南開大學
Nankai University

计算机学院

并行程序设计实验报告

MPI 加速的高斯消去算法

2022 年 7 月

目录

1 实验目标

1. 通过 MPI 实现集群加速的高斯消去算法和消元子模式的高斯消去算法。
2. 编写测试脚本，通过运行计时的方式，测试不同规模下不同线程数量、不同优化策略的加速比。
3. 对实验结果进行分析，找出性能差异的原因。

2 核心代码

2.1 普通高斯消去

本次实验中普通高斯消去的并行化采用了主从模式，每次消元时，主进程将当前行进行广播，之后向每个从进程发送其应处理的任务，而线程处理完任务后将结果发回主进程。具体的主进程代码如下

```
#define N 1024
#define ele_t float
#define ZERO 1e-5

void run_master(ele_t *_mat)
{
    ele_t(*mat)[N] = (ele_t(*)[N])_mat;
    MPI_Request* request=new MPI_Request[world_size];
    for (int i = 0; i < N; i++)
    { // i: 当前行/枢轴位置
        int n_lines = (N - i - 1) / world_size + 1; // 进程处理任务数量
        n_lines = n_lines == 1 ? 0 : n_lines; // 如果只有一行，则不需要发送数据

        if (n_lines)
        {
            MPI_Bcast(mat[i], sizeof(ele_t) * N, \
                MPI_BYTE, 0, MPI_COMM_WORLD); // 发送当前消元行
            for (int th = 1; th < world_size; th++)
            { // 向各进程发送被消元行
                MPI_Send(mat[i + 1 + (th - 1) * n_lines], \
                    sizeof(ele_t) * N * n_lines, MPI_BYTE, \
                    th, 0, MPI_COMM_WORLD);
                // MPI_Isend(mat[i + 1 + (th - 1) * n_lines], \
                //     sizeof(ele_t) * N * n_lines, MPI_BYTE, \
                //     th, 0, MPI_COMM_WORLD, &request[th]);
                // 反向优化的 Isend?
            }
        }
```

```
29
30  // #pragma omp parallel for num_threads(2)
31          for (int j = i + 1 + (world_size - 1) * n_lines; j < N; j++)
32          { // 主进程计算剩下的最后几行
33              if (abs(mat[i][i]) < ZERO)
34                  continue;
35              ele_t div = mat[j][i] / mat[i][i];
36  // #pragma omp simd
37              for (int k = i; k < N; k++)
38                  mat[j][k] -= mat[i][k] * div;
39          }
40
41          if (n_lines)
42          {
43              for (int th = 1; th < world_size; th++)
44              { //从从进程回收数据
45                  MPI_Recv(mat[i + 1 + (th - 1) * n_lines], \
46                      sizeof(ele_t) * N * n_lines, MPI_BYTE, \
47                      th, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
48              }
49          }
50      }
51  }
```

这里采取的任务划分方式为静态均分，除主进程外，每个进程分配到的行数为 $\frac{\text{总行数}}{world\_size}+1$，而主进程则分配到余下的行数，这种分配方式下主进程也能分配到一定数量的任务，能够防止其出现空载的情况。

而从进程具体代码如下：

```
1   void run_slave()
2   {
3       // int i;       // 枢轴位置
4       int n_lines = (N - 1) / world_size + 1;       // 需要消元的行数
5       ele_t lines_i[N]; // 当前消元行
6       ele_t(*mat)[N] = (ele_t(*)[N])new ele_t[n_lines * N]; // 被消元矩阵
7
8       for (int i = 0; i < N; i++)
9       { // i 是确定的，和主进程同步
10          n_lines = (N - i - 1) / world_size + 1;
11          if (n_lines == 1)
12              break;
13          MPI_Bcast(lines_i, sizeof(ele_t) * N, MPI_BYTE, 0, MPI_COMM_WORLD);
```

```
14          MPI_Recv(mat, n_lines * N * sizeof(ele_t), MPI_BYTE, \
15              0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
16  // #pragma omp parallel for num_threads(2)
17          for (int j = 0; j < n_lines; j++)
18          {
19              if (abs(lines_i[i]) < ZERO) // 枢轴为 0, 不需要消元
20                  continue;
21              ele_t div = mat[j][i] / lines_i[i];
22  // #pragma omp simd
23              for (int k = i; k < N; k++)
24                  mat[j][k] -= lines_i[k] * div;
25          }
26          MPI_Send(mat, n_lines * N * sizeof(ele_t), MPI_BYTE, \
27              0, 1, MPI_COMM_WORLD);
28      }
29      delete[] mat;
30  }
```

2.2 消元子模式的高斯消去

消元子模式下高斯消去算法的并行策略和普通高斯消去算法的思路相似，也采取了同样的静态任务划分方式，但是由于特殊高斯消去算法需要通过遍历整个消元子矩阵来确定当前行是进行升格操作还是消元操作，从进程无法获知确定的校园次数，因此每次消元主进程还需要发送当前消元子的信息和枢轴位置给被消元行。

消元子模式的高斯消去并行算法的主进程代码如下：

```
1   void run_master(mat_t (*ele)[COL / mat_L + 1], mat_t (*row)[COL / mat_L + 1])
2   {
3       bool upgraded[ROW] = {0};
4
5       for (int j = COL - 1; j >= 0; j--)
6       { // 遍历消元子
7           if (!(ele[j][j / mat_L] & ((mat_t)1 << (j % mat_L))))
8           { // 如果不存在对应消元子则进行升格
9               for (int i = 0; i < ROW; i++)
10              { // 遍历被消元行
11                  if (upgraded[i])
12                      continue;
13                  if (row[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
14                  {
15                      memcpy(ele[j], row[i], (COL / mat_L + 1) * sizeof(mat_t));
16                      upgraded[i] = true;
```

```
                    break;
                }
            }
        }

        int n_workload = ROW / world_size + 1;
        // 发送当前消元子、被消元行升级情况
        MPI_Bcast(&j, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(ele + j, (COL / mat_L + 1) * sizeof(mat_t), MPI_BYTE, 0, MPI_COMM_WORLD);
        MPI_Bcast(upgraded, ROW * sizeof(bool), MPI_BYTE, 0, MPI_COMM_WORLD);

        for (int th = 1; th < world_size; th++)
        { // 发送被消元行
            int i_offset = n_workload * (th - 1);
            MPI_Send(row + i_offset, n_workload * (COL / mat_L + 1) * sizeof(mat_t), \
                MPI_BYTE, th, 0, MPI_COMM_WORLD);
        }

#pragma omp parallel for num_threads(2)
        for (int i = n_workload * (world_size - 1); i < ROW; i++)
        { // 遍历被消元行
            if (upgraded[i])
                continue;
            if (row[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
            { // 如果当前行需要消元
#pragma omp simd
                for (int p = 0; p <= COL / mat_L; p++)
                    row[i][p] ^= ele[j][p];
            }
        }

        for (int th = 1; th < world_size; th++)
        { // 回收
            int i_offset = n_workload * (th - 1);
            MPI_Recv(row + i_offset, n_workload * (COL / mat_L + 1) * sizeof(mat_t), \
                MPI_BYTE, th, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    int j = -1; //结束从进程
    MPI_Bcast(&j, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

对应的从进程代码如下：

```
void run_slave()
{
    int j;
    int n_workload = ROW / world_size + 1;
    int i_offset = n_workload * (world_rank - 1);
    mat_t ele_j[COL / mat_L + 1];
    mat_t row[n_workload][COL / mat_L + 1];
    bool upgraded[ROW] = {0};

    while (true)
    {
        MPI_Bcast(&j, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (j == -1)
            break;
        MPI_Bcast(ele_j, (COL / mat_L + 1) * sizeof(mat_t), \
            MPI_BYTE, 0, MPI_COMM_WORLD);
        MPI_Bcast(upgraded, ROW * sizeof(bool), MPI_BYTE, \
            0, MPI_COMM_WORLD);
        MPI_Recv(row, n_workload * (COL / mat_L + 1) * sizeof(mat_t), \
            MPI_BYTE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

#pragma omp parallel for num_threads(2)
        for (int i = i_offset; i < i_offset + n_workload; i++)
        {
            if (upgraded[i])
                continue;
            if (row[i - i_offset][j / mat_L] & ((mat_t)1 << (j % mat_L)))
            { // 如果当前行需要消元
#pragma omp simd
                for (int p = 0; p <= COL / mat_L; p++)
                    row[i - i_offset][p] ^= ele_j[p];
            }
        }

        MPI_Send(row, n_workload * (COL / mat_L + 1) * sizeof(mat_t), \
            MPI_BYTE, 0, 1, MPI_COMM_WORLD);
    }
}
```

3 实验环境

本次实验在笔记本电脑上通过 WSL 进行，WSL 系统为 Debian11，内核版本 5.10.16.3-microsoft-standard-WSL2，GCC 编译器版本 10.2.1 20210110，MPI 实现为 MPICH，版本 3.4.1。

硬件方面，电脑的 CPU 为英特尔 12 代 i7-12700H，为 6 大核 +8 小核的混合架构，L1d cache 480 KiB，L1i cache 320 KiB，L2 cache 12.5 MiB，L3 cache 24 MiB。

4 实验过程

首先需要生成测试数据，之后编写测试用的 Shell 脚本，通过编译时宏定义改变数据规模，并通过 mpirun 命令的-n 参数指定不同的线程数，将不同情况的接轨导出为 csv 格式的数据，绘制折线图，本次的编译均使用了-O0 优化选项。

Shell 脚本中普通高斯消去的编译与运行命令如下：

```
mpic++ -O0 -march=native -fopenmp -DN=$((128 * i)) ./gauss.cpp -o ./gauss_test
mpirun -n 1 ./gauss_test >>./gauss_timing_$timestr.csv
mpirun -n 2 ./gauss_test >>./gauss_timing_$timestr.csv
mpirun -n 4 ./gauss_test >>./gauss_timing_$timestr.csv
mpirun -n 8 ./gauss_test >>./gauss_timing_$timestr.csv
```

Shell 脚本中特殊高斯消去的编译与运行命令如下：

```
mpic++ -march=native -w -pthread -DDATA=\"${data_path}${file}/\" \
    -DCOL=${attr[1]} -DELE=${attr[2]} -DROW=${attr[3]} \
    -O0 -fopenmp -DSEPR=\",\" -DNUM_THREADS=1 \
    ./groebner.cpp -o ./groebner
mpirun -n 1 ./groebner >>groebner_$timestr.csv
mpirun -n 2 ./groebner >>groebner_$timestr.csv
mpirun -n 4 ./groebner >>groebner_$timestr.csv
mpirun -n 8 ./groebner >>groebner_$timestr.csv
```

由于 MPI 实验是在本地完成的，因此实验时注释了 OpenMP 相关的并行化代码，以最大程度体现 MPI 并行化加速的效果。

5 实验结果

5.1 普通高斯消去

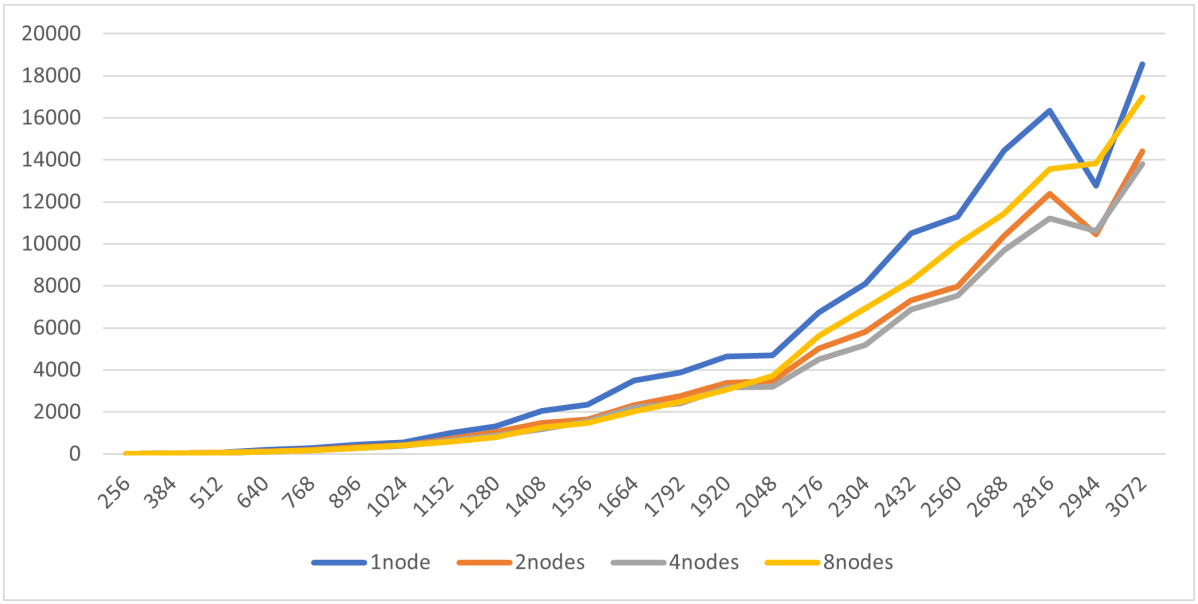

图 5.1: 高斯消去算法运行时间（单位：ms）-输入矩阵行数

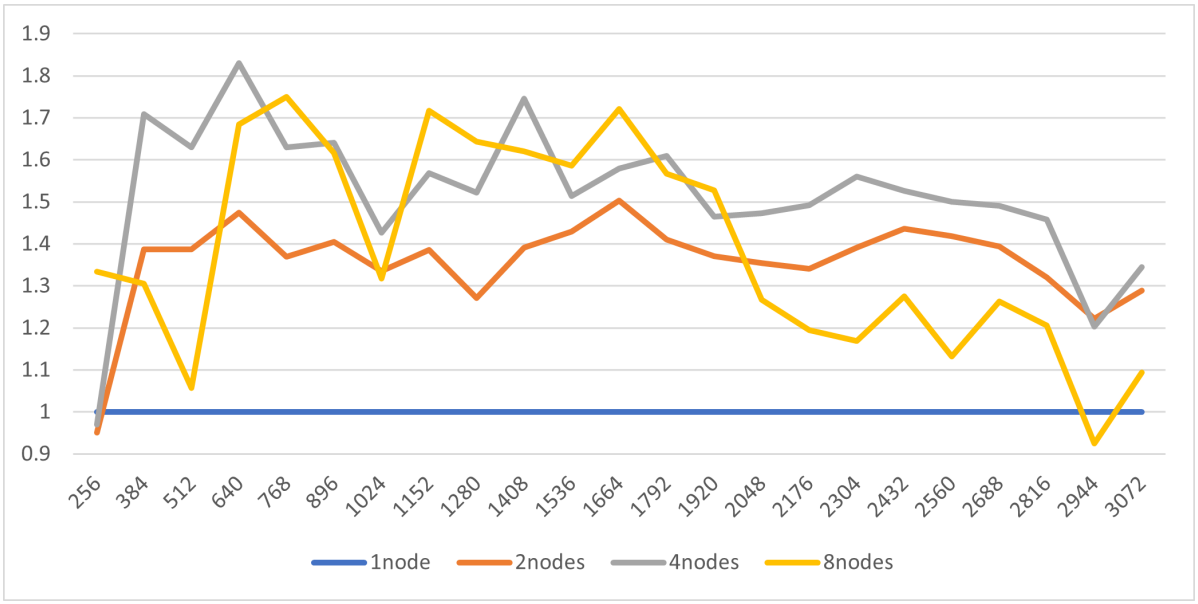

图 5.2: 高斯消去算法加速比-输入矩阵行数

5.2 消元子模式高斯消去

数据集 \ 进程数	1	2	4	8
2_254_106_53	0.11993	0.369463	1.06552	1.64307
3_562_170_53	0.192358	0.923936	1.19662	2.08669
4_1011_539_263	3.04667	6.46555	6.71845	9.36801
5_2362_1226_453	31.0493	45.6444	56.998	55.6536
6_3799_2759_1953	368.317	384.878	487.475	544.918
7_8399_6375_4535	5095.38	5046.97	5169.95	5678.16

表 1: 特殊高斯消去算法运行时间（单位：ms）

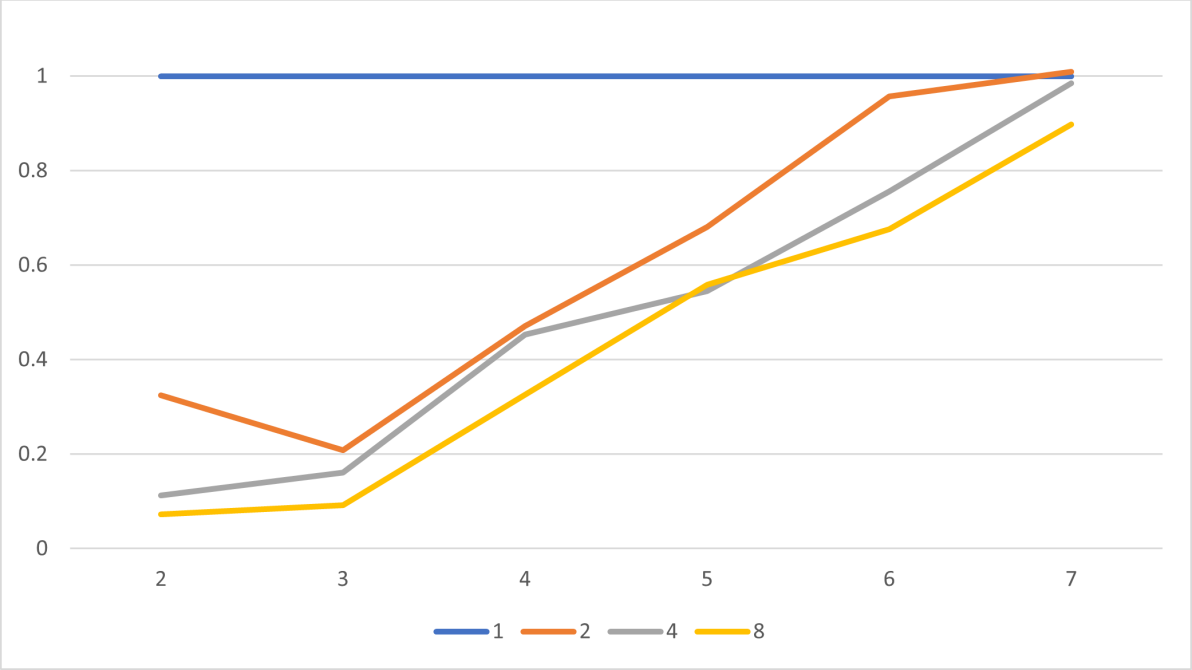

图 5.3: 特殊高斯消去算法加速比-输入矩阵行数-节点数

8 号数据集下，2 进程并行的加速比只有 0.3 左右，甚至还不如 7 号数据集，笔者也发现了代码中通讯部分的不合理之处，因此也就没有再进行测试。

6 结果分析

6.1 进程间通讯消耗

不难发现，如图5.2所示，在本次 MPI 并行化实验中，普通高斯消去算法的并行加速比，相比此前多次实验中的加速比要低很多。尤其是图5.3所示的特殊高斯消去算法并行化，并行算法的运行时间几乎全部长于串行算法的运行时间。

笔者猜测，导致并行算法性能低下的主要原因在于通讯的开销。本次的实验代码是在 Pthread 实验的代码基础上修改而来的，但是 MPI 的并行模型属于非统一内存并行，因此进行计算时需要将每个进程需要的数据由主进程进行发送，而 Pthread 实验的代码并没有考虑这一消耗。因此在普通高斯消去和特殊高斯消去的 MPI 并行化实现中，大量不必要的信息被重复发送了多次。

这一点也可以通过复杂度分析验证，例如，普通高斯消去算法的复杂度为 $O(n^3)$，而代码中进行通讯的代码看似位于第二层循环上，复杂度应为 $O(n^2)$。因此，随着问题规模的增长，通讯的代价相比并行的收益应该会变得很低。

然而，通讯的数据数量和输入矩阵的行数是相关的。那么，主进程向子进程发送的矩阵行数之和和输入矩阵的行数应呈线性关系，而非常数关系。因此，在并行化实现中，通讯的复杂度应该为 $O(num\_threads * n^3)$。这和整个算法的复杂度是一致的，但是当线程数增长时通讯的时间复杂度也会在 $O(n^3)$ 的基础上线性增长。所以，增加线程所带来的通讯消耗，是输入数据规模增大也无法弥补的。

为了验证如上的想法，笔者将代码中进行实际运算的部分进行了注释，只留下进行通讯的部分，并进行时间的测量。之后将测量的结果看作为算法运行中的通讯时间，联合之前测得的完整运行时间，计算出比值。

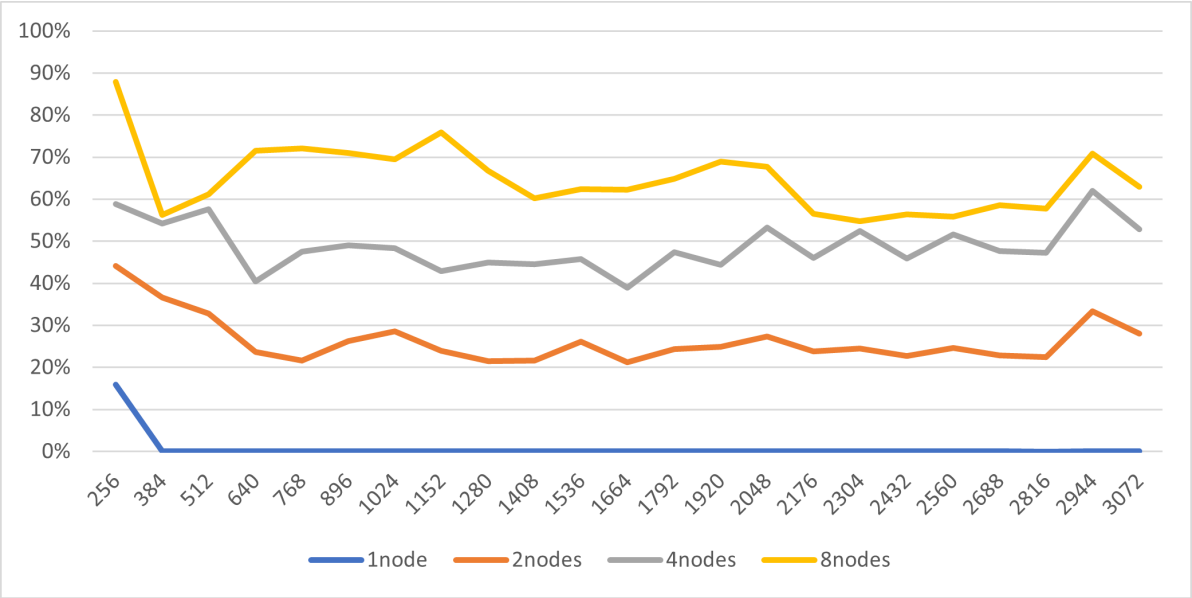

图 6.4: 普通高斯消去算法并行化中通讯时间占比-输入矩阵行数

如图6.4所示，并行化 MPI 算法运行时间中很大一部分被花在了通讯上。2 线程的通讯时间平均约为 26%，4 线程的通讯时间占比平均约为 48%，并且这一占比并没有呈现出随着输入数据规模而变化的趋势，所以这一数据印证了上文中的复杂度分析的结果。

要消除通讯的影响，实现更高效率的并行化，需要考虑修改 Pthread 实验中迁移而来的主从模式的并行化思维。以特殊高斯消去为例，每个线程所分配到的被消元行是固定的，因此完全没有必要将消元行在主从线程间进行多次发送。

关于该问题的优化，笔者计划将其作为期末结课报告的一部分，因此就不再赘述了。

6.2 异步发送 Isend 的优化

在编写主进程的代码时，笔者尝试通过 MPI_Isend 函数来对矩阵进行异步发送，让主进程在等待接收的同时可以进行运算。然而这一优化却完全无法带来加速比上的收益。

笔者思考后发现异步通讯的优化关键在于收发等待的时间，而非收发的时间，而在主进程发送数据时，从进程早已处于等待状态，而通讯和运算的并行化并不能被自动完成，因此异步发送也就不会带来任何性能收益。

7 总结

在本次实验中，我们通过 MPI 实现了普通高斯消去算法和消元子模式高斯消去算法的并行化，并对并行算法的性能进行了测量和分析。通过这次试验，我们对 MPI、通讯复杂度、非同一内存结构并行程序设计有了更深刻的了解。