南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

高斯消去法的 MPI 编程

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 6 月 9 日

摘要

本实验对普通高斯消元算法和特殊高斯消元算法进行了各种不同形式的 MPI 优化，在普通高斯消元部分，将 MPI 与 SIMD，Pthread，OpenMP 等结合，并讨论不同通信方式的编程实现方式。通过不同数量进程和不同平台下的性能测试与对比，解释各平台的特点，以及各算法及进程数量的影响。

关键字：并行，高斯消去，特殊高斯消去，MPI，通信

目录

一、 实验背景

（一） 实验目的

在本次实验将在 MPI 编程下探究普通高斯消去算法以及特殊高斯消去算法的优化程度，通过性能测试探究 MPI 编程的原理以及其对高斯消去算法的优化情况，同时结合先前实验 SIMD、Pthread&OpenMP 编程讨论，并加以总结。该实验同时也是期末大作业的一个分支。

（二） 实验环境

本次实验主要在以下两个平台进行。了解到 O2 优化或许会降低实验结果可靠性，在本次实验中，不开启任何形式的 O2 优化。

注意：由于本实验需要 MPI 环境，无法在 Windows 环境下完成，因此本实验 x86 平台实验环境替换为 Ubuntu 虚拟机。

- x86 平台：Ubuntu 22.04.4 LTS 虚拟机，windows 11 ，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，mpic++ 编译器。
- ARM 平台：华为鲲鹏服务器，mpic++ 编译器。

二、 普通高斯消去算法

（一） 算法设计

1. 平凡算法

高斯消去法主要分为消去和回代两个部分，如图1。消去部分复杂度较高，为 O(n^3)，回代部分也有 O(n^2) 的复杂度。

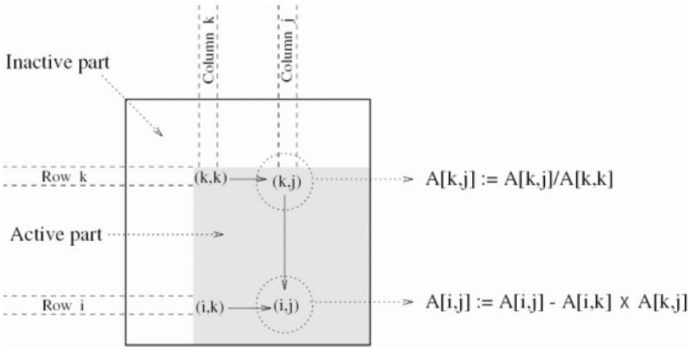


图 1: 高斯消元法示意图

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，全部结束后结果如图2所示，然后从最后一行向上回代。

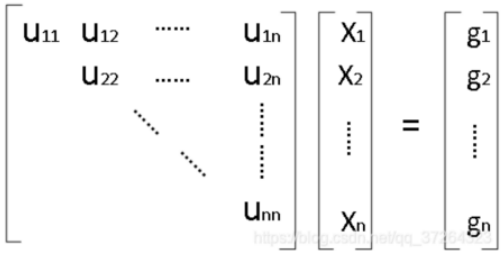


图 2: 消去过程结果示意图

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 x_{i+1},x_{i+2},..., x_n 计算出 x_i。

在本实验中，我们主要考虑高斯消去法的消去过程，即将矩阵消去成上三角矩阵的过程。

2. MPI 一维行划分

- 划分方式 1

  注：该部分除法 $\frac{n}{m}$ 均为整数除法，等效于 $\frac{n-n\%m}{m}$。

  在 MPI 一维行划分算法中，主要实现方式是把每一行分配给每一个进程，假设矩阵规模为 n*n，进程数量为 m(从第 0 个到第 $m-1$ 个)，则对于进程 $i(i!=m-1)$, 分配 $\frac{n}{m}i$ 至 $\frac{n}{m}(i+1)-1$ 行。

  对于进程 $m-1$，要将从 $\frac{n}{m}(m-1)$ 到最后的所有行分配，以保证所有行都被处理。

  相关算法实现在下一部分编程实现中展示，该方法对应后续性能测试实验的 mpi 组。

- 划分方式 2

  考虑一种负载更加均衡的方式，以余数的对应关系给各进程分配行，使得每个进程的负载更加均衡。

  对于进程 i，所有除以进程数量余数为 i 的行将分配给它。

  相关算法实现在下一部分编程实现中展示，该方法对应后续性能测试实验的 mpi_2 组。

3. 流水线划分优化

对于一维行划分方式 1 进行流水线划分优化，每一个进行计算的进程至传递给后一个进程和 0 号进程（为保证输出结果正确），非进行计算且在进行计算之后的进程则从前一个进程处接受，并传递给后一个进程（如果不是最后一个进程）。

相关算法实现在下一部分编程实现中展示，该方法对应后续性能测试实验的 mpi_pipe 组。

4. 非阻塞通信实现

对于两种一维行划分方式的阻塞通信改为非阻塞通信，以降低等待时间，通过 wait 来保证算法正确性。

相关算法实现在下一部分编程实现中展示，该方法对应后续性能测试实验的 mpi_i，mpi_i2 组。

（二） 编程实现

1. 整体模块

a) MPI 编程的特殊设置

在函数外部每次调用算法函数之前，使用 Barrier 保证程序在同一算法中，增加安全性。在算法函数执行完成后，所有算法使用 Barrier 以保证当时获得的结果矩阵的正确性。在计时开始之前，使用 Barrier 保证计算时间开始之前所有进程同步，即确定计时正确性。

虽然在该实验背景下，非 MPI 算法最终的测算结果可能是多个进程中最慢的进程结果，但我采用多次计算取平均值的方法，且各进程执行相同代码花费的时间相差并不多，仍能较好的反映实验现象。

由于运行环境的波动性，我认为将多种算法打包放在单次实验中执行可以更好的保证测试结果的可信性，控制变量。

b) 数组初始化

在函数中，使用当前时间作为随机种子，以保证每一次实验处理的矩阵都各不相同。同时对矩阵进行了一些限制，以更好的避免 inf 和 nan 的出现，保证算法的稳定性以及正确性。

数组初始化函数

```
void reset(int n) {
    srand(time(NULL));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < i; j++)
            A[i][j] = 0;
        A[i][i] = 1.0;
        for(int j = i + 1; j < n; j++)
            A[i][j] = (rand() % 1000 + 1) / 10.0;
    }
    for(int k = 0; k < n; k++)
        for(int i = k + 1; i < n; i++)
            for(int j = 0; j < n; j++)
                A[i][j] += A[k][j];
}
```

c) 正确性检查

在该函数中，使用新建立的算法与平凡算法进行对比，以确定新建立的算法的正确性，同时如上文所述，根据 MPI 编程环境的特点增加 Barrier。后续每个算法都经过了正确性的检查。

正确性检查函数

```
void check(void (*m1)(int), void (*m2)(int)) {
    reset(20);
    float B[20][20];
    for(int i = 0; i < 20; i++) for(int j = 0; j < 20; j++) B[i][j] = A[i][j
        ];
    MPI_Barrier(MPI_COMM_WORLD);
    m1(20);
```

```
    if(mpi_rank==0){（打印方法1结果矩阵）}
    for(int i = 0; i < 20; i++) for(int j = 0; j < 20; j++) A[i][j] = B[i][j
        ];
    MPI_Barrier(MPI_COMM_WORLD);
    m2(20);
     if(mpi_rank==0){（打印方法2结果矩阵）}
}
```

d) 计时

在本次实验中，使用 MPI 自带的计时方法进行计时，大大提高了计时的精度。同时通过 Barrier 的设置保证算法正确性和计时合理性。

计时函数

```
void op(void (*method)(int), int n, int t=5) {
    reset(n);
    if(mpi_rank==0)cout << "␣n=" << n << "␣t=" << t;
    MPI_Barrier(MPI_COMM_WORLD);
    double start_time, end_time;
    if(mpi_rank == 0) start_time = MPI_Wtime();
    for(int i = 0; i < t; i++){
        MPI_Barrier(MPI_COMM_WORLD);
        method(n);
    }
    if(mpi_rank == 0) end_time = MPI_Wtime();
    double pertime = (end_time - start_time) / t;
    if(mpi_rank == 0) cout << "␣pertime=" << pertime << "s\n";
}
```

e) 平凡算法的实现

平凡算法如下所示。

平凡算法

```
void common(int n) {
    for (int k = 0; k < n; ++k) {
        for (int j = k + 1; j < n; ++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][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

2. MPI 一维行划分-1

MPI 一维行划分算法 1 实现如下所示，对应后续实验的 mpi 组。

MPI 一维行划分算法 1

```
void mpi(int n) {
    int per = n / mpi_size;
    int r1 = mpi_rank * per;
    int r2 = (mpi_rank == mpi_size - 1) ? (n - 1) : ((mpi_rank+1) * per - 1);
    for (int k = 0; k < n; k++) {
        if (r1 <= k && k <= r2) {
            for (int j = k + 1; j < n; j++)
                A[k][j] /= A[k][k];
            A[k][k] = 1.0;
            for (int j = 0; j < mpi_size; j++) {
                if(j!=mpi_rank)
                MPI_Send(&A[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
        } else {
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
        }
        for (int i = max(r1,k+1); i <= r2; i++) {
            for (int j = k + 1; j < n; j++)
                A[i][j] -= A[k][j] * A[i][k];
            A[i][k] = 0;
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

3. MPI 一维行划分-2

MPI 一维行划分算法 2 实现如下所示，对应后续实验的 mpi_2 组。

MPI 一维行划分算法 2

```
void mpi_2(int n) {
    for (int k = 0; k < n; k++) {
        if (k%mpi_size==mpi_rank) {
            for (int j = k + 1; j < n; j++)
                A[k][j] /= A[k][k];
            A[k][k] = 1.0;
            for (int j = 0; j < mpi_size; j++) {
                if(j!=mpi_rank)
                MPI_Send(&A[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
        } else {
            MPI_Status status;
```

```
            MPI_Recv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
        }
        for (int i = k+1; i <n; i++) {
            if(i%mpi_size==mpi_rank){
                for (int j = k + 1; j < n; j++)
                    A[i][j] -= A[k][j] * A[i][k];
                A[i][k] = 0;
            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

4. 流水线划分优化

MPI 一维行划分流水线优化算法实现如下所示，对应后续实验的 mpi_pipe 组。

MPI 流水线优化算法

```
void mpi_pipe(int n) {
    //分配进程，同行划分1的2-4行
    for (int k = 0; k < n; k++) {
        if (r1 <= k && k <= r2) {
            //除法，同行划分1的6-9行
            if (mpi_rank != mpi_size - 1)
                MPI_Send(&A[k][0], n, MPI_FLOAT, mpi_rank + 1, 0,
                    MPI_COMM_WORLD);
            if (mpi_rank != 0)
                MPI_Send(&A[k][0], n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
        } else if(k<r1){
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, mpi_rank - 1, 0, MPI_COMM_WORLD,
                &status);
            if (mpi_rank != mpi_size - 1) {
                MPI_Send(&A[k][0], n, MPI_FLOAT, mpi_rank + 1, 0,
                    MPI_COMM_WORLD);
            }
        }else if(!mpi_rank){
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
        }
        //减法，同行划分1的18-22行
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

5. 非阻塞通信实现

MPI 一维行划分非阻塞通信实现如下所示，对应其对于两种 MPI 行划分的优化分别对应后续实验的 mpi_i，mpi_i2 组。

MPI 非阻塞通信

```
//send代码替换如下
MPI_Request request;
MPI_Isend(&A[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD, &request);
//recv代码替换如下
MPI_Status status;
MPI_Request request;
MPI_Irecv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &request
    );
MPI_Wait(&request, &status);
```

6. MPI with Pthread

将 mpi_i2 组与 Pthread 的信号量同步方式（三层循环全部纳入）相结合，在后续实验中对应 mpii2_pt 组，未进行 MPI 优化的 Pthread 优化记为 pt 组。

主调函数的 mpi 优化方式在下面给出，被调函数及其他细节可以由仓库中代码看到，或者也可以从仓库中 Pthread（stat3 组）优化实验中看到具体算法细节。

MPI with Pthread

```
void mpii2_pt(int n) {
    for (int k = 0; k < n; k++) {
        if (k % mpi_size == mpi_rank) {
            //信号量同步计算过程
            //同mpi_i2 isend
        } else {
            //同mpi_i2 irecv
        }
        //同mpi_i2减法
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

7. MPI with OpenMP

将 mpi_i2 组与 OpenMP 多线程优化（常规划分方式）相结合，在后续实验中对应 mpii2_mp 组，未进行 MPI 优化的 OpenMP 优化记为 mp 组。

mpi 优化方式在下面给出，其他细节可以由仓库中代码看到，或者也可以从仓库中 OpenMP（mp 组）优化实验中看到具体算法细节。

MPI with OpenMP

```
void mpii2_mp(int n) {
    int i, j, k;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
```

```
    for (k = 0; k < n; k++) {
        #pragma omp single
        {
            //同mpi_i2除法及通信
        }
        #pragma omp for
        //同mpi_i2减法
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

8. MPI with SSE

将 mpi_i2 组与 SSE 向量化优化相结合，该部分仅能够在 x86 平台下测试，对应 mpii2_sse 组，为进行 MPI 优化的 SSE 优化记为 sse 组。

mpi with sse 优化方式在下述代码中给出。

MPI with SSE

```
//将mpi_i2组的减法过程（即对应于mpi_2组的17-18行），替换为如下代码
int j;
__m128 aik=_mm_set_ps(A[i][k],A[i][k],A[i][k],A[i][k]);
for(j=k+1;j<=n-4;j+=4){
    __m128 akj=_mm_loadu_ps(A[k]+j);
    __m128 multi=_mm_mul_ps(aik,akj);
    __m128 aij=_mm_loadu_ps(A[i]+j);
    aij=_mm_sub_ps(aij,multi);
    _mm_storeu_ps(A[i]+j,aij);
}
for (;j<n; j++)
    A[i][j] -= A[i][k] * A[k][j];
```

9. MPI with NEON

将 mpi_i2 组与 NEON 向量化优化相结合，该部分仅能够在 ARM 平台下测试，对应 mpii2_neon 组，为进行 MPI 优化的 NEON 优化记为 neon 组。

mpi with neon 优化方式在下述代码中给出。

MPI with NEON

```
//将mpi_i2组的减法过程（即对应于mpi_2组的17-18行），替换为如下代码
int j;
float32x4_t aik=vmovq_n_f32(A[i][k]);
for(j=k+1;j<=n-4;j+=4){
    float32x4_t akj=vld1q_f32(A[k]+j);
    float32x4_t multi=vmulq_f32(aik,akj);
    float32x4_t aij=vld1q_f32(A[i]+j);
    aij=vsubq_f32(aij,multi);
    vst1q_f32(A[i]+j,aij);
```

```
10 }
11 for (;j<n; j++)
12     A[i][j] -= A[i][k] * A[k][j];
```

（三） 性能测试

1. 各种 MPI 算法

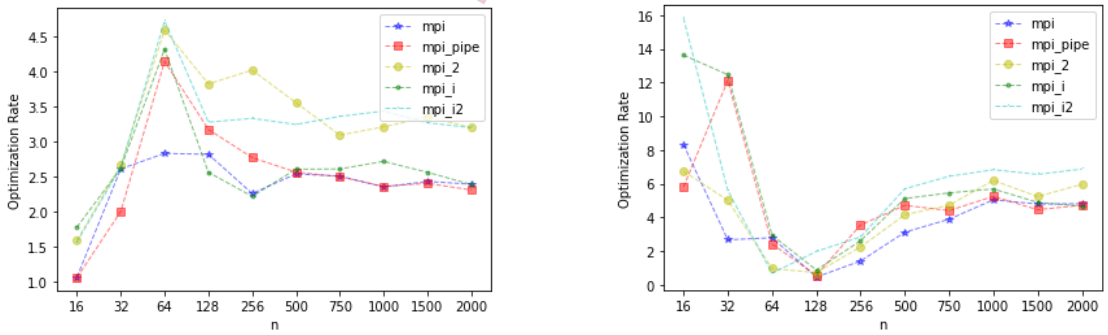
a) x86 平台

在 x86 平台下，对 4 进程和 8 进程的 mpi 优化程序进行性能测试，结果如表1所示。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0174	0.0912	0.510	5.05	25.1	175	585	1368	4475	10504
mpi_4	0.0163	0.0349	0.180	1.79	11.1	68.9	233	581	1839	4388
mpi_pipe_4	0.0164	0.0455	0.123	1.59	9.05	68.2	234	579	1860	4543
mpi_2_4	0.0109	0.0342	0.111	1.32	6.24	49.2	189	426	1334	3279
mpi_i_4	0.00977	0.0348	0.118	1.97	11.3	67.1	224	503	1745	4394
mpi_i2_4	0.0111	0.0346	0.108	1.54	7.53	53.9	174	398	1368	3283
$common_8$	2.262	1.297	0.584	5.91	35.1	260	869	2148	7185	17006
mpi_8	0.272	0.488	0.210	13.2	25.1	83.7	224	426	1502	3539
mpi_pipe_8	0.390	0.107	0.242	10.9	9.93	55.2	198	410	1615	3607
mpi_2_8	0.336	0.259	0.611	8.38	15.7	62.8	184	347	1374	2857
mpi_i_8	0.166	0.104	0.199	6.85	13.5	50.9	160	378	1472	3647
mpi_i2_8	0.143	0.232	0.863	3.01	12.4	45.7	135	315	1097	2479

表 1: x86 平台性能测试结果 (下标表示进程数量)(单位:ms)

对比同进程下与平凡算法相比较，结果如图3所示，左图为 4 进程的对比结果，右图为 8 进程的对比结果。



(a) 4 进程 (b) 8 进程

图 3: 各进程 x86 平台加速比

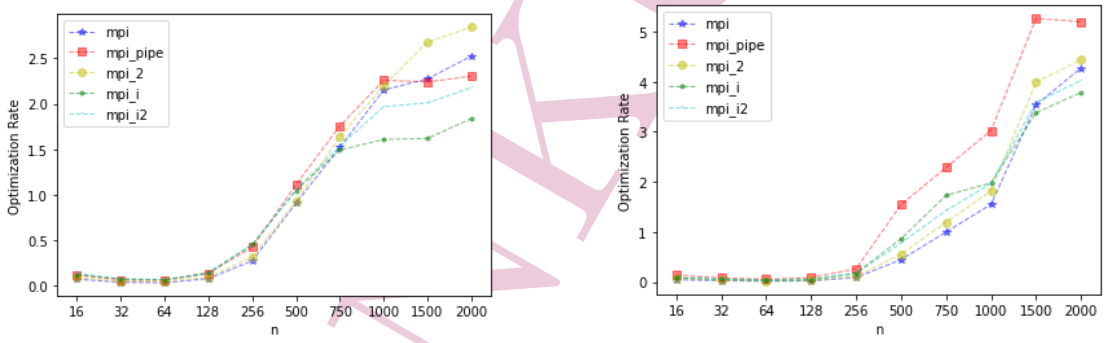
b) ARM 平台

在 ARM 平台下，对 4 进程和 8 进程的 mpi 优化程序进行性能测试，结果如表2所示。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0978	0.103	0.179	0.777	5.73	56.4	199	482	1421	3431
mpi_4	1.36	2.72	5.41	9.98	20.8	62.2	131	224	625	1360
mpi_pipe_4	0.827	1.60	3.04	5.73	13.5	50.8	114	213	634	1490
mpi_2_4	1.07	2.11	3.96	8.30	18.1	60.9	122	219	531	1206
mpi_i_4	0.793	1.42	2.68	5.33	12.69	53.9	134	300	878	1867
mpi_i2_4	0.707	1.36	2.60	5.81	12.7	53.5	130	245	707	1575
$common_8$	0.146	0.258	0.293	0.887	5.85	60.3	206	483	2009	4184
mpi_8	3.07	9.40	17.3	35.1	64.7	135	207	312	568	980
mpi_pipe_8	1.06	3.15	5.19	10.1	22.2	39.0	90.3	160	382	804
mpi_2_8	2.38	7.36	13.1	27.4	52.6	109	173	266	503	943
mpi_i_8	1.71	4.47	8.24	17.5	32.7	69.9	119	244	594	1106
mpi_i2_8	1.62	4.64	8.72	18.8	35.1	77.3	145	245	560	1040

表 2: ARM 平台性能测试结果 (下标表示进程数量)(单位:ms)

对比同进程下与平凡算法相比较，结果如图4所示，左图为 4 进程的对比结果，右图为 8 进程的对比结果。



(a) 4 进程 (b) 8 进程

图 4: 各进程 ARM 平台加速比

2. MPI with SIMD,Pthread,OpenMP

a) x86 平台

在 x86 平台下，对 4 进程的 mpi 优化程序进行性能测试，其中 pthread 和 openmp 的线程数均为 8，结果如表3所示。

与平凡算法相比较，结果如图5所示。

b) ARM 平台

在 ARM 平台下, 对 4 进程和 8 进程的 mpi 优化程序进行性能测试, 其中 pthread 和 openmp 的线程数均为 8，结果如表2所示。

对比同进程下与平凡算法相比较，结果如图6所示，左图为 4 进程的对比结果，右图为 8 进程的对比结果。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0174	0.0912	0.510	5.05	25.1	175	585	1368	4475	10504
pt_4	5.61	8.06	15.7	29.7	51.6	210	540	1357	3746	8471
$mpii2\_pt_4$	27.1	19.5	38.8	129	160	377	742	1654	3894	8554
mp_4	173	345	686	1384	2575	5100	8240	12303	17900	27198
$mpii2\_mp_4$	191	361	719	1471	2756	5519	8990	11564	16970	23697
sse_4	0.00746	0.0418	0.334	1.98	11.97	99.2	321	755	2565	5838
$mpii2\_sse_4$	0.0115	0.0286	0.107	0.550	4.39	29.1	84.9	206	745	1878

表 3: x86 平台性能测试结果 (4 进程)(单位:ms)

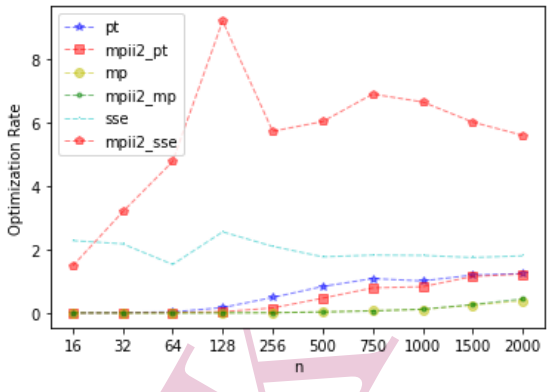
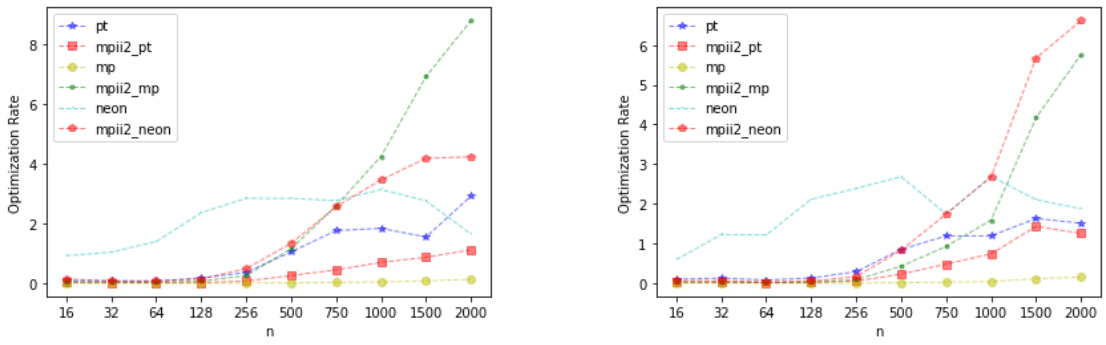


图 5: x86 平台优化比

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0978	0.103	0.179	0.777	5.73	56.4	199	482	1421	3431
pt_4	1.16	1.40	2.48	4.54	15.9	54.0	114	263	923	1179
$mpii2\_pt_4$	5.12	10.0	19.5	41.6	87.4	222	451	694	1648	3081
mp_4	0.641	0.450	0.804	1.35	4.48	12.7	29.3	59.9	188	444
$mpii2\_mp_4$	2.26	3.27	6.83	11.5	22.8	49.2	77.2	114	206	391
$neon_4$	0.107	0.099	0.129	0.331	2.02	19.9	72.3	154	516	2093
$mpii2\_neon_4$	0.718	1.36	2.69	5.53	11.8	42.5	78.2	140	341	813
$common_8$	0.146	0.258	0.293	0.887	5.85	60.3	206	483	2009	4184
pt_8	1.50	2.04	3.65	6.95	20.1	70.9	173	406	1233	2767
$mpii2\_pt_8$	6.05	14.4	27.1	55.7	114	265	430	655	1408	3329
mp_8	1.25	0.844	1.31	2.53	5.45	12.5	34.0	66.0	260	647
$mpii2\_mp_8$	5.34	8.42	17.2	34.7	68.0	142	223	305	482	727
$neon_8$	0.247	0.211	0.241	0.420	2.45	22.5	118	179	954	2234
$mpii2\_neon_8$	2.40	4.49	9.56	16.7	35.8	72.9	119	181	354	632

表 4: ARM 平台性能测试结果 (下标表示进程数量)(单位:ms)



(a) 4 进程 (b) 8 进程

图 6: 各进程 ARM 平台加速比

（四） 总结

1. 各种 MPI 算法

各 mpi 优化算法都展示出了较好的优化性能。8 进程的优化性能往往好于 4 进程。在 x86 平台下，流水线优化算法 (mpi_pipe) 并未能展现出相比于普通 mpi 算法的优势，第二种更为均衡的行划分方式的性能相比于第一种有着较为明显的性能提升，两种划分方式的非阻塞通信效率均高于阻塞通信。

这体现了更加均衡的划分方式有利于提升程序性能，非阻塞通信在不必要的阶段不进行等待也能够提升程序性能，流水线优化算法未能取得较好的成绩，可能是因为尤其是在第一种划分方式下，排队等待接收的开销相比于单一进程进行消元的开销较为显著，可能也于机器性能有关。

在 ARM 平台上，反而是阻塞通信组好于非阻塞通信组，这可能是由于 ARM 平台上由于数据传输足够快，对于非阻塞通信的判定开销反而高于阻塞的等待开销导致。pipe 组虽然在 4 进程上没有取得太好的成绩，但是在 8 进程上成绩非常显著，说明在进程较多的环境下，流水线优化对于程序性能的提升还是非常有必要的，

同时，我们会发现无论是在 x86 平台上还是 ARM 平台上，没有使用优化的平凡算法因为 barrier 的缘故造成了更多的运行时间，这绝对不是多次实验取最大值的结果。这在 x86 平台上尤为显著，这可能是因为个人设备的性能不足导致，在 ARM 平台上虽然问题依然存在，但没有特别显著，这得益于鲲鹏服务器优异的性能。

2. MPI with SIMD,Pthread,OpenMP

x86 平台上，仅有 sse 组和 MPI with sse 组取得了较好的成绩。普通的 Pthread 和 OpenMP 算法相比于不在 MPI 环境下运行的时候慢了特别多（这对于平凡算法并没有如此显著，但他们几乎是数量级的损耗）。原因也很容易理解，即使我已经为我的虚拟机分配了等同于我物理机的核心数量，但他依然还是太少了，在多线程和多进程的工作背景下造成的计算资源完全不够用。

ARM 平台上，neon 组也保持了非常好的表现，同时于 mpi 的结合更是取得了非常优异的结果，这也说明了向量化编程的广泛适配性和极强的性能。此外，OpenMP with MPI 的异军突起值得我们注意。或许在鲲鹏服务器下，计算资源足够多，OpenMP 和 MPI 的配合可以取得非常不错的成绩。Pthread with MPI 组甚至几乎没能超过仅 Pthread 组，这可能是我们 MPI 的算法不太适用，或者在结合过程和结合方式上还有很大提升空间。

三、 特殊高斯消去算法

（一） 算法设计

1. 平凡算法

特殊高斯消去计算来自一个实际的密码学问题–Gröbner 基的计算。

运算均为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1。1. 其加法运算实际上为异或运算:0+0=0、0+1=1、1+0=1、1+1=0 由于异或运算的逆运算为自身，因此减法也是异或运算。乘法运算 0*0=0、0*1=0、1*0=0、1*1=0。因此，高斯消去过程中实际上只有异或运算–从一行中消去另一行的运算退化为减法。

考虑单次完成数据读取的简单情况，此特殊的高斯消去计算过程 (串行算法) 如下:

对于每个被消元行，检查其首项，如有对应消元子则将其减去 (异或) 对应消元子，重复此过程直至其变为空行 (全 0 向量) 或首项无对应消元子。

如果某个被消元行变为空行，则将其丢弃，不再参与后续消去计算；如其首项被没有对应消元子，则将它“升格”为消元子，在后续消去计算中将以消元子身份而不再以被消元行的身份参与;

重复上述过程，消元子和被消元行共同组成结果矩阵–可能存在很多空行。

2. MPI 算法

考虑这样一个方式，若可供分配 n 个进程，则将 n-1 个进程用于在他们各自的程序中消元他们被分配到的行，再将他们的消元结果发送给 0 号进程，0 号进程使用他们的消元结果进行进一步消元。这样的分配方式可以在保证消元结果正确性的前提下，最大限度的应用进程的负载，即使最后一部分工作仍然需要 0 号进程独立完成。

（二） 编程实现

在这部分展示了应用 MPI 计时器重写的计时功能，这可以更加精确的获得较短时间的样例的运行时间，对该部分实验非常有帮助。此外，展示了 mpi 的优化算法。对于读写操作以及平凡算法的建立，除下文中说明的一处修改以外没有其他过多修改，可于仓库中查看详细代码，并从从先前编程实验报告中得到详细解释。

MPI 计时

```
void op(void(*method)(), int t=5) {
    double pertime,start_time,end_time;
    for (int i = 0; i < t; i++) {
        memset(row, 0, sizeof(row));
        memset(xyz, 0, sizeof(xyz));
        memset(is, 0, sizeof(is));
        read();
        MPI_Barrier(MPI_COMM_WORLD);
        if(mpi_rank == 0) start_time = MPI_Wtime();
        method();
        if (mpi_rank == 0) {
            end_time = MPI_Wtime();
            pertime+=(end_time-start_time);
```

```
            write();
        }
    }
    if (mpi_rank == 0) cout << pertime/t*1000 << "ms\n";
}
```

MPI 优化算法

```
void mpi() {
    int rows_per_proc = nrow / (mpi_size - 1);
    int start = (mpi_rank - 1) * rows_per_proc;
    int end = (mpi_rank == mpi_size - 1) ? nrow : start + rows_per_proc;
    if (mpi_rank != 0) {
        for (int i = start; i < end; i++) {
            bool isend = false;
            for (int j = xlen - 1; j >= 0 && !isend; j--) {
                while (!isend) {//这里的判断条件进行了调整, 以使得消元结束后立即停止, 平凡算法也进行同步调整
                    if (!row[i][j]) break;
                    int f = -1;
                    for (int u = 31; u >= 0; u--) {
                        if ((row[i][j]) & (1 << u)) {
                            f = u + 32 * j;
                            break;
                        }
                    }
                    if (f == -1) break;
                    if (!is[f]) {
                        for (int k = 0; k < xlen; k++)
                            xyz[f][k] = row[i][k];
                        is[f] = true;
                        isend = true;
                    } else {
                        for (int k = 0; k < xlen; k++)
                            row[i][k] ^= xyz[f][k];
                    }
                }
            }
            MPI_Send(row[i], xlen, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    } else {
        for (int i = 0; i < nrow; i++) {
            MPI_Status status;
            MPI_Recv(row[i], xlen, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            bool isend = false;
            //消元, 同7-29行
        }
    }
```

```
40      MPI_Barrier(MPI_COMM_WORLD);
41  }
```

（三） 性能测试

在各组实验中，测试 ARM 平台（因为该环境更加稳定）下，4 进程和 8 进程的平凡算法和 MPI 优化算法的表现。xsize 设置为矩阵列数，maxrow 设置为被消元行数，以避免无意义的 0 计算开销。各组计算五次取得平均值，结果如表5所示。

样例编号	1	2	3	4	5	6	7
矩阵列数	130	254	562	1011	2362	3799	8399
非零消元子数	22	106	170	539	1226	2759	6375
被消元行数	8	53	53	263	453	1953	4535
common_4	0.0525	0.105	0.114	2.45	15.7	377	4512
mpi_4	0.209	0.844	0.979	4.92	22.5	155	1622
common_8	0.0793	0.135	0.146	2.48	15.9	383	5124
mpi_8	0.263	1.06	1.096	5.93	18.3	140	1115

表 5: ARM 平台测试结果

（四） 总结

可以看出，MPI 优化算法在较大规模消元下，二次消元的优化效果非常显著。虽然在规模较小下并不显著，但在规模足够大的情况下，如第七组，在 4 进程中得到了将近 3 倍的加速比，在 8 进程中更是超过 4 倍。这得益于其更好的使用了更多的进程进行计算。

四、 总结

在本次实验中，我对普通高斯消元算法和特殊高斯消元算法进行了各种不同形式的 MPI 优化，在普通高斯消元部分，我还将 MPI 与 SIMD，Pthread，OpenMP 等结合起来，还讨论了不同通信方式对性能的影响。

本次实验使我很好的了解了 MPI 的编程方式和运行流程，对各进程之前的通信也熟练了解。通过不同数量进程和不同平台下的性能测试与对比，我也更加了解了各平台的特点，以及各算法及进程数量的影响。本次实验中，无论是普通高斯消元算法还是特殊高斯消元算法，都得到了较好的优化性能。

这次实验也是期末实验报告中的重要实验环节。相关完整代码和部分运行结果可在GitHub仓库中查看。