1.3 二维划分

算法设计 二维划分的流程如图3.6和图3.7所示。除法为：首先持有 A[i][k]($k \le i < n$) 的进程将 A[i][k] 广播给同行其他进程，然后持有 A[k][j]($k+1 \le j < N$) 的进程做除法。消去为：首先持有 A[k][j]($k+1 \le j < n$) 的进程将 A[k][j] 广播给同列其他进程，然后持有 A[i][j]($k+1 \le i < n, k+1 \le j < n$) 的进程做消去。

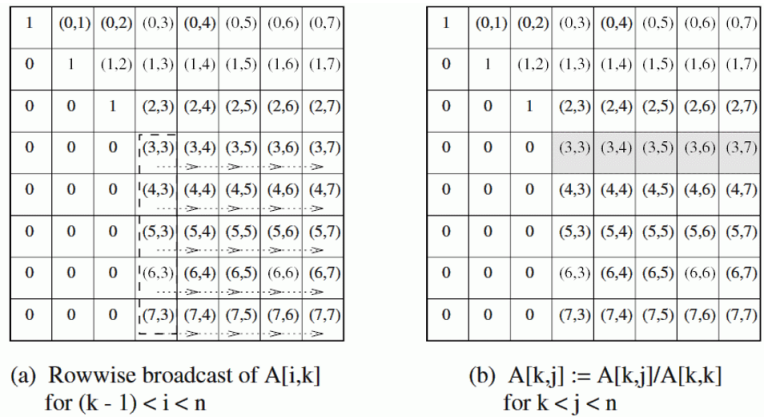

图 3.6: 二维划分除法流程

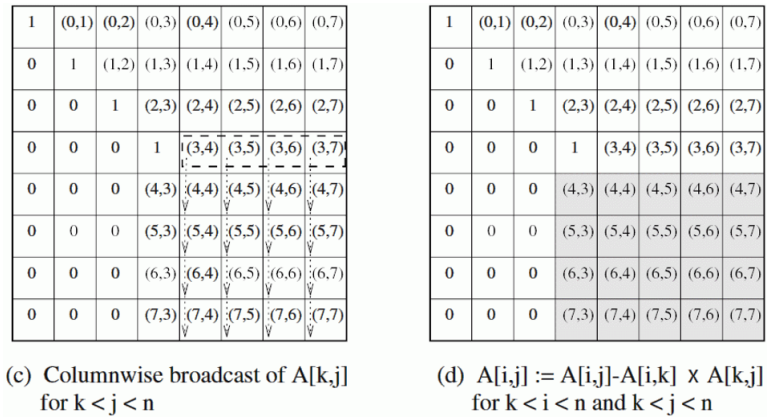

图 3.7: 二维划分消去流程

然而，这种划分方式要求处理器数量大于等于 N^2，这在本实验中的 8 个进程显然不适用。为了解决这个问题，我们可以修改策略，让每个进程负责一个子矩阵，而不是单独负责一个元素。在这种情况下，我们可以尝试使用二维块划分算法，将矩阵划分为 3×3 的子矩阵，其中一个进程负责两个子矩阵，而其余七个进程各自负责一个子矩阵。二维划分的一个麻烦之处在于需要手动分配进程，而不能像一维情况下那样使用公式（例如取模运算）来获取进程的 ID。因此，我们采用了八个 if 判断语句来指定线程，这些 if 语句分别用于广播和消去划分的两个阶段。

为了实现更均衡的任务分配，考虑到大部分消元操作集中在右下角矩阵，我们采用了两种方法：首先，将矩阵从左到右、从上到下以 N/3 和 N*2/3 的位置进行划分，这样右下角的子矩阵相对较小；其次，将左上角的两个子矩阵分配给一个进程，而其他进程各自持有一个子矩阵。通过这样的划分方式，我们能够在一定程度上实现更均衡的任务分配，确保消去操作能够更均匀地分布在各个进程之间。

在进行消去任务的分配时，我们面临了一些困难，因为我们认为有太多的情况需要逐一讨论。然而，经过仔细思考，我们发现实际上，对于每个进程而言，通常只有两种情况，最多可能有三种情况

需要讨论。这是因为每次消去操作都是对一个正方形的子矩阵进行消去，这个” 正方形” 的特性使得许多不会出现的情况被排除掉，如图3.8所示。

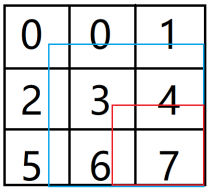

图 3.8: 二维划分原理图

我们以较复杂的 0 号和 3 号进程为例，展示判定代码：

```
//0 号进程任务分配
if(myid == 0){
    if(k + 1 >= N / 3) continue;
    for(int i = k + 1;i < N / 3; i++)
        for(int j = k + 1;j < N * 2 / 3; j++)
            A[i][j] = A[i][j] - A[i][k] * A[k][j];
}
//3 号进程任务分配
if(myid == 3){
    if(k + 1 >= N * 2 / 3) continue;
    if(k + 1 <= N / 3){
        for(int i = N / 3;i < N * 2 / 3; i++)
            for(int j = N / 3;j < N * 2 / 3; j++)
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
    }
    else{
        for(int i = k + 1;i < N * 2 / 3; i++)
            for(int j = k + 1;j < N * 2 / 3; j++)
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
    }
}
```

复杂度分析 在每个消去步中，一个进程最多需进行 n^2/p 次乘法和减法运算，数据通信量为 $n/\sqrt{p}$。并行时间约为 $O(2n^3/p)$，代价为 $2n^3$，为串行时间的 3 倍。若改为循环划分，负载会更均衡，空闲时间会更少。

实验结果 在金山云平台、8 进程数下，分别测得规模 512、1024、2048 的程序运行时间如表3所示。块二维划分的执行时间较多，这是因为通信开销过大，且负载不如循环划分均匀，等待开销过大。

数据规模	512	1024	2048
时间 (ms)	2340.16	8889.162	34397.196

表 3: 块二维划分

3.1.4 流水线划分

算法设计 流水线算法与普通的块划分算法有所不同，**流水线算法的关注点是进程，而不是消去步骤。在流水线算法中，每个进程只需要关注在 $k = 0, 1, ..., n-1$ 时需要执行的任务**。进程的任务包括行的除法运算和将除法结果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的除法结果时，它首先将结果转发给下一个进程，然后对自己负责的行进行消去操作。对于单个进程来说，它在完成一行的除法运算之后立即转发该行，然后继续执行下一行的除法运算，如此重复，直到完成所有的除法步骤。与普通的块划分算法相比，流水线算法的关注点更加集中在进程的执行顺序和任务分配上，通过适当的数据转发和任务交替执行，实现了流水线式的并行计算。

在流水线算法中，进程间形成了一个逻辑链结构。不失一般性，假设矩阵规模为 5*5，有 5 个进程执行，每个进程操作一行数据，则算法流程如图3.9、3.10和图3.11所示。

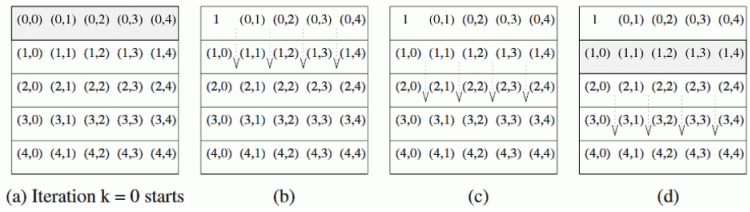

图 3.9: 流水线划分流程图-1

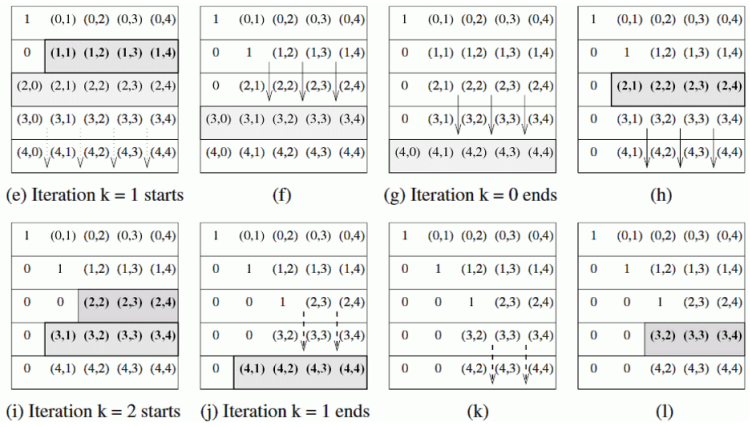

图 3.10: 流水线划分流程图-2

图 3.11: 流水线划分流程图-3

让我们以 $k = 0$ 的迭代为例来说明。首先，进程 P0 执行第 0 行的除法运算，将结果 A[0] 发送给进程 P1。一旦进程 P1 接收到结果，它立即将其转发给进程 P2，并开始对自己负责的第 1 行进行消去操作。同时，进程 P0 可以继续执行第 2 行的除法运算，并将结果转发给下一个进程。这样依次进行，以此类推。

这个过程中，每个进程根据接收到的除法结果，立即将其转发给下一个进程，并同时进行自己负责的行的消去操作。通过这种方式，各个进程之间实现了数据的流动和任务的交替执行，从而实现了并行计算的效果。

复杂度分析 n 个消去步骤，每个步骤的启动间隔是常量个操作步（一次通信、计算）；最后一个消去步骤，仅对一个矩阵元素进行计算，因此总操作步为 $O(n)$ 复杂度。

对于每个操作步，元素的传输复杂度为 $O(n)$，元素的除法或消去操作也是 $O(n)$。

因此，并行时间复杂度为 $O(n^2)$，代价为 $O(n^3)$。**这是一个代价最优的算法**。

然而，由于进程数远少于矩阵行数，无法让每个进程保存矩阵的一行，因此该算法没有应用于该实验。

3.1.5 对比

下面对比各任务划分方式的运行时间如图3.12所示。

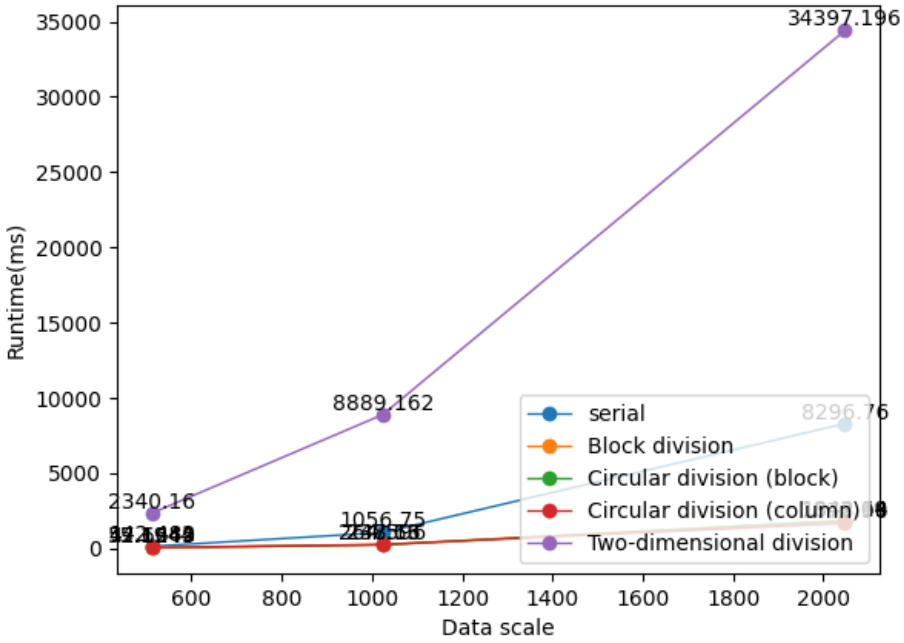

图 3.12: 不同任务划分对比

从总体上看，根据性能表现排序，循环块划分的性能最好，其次是块划分，然后是循环列划分，最后是串行和二维划分。这种排序的主要原因有以下几点：

1. 循环块划分减少了空闲等待时间：通过将任务均匀分配给不同的进程，循环块划分可以减少进程之间的空闲等待时间，提高并行计算的效率。

2. 循环列划分虽然减少了等待时间但没有利用缓存的空间局部性：尽管循环列划分在减少等待时间方面有一定优势，但它没有充分利用缓存的空间局部性。这可能导致缓存未能有效地存储和重用数据，从而影响性能。

3. 二维划分的通信开销过大：二维划分方式由于进程之间的通信开销较大，会导致性能下降。在这种划分方法中，进程需要频繁地进行通信和同步操作，这会增加额外的开销，降低了整体性能。

4. 消去部分没有分配给多个进程并行，导致时间最长：对于串行和二维划分，消去部分没有被分配给多个进程并行执行。这意味着消去操作在这些划分方法中是顺序执行的，导致时间最长。

总结一下，循环块划分的性能较好，主要是由于减少了空闲等待时间；块划分在性能上稍逊一筹；循环列划分、串行和二维划分的性能相对较差，分别受限于空间局部性、通信开销和顺序执行的影响。

使用 perf 分析各划分方式，图3.13为块划分，图3.14为循环块划分，图3.15为循环列划分。

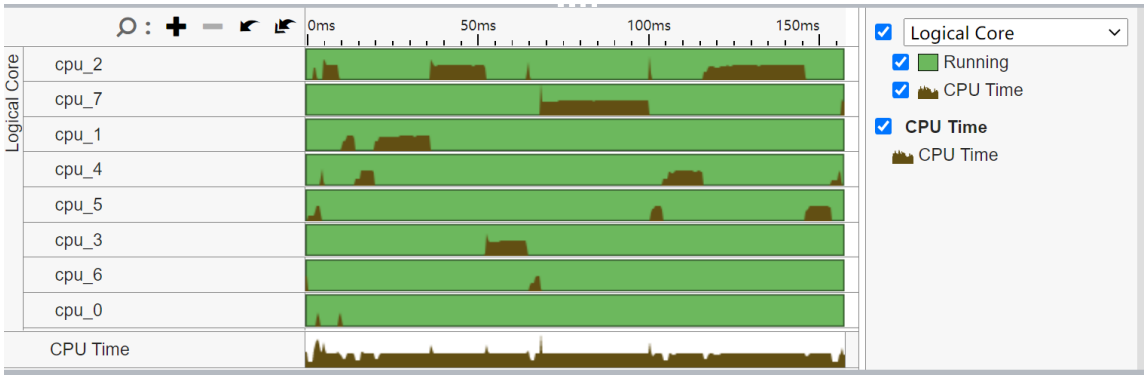

图 3.13: 块划分

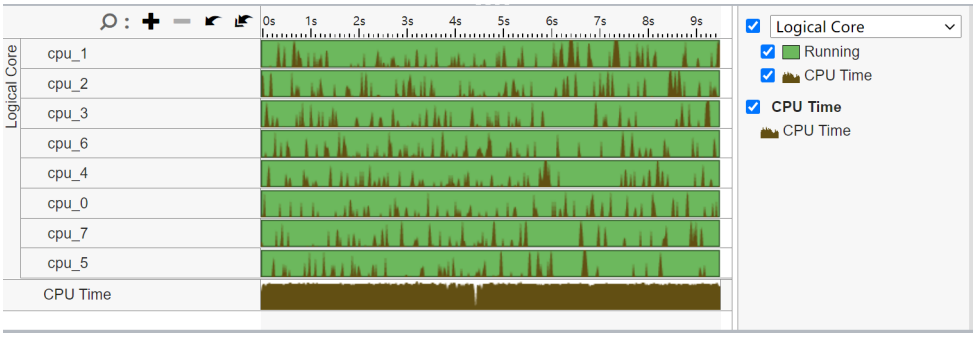

图 3.14: 循环块划分

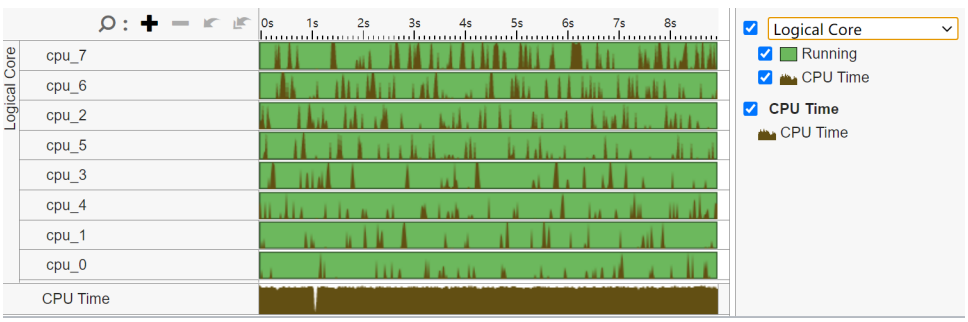

图 3.15: 循环列划分

可以看到，加入循环划分后各进程的负载更均衡，没有出现过长的等待时间，各进程的执行情况非常类似。根据 Vtune 得到的数据，我们列出表6：

	块划分	循环块划分	循环列划分
CPI	0.338	0.312	0.290
Clockticks	211,200,000	8,665,600,000	12,897,600,000
CPU Time	59.848ms	2.338s	3.464s
Averange CPU Frequency	3.8 GHz	3.7 GHz	3.7 GHz

表 4: 各划分的性能指标

3.2 MPI 编程方法

下面我们就一维列循环划分为例，比较阻塞通信、非阻塞通信、单边通信的区别。以下实验数据是在规模 512、1024、2048 的矩阵、金山云平台条件下完成。

3.2.1 阻塞通信

阻塞通信使用广播命令 MPI_Bcast(A[k], N, MPI_FLOAT, k % n_threads, MPI_COMM_WORLD)。此时进程需要等到接收到 A[k] 一行的数据后再继续后续运算，在等待时 CPU 核空闲，导致空闲开销。分别在规模 512、1024、2048 下测得数据如表11所示。

	512	1024	2048
阻塞通信	66.6861	373.416	1820.39

表 5: 阻塞通信

3.2.2 非阻塞通信

为减少阻塞通信的空闲开销，MPI 还提供一些支持非阻塞通信的函数。在非阻塞通信中，调用返回不代表通信完成，进程在通信未完成时就可以进行计算，实现计算和通信的重叠。对于 MPI_Bcast，我们使用组通信的非阻塞变体 MPI_Ibcast 函数，在通信时进行 j 下标的计算 (不需要用到 A[k])，计算好后使用 MPI_Wait 函数来判断非阻塞通信是否完成。伪代码如下：

```
for(int i = 0;i < n_threads; i++)
    MPI_Ibcast(A[k], N, MPI_FLOAT, k % n_threads, MPI_COMM_WORLD,&req[i]);
//通信和计算重叠
int j = k + 1;
while ((j - n_threads * (j / n_threads)) != myid) j ++;
MPI_Wait(&req[myid], &status[myid]);
//消去...
```

在规模 512、1024、2048 下测试，得到数据如表6所示。

	512	1024	2048
非阻塞通信	133.59	459.964	2119.154

表 6: 非阻塞通信

3.2.3 单边通信

单边通信的基本思想是数据传输和同步解耦。在单边通信中，每个进程暴露一部分内存空间给其他进程，其他进程可直接读写此空间，无需远端进程同步即可操作数据。在本实验中，我们使用 MPI_WIN_CREATE 来设置 buffer 为远端可访问的，代替 MPI_BCast。在规模 512、1024、2048 下测得数据为表7所示。

	512	1024	2048
单边通信	60.5913	305.116	1781.65

表 7: 单边通信

3.2.4 比较

比较串行和 MPI 不同通信方式结果，如图3.16所示。

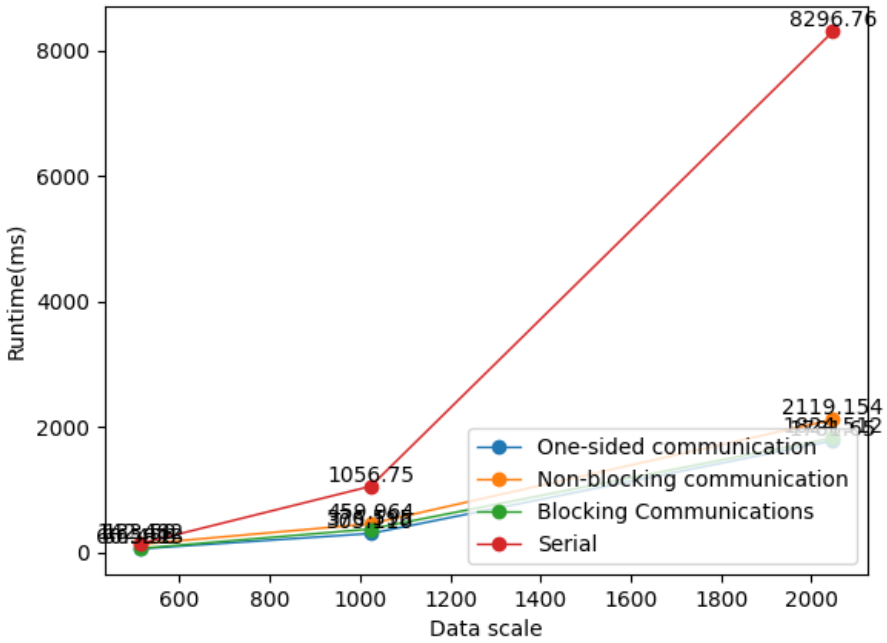

图 3.16: 不同通信方式比较

可以看到，最快的是单边通信，这是因为它直接划分了一块所有进程都可访问的内存，而省略了数据发送和接受的开销，以及等待的开销。使用 Vtune 分析，图3.17为单边通信的效果图，图3.18为非阻塞通信的效果图。可以看到，相较图3.15，各进程等待通信的时间间隔减少。

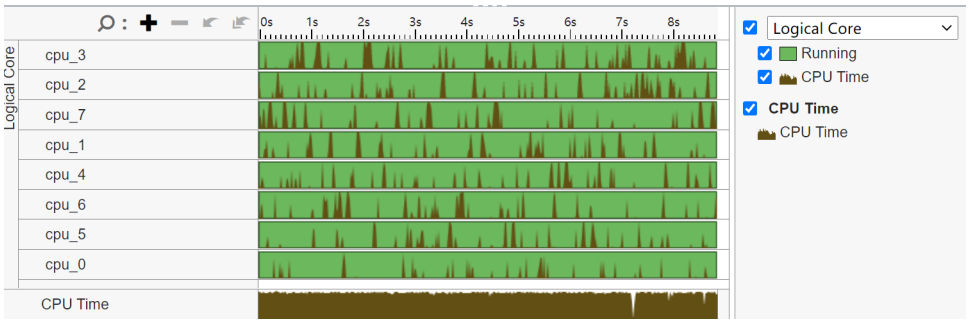

图 3.17: 单边通信

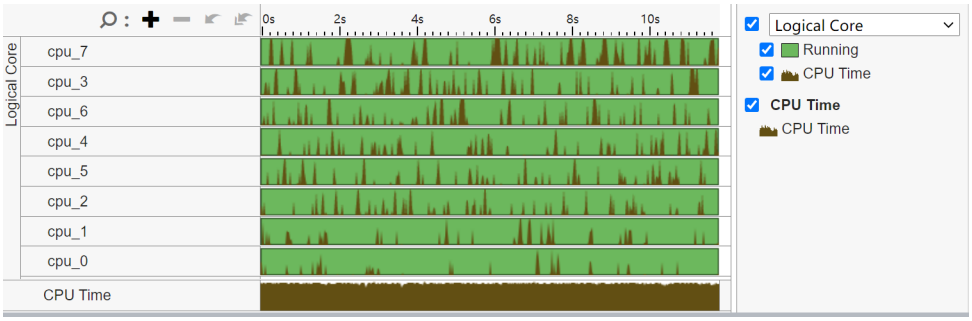

图 3.18: 非阻塞通信

表8为 Vtune 分析下，不同通信方式的各性能指标。可以看到，单边通信的 CPU Time 最低，性能的确最好。

	阻塞通信	非阻塞通信	单边通信
CPI	0.290	0.394	0.289
Clockticks	12,897,600,000	14,568,000,000	9,944,000,000
CPU Time	3.464s	3.863s	2.667s
Averange CPU Frequency	3.7 GHz	3.8 GHz	3.7 GHz

表 8: 不同通信方式对比

3.3 结合

3.3.1 与 SIMD 结合

实验结果 在金山云平台上，将 MPI 与 SIMD 结合，在消去部分使用 SSE 或 AVX 指令，得到结果如表9所示。可以看到，加入 SIMD 均有较大程度的加速，加速比由 4.5 提升到了 7，且 AVX 加速效

数据规模	512	1024	2048
MPI+SSE	27.4656	151.948	1172.15
MPI+AVX	29.5945	109.55	942.961

表 9: MPI+SIMD

果较 SSE 又更优。

复杂度分析 对于使用 SSE 的指令, 一次可以展开 8 个循环, 因此在计算复杂度中的加速比为 8, 计算复杂度为 $O(3n(n-1)/16)$; 通信复杂度不变, 仍为 $\sum_{k=0}^{n-1}(t_s+t_w(n-k-1))logn = t_snlogn+t_w(n(n-1)/2)logn$, 因此总代价为 $O(n^3logn)$ 量级。
对于 AVX 指令,一次可以展开 16 个循环,因此在计算复杂度中的加速比为 16,计算复杂度为 $O(3n(n-1)/32)$; 通信复杂度不变, 仍为 $\sum_{k=0}^{n-1}(t_s+t_w(n-k-1))logn = t_snlogn+t_w(n(n-1)/2)logn$, 因此总代价为 $O(n^3logn)$ 量级。

3.3.2 与 Pthread 结合

实验结果 将 MPI 与 Pthread 结合, 结果如表10所示。由于采用了动态创建线程的方法, 因此时间较

数据规模	512	1024	2048
MPI+Pthread	455.945	1781.54	7426.654

表 10: MPI+Pthread

单纯 MPI 变慢, 但仍然优于串行。

复杂度分析 由于创建了 4 个线程参与计算,因此在计算复杂度中的加速比为 4,计算复杂度为 $O(3n(n-1)/8)$; 通信复杂度不变, 仍为 $\sum_{k=0}^{n-1}(t_s+t_w(n-k-1))logn = t_snlogn+t_w(n(n-1)/2)logn$; 此外, 线程的创建和销毁也是一笔开销, 在每个循环内动态创建进程, 记创建代价为 t_create, 则复杂度为 $O(n*t_create)$。

3.3.3 与 OpenMP 结合

实验结果 在消去部分使用 pragma omp parallel for num_threads(NUM_THREADS) 语句, 将消去的最内层循环分配给 4 个进程进行, 这样将 MPI 与 OpenMP 结合起来。结果如表11所示。同理, 由

数据规模	512	1024	2048
MPI+OMP	445.364	1768.156	7426.156

表 11: MPI+OpenMP

于这是在循环内部动态创建线程, 因此时间较单纯 MPI 较慢, 但仍优于串行。

复杂度分析 由于创建了 4 个线程参与计算,因此在计算复杂度中的加速比为 4,计算复杂度为 $O(3n(n-1)/8)$; 通信复杂度不变, 仍为 $\sum_{k=0}^{n-1}(t_s+t_w(n-k-1))logn = t_snlogn+t_w(n(n-1)/2)logn$; 此外, 线程的创建和销毁也是一笔开销, 在每个循环内动态创建进程, 记创建代价为 t_create, 则复杂度为 $O(n*t_create)$。

3.3.4 比较

比较各数据规模下 MPI 和 SIMD、Pthread、OpenMP 结合的实验结果, 如图3.19所示。

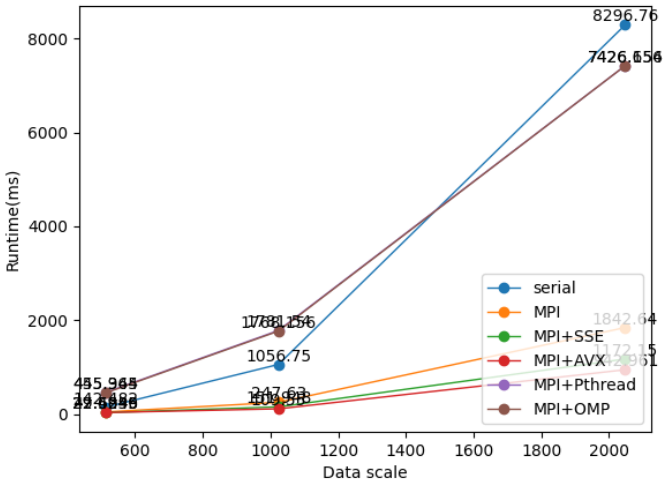

图 3.19: 不同并行策略对比

根据图表显示的数据，可以得出以下结论：MPI+AVX 的性能表现最佳，其加速比达到了 8.7；其次是纯 MPI，其加速比为 4.5；对于 MPI+Pthread 和 MPI+OpenMP，由于程序的前后依赖性，需要在循环内动态创建线程，这导致了较大的开销，使得加速比仅为 1.2。总体而言，采用并行方法可以提高效率。

3.4 不同平台

分别在 Windows 和 Linux 平台，规模 512、1024、2048 下，执行串行和 MPI 代码，结果如图3.20所示。

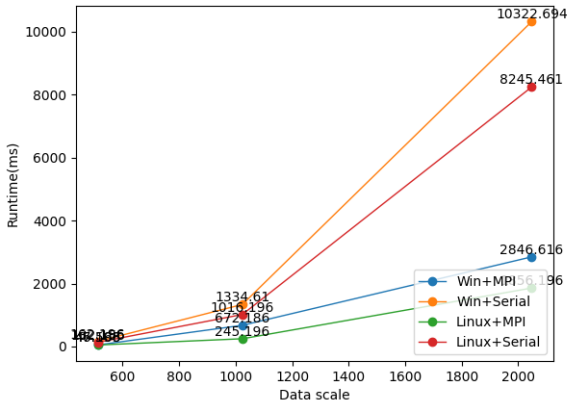

图 3.20: 不同平台对比

可以看到，无论是串行还是 MPI 并行，Windows 平台较 linux 平台运行时间都较长。