南開大學
Nankai University

计算机学院

并行程序设计调研报告

并行体系结构调研

姓名：苏胤华 赵元鸣

学号：2213893 2211757

专业：计算机科学与技术

2024 年 6 月 9 日

目录

1 普通高斯消去法 mpi 并行优化

1.1 思路探索

MPI 是一种用于编写并行程序的通信库接口，全称为 Message Passing Interface（消息传递接口），可以理解为是一种独立于语言的信息传递标准，其有着多种具体实现。本文将探究 ARM 平台上的 MPICH 和 x86 平台上的 MS-MPI。

MPI 库的优点在于其具有可移植性和扩展性。它支持不同操作系统和硬件平台，并且可以轻松地增加处理器数量来提高计算性能。MPI 的标准由 MPI 论坛制定，该论坛由计算机科学家、计算机厂商和用户组成，旨在确保 MPI 的兼容性和可扩展性。通过 MPI 标准的制定，各种不同的 MPI 实现版本可以在不同的计算环境下无缝运行，这包括 MPICH、Open MPI、Intel MPI 等。

MPI 优化相比此前的多线程优化会更加复杂，因为其各个进程之间的内存不共享，因此进程间要不停相互传递数据，所以其任务分配和多进程的进行方式要谨慎构思。多线程优化中，所有线程共享同一个内存地址空间，数据传递相对简单，线程之间的通信可以通过共享内存直接完成。然而，在 MPI 并行优化中，每个进程拥有独立的内存空间，进程之间的数据传递只能通过消息传递实现，这使得并行程序的设计和实现更加复杂。

在进行 MPI 并行优化时，首先需要对任务进行合理划分。常见的任务划分方式有块划分和循环划分等。块划分中，任务被分成若干个连续的块，每个进程负责一个或多个块。循环划分中，任务按照轮流的方式分配给各个进程。每种任务划分方式都有其优缺点，具体选择哪种方式需要根据具体问题的特性和计算资源的情况来决定。

例如，在矩阵运算中，可以采用行划分的方法，每个进程负责若干行的计算。当一个进程完成对某行的计算后，它需要将计算结果传递给其他进程以便于后续的计算。这样，通过不断地消息传递和同步，各个进程可以协同工作，最终完成整个计算任务。

此外，MPI 程序的性能优化还需要考虑负载均衡问题。在实际应用中，由于任务的复杂性和不确定性，可能会导致某些进程的计算任务过重，而其他进程相对空闲。为了解决这个问题，可以采用动态负载均衡技术，通过实时监控各个进程的计算负载，动态调整任务分配策略，确保各个进程的计算任务尽可能均衡，从而提高整体计算效率。

在本文中，我们重点探讨 ARM 平台上的 MPICH 和 x86 平台上的 MS-MPI。MPICH 是一种广泛使用的 MPI 实现，具有良好的可移植性和扩展性，适用于各种不同的计算环境。MS-MPI 则是 Microsoft 为 Windows 平台开发的 MPI 实现，具有良好的兼容性和性能，适用于在 Windows 环境下进行高性能计算。

具体的优化思路是：在 MPI 并行优化中，会存在多个进程，每个进程持有一定的任务量，即行向量，负责对这些行向量进行行消去和标准化（即将对角线元素变为 1）。由于各个进程之间的数据互不连通，因此需要通过消息传递的方法来进行通信和同步。在这种情况下，每个进程独立处理其分配的行向量，同时通过 MPI 库提供的消息传递机制，与其他进程交换必要的数据，从而实现进程间的协作和数据共享。这样的设计不仅提高了并行计算的效率，还确保了各进程在计算过程中的数据一致性和完整性。

任务的划分方式包括块划分和循环划分。块划分方式中，整个任务被分成若干个连续的块，每个进程负责一个或多个块的计算，确保任务负载在进程之间的合理分配。循环划分方式中，任务按照轮流的方式分配给各个进程，使得每个进程负责间隔相等的任务子集，从而平衡了各进程的工作量，避免了某些进程过载或空闲的情况。这两种任务划分方式各有优缺点，选择哪种方式应根据具体应用的需求和计算资源的情况来决定，以达到最优的计算性能和资源利用效率。

具体执行过程中，当一个进程完成对某行向量的消元操作并将对角线元素标准化为 1 后，它需要

将该行的计算结果传递给其他进程。这样做是为了确保其他进程在进行后续计算时，可以使用最新的已标准化数据，从而保证计算过程的正确性和一致性。这一过程通过 MPI 的消息传递功能来实现，例如使用 MPI_Send 和 MPI_Recv 函数进行数据交换。通过这种方法，各个进程能够协同工作，共同完成整个计算任务。在最终的消元过程结束后，所有处理的结果将汇总到 0 号进程中，得到最终结果。这种方法不仅有效利用了计算资源，还保证了计算结果的正确性和一致性。然而，实现这一优化方法需要谨慎设计通信模式和任务分配策略，以尽量减少通信开销，提高并行计算的效率。

1.2 代码实现

因为篇幅原因，本人不过多展示，只展示部分的普通优化实现，更多如与 SIMD 和多线程结合、非阻塞通信等部分的代码会在仓库中储存。

平凡算法呈现如下：

平凡算法

```
void LU() {
    for (int k = 0; k < N; k++) {
        for (int j = k + 1; j < N; j++) {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;

        for (int i = k + 1; i < N; i++) {
            for (int j = k + 1; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

ARM 平台最终优化算法呈现：

ARM 上 MPI 优化算法

```

void mpi_optimize(int argc, char* argv[]) {
    double start_time = 0;
    double end_time = 0;
    MPI_Init(&argc, &argv);
    int total = 0;
    int rank = 0;
    int i = 0;
    int j = 0;
    int k = 0;
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &total);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    cout << rank << " of " << total << " created" << endl;
```

```
    int begin = N / total * rank;
    int end = (rank == total - 1) ? N : N / total * (rank + 1);
    cout << "rank " << rank << " from " << begin << " to " << end << endl;
    if (rank == 0) {
        A_init();
        cout << "initialize success:" << endl;
        print(A);
        cout << endl << endl;
        for (j = 1; j < total; j++) {
            int b = j * (N / total), e = (j == total - 1) ? N : (j + 1) * (N / total);
            for (i = b; i < e; i++) {
                MPI_Send(&A[i][0], N, MPI_FLOAT, j, 1, MPI_COMM_WORLD);发送数据
            }
        }

    }
    else {
        A_initAsEmpty();
        for (i = begin; i < end; i++) {
            MPI_Recv(&A[i][0], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
        }

    }

    MPI_Barrier(MPI_COMM_WORLD);
    start_time = MPI_Wtime();
    for (k = 0; k < N; k++) {
        if ((begin <= k && k < end)) {
            for (j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
            for (j = 0; j < total; j++) { //
                if(j!=rank)
                    MPI_Send(&A[k][0], N, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
        }
        else {
            int src;
            if (k < N / total * total)
                src = k / (N / total);
            else
                src = total-1;
            MPI_Recv(&A[k][0], N, MPI_FLOAT, src, 0, MPI_COMM_WORLD, &status);
        }
        for (i = max(begin, k + 1); i < end; i++) {
            for (j = k + 1; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
```

```
            A[i][k] = 0;
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        end_time = MPI_Wtime();
        printf("耗时: %.4lf ms\n", 1000 * (end_time - start_time));
        print(A);
    }
    MPI_Finalize();
}
```

x86 优化算法呈现:

x86 上 MPI 优化算法

```

double mpi_optimize(int argc, char* argv[]) {
    double start_time = 0;
    double end_time = 0;
    MPI_Init(&argc, &argv);
    int total = 0;
    int rank = 0;
    int i = 0;
    int j = 0;
    int k = 0;
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &total);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int begin = N / total * rank;
    int end = (rank == total - 1) ? N : N / total * (rank + 1);
    if (rank == 0) {  //0号进程初始化矩阵
        A_init();

        for (j = 1; j < total; j++) {
            int b = j * (N / total), e = (j == total - 1) ? N : (j + 1) * (N / total);
            for (i = b; i < e; i++) {
                MPI_Send(&A[i][0], N, MPI_FLOAT, j, 1, MPI_COMM_WORLD);
            }
        }

    }
    else {
        A_initAsEmpty();
        for (i = begin; i < end; i++) {
            MPI_Recv(&A[i][0], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
        }

    }

```

```
35      MPI_Barrier(MPI_COMM_WORLD);
36      start_time = MPI_Wtime();
37      for (k = 0; k < N; k++) {
38          if ((begin <= k && k < end)) {
39              for (j = k + 1; j < N; j++) {
40                  A[k][j] = A[k][j] / A[k][k];
41              }
42              A[k][k] = 1.0;
43              for (j = 0; j < total; j++) { //
44                  if (j != rank)
45                      MPI_Send(&A[k][0], N, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
46              }
47          }
48          else {
49              int src;
50              if (k < N / total * total)
51                  src = k / (N / total);
52              else
53                  src = total - 1;
54              MPI_Recv(&A[k][0], N, MPI_FLOAT, src, 0, MPI_COMM_WORLD, &status);
55          }
56          for (i = max(begin, k + 1); i < end; i++) {
57              for (j = k + 1; j < N; j++) {
58                  A[i][j] = A[i][j] - A[i][k] * A[k][j];
59              }
60              A[i][k] = 0;
61          }
62      }
63      MPI_Barrier(MPI_COMM_WORLD);
64      if (rank == 0) {
65          end_time = MPI_Wtime();
66          printf("耗时: %.4lf ms\n", 1000 * (end_time - start_time));
67      }
68      MPI_Finalize();
69      return end_time - start_time;
70  }
```

1.3 优化步骤

1.3.1 普通优化结果

通过测试，MPI 在小规模数据情况下可能会出现负优化的效果，这是因为各进程间频繁通信导致的额外开销。因此，对于小规模数据，MPI 并不一定适合。然而，随着数据量的增大，更多进程的优势才会显现出来。例如，在 ARM 平台上，直到数据规模达到 3000 时，12 进程才成为最优选择，这表明了 MPI 在大规模数据处理时的潜力。

MPI 展现出了很好的加速效果，特别是在 x86 平台上。在数据规模较大时，例如在 3000 的情况下，4 进程已经有了明显的优化效果，加速比稳定在 2.5 以上。随着进程数的增加至 6 和 8，加速比进

一步提升，8 进程的加速比甚至超过了 4。这说明 MPI 在 x86 平台上具有出色的性能表现，特别是在处理大规模数据时。

MPI 在大规模数据处理时表现出了良好的加速效果，在 x86 平台尤其突出。尽管在小规模数据情况下可能存在负优化的问题，但随着数据量的增大，MPI 能够充分发挥多进程并行计算的优势，实现更高的计算效率。

在 x86 平台优化结果如下：

表 1: 不同进程数下的计算时间（单位：毫秒）

数据规模	100	500	1000	1500	2000	3000
平凡算法	0.46	130.47	1129.43	3743.72	9002.46	30394.11
4 进程	0.56	52.41	419.88	1354.87	2944.36	10150.49
6 进程	1.00	39.53	293.73	998.77	2298.79	8292.88
8 进程	1.69	40.43	276.19	888.50	1887.85	7247.39

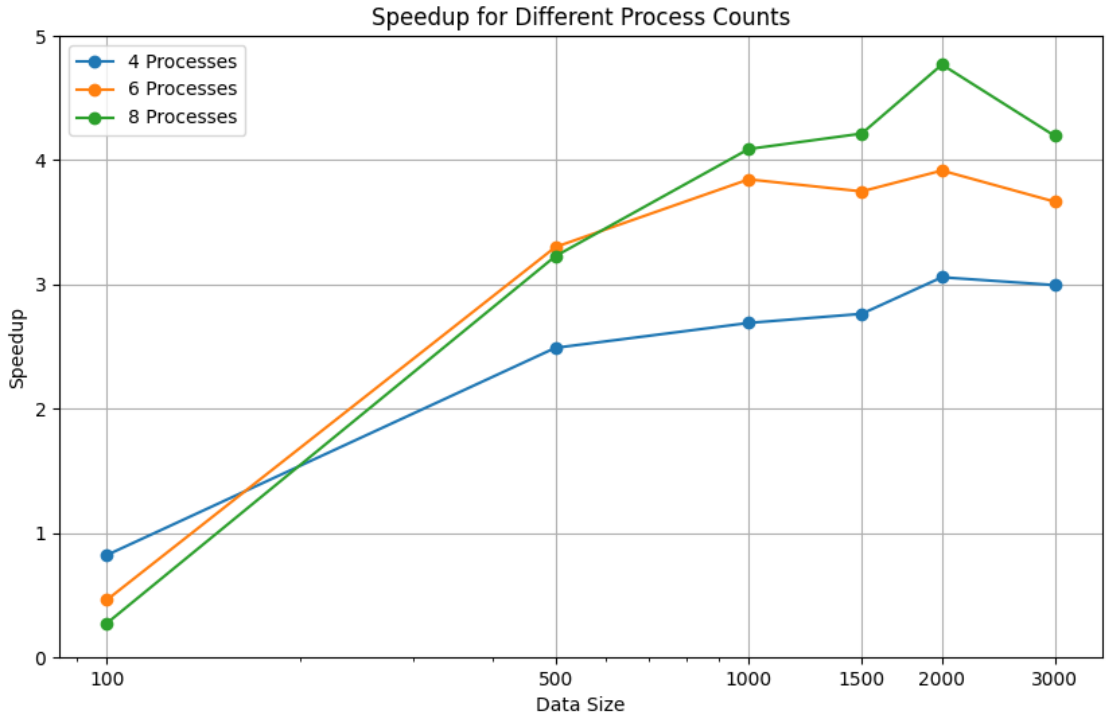

图 1.1: x86 平台 MPI 优化加速比

ARM 平台 在 ARM 平台的不同进程下，经测试，运行时间如表所示

表 2: 不同进程数下的计算时间（单位：秒）

数据规模	100	500	1000	1500	2000	3000
平凡算法	0.029	0.294	2.454	8.982	20.938	73.967
4 进程	0.078	0.793	1.847	5.466	13.082	45.029
8 进程	0.186	0.907	2.382	4.678	8.722	29.708
12 进程	0.327	1.506	3.434	6.085	9.403	25.719

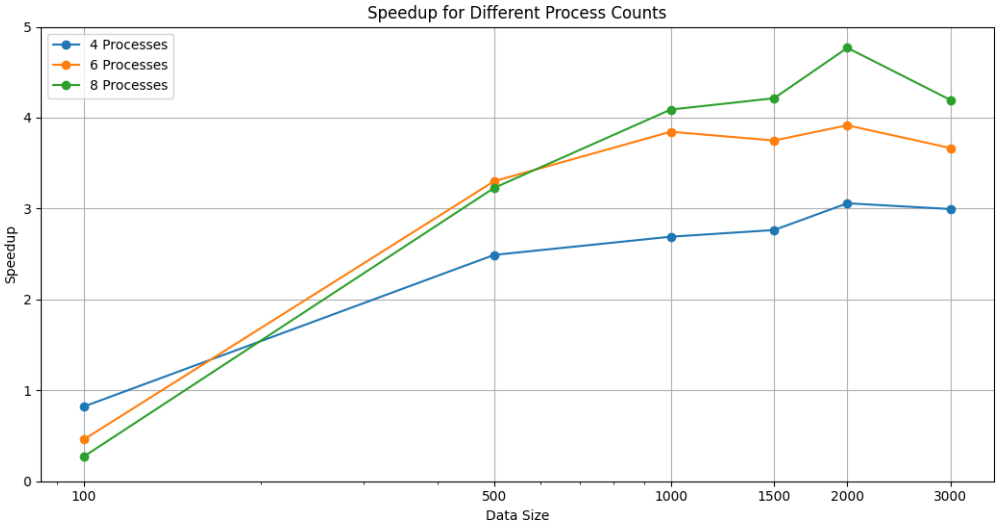

图 1.2: ARM 平台 MPI 优化加速比

1.3.2 不同任务划分测试

此外，我们还对不同的任务划分的优化结果进行了探究，包括块划分、优化后的块划分，以及循环划分，得到结果如下

表 3

Data Size	1000	1500	2000	3000
x86 块划分 (ms)	419.29	1274.28	3062.72	10166.89
x86 优化块划分 (ms)	404.36	1351.04	2994.42	11540.00
x86 循环划分 (ms)	363.10	1039.09	2762.43	9632.18
ARM 块划分 (s)	2.34	5.60	15.70	50.85
ARM 优化块划分 (s)	2.11	6.26	14.90	46.52
ARM 循环划分 (s)	1.59	6.17	13.34	40.81

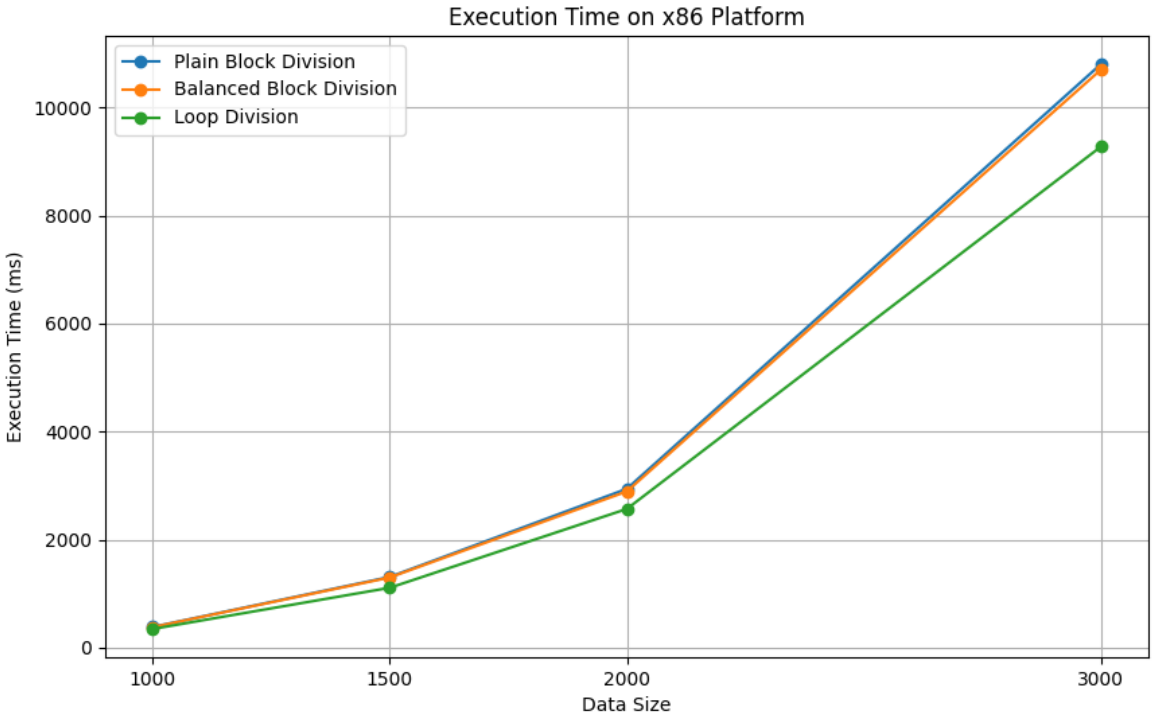

图 1.3: x86 平台运行时间对比

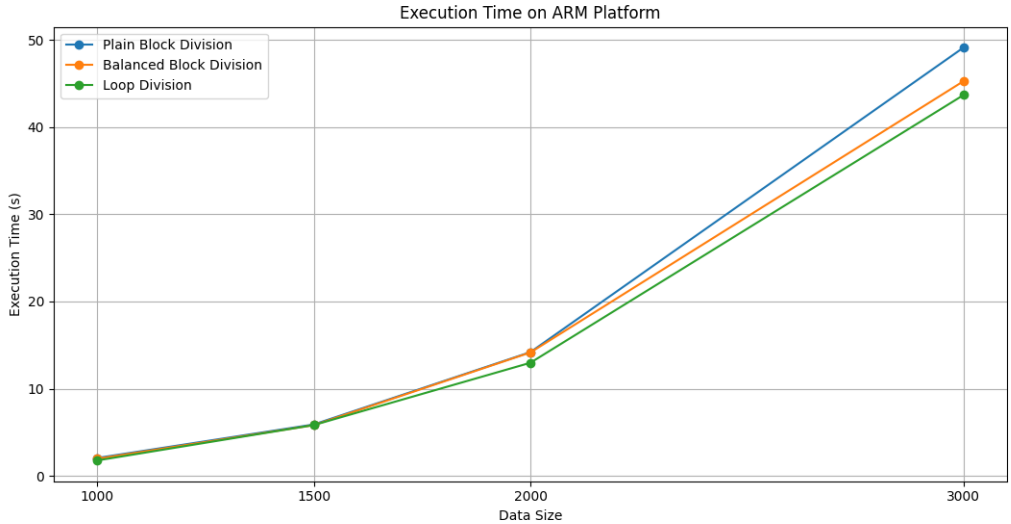

图 1.4: ARM 平台运行时间对比

可以看到，对末尾任务进行平均划分的块划分方式和普通块划分差别不大，这是因为在使用 4 个进程时，末尾的多余任务通常少于 4 个行向量。除非每个行向量的计算量非常巨大，否则这种方法和普通块划分在性能上几乎没有差别。这是因为大部分计算任务在两种块划分方式下都会被均匀地分配到各个进程中，剩余的少量任务对整体计算时间的影响是非常有限的。

循环划分是所有块划分方式中耗时最少的一种，这是因为这种方式的任务划分非常均衡。每个进

程几乎同时开始和结束计算任务，从而最大限度地利用了并行计算资源。循环划分的这一优势在不同的平台上表现出一致性，特别是在 x86 平台和 ARM 平台上都表现出了显著的优越性。在 x86 平台上，循环划分同样能够均匀分配计算任务，减少了进程间的等待时间，从而提高了整体计算效率。

在 ARM 平台上，循环划分的效果也非常明显，因为 ARM 处理器的计算能力和内存带宽相对较低，因此更加需要高效的任务划分来提高并行计算的效率。通过均匀分配任务，循环划分减少了因任务不均衡带来的计算资源浪费，使得每个进程都能尽可能多地参与计算工作，从而缩短了总的计算时间。

总结来看，尽管对末尾任务进行平均划分的块划分方式在一定程度上改进了任务分配的均衡性，但由于多余任务的数量较少，其效果并不明显。而循环划分方式由于其高度均衡的任务分配，显著提高了计算效率，在 x86 平台和 ARM 平台上均表现出明显的优势。因此，循环划分是高性能计算中非常有效的一种任务分配策略，无论是在何种计算平台上，都能显著减少计算时间，提高并行计算的性能。

1.3.3 非阻塞通信

在此前的 MPI 优化中，均采用了阻塞方式的通信。阻塞通信的优点在于其简单易用，能够方便地进行进程间的同步。由于发送进程在发送消息后必须等待接收进程的响应，因此可以避免出现竞争条件。这种方式确保了数据传递的顺序性和一致性，减少了由于数据竞争导致的错误。然而，阻塞通信也有显著的缺点，即会导致整个程序的效率降低。这是因为发送进程必须等待所有接收进程处理完消息之后才能继续执行其后续操作。这种等待机制会导致进程闲置时间的增加，特别是在处理大规模数据或需要频繁通信的情况下，效率问题尤为突出。

在高斯消元算法中，每个进程需要处理的数据相互独立，不会相互干扰，因此理论上适合采取非阻塞通信方式。非阻塞通信允许发送进程在发送消息后立即返回，不必等待接收进程的响应，从而可以继续进行其他计算任务。这种方式能够显著提高并行计算的效率，因为进程可以更好地利用计算资源，减少因等待而浪费的时间。因此，尽管非阻塞通信可以提高效率，但在具体的实现过程中，依然需要确保数据接收的完整性和顺序性。

为了在非阻塞通信中实现这一点，可以在发送消息后使用 MPI 的非阻塞通信函数（如 MPI_Isend 和 MPI_Irecv），并在需要确保数据接收完毕时使用 MPI_Wait 或 MPI_Waitall 函数。这些函数能够确保所有非阻塞通信操作完成后再进行下一步计算，从而在保持非阻塞通信效率的同时，确保数据的一致性和正确性。因此，在高斯消元算法的并行优化中，非阻塞通信版本的代码不仅能够提高整体效率，还需要在关键步骤中进行适当的同步，以确保计算结果的正确性。

表 4: Execution Time for Different MPI Versions (in milliseconds)

Data Size	500	1000	1500	2000	3000
Plain Algorithm	126.25	1091.32	3683.89	9197.15	27283.85
Blocking MPI	48.26	379.51	1334.93	3247.27	11314.29
Non-Blocking MPI	50.97	315.82	1054.96	3018.88	9701.10

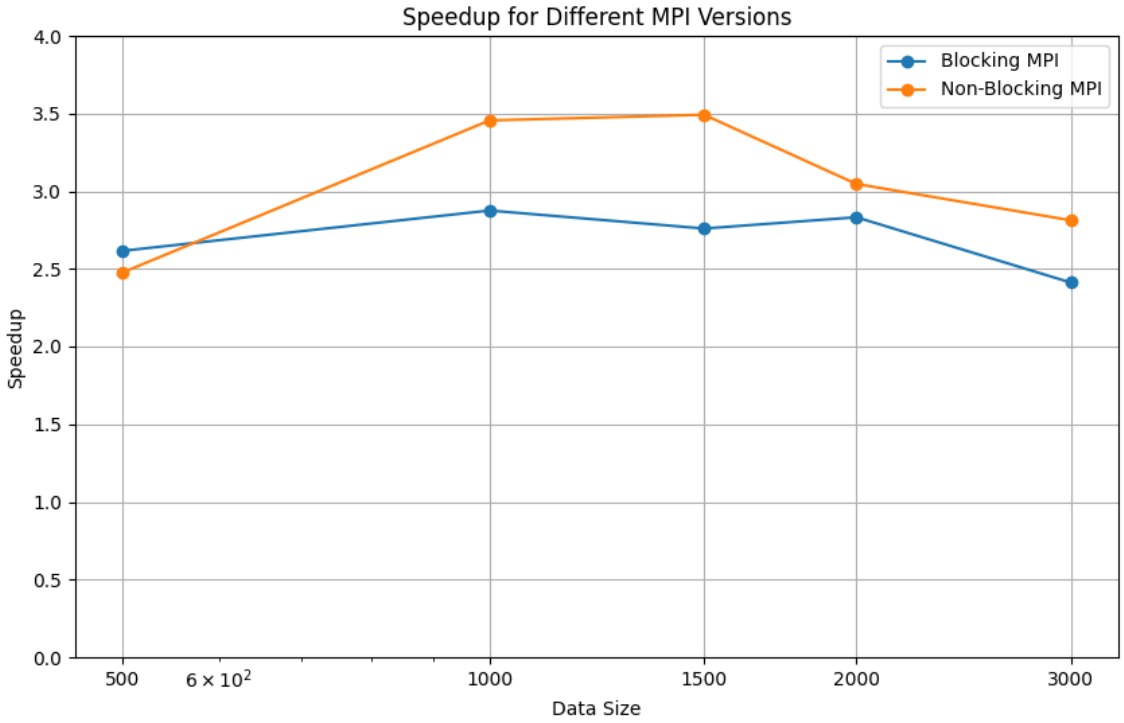

图 1.5: 非阻塞通信优化结果

1.3.4 与 SIMD 和多线程结合

表 5: Execution Time for Different MPI Versions (in milliseconds)

Data Size	500	1000	1500	2000	3000
Ordinary MPI	57.47	339.72	1384.00	3226.84	11124.79
Non-blocking MPI	42.52	317.68	1097.53	2913.79	11040.18
Non-blocking + AVX	27.42	136.58	835.97	1866.90	6334.85
Non-blocking + OpenMP	19.14	98.43	311.31	684.30	3850.21
Non-blocking + OpenMP + AVX	16.59	58.25	172.00	337.71	1536.71

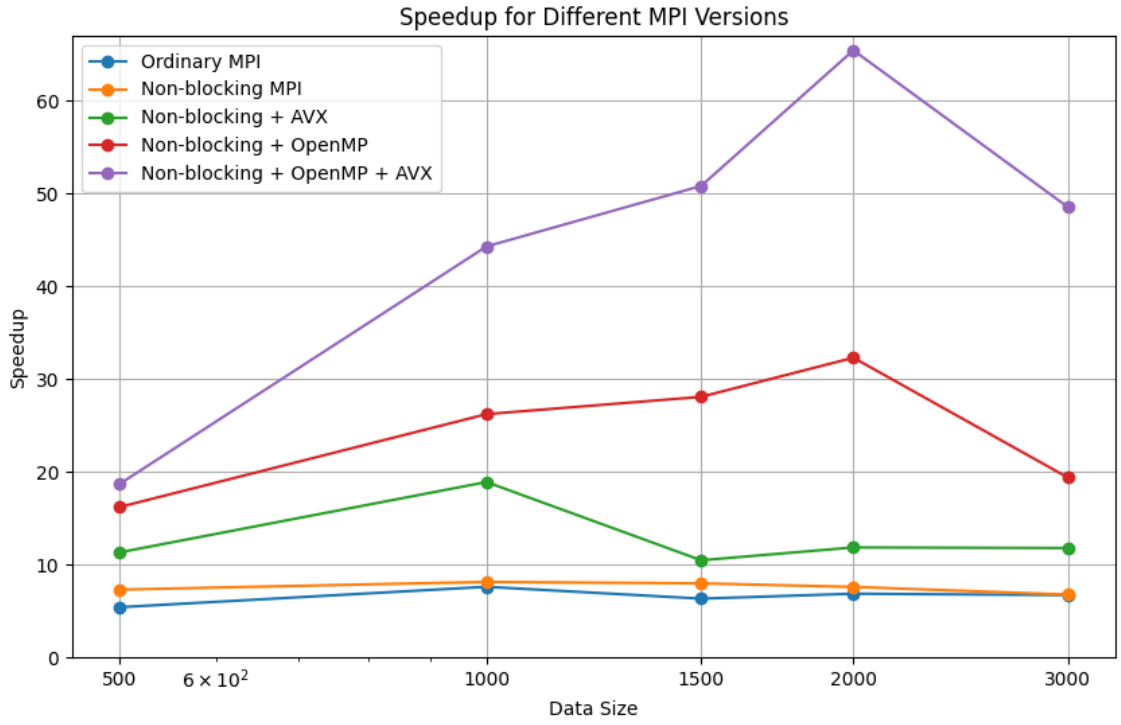

图 1.6: SIMD 与多线程结合结果

可以看到，与 OpenMP 和 AVX 结合后的程序性能显著提高，加速比竟然达到惊人的 66 倍，大大加快了执行速度。这种显著的性能提升，表明了非阻塞通信与现代处理器的并行计算技术相结合所能带来的巨大优势。

当数据量较小时（例如 500），加速比的提升无论是和 AVX、OpenMP 单独结合还是同时结合二者，效果都不是那么明显。这三种方式之间的性能差距也不大，推测原因是数据量较小，计算任务不足以充分利用 AVX 和 OpenMP 的并行计算优势。在此前对 SIMD（单指令多数据流）和 OpenMP 的探究中，也出现了类似的现象。当数据量较小时，并行计算的优势无法完全体现出来，因为并行处理带来的开销（例如线程启动、数据分配等）可能抵消了其带来的性能提升。因此，当数据量较小时，虽然使用 AVX 和 OpenMP 能够提升性能，但提升幅度并不如预期的那么大。

然而，当数据量增加时，加速比显著提升。数据量增加意味着更多的计算任务和数据处理需求，这正是并行计算和 SIMD 技术所擅长的领域。在数据量达到 1500 时，加速比显著提高，这说明 AVX 和 OpenMP 技术在大规模数据处理中的优势得到了充分发挥。AVX 技术能够在一次指令中处理多个数据，极大地提高了处理器的计算能力；OpenMP 则通过多线程并行计算，进一步提升了计算效率。

然而，在数据量达到 3000 规模时，加速比开始下降。可能的原因是进程间通信增多或计算量增大，导致了额外的开销。特别是当使用 AVX 和 OpenMP 同时结合时，虽然理论上能够获得更高的加速比，但实际上由于进程间需要频繁通信、数据同步以及负载均衡等问题，加速比反而有所下降。

这种现象表明，虽然非阻塞通信结合 AVX 和 OpenMP 能够显著提升计算性能，但在实际应用中，还需要考虑进程间通信和同步带来的开销。这些因素在大规模数据处理时尤为重要，需要通过优化通信策略和任务分配来进一步提升性能。

所以，非阻塞通信与 AVX、OpenMP 的有机结合使得算法在大规模数据处理中的性能得到了显著提升。尽管在小规模数据处理时效果不明显，但在大数据量情况下，这种结合能够大幅度加速计算，

充分发挥并行计算的优势。

1.3.5 流水线形式优化

在原先的算法中，每个进程会将消元完毕的行向量传递到整个进程域，然而在块划分中，实际上只有该进程之后的进程需要使用该行向量。因此，只需将数据传递给其之后的进程，就像流水线一般操作。这种方法可以减少约一半的数据传递操作，从而提高计算效率。最终在最后一个进程中得到最终结果。进过代码优化后的优化结果如下：

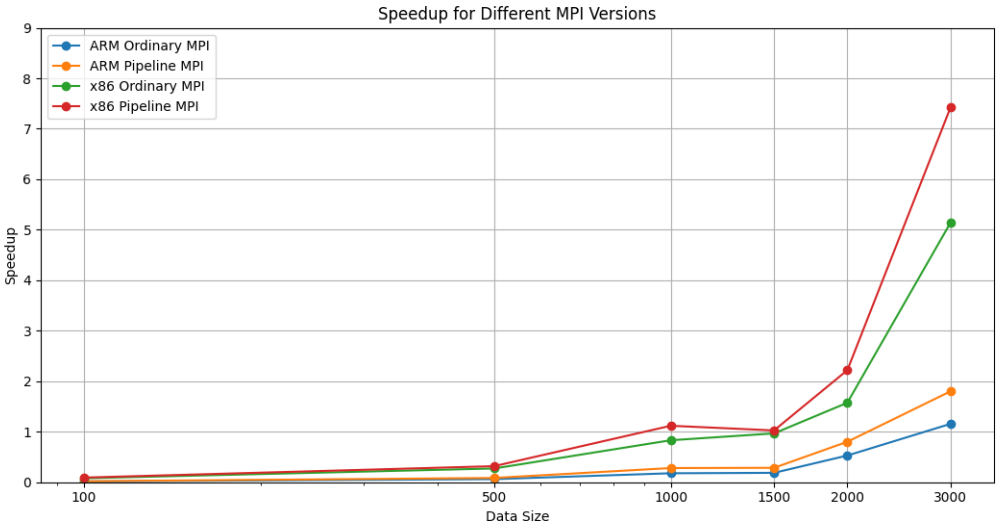

图 1.7: 流水线优化结果

表 6: Execution Time for Different MPI Versions (in seconds and milliseconds)

Data Size	ARM Ordinary MPI (s)	ARM Pipeline MPI (s)	x86 Ordinary MPI (ms)	x86 Pipeline MPI (ms)
100	2.06	1.63	355.13	302.26
500	5.55	3.96	1208.83	1032.46
1000	13.99	8.80	2954.91	2195.13
1500	49.80	32.28	9473.24	8927.68
2000	43.88	28.91	14693.55	10418.36
3000	54.89	35.21	12321.15	8532.29

减少信号传递次数后的流水线版本 MPI 在速度上确实有提升，使得算法执行更快。然而，提升幅度并不是非常显著。推测原因可能是因为在原先的运行中，数据传递所占用的时间并不算高，因此优化后的提升幅度相对有限。不过，这仍然说明了这种优化思路的正确性。通过减少不必要的数据传递，可以有效地减少通信开销，提高并行计算的效率。

在大多数并行计算任务中，数据传递和通信往往是影响性能的关键因素之一。尽管在本次优化中，数据传递占用的时间并不显著，但在更大规模或更复杂的计算任务中，这种优化思路可能会带来更明显的性能提升。流水线版本的 MPI 优化通过减少进程间的通信次数，充分利用了进程间的数据依赖关系，体现了有效的优化策略。

个人认为这种优化方法的一个重要优点是其适用性强，不仅限于当前的高斯消元算法。在其他需要频繁数据传递的并行计算任务中，同样可以采用类似的流水线优化策略。通过分析和理解进程间的数据依赖关系，减少不必要的数据传递和通信，可以提高整体计算效率。

1.4 结果分析

在此次实验进行的十分广泛的研究，我们对普通高斯消元法进行了多种 MPI 优化，包括普通 MPI、非阻塞 MPI、不同任务划分，与先前实验 SIMD 和多线程编程的结合，以及更多 ARM 和 x86 平台的不同优化版本。这些优化在不同程度上提升了算法的执行效率。普通 MPI 实现了基本的并行计算，但由于采用阻塞通信方式，进程间的等待时间较长，整体性能提升有限。而非阻塞 MPI 通过采用非阻塞通信方式，减少了进程间的等待时间，显著提升了计算效率，相比于普通 MPI 有较为明显的改进。非阻塞 +AVX 进一步利用 AVX 指令集加速了向量运算，进一步提升了性能。非阻塞 +OpenMP 通过多线程并行化进一步提升了计算效率，尤其在较大数据规模下效果显著。非阻塞 +OpenMP+AVX 结合了两者的优势，实现了最大程度的优化，达到了最高的加速比。

本次最显著的优化方式是与 SIMD 和多线程结合，以及采用流水线形式优化。与 SIMD 和多线程结合后的程序性能大幅增加，加速比最高达到了 66 倍，这种显著的性能提升显示了非阻塞通信与现代处理器并行计算技术相结合的巨大优势。而流水线形式优化则通过减少不必要的数据传递，使得算法执行更快，尽管提升幅度相对有限，但仍然验证了这种优化思路的正确性。

然后通过对不同任务划分的探究，发现循环划分在所有块划分方式中耗时最少，表现出显著的优越性。这种任务划分方式能够充分利用并行计算资源，减少了进程间的等待时间，提高了整体计算效率。在非阻塞通信方面，尽管在小规模数据处理时效果不明显，但在大规模数据情况下，非阻塞通信结合 AVX、OpenMP 等技术能够大幅度加速计算，充分发挥并行计算的优势。最后，在流水线形式优化中，减少了不必要的数据传递，通过分析和理解进程间的数据依赖关系，提高了整体计算效率。这些优化方法也不仅限于高斯消元算法，也可应用于其他需要频繁数据传递的并行计算任务中，从而进一步提高计算性能，这也是本次实验的价值所在。

2 特殊高斯消元法 mpi 并行优化

2.1 算法设计

MPI（Message Passing Interface）消息传递接口是一种高效的并行编程方法，与 pthread 不同，pthread 申请的是多线程，MPI 申请的则是多进程，pthread 的线程之间通过共享内存进行通信和数据交换，通信开销较低，同时 pthread 提供了多种同步机制，如互斥锁（mutex）、条件变量（condition variable）和读写锁，以协调线程之间对共享资源的访问。对于 mpi，多个进程在各自独立的内存空间中运行，需要通过显式的消息传递进行通信。MPI 创建的是进程，进程之间不共享内存，各自拥有独立的资源。

在本次实验中，使用 mpi 的编程方法对程序进行了优化，由于 mpi 各进程之间内存不共享，因此如何进行任务的划分以及进程之间的通信是本次实验中主要探究的问题。对于任务划分，块划分与循环划分这两种划分方式在特殊高斯消元中当中都是可行的，块划分指的是将任务划分成连续的块，每个块分配给一个进程处理，每个进程负责相同大小的一个连续部分的消元任务，而循环划分指的是，每个进程负责第 i，i+p，i+2p... 这些消元行的消元任务。对于通信方式，各个进程之间对于被消元行的消元本身是并不互斥的，只需要使用先前获得的消元子进行消元即可，但是当该被消元子消元之后需要升格为消元子之后就涉及进程之间的通信，如果使用非阻塞通信，可能会出现两个进程同时需要消元子更新的过程，虽然会导致代码的复杂性增加但是非阻塞通信也可以提升程序的效率，因此本次实验中的通信方式选择非阻塞通信，也就是消元子更新信息的传递不会停止任务。

同时，本次实验在 Windows 和鲲鹏服务器平台进行，设计了 mpi 与其他并行方法结合对比的实验，在 arn 架构下编写了以下几种并行方法：mpi、mpi+neon、mpi+openmp（4）、mpi+openmp（4）

+neon、mpi+openmp (8) +neon 这五个对比实验，在 windows 架构下编写了 mpi、mpi+avx+openmp 的方法，最后在 mpi+openmp (8) +neon 的条件下对比循环划分与块划分的任务划分方式。

2.2 编程实现

根据算法设计中给出的设计实现如下代码，这里仍以 mpi+openmp (8) +neon 方法，块任务划分的方法为例，代码如下：

MPI+openmp+neon 算法

```
void GE_mpi_neon_openmp(int rank, int size, int num_threads) {
    //读文件
    MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int num = (flag == -1) ? maxrow : flag;
    auto start = chrono::high_resolution_clock::now();
    vector<MPI_Request> send_requests;
    #pragma omp parallel for num_threads(num_threads)
    for (int i = rank; i < num; i += size) {
        while (findfirst(i) != -1) {
            int first = findfirst(i);
            if (ifBasis[first] == 1) {
                // 使用NEON异或消元
            } else {
                for (int j = 0; j < maxsize; j += 4) {
                    uint32x4_t neon_rows =
                        vld1q_u32(reinterpret_cast<uint32_t*>(&gRows[i][j]));
                    vst1q_u32(reinterpret_cast<uint32_t*>(&gBasis[first][j]),
                        neon_rows);
                }
                ifBasis[first] = 1;
                ans.insert(pair<int, int*>(first, gBasis[first]));
                // 将新消元子发送给其他进程
                for (int dest = 0; dest < size; dest++) {
                    if (dest != rank) {
                        MPI_Request request;
                        MPI_Isend(&gBasis[first], maxsize, MPI_INT, dest, first,
                            MPI_COMM_WORLD, &request);
                        #pragma omp critical
                        {
                            send_requests.push_back(request);
                        }
                    }
                }
                break;
            }
        }
    }
    // 持续处理直到所有发送操作完成
    bool all_done = false;
```

```
37      while (!all_done) {
38          all_done = true;
39          // 接收来自其他进程的新消元子，并尝试消元
40          for (int source = 0; source < size; source++) {
41              if (source != rank) {
42                  MPI_Status status;
43                  int flag;
44                  MPI_Iprobe(source, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
45                  if (flag) {
46                      int first = status.MPI_TAG;
47                      int* received_basis = new int[maxsize]; // 分配接收缓冲区
48                      MPI_Recv(received_basis, maxsize, MPI_INT, source, first,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
49                      #pragma omp critical
50                      {
51                          ifBasis[first] = 1;
52                          // 使用NEON异或消元
53                          // 尝试对接收到的新消元子进行消元操作
54                          while (try_eliminate(received_basis)) {
55                              // 如果消元后产生新的消元子，继续传播给其他进程
56                              for (int dest = 0; dest < size; dest++) {
57                                  if (dest != rank) {
58                                      MPI_Request request;
59                                      MPI_Isend(received_basis, maxsize, MPI_INT, dest,
                                            first, MPI_COMM_WORLD, &request);
60                                      send_requests.push_back(request);
61                                  }
62                              }
63                          }
64                      }
65                      delete[] received_basis; // 清理接收缓冲区
66                  }
67              }
68          }
69          // 检查是否所有发送操作完成
70          for (auto& request : send_requests) {
71              int flag;
72              MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
73              if (!flag) {
74                  all_done = false;
75              }
76          }
77      }
78  }
79  bool try_eliminate(int* basis) {
80      int first = findfirstbasis(basis); // 找到首项的位置
81      if (ifBasis[first] == 1) {
82          // 使用NEON异或消元
83          // 检查新的basis是否为零行
```

```
84          for (int j = 0; j < maxsize; ++j) {
85              if (basis[j] != 0) {
86                  return true; // 产生新的非零行，消元成功
87              }
88          }
89          return false; // 全零行，无需进一步操作
90      }
91      return false; // 无法消元
92  }
```

在代码当中，首先还是读取所有消元子，然后使用 Bcast 方式将读取的文件广播到所有进程，新建三个变量 send_request：后续消元子需要更新的时候使用 MPI_Send 把新消元子传输到其他进程，将返回的 MPI_Request 储存在这个向量当中，后续使用 MPI_Wait 检查发送是否完成，确保所有发送操作都被正确完成；send_indices：储存每个发送请求对应的消元子首项，用于记录记录和跟踪哪些消元子已经被发送，以及对应的请求是什么；received_basis_data：是一个 map 类型的变量，后续用来储存接收到的新消元子。接着进入消元过程，openmp 与 neon 消元的过程没有对先前的代码做出改进，在需要升格消元子的时候，除自身消元子更新，使用 MPI 编程将新消元子传输给其他进程，再储存 request 信息。结束消元循环之后需要接受来自其他进程的消元子，这里使用了一个循环，循环条件是有未结束的 request 信息那，之后先使用 MPI_Iprobe 检测是否有需要接受的信息，如果有就说明接收到了其他进程的消元子，但是自身可能已经拥有该消元子首项对应消元子，因此需要尝试对该消元子消元，如果又产生了需要升格的消元子，那么需要继续传递到其他进程，while 循环也是为了确保不会出现死锁的情况，因为接受信息就有可能传递新消元子的信息，只接受一次信息可能会导致死锁情况出现，所以每次循环中都对所有 request 做一次检查，只要有未成功的 request 就要重新循环接受信息。

2.3 性能测试

本次实验当中 mpi 进程数量均为 4，以下是在鲲鹏服务器下进行的实验结果：

arm 实验结果	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10	例 11
平凡算法	0	12	16	418	1613	21431	121943	919782	1315360	3546079	1375
MPI	2	8	5	91	294	6732	39231	306386	381401	747354	257
MPI+neon	1	8	9	224	248	5177	31260	237414	287429	542046	149
MPI+openmp	2	17	6	232	184	2785	15885	132367	184115	292446	99
MPI+neon+openmp4	1	17	5	94	207	1985	12195	94417	126355	247669	56
MPI+neon+openmp8	2	15	11	76	213	1604	6170	53725	65889	152318	48

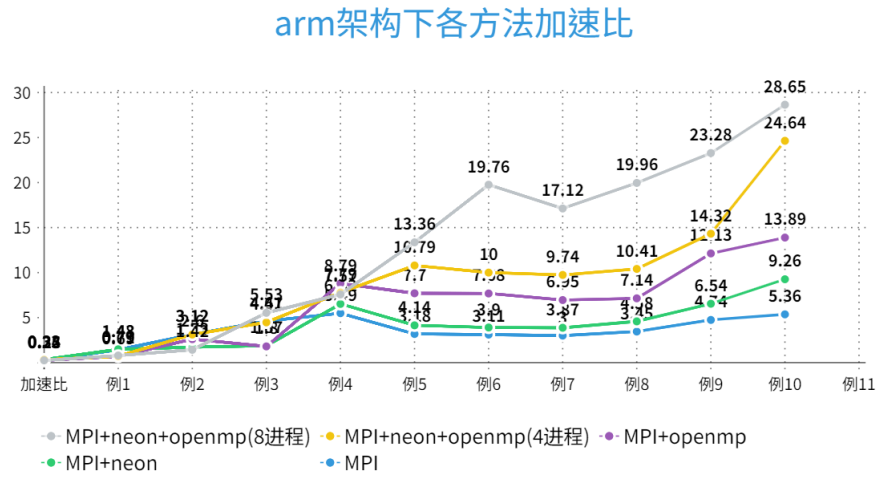

图 2.8: arm

以下是在 windows 下运行的实验结果：

windows	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10	例 11
平凡算法	0	9	10	268	836	9233	52857	361128	682589	2122691	676
MPI	0	2	2	69	185	2617	11788	84371	165929	552470	178
MPI+avx+openmp(8)	1	5	6	65	143	1095	5951	27151	48096	144247	49

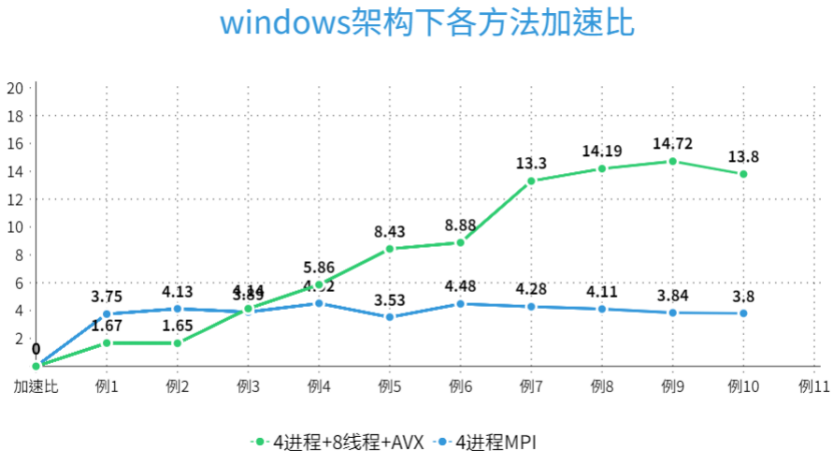

图 2.9: Windows

2.4 结果分析

windows 架构下进行的实验，只使用 mpi 加速的时候，加速比没有随问题规模变化产生较大的浮动，基本稳定在 4 左右的加速比，而 mpi 与 avx、openmp 结合之后，加速比就受到了问题规模的影响，矩阵规模较小的例 1-3 测试加速比较低不具备太大的参考价值，因为创建线程，进程间进行通信的成本可能还大于直接计算的成本，在例 4-6 中规模的问题规模测试中，基本能取得 6-8 倍的加速比，而随着例 7-10 中问题规模的膨胀，加速比反而得到了进一步的提升，加速比稳定在 14 左右。对于两种对比实验下加速比和问题规模大小不同的关系，推测是在小中规模下的并行方法并没有完全发挥对应的优势，在小规模实验下，avx 与 openmp 的额外开销对结果的影响较为明显，例如在例 2、3 中使用 avx 与 openmp 甚至起到了负优化的影响，而随着问题规模的逐渐扩大，计算密度也越来越大，AVX

能够充分利用向量寄存器，OpenMP 能够更有效地分配和管理线程，导致加速比逐渐增大，实际上问题规模在例 6-例 7 的扩大倍数是最大的，这也对应了加速比在例 6-例 7 产生了大幅度变化，说明问题规模越大越能发挥 avx、openmp 这类并行方法的作用，而 mpi 网络通信的开销较为稳定，可以取得一个稳定的加速比效果。

arm 架构下，MPI 算法加速比在 4-5 左右，MPI+neon 算法加速比也在 5 左右，但是在稀疏矩阵上取得了显著的性能提升，加速比可以达到 10 倍左右，而 MPI+openmp 算法的加速比较不稳定，浮动区间在 6-12 之间，MPI+neon+openmp 算法随着问题规模的扩大加速比不断扩大，MPI+neon+openmp(8 线程) 的算法取得了本学期并行编程的最好加速比，在大规模问题上可以取得 20 倍以上的加速比，说明 mpi 与 neon、openmp 的结合是非常成功的。同时注意到 neon 相比于 mpi 和 openmp 方法对稀疏矩阵可能有比较好的加速效果，因为使用 MPI+neon 和使用 MPI+openmp 在例 10-例 11 加速比的变化中，前者是显著高于后者的。

最后观察结果输出的文件大小，发现在大规模下的输出与先前实验产生的结果存在几 kb 的区别，分析本次实验中设计的算法认为可能是因为 MPI 的通信取决于各个进程消元进度的差别，也就是进度越快的进程产生的消元子会更早的被其他进程接受并使用，并且由于每个进程可能需要对接收到的消元子进一步尝试消元再传输给其他进程，因此在整个消元过程中，每个进程所拥有的消元子可能是不同的，这也导致了结果的不同，并且由于这个因素，对于消元结果的正确性验证来说也更为困难，但是由于大小上并没有产生特别明显的偏差，应该可以说明本次实验设计的 MPI 编程算法大体是正确的。

3 实验分工以及仓库地址

本次实验由赵元鸣 2211757 进行普通高斯消去法部分的研究，由苏胤华 2213893 进行特殊高斯消去法部分的研究。

我们的代码仓库地址为：https://github.com/NanGong-WenYa/Parallel_Programming，欢迎您浏览。