南开大学 Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计 MPI 实验报告

MPI 实现并行高斯消去法

蒋薇

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 6 月 4 日

摘要

运用 mpi 相关知识在 arm 平台实现高斯消去并行化，与在本机 x86 平台运行对比，在 mpi 一维划分与循环划分对比，mpi 结合 omp 与 simd 编程，对比分析。

关键字：Parallel，高斯消去，MPI

目录

一、 普通高斯消去基础 MPI 并行化实验

(一) 高斯消去串行实现

Algorithm 1 高斯消去实现伪代码

procedure LU(A)
  $begin$
  $fork := 1 tondo$
  $forj := k + 1 tondo$
  $A[k, j] := A[k, j]/A[k, k];$
  $endfor;$
  $A[k, k] := 1.0$
  $fori := k + 1 tondo$
  $forj := k + 1 tondo$
  $A[i, j] := A[i, j] - A[i, k] * A[k, j];$
  $endfor;$
  $A[i, k] := 0;$
  $endfor;$
  $endfor;$
  $endLU$
end procedure

主要代码

高斯消去串行算法

```
void Gauss_calculation(void)//Gauss消去法解线性方程组
{
        double get_A = 0.0;
        for (int i = 1; i < RANK; i++)
        {
                for (int j = i; j < RANK; j++)
                {
                        get_A = A[j][i - 1] / A[i - 1][i - 1];
                        b[j] = b[j] - get_A * b[i - 1];
                        for (int k = i - 1; k < RANK; k++)
                        {
                                A[j][k] = A[j][k] - get_A * A[i - 1][k];
                        }
                }
        }
```

实验截图如图1所示

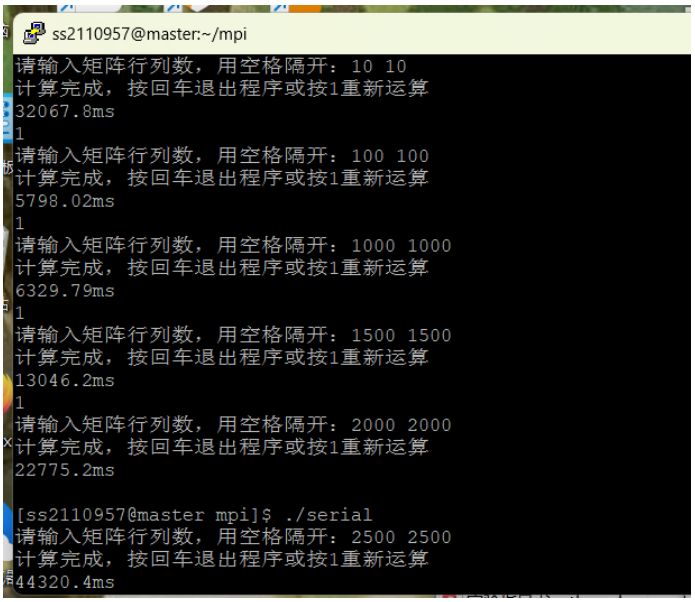


图 1: 串行实验截图

实验结果表

矩阵规模 n*n	运行时间 ms
10*10	32067.8
100* 100	5798.02
1000 * 1000	6329.79
1500 * 1500	13046.2
2000 * 2000	22775.2
2500 * 2500	44320.4

表 1: 串行性能测试结果 (单位:ms)

(二) mpi 行主元高斯消去

描述：矩阵进行行交叉分解，即第 0 行分给 0 号进程，第 1 行分给 1 号进程,....., 第 p-1 行分给 p-1 号进程，第 p 行分给 0 号进程，依此循环分配；求得第 k 行的主元（枢纽），并将该行所有元素及枢纽位置发送给每个进程，每个进程按照该行元素及枢纽位置将各进程中枢纽列的 k + 1 行至 dim - 1 行全部消为 0；其中 k 从 0，1，2，....，到 dim - 2。

mpi 高斯消去并行算法

```
1    void distributeTask(const Matrix& lAb, Matrix& Ab)
2    {
3        int counts = dim / numprocs;
4        for (int i = 0; i < counts; ++i)
5            MPI_Scatter(&lAb(i * numprocs, 0), dim + 1, MPI_DOUBLE, &Ab(i
                 , 0), dim + 1, MPI_DOUBLE, masterNode, MPI_COMM_WORLD);
6        if (counts != partRow)
```

```
                MPI_Scatter(&lAb(counts * numprocs, 0), dim + 1, MPI_DOUBLE,
                    &Ab(partRow - 1, 0), dim + 1, MPI_DOUBLE, masterNode,
                    partComm);
}
//创建新的通信域，方便不能整除时的播撒数据
        MPI_Group worldGroup;
        MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);
        MPI_Group partGroup;

        int* groupRank = new int[remainder];
        for (int i = 0; i < remainder; ++i)
                groupRank[i] = i;
        MPI_Group_incl(worldGroup, remainder, groupRank, &partGroup);
        MPI_Comm_create(MPI_COMM_WORLD, partGroup, &partComm);
        delete[] groupRank;
```

(三) mpi 通信实现方法：

1. 方法一：MPI 群集通信函数

进行第 k 次迭代时，将所得的枢纽及枢纽所在行的元素利用 MPI 群集通信函数广播至每个进程

mpi 高斯消去并行算法群集通信

```
    //群集通信函数实现并行
    if (myid == masterNode)
        rec1 = MPI_Wtime();
        MPI_Bcast(&sta(0, 0), dim + 1,MPI_DOUBLE, source, MPI_COMM_WORLD);
        MPI_Bcast(&picked, 1, MPI_DOUBLE, source, MPI_COMM_WORLD);
        if (myid == masterNode){
                        rec2 = MPI_Wtime();
                        dur2 += rec2 - rec1;}
        int tmp = loc[k];
        loc[k] = loc[picked];
        loc[picked] = tmp;
        if (myid == masterNode)
                s1 = MPI_Wtime();
                for (int i = p; i < partRow; ++i){
                        double t = Ab(i, loc[k]) / sta(0, loc[k]);
                for (int j = k; j < dim + 1; ++j)
                                Ab(i, loc[j]) -= t * sta(0, loc[j]);}
                if (myid == masterNode){
                        s2 = MPI_Wtime();
                        dur += s2 - s1;}
            }
```

2. 方法二：非阻塞通信

采用非阻塞通信，各进程组成环，0 号进程将消息发送给 1 号，1 号发送给 2 号，.....，p - 1 号发送给 0 号；这一过程中使用 MPI_Isend 非阻塞发送，可实现异步计算

mpi 高斯消去并行算法群集通信

```
//非阻塞通信实现异步计算
if (abs(Ab(p, loc[j])) > max){
    max = abs(Ab(p, loc[j]));picked = j;}
MPI_Isend(&Ab(p, 0), dim + 1, MPI_DOUBLE, down, picked, MPI_COMM_WORLD, &
    request);
for (int j = 0; j < dim + 1; ++j){
    sta(0, j) = Ab(p, j);p++;}
 else{
 MPI_Irecv(&sta(0, 0), dim + 1, MPI_DOUBLE, up, MPI_ANY_TAG,
     MPI_COMM_WORLD, &request);
 if (myid == masterNode)
 rec1 = MPI_Wtime();
 MPI_Wait(&request, &status);
 if (myid == masterNode){
     rec2 = MPI_Wtime();
     dur2 += rec2 - rec1;}
     picked = status.MPI_TAG;
     if (down != source)
     MPI_Isend(&sta(0, 0), dim + 1, MPI_DOUBLE, down, picked,
         MPI_COMM_WORLD, &request);
```

注意：int myid, numprocs, masterNode;//进程标识号与进程总数，设为全局变量可在任意个函数中访问

int partRow, up, down;//各进程分配到的行数，相邻的上下两个进程

int dim = 1500;//问题规模

$MPI_CommpartComm$;//不能整除的部分在该通信域做一次 scatter/gather 即可

(四) 结果展示

MPI 相关实验完整代码： 请点击这里

1. 群集通信

使用群集通信的结果：符合预期，计算时间下降，通信略有增加

实验截图如图2所示

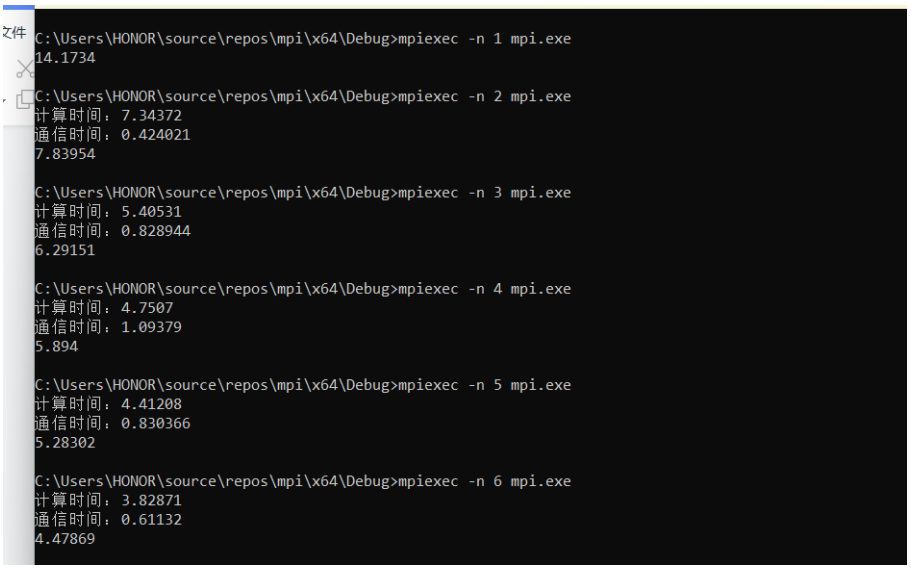


图 2: arm 群集实验

进程数	计算时间	通信时间	seqGaussSolver() 时间
1	0	0	2.41192
2	1.12359	1.13868	2.31154
3	0.689188	1.68915	2.45711
4	0.629044	1.32789	2.02197
5	0.286778	0.731163	1.09932
6	0.241671	0.622755	0.943172

表 2: arm 平台问题规模 1500*1500 集群通信

实验截图如图3所示

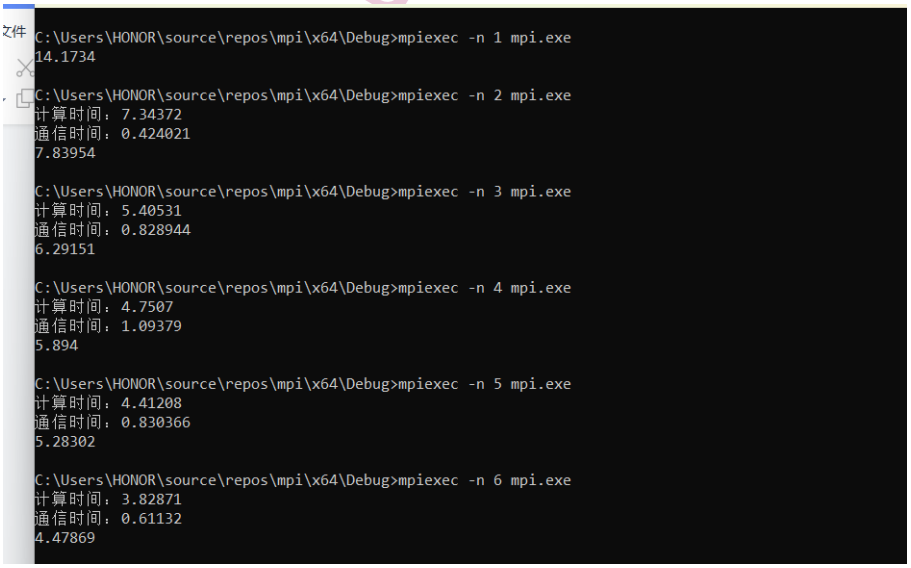


图 3: x86 群集实验

进程数	计算时间	通信时间	seqGaussSolver() 时间
1	0	0	14.1734
2	7.34372	0.424021	7.83954
3	5.40531	0.828944	6.29151
4	4.7507	1.09379	5.894
5	4.41208	0.830366	5.28302
6	3.82871	0.61132	4.47869

表 3: x86 平台问题规模 1500*1500 集群通信

2. 使用非阻塞通信

实验截图如图4所示

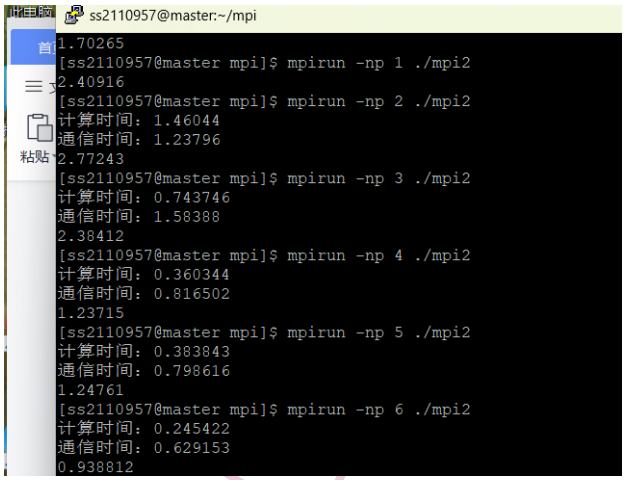


图 4: arm 非阻塞通信实验

进程数	计算时间	通信时间	seqGaussSolver() 时间
1	0	0	2.40916
2	1.46044	1.23796	2.77243
3	0.743746	1.58388	2.38412
4	0.360344	0.816502	1.23715
5	0.383843	0.798616	1.24761
6	0.245422	0.629153	0.938812

表 4: arm 平台问题规模 1500*1500 非阻塞通信

实验截图如图5所示

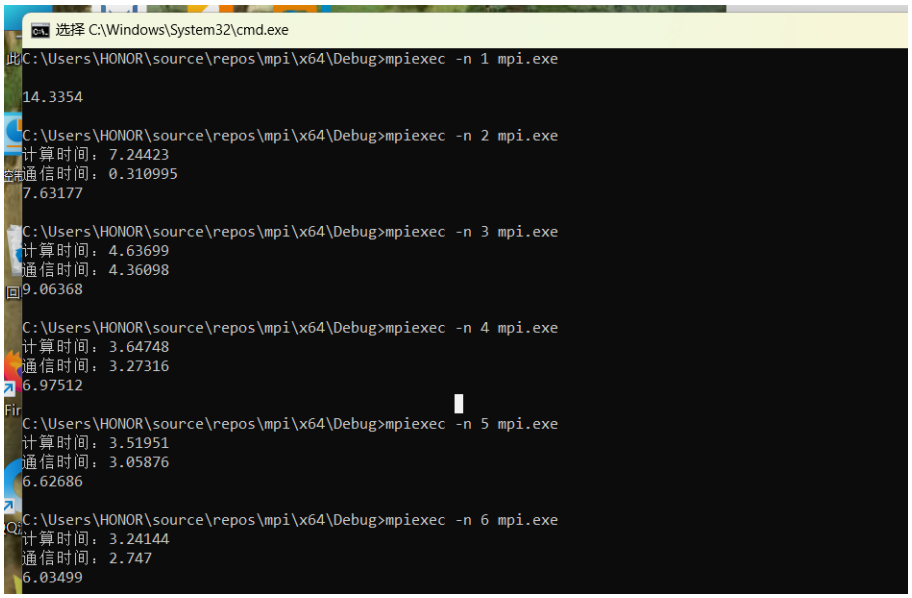


图 5: x86 非阻塞实验

进程数	计算时间	通信时间	seqGaussSolver() 时间
1	0	0	14.3354
2	7.24423	0.310995	7.63177
3	4.63699	4.36098	9.06368
4	3.64748	3.27316	6.97512
5	3.51951	3.05876	6.62686
6	3.24144	2.747	6.03499

表 5: x86 平台问题规模 1500*1500 非阻塞通信

仅在进程调用为 2 的时候符合预期，其余并行部分通信与计算基本 1 比 1, 将问题规模增至 4000，情况依旧。

（五） 一维划分与循环划分

1. 一维划分

一维（行）块划分：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 $[i*(n-n$

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。

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。

注意：要想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。

2. 循环划分

mpi 高斯消去并行算法循环划分

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
// 只有是0号进程, 才进行初始化工作
if (rank == 0) {
    init_matrix();}
start_time = MPI_Wtime();
int task_num = rank < N% size ? N / size + 1 : N / size;
// 0号进程负责任务的初始分发工作
auto* buff = new float[task_num * N];
if (rank == 0) {}
        int count = p < N% size ? N / size + 1 : N / size;
        MPI_Send(buff, count * N, MPI_FLOAT, p, 0, MPI_COMM_WORLD);}}
// 非0号进程负责任务的接收工作
else {
    MPI_Recv(&matrix[rank][0], task_num * N, MPI_FLOAT, 0, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);}
// 做消元运算
for (int k = 0; k < N; k++) {
    // 如果除法操作是本进程负责的任务, 并将除法结果广播
    if (k % size == rank) {
        for (int j = k + 1; j < N; j++) {
            matrix[k][j] /= matrix[k][k];}
        matrix[k][k] = 1;
        for (int p = 0; p < size; p++) {
            if (p != rank) {
                MPI_Send(&matrix[k][0], N, MPI_FLOAT, p, 1,
                    MPI_COMM_WORLD); }}}
    // 其余进程接收除法行的结果
    else {
        MPI_Recv(&matrix[k][0], N, MPI_FLOAT, k % size, 1, MPI_COMM_WORLD
            , MPI_STATUS_IGNORE);}
end_time = MPI_Wtime();
return (end_time - start_time) * 1000;}
```

时间 (规模 n 进程数)	1	2	3	4	5
100	0.347376	2.90585	5.39255	7.38597	9.41873
500	47.5321	39.7689	41.5428	43.7205	49.9389
1000	382.366	282.635	278.956	226.673	206.184
1500	1253.78	784.972	936.425	998.859	908.286
2000	2968.3	1753.53	1842.22	1753.51	1663.67

表 6: arm 平台循环划分

3. 流水线划分

流水线划分：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。

mpi 高斯消去并行算法流水线划分

```
MPI_Scatterv(matrix, thread_count, s_count, MPI_FLOAT, space, thread_count[
    m_rank], MPI_FLOAT, 0, MPI_COMM_WORLD);
    for (k = 0; k < s_count[m_rank] / ROW; k++){
        MPI_Recv(local, ROW, MPI_FLOAT, m_rank - 1, k, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        if (m_rank != m_size - 1){
            MPI_Send(local, ROW, MPI_FLOAT, m_rank + 1, k, MPI_COMM_WORLD);}
        for (i = 0; i < thread_count[m_rank] / ROW; i++){
            for (j = k + 1; j < ROW; j++){
                space[i][j] -= space[i][k] * local[j];}
            space[i][k] = 0;}}
```

(六) mpi 结合多线程与 simd 编程

1. 描述

对于一维行划分，可看作是将第二层循环拆分，分配给不同进程；这样，继续进行多线程并行，即可继续对第二层循环进行划分，即，将进程负责的行划分给内部的多个线程，也可以对最内层循环进行划分，即，将进程负责的所有行的不同列分配给不同线程；再继续结合 SIMD 并行化，则只能对最内层循环进行向量化。

2. 主要代码

mpi 高斯消去并行算法结合多线程与 simd

```
mpi_omp_SIMD(){
#pragma omp parallel num_threads(NUM_THREADS), shared(matrix), private(i,j,k,
    diver,divee,mult1,mult2,sub1,m_size,m_rank)
    for (k = 0; k < ROW; ++k) {
        if (k >= r1 && k <= r2){
            diver = vld1q_dup_f32(&matrix[k][k]);
#pragma omp single
    for (j = k + 1; j < ROW && ((ROW - j) & 3); ++j){
            matrix[k][j] = matrix[k][j] / matrix[k][k];}
    for (; j < ROW; j += 4){
                divee = vld1q_f32(&matrix[k][j]);
                divee = vdivq_f32(divee, diver);
                vst1q_f32(&matrix[k][j], divee);}
#pragma omp barrier
```

```
    matrix[k][k] = 1.0;
    for (j = 0; j < m_size; ++j){
         MPI_Send(&matrix[k][0], ROW, MPI_FLOAT, j, 1, MPI_COMM_WORLD);}}
        else{
            MPI_Recv(&matrix[k][0], ROW, MPI_FLOAT, j, 1, MPI_COMM_WORLD, &
                status);}
#pragma omp for schedule(dynamic)
    for (i = max(r1, k + 1); i < r2; ++i){
        mult1 = vld1q_dup_f32(&matrix[i][k]);
        for (j = k + 1; j < ROW && ((ROW - j) & 3); ++j){
            matrix[i][j] = matrix[i][j] - matrix[i][k] * matrix[k][j];}
            for (; j < ROW; j += 4){
                sub1 = vld1q_f32(&matrix[i][j]);
                mult2 = vld1q_f32(&matrix[k][j]);
                mult2 = vmulq_f32(mult1, mult2);
                sub1 = vsubq_f32(sub1, mult2);
                vst1q_f32(&matrix[i][j], sub1);}
            matrix[i][k] = 0.0;}
#pragma omp barrier}
    MPI_Barrier(MPI_COMM_WORLD);
    if (m_rank == 0){
        end_time = MPI_Wtime();
        cout << "mpi_omp_SIMD:" << (end_time - start_time) * 1000 << "ms" <<
            endl;}
MPI_Final
```

3. 结果展示

实验截图如图6所示

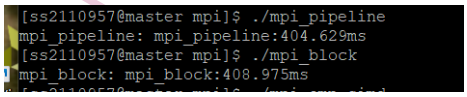


图 6: 不同方法测试时间

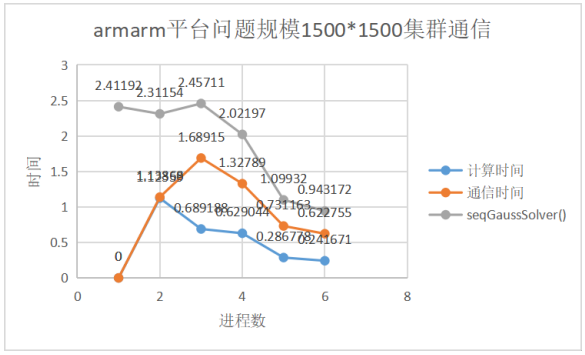
方法	plain()	mpi()	mpi$_o mp_s imd()$	mpi$_b lock()$	mpi$_p ipeline()$
时间 (ms)	389.148	458.92	407.58	498.975	404.429

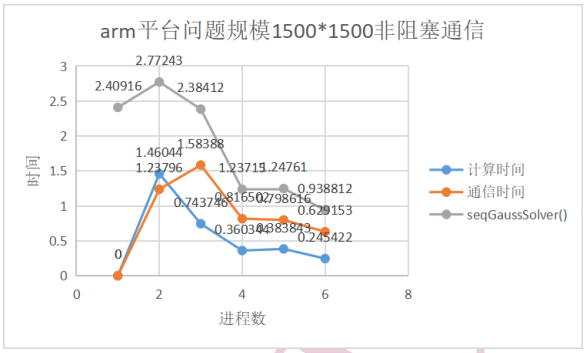
表 7: 不同方法测试

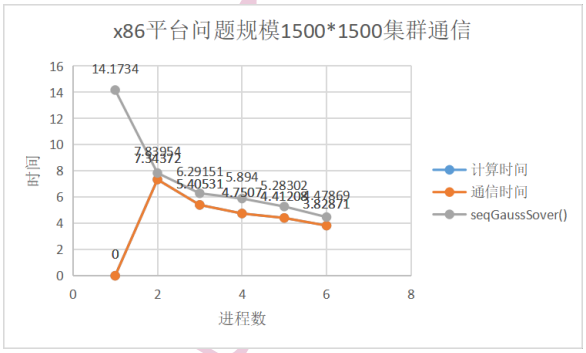
(七) 对比分析

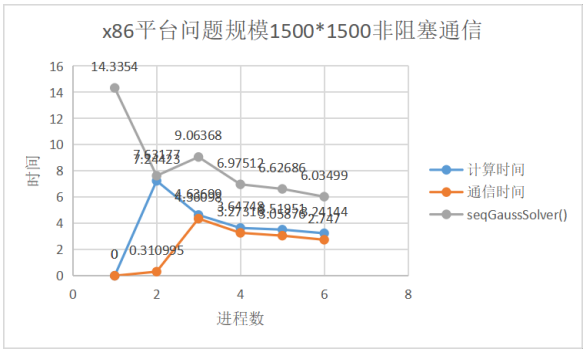
1. arm 平台与 x86

实验截图如图6所示

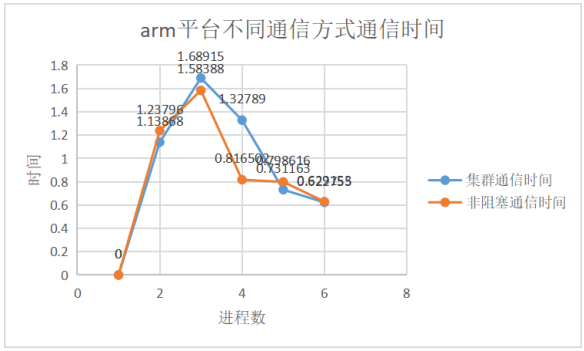

armarm平台问题规模1500*1500集群通信
时间
进程数
计算时间
通信时间
seqGaussSolver()

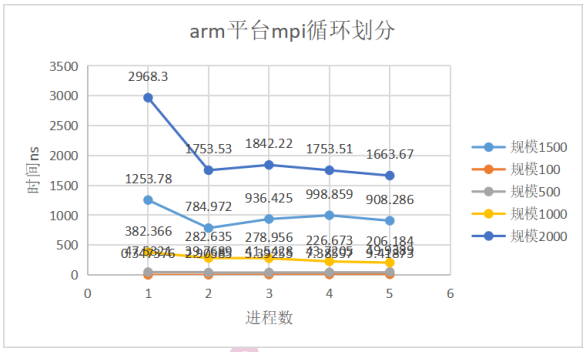

arm平台问题规模1500*1500非阻塞通信
时间
进程数
计算时间
通信时间
seqGaussSolver()


x86平台问题规模1500*1500集群通信
时间
进程数
计算时间
通信时间
seqGaussSover()

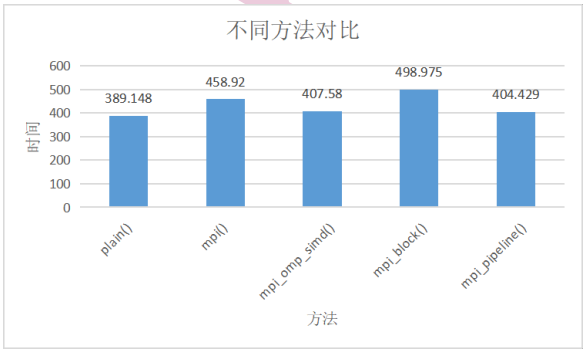

x86平台问题规模1500*1500非阻塞通信
时间
进程数
计算时间
通信时间
seqGaussSolver()

2. arm 平台不同通信方式





3. 不同划分 (循环划分为例)



4. VS 测试

性能分析的“摘要”中会显示 CPU 的使用情况，下边会显示函数名称，其中“非独占时间百分比”是指的包括了子函数执行时间的总执行时间；“独占时间百分比”是不包括子函数执行时间的函数体执行时间，函数执行本身花费的时间，不包括子 (函数) 树执行的时间。点击“main”会出现以下各个函数的时间占比。此时可以知道程序中的各个函数所用时间的百分比，进而进行优化。