南開大學
Nankai University

计算机学院

并行程序设计实验报告

MPI 编程实验

姓名：陆遥

学号：2211843

专业：计算机科学与技术

2024 年 6 月 9 日

目录

1 问题描述

数学上，高斯消元法（或译：高斯消去法），是线性代数规划中的一个算法，可用来为线性方程组求解。但其算法十分复杂，不常用于加减消元法，求出矩阵的秩，以及求出可逆方阵的逆矩阵。不过，如果有过百万条等式时，这个算法会十分省时。一些极大的方程组通常会用迭代法以及花式消元来解决。当用于一个矩阵时，高斯消元法会产生出一个“行梯阵式”。高斯消元法可以用在电脑中来解决数千条等式及未知数。亦有一些方法特地用来解决一些有特别排列的系数的方程组。

2 普通高斯消去算法

2.1 串行算法设计

高斯消去的计算模式如图 2.1 所示，主要分为消去过程和回代过程。在消去过程中进行第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 k + 1 至 N 行进行减去第 k 行的操作，全部结束后，得到如图 2.2 所示的结果。而回代过程从矩阵的最后一行开始向上回代，对于第 i 行，利用已知的 xi+1, xi+2, ..., xn 计算出 xi。串行算法如下面伪代码所示。

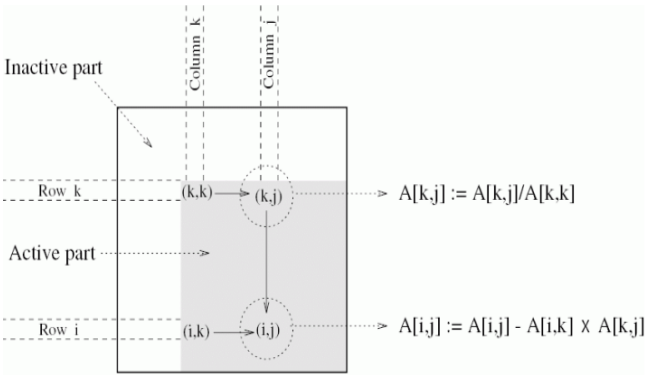

图 2.1: 高斯消去法示意图

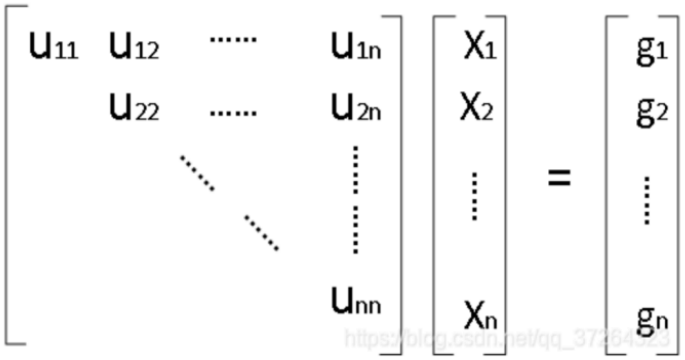

图 2.2: 高斯消去法消去过程结束后，结果的示意图

2.2 MPI 对比实验设计

2.2.1 问题规模大小

问题规模的大小主要体现在矩阵规模上，对于不同的并行方案，不同的问题规模会体现不同的并行优化效果，有时甚至会出现负优化的现象，这都是非常正常的。在实验中设置不同的问题规模，有助于在各规模范围内找到最优的并行优化方案。

2.2.2 设置不同进程数：节点数的影响

我们应该如何设置进程的数量，是否是越多越好，还有需要在一个合理的区间内。区别于普通的可执行程序只创建一个进程，通过 mpiexec −n num ./test 运行的程序会根据 −n 选项后的参数 num，创建 num 个进程运行 test 可执行程序。因此可以在实验中通过设置不同的 num，探索进程数对并行优化效果的影响。

2.2.3 数据划分：负载均衡问题

以 MPI 按一维（行）块划分消去并行处理为例，假设设置 m 个 MPI 进程、问题规模为 n，则给每一个进程分配 n/m 行的数据，对于第 i 个进程，其分配的范围为 [i ∗ (n − n%m)/m, i ∗ (n − n%m)/m + (n − n%m)/m − 1]，而最后一个进程分配 [(m − 1) ∗ (n − n%m)/m, n − 1] 行。注意，这种简单策略是将余数部分（n%m 行）都分配给了最后一个进程，大家可思考稍复杂些但负载更为均衡的分配策略——将余数部分均匀分配给前 n%m 个进程，每个进程一行。初始化矩阵等工作由 0 号进程实现，然后将分配的各行发送给各进程。在第 k 轮消去步骤，负责第 k 行的进程进行除法运算，将除法结果一对多广播给编号更大的进程，然后这些进程进行消去运算。消除过程完成后，可将结果传回 0 号进程进行回代，也可由所有节点进行并行回代。

2.2.4 任务划分：不同的循环划分方法

一维列（循环）块划分与一维行划分略有不同，在除法阶段，需要持有对角线上元素的进程将其广播给其他进程，然后所有进程对自己所负责的列元素进行除法计算；接下来无需广播除法结果，因为需要除法结果的后续行都由同一个进程负责，直接在本地进行消去计算即可。

2.2.5 探索更佳性能：与 SIMD 结合

结合之前学习的 SIMD 向量化知识，将 MPI 并行化与 SIMD 结合，使其能达到更好的性能。MPI 并行化并没有改变除法与消去内部的代码，因此将其用 SIMD 相关指令集进行向量化是可行的。

2.2.6 多线程与多进程：与 OpenMP 与 Pthreads 结合

进程 (process) 是操作系统进行资源分配的最小单元，线程 (thread) 是操作系统进行运算调度的最小单元。

通常情况下，我们在运行程序时，比如在 Linux 下通过./test 运行一个可执行文件，那么我们就相当于创建了一个进程，操作系统会为这个进程分配 ID 和堆栈空间等资源。如果这个可执行程序是个多线程程序（比如由 OpenMP 或 Pthread 编写的 cpp 文件编译而来），那么这个./test 进程在执行到某个特定位置时会创建多个线程继续执行。所以，进程可以包含多个线程，但每个线程只能属于一个进程。

2.2.7 通信方式：阻塞通信与非阻塞通信

顾名思义，标准模式的非阻塞通信就是在标准模式的基础上，增加了一些额外的功能，使得通信过程更加灵活。MPI 库提供了 MPI-Isend 和 MPI-Irecv 函数，可以实现非阻塞式的消息发送和接收。

非阻塞式通信中，发送和接收进程发起通信后立即返回，而不会等待通信操作完成。发送进程可以立即执行后续代码，而接收进程在通信完成之前不需要等待可以利用该时间执行其他操作。相比较而言，非阻塞式通信提供了更大的灵活性，但需要额外的代码来处理通信完成后的操作，以确保通信操作的正确性。

MPI-Isend 是 MPI-Send 的非阻塞版本，它启动一个标准非阻塞发送，调用后立即返回。该调用的返回只表示该消息可以被发送，并不意味着消息已经成功发送出去。

2.2.8 流水线算法的尝试

流水线算法是并行计算中一个非常有效的、常用的手段，它根据计算的依赖和递推关系制定多任务流水线流程。流水线技术是一种在程序执行时多条指令重叠进行操作的一种准并行处理实现技术，各种部件同时处理是针对不同指令而言的，它们可同时为多条指令的不同部分进行工作，以提高各部件的利用率和指令的平均执行速度。流水线的基本原理是把一个重复的过程分解为若干个子过程，前一个子过程为下一个子过程创造执行条件，每一个过程可以与其它子过程同时进行。流水线各段执行时间最长的那段为整个流水线的瓶颈，一般地，将其执行时间称为流水线的周期。

本实验中，我们将进入流水线算法，探索其是否能进一步的提高优化性能。

2.3 代码实现

篇幅所限，部分函数仅展示部分代码，省略与其他函数重复部分。

2.3.1 普通 MPI

每一个进程都会负责一部分行的消去工作，而在除法步骤处，则是会轮到自己负责行时进行除法运算，而没轮到时则只接受其他进程的除法结果。

```
1  void m_gauss_mpi_1(int n, float **m, int rank, int size)
2  {
3      if (rank != size - 1){
4          for (int k = 0; k < n; k++){
5              if (k <= rank * (n - n % size) / size + (n - n % size) / size - 1 && k >=
                   rank * (n - n % size) / size){
6                  for (int j = k + 1; j < n; j++){
7                      m[k][j] = m[k][j] / m[k][k];
8                  }
9                  m[k][k] = 1.0;
10                 for (int j = 0; j < size; j++)
11                     if (j != rank)
12                         MPI_Send(&m[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
13             }
14             else
15                 MPI_Recv(&m[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Algorithm 1: MPI 版本的普通高斯消元

Data: 系数矩阵 A[n,n], 进程号 myid, 负责的起始行 r1, 负责的终止行 r2, 进程总数 num

Result: 上三角矩阵 A[n,n]

```
1  for k = 0 to n-1 do
2      if r1 <= k <= r2 then
3          for j = k + 1; j < n; j++ do
4              A[k,j]=A[k,j]/A[k,k];
5          A[k,k] ← 1.0;
6          for j = 0; j < num; j++ do
7              MPI_Send(&A[k,0], n, MPI_FLOAT, j, ...);
8      else
9          MPI_Recv(&A[k,0], n, MPI_FLOAT, j, ...);
10     for i ← r1 to r2 do
11         for j = k + 1; j < n; j++ do
12             A[i,j] ← A[i,j]-A[k,j]*A[i,k];
13         A[i,k] ← 0;
14     k++;
15     ...
```

图 2.3: MPI 版本高斯消去伪代码

```
16              for (int i = rank * (n - n % size) / size; i < rank * (n - n % size) /
                    size + (n - n % size) / size; i++){
17                  for (int j = k + 1; j < n; j++){
18                      m[i][j] = m[i][j] - m[i][k] * m[k][j];
19                  }
20                  m[i][k] = 0;
21              }
22          }
23      }
24  }
```

2.3.2 负载均衡 MPI

区别在于将余数部分均匀分配给前 n%m 个进程，每个进程一行。

```
1  void m_gauss_mpi_6(int n, float** m, int rank, int size)
2  {
```

```
    if (rank < n / size && rank != size - 1){
        for (int k = 0; k < n; k++){
            if (k <= rank * (n - n % size) / size + (n - n % size) / size && k >=
                rank * (n - n % size) / size){
                for (int j = k + 1; j < n; j++){
                    m[k][j] = m[k][j] / m[k][k];
                }
                m[k][k] = 1.0;
                for (int j = 0; j < size; j++)
                    if (j != rank)
                        MPI_Send(&m[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
            else
                MPI_Recv(&m[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int i = rank * (n - n % size) / size; i <= rank * (n - n % size) /
                size + (n - n % size) / size; i++){
                for (int j = k + 1; j < n; j++){
                    m[i][j] = m[i][j] - m[i][k] * m[k][j];
                }
                m[i][k] = 0;
            }
        }
    }
}
```

2.3.3 循环划分 MPI

```
void m_gauss_mpi_2(int n, float** m, int rank, int size)
{
    if (rank != size - 1){
        for (int k = 0; k < n; k++){
            if (k <= rank * (n - n % size) / size + (n - n % size) / size - 1 && k >=
                rank * (n - n % size) / size){
                for (int j = 0; j < size; j++)
                    if (j != rank)
                        MPI_Send(&m[k][k], 1, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
            else
                MPI_Recv(&m[k][k], 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int j = rank * (n - n % size) / size; j < rank * (n - n % size) /
                size + (n - n % size) / size; j++){
                m[k][j] = m[k][j] / m[k][k];
            }
            m[k][k] = 1.0;
            for (int i = k + 1; i < n; i++){
                for (int j = rank * (n - n % size) / size; j < rank * (n - n % size)
```

```
                    / size + (n - n % size) / size; j++)
                {
                    m[i][j] = m[i][j] - m[i][k] * m[k][j];
                }
                m[i][k] = 0;
            }
        }
    }

}
```

2.3.4 OpenMP+MPI

```
void m_gauss_mpi_4(int n, float** m, int rank, int size)
{
    if (rank != size - 1){
        #pragma omp parallel for
        for (int k = 0; k < n; k++){
            if (k <= rank * (n - n % size) / size + (n - n % size) / size - 1 && k >=
                rank * (n - n % size) / size){
                #pragma omp parallel for
                 . . . . . .
            }
            else
                 . . . . . .
            #pragma omp parallel for
            for (int i = rank * (n - n % size) / size; i < rank * (n - n % size) /
                size + (n - n % size) / size; i++){
                #pragma omp parallel for
                . . . . . .
            }
        }
    }
}
```

2.3.5 Pthreads+MPI

```
typedef struct
{
    int t_id;
  int rank;
}threadParam_t;
void *threadFunc(void *param)
{
    threadParam_t *p = (threadParam_t*)param;
    int t_id = p->t_id;
    int rank = p->rank;
```

```
    for(int k=0;k<n;k++){
    int total_begin = max(average_row*rank,k+1);
      int total_end = average_row*(rank+1);
    int avgnum = (total_end-total_begin)/NUM_THREADS;
      int begin = total_begin + avgnum*t_id;
      int end = total_begin + avgnum*(t_id+1);
      if(t_id==6)
        end = total_end;
    sem_wait(&sem_workerstart[t_id]);
    if(avgnum>0)
    {
         for(int i=begin;i<end;i++){
                . . . . . .
         }
    }
    sem_post(&sem_main);
    sem_wait(&sem_workerend[t_id]);
    }
    pthread_exit(NULL);
}
int main(int argc,char* argv[])
{
    average_row = n/size;
    double tb = MPI_Wtime();
    sem_init(&sem_main,0,0);
    for(int t_id=0;t_id<NUM_THREADS;t_id++)
    pthread_create(&handles[t_id],NULL,threadFunc,(void*)(&param[t_id]));
    for(int k=0;k<n;k++)
    {
    if(k>=average_row*rank && k<average_row*(rank+1))
    {
            for(int j=k+1;j<n;j++)
                A[k][j] = A[k][j]/A[k][k];
            A[k][k] = 1.0;
        for(int j=0;j<size;j++)
      {
        if(rank != j)
            MPI_Send(&A[k][0],n,MPI_FLOAT,j,j,MPI_COMM_WORLD);
      }
    }
    else
      MPI_Recv(&A[k][0],n,MPI_FLOAT,k/average_row,rank,MPI_COMM_WORLD,&status);
        for(int i=0;i<NUM_THREADS;i++)
      sem_post(&sem_workerstart[i]);
    for(int i=0;i<NUM_THREADS;i++)
      sem_wait(&sem_main);
    for(int i=0;i<NUM_THREADS;i++)
      sem_post(&sem_workerend[i]);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

2.3.6 SIMD 部分示例代码

```
void m_gauss_simd(int n)
{
    __m128 vt, va, vaik, vakj, vaij, vx;
    for(int k = 0; k < n; k++){
        vt = _mm_set_ps1(m[k][k]);
        int j;
        for(j = k+1; j+4 <= n; j+=4){
            va = _mm_loadu_ps(&m[k][j]);
            va = _mm_div_ps(va, vt);
            _mm_storeu_ps(&m[k][j], va);
        }
        m[k][k] = 1.0;
        for(int i = k+1; i < n; i++){
            vaik = _mm_set_ps1(m[i][k]);
            int j;
            for(j = k+1; j+4 <= n; j+=4){
                vakj = _mm_loadu_ps(&m[k][j]);
                vaij = _mm_loadu_ps(&m[i][j]);
                vx = _mm_mul_ps(vakj, vaik);
                vaij = _mm_sub_ps(vaij, vx);
                _mm_storeu_ps(&m[i][j], vaij);
            }
            m[i][k] = 0;
        }
    }
}
```

2.3.7 非阻塞通信

```
int main(int argc,char* argv[])
{
    int average_row = n/size;
    for(int k=0;k<n;k++){
    if(k>=average_row*rank && k<average_row*(rank+1)){
            for(int j=k+1;j<n;j++)
                A[k][j] = A[k][j]/A[k][k];
            A[k][k] = 1.0;
    }
    MPI_Ibcast(&A[k][0],n,MPI_FLOAT,n/average_row,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&status);
        for(int i=max(average_row*rank,k+1);i<average_row*(rank+1);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;
        }
    }
    MPI_Finalize();
    return 0;
}
```

2.3.8 流水线算法

```
int main(int argc,char* argv[])
{
    MPI_Init(&argc,&argv);
    int average_row = n/size;
    for(int k=0;k<n;k++){
    if(k>=average_row*rank && k<average_row*(rank+1)){
            for(int j=k+1;j<n;j++)
                A[k][j] = A[k][j]/A[k][k];
            A[k][k] = 1.0;
      if(rank != size-1)
          MPI_Send(&A[k][0],n,MPI_FLOAT,rank+1,rank+1,MPI_COMM_WORLD);
    }
    else {
      if(rank > k/average_row){
        MPI_Recv(&A[k][0],n,MPI_FLOAT,rank-1,rank,MPI_COMM_WORLD,&status);
         }
      if(rank > k/average_row && rank != size-1){
        MPI_Send(&A[k][0],n,MPI_FLOAT,rank+1,rank+1,MPI_COMM_WORLD);
      }
    }
        for(int i=max(average_row*rank,k+1);i<average_row*(rank+1);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;
        }
    }
    MPI_Finalize();
    return 0;
}
```

2.4 实验结果

2.4.1 实验平台运行

在 VS2022 中运行实验程序。

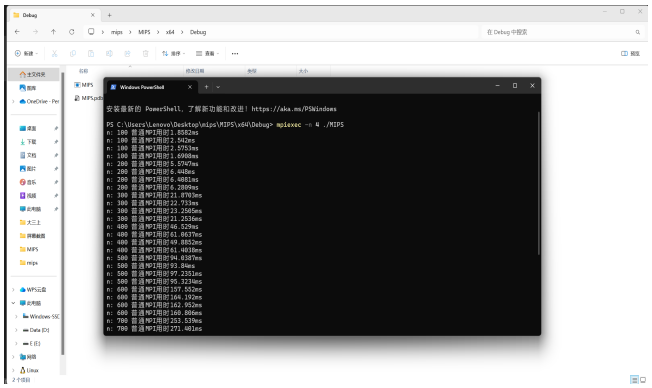

图 2.4: 实验平台运行图

2.4.2 问题规模大小

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0053	0.1542	0.6244	1.2408	152.736	559.112	981.917
块划分 MPI 算法	0.0075	0.1492	0.3181	0.6088	57.603	178.692	351.133

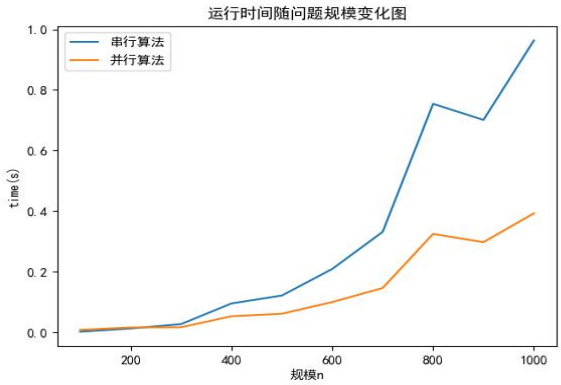

图 2.5: 运行时间随问题规模变化图

2.4.3 设置不同进程数：节点数的影响

设置问题规模为 1000，得到不同进程数下具体运行时间。

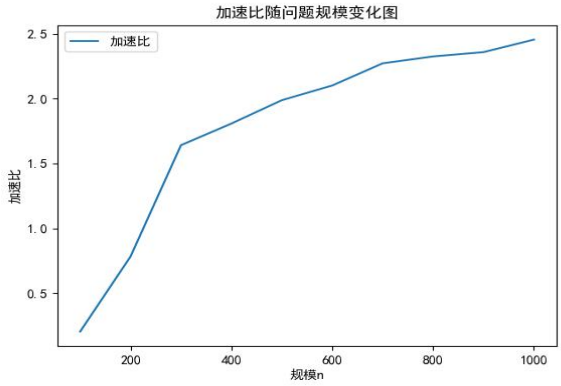

图 2.6: 加速比随问题规模变化图

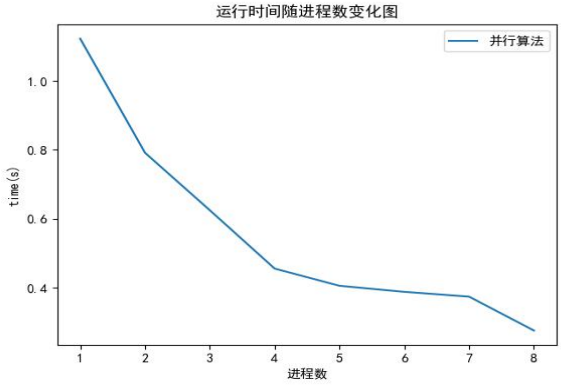

图 2.7: 运行时间随进程数变化图

2.4.4 数据划分：负载均衡问题

2.4.5 任务划分：不同的循环划分方法

2.4.6 探索更佳性能：与 SIMD 结合

2.4.7 多线程与多进程：与 OpenMP 与 Pthreads 结合

2.4.8 通信方式：阻塞通信与非阻塞通信

2.4.9 流水线算法的尝试

2.4.10 实验结果总结

问题规模大小 问题规模较小时，总体并行优化算法相较于传统串行用时相似甚至较长。这是因为进程创建、各种参数变量的加入以及运算步骤的复杂化等因素都会延长算法运算的时间，而因为问题规模较小，其优化的效果没有很好地表现出来。随着问题规模的增大，并行算法的优化效果才开始显现。表现为加速比的逐步上升。

设置不同进程数：节点数的影响 在进程数小于等于 8 时，程序的运行时间的确随着进程数的增加而逐步减少。这是因为电脑 CPU 有足够的节点来并行处理这些进程。而进程数的设置并非越多越好，一旦超出电脑 CPU 能同时处理的范围，就是出现进程的拥挤，反而降低了程序的运行效率。

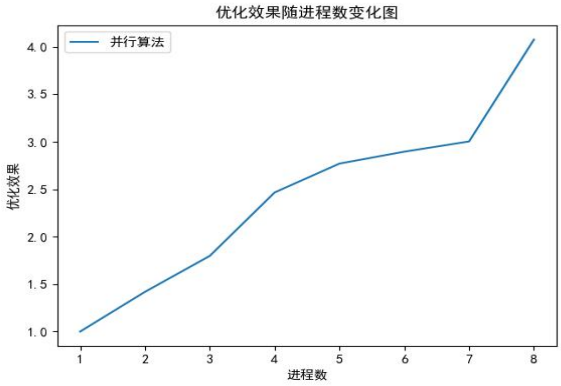

图 2.8: 加速比随进程数变化图

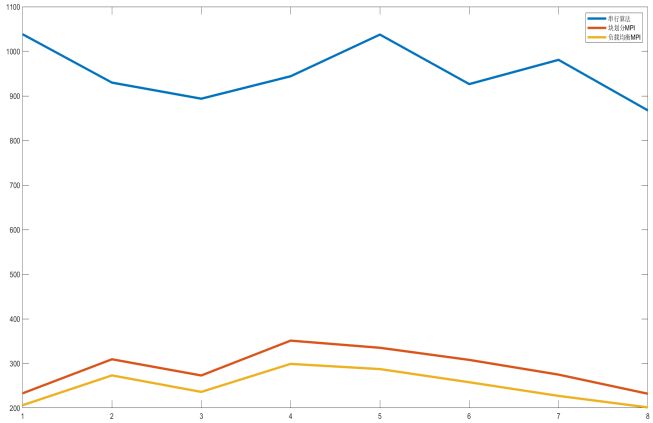

图 2.9: n=1000 时不同数据划分下运行时间图

数据划分：负载均衡问题 将余数部分均匀分配给前 n%m 个进程，每个进程一行，这样的负载更为均衡，使得最后一个进程不需要相较于其他进程额外地处理相关运算，从而能够减少程序运行时间，这样更为合理的数据划分方式拥有更高的加速比。

任务划分：不同的循环划分方法 从实验结果来看，采用循环划分的方式虽然也能明显起到并行优化的效果，但是相较于块划分的方式，呈现出较低的运行效率。循环划分与块划分的不同之处在于循环划分使用了按列分配的方式。在按行分配与按列分配的问题上，我们一直都认为按行分配是更优的选择，而事实也确实如此，两个按列分配方案的运行时间都要慢于按行分配的时间，这其中主要是 cache 高速缓存行内存存储访问的问题，使得在进行数据查找能更快地获取同一行的数据。这一点在之前的实验中有所涉及。

探索更佳性能：与 SIMD 结合 我们在用 MPI 并行化时，没有改动内层循环进行具体数据运算部分，在这里加上 SIMD 编程可以进一步优化性能。实验结果也体现了这一点，加上 SIMD 编程优化的方案在比普通 MPI 算法的运行时间更短。

多线程与多进程：与 OpenMP 与 Pthreads 结合 通常情况下，我们在运行程序时，比如在 Linux 下通过./test 运行一个可执行文件，那么我们就相当于创建了一个进程，操作系统会为这个进程分配 ID 和堆栈空间等资源。如果这个可执行程序是个多线程程序（比如由 OpenMP 或 Pthread 编写的 cpp

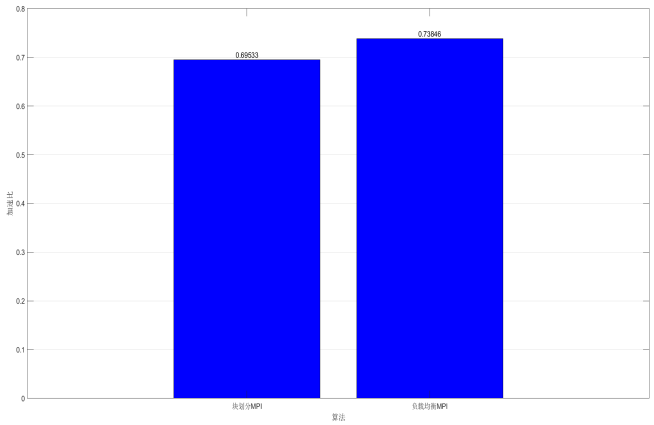

图 2.10: 加速比与数据划分关系图

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0053	0.1542	0.6244	1.2408	152.736	559.112	981.917
块划分 MPI 算法	3.6321	0.1492	0.3181	0.6088	57.603	178.692	351.133
循环划分 MPI 用时	3.7725	0.1894	0.4732	0.8411	97.793	394.41	570.436

文件编译而来），那么这个./test 进程在执行到某个特定位置时会创建多个线程继续执行。从实验结果来看，这样做确实可以进一步地减少程序运行时间，提高了性能。

通信方式：阻塞通信与非阻塞通信 非阻塞式通信中，发送和接收进程发起通信后立即返回，而不会等待通信操作完成。发送进程可以立即执行后续代码，而接收进程在通信完成之前不需要等待可以利用该时间执行其他操作。从实验结果来看，非阻塞式通信确实较少的程序运行时间，但是不明显，因为其需要额外的代码来处理通信完成后的操作，以确保通信操作的正确性，这样做便会相应地延长运行时间。

流水线算法的尝试 流水线算法是并行计算中一个非常有效的、常用的手段，它根据计算的依赖和递推关系制定多任务流水线流程。流水线技术是一种在程序执行时多条指令重叠进行操作的一种准并行处理实现技术，各种部件同时处理是针对不同指令而言的，它们可同时为多条指令的不同部分进行工作，以提高各部件的利用率和指令的平均执行速度。

从实验结果来看，流水线算法对总体运行时间影响不大，但是有效减少了 0 号进程运行时间。

3 链接

github 项目链接:https://github.com/TimeIsAPlace/MPI.git

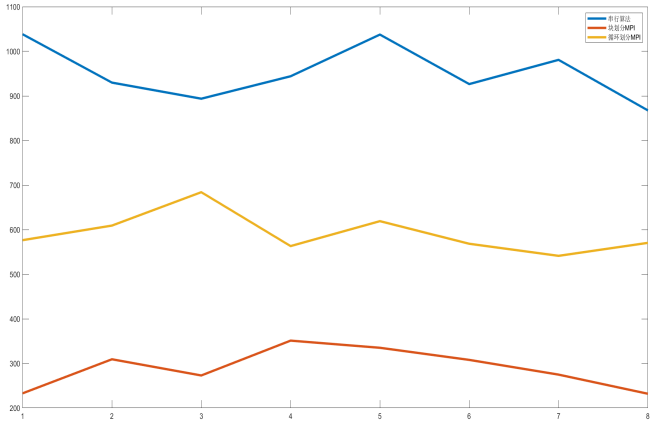

图 2.11: n=1000 时不同划分方式下运行时间图

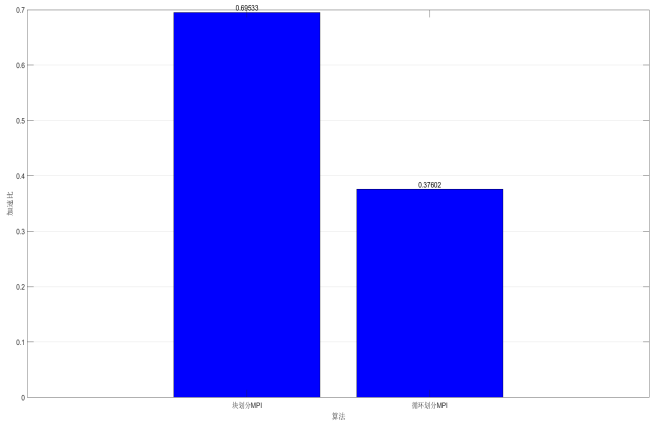

图 2.12: 加速比与划分方式关系图

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0053	0.1542	0.6244	1.2408	152.736	559.112	981.917
块划分 MPI 算法	3.6321	0.1492	0.3181	0.6088	57.603	178.692	351.133
SIMD+MPI 用时	4.8612	0.3016	0.4512	0.5113	41.476	147.361	254.745

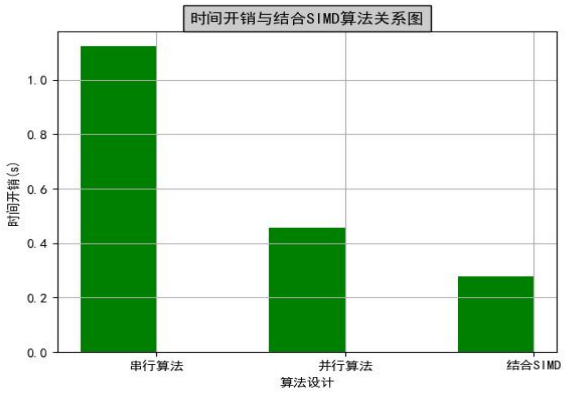

图 2.13: 运行时间与 SIMD 算法关系图

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0053	0.1542	0.6244	1.2408	152.736	559.112	981.917
块划分 MPI 算法	3.6321	0.1492	0.3181	0.6088	57.603	178.692	351.133
OpenMP+MPI 用时	1.5341	0.1385	0.1732	0.4198	31.364	98.341	189.412
Pthreads+MPI 用时	2.1643	0.1396	0.1481	0.3834	40.145	149.245	315.764

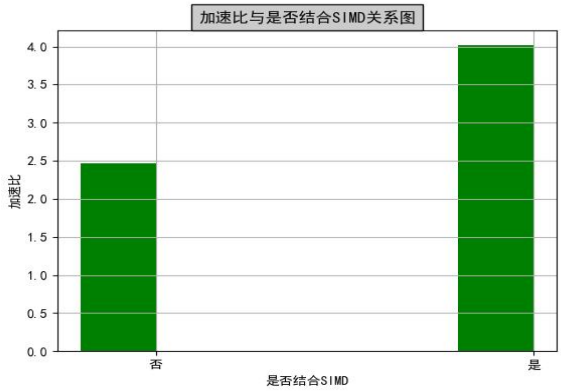

图 2.14: 加速比与 SIMD 算法关系图

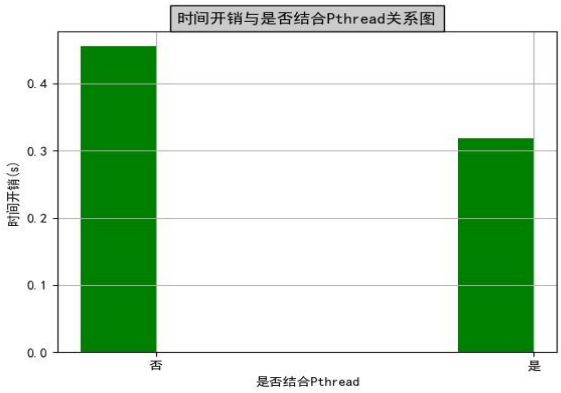

图 2.15: 运行时间与 Pthreads 关系图

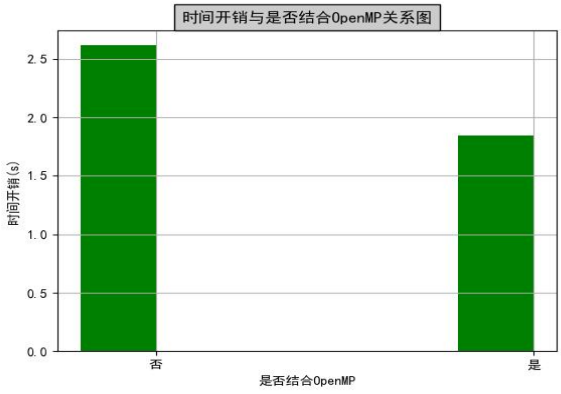

图 2.16: 运行时间与 OpenMP 关系图

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0053	0.1542	0.6244	1.2408	152.736	559.112	981.917
阻塞通信 MPI 算法	3.6321	0.1492	0.3181	0.6088	57.603	178.692	351.133
非阻塞通信 MPI 算法	3.5612	0.1385	0.3018	0.5831	54.113	170.515	342.375

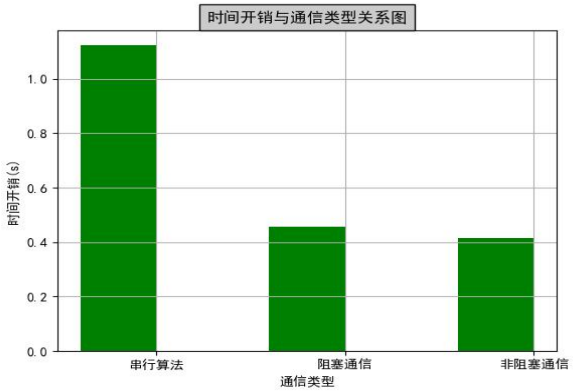

图 2.17: 运行时间与是否阻塞通信关系图

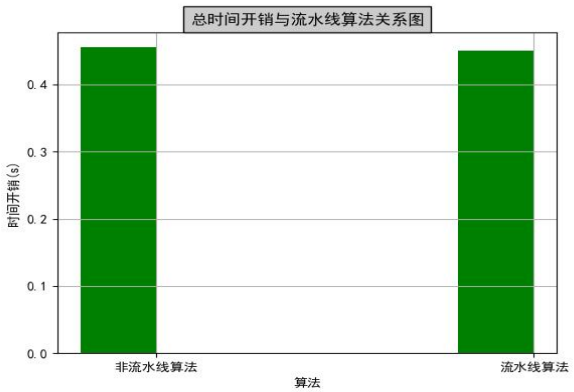

图 2.18: 运行时间与流水线算法关系图

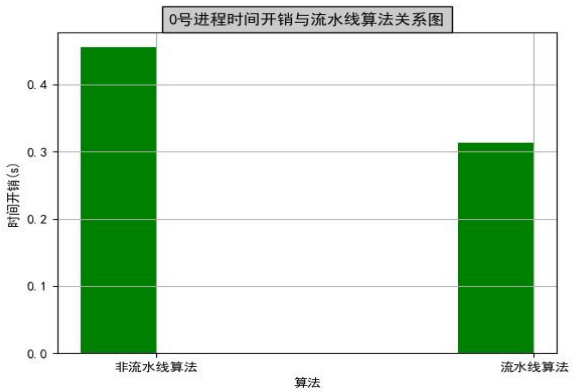

图 2.19: 0 号进程运行时间与流水线算法关系图