南開大學
Nankai University

计算机学院

并行程序设计实验报告报告

高斯消去 MPI 优化

姓名： 高祎珂

学号：2011743

专业：计算机科学与技术

2022 年 6 月 23 日

目录

1 实验介绍

1.1 实验简介

对于普通高斯消去在之前的实验中有过详细介绍，此处不再叙述，此实验就是利用 mpi 对高斯消去进行进一步优化。在程序中，不同的进程需要相互的数据交换，特别是在科学计算中，需要大规模的计算与数据交换，集群可以很好解决单节点计算力不足的问题，但在集群中大规模的数据交换是很耗费时间的，MPI 可以在多节点的情况下快速进行数据交流，它可移植性强，功能强大，也是目前主要的并行化采用的方式。此实验利用 MPI 的消息传递进行高斯消去。

1.2 实验设计

1. 首先生成后续进行高斯消去的矩阵，编写代码实现高斯消去串行算法。
2. 在 ARM 平台上，进行 MPI 实验，根据伪代码编写程序，按块划分，计算各进程的任务号，进行 MPI 的消息传递以及之后的计算，查看不同数据规模和不同并发度的优化效果。
3. 加入 SIMD 和 OpenMP 进行优化，编写程序，查看优化效果。
4. 在 X86 平台上进行同样操作，分析在 X86 平台上的优化表现，最后比较一下两平台。
5. 编写流水线算法，比较流水线算法与块划分的差异。

1.3 理论分析

MPI 如果开辟出 n 个进程，理论上应该能达到 n 倍的加速比，但是由于这个过程的通信开销，达不到这么高的优化，开销的大小这需要通过实验来验证。理论上加入 SIMD 和 OMP 的优化的话，优化效果可以达到 n×m 的效果，流水线算法会减少等待时间，理论上效果应该是会比块划分更优，这些理论猜想都需要实验结果来验证。

2 X86 平台

2.1 算法设计

对于块划分，设置 id 为 0 的进程执行本应该进行串行的操作，记录时间，分配任务，其他进程进行出发和消去，最后一个进程执行 per_block+remian 行，进行中间的消息传递和任务分配，执行 SIMD 时，修改执行消去的代码，实现 SIMD，流水线算法改变任务划分以及消息传递方式，一个进程负责行的除法运算完成之后，消息转发给下一个进程；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。

代码链接为github, 这里只展示一下核心代码。

MPI 块任务划分

```
1     void eliminate(float mat[][N], int rank, int num_proc)
2 {
3     int block = N / num_proc;
```

```
    //    未能整除划分的剩余部分
    int remain = N % num_proc;

    int begin = rank * block;
    //    当前进程为最后一个进程时，需处理剩余部分
    int end = rank != num_proc - 1 ? begin + block : begin + block + remain;
    for (int k = 0; k < N; k++)
    {
        //        判断当前行是否是自己的任务
        if (k >= begin && k < end)
        {
            for (int j = k + 1; j < N; j++)
                mat[k][j] = mat[k][j] / mat[k][k];
            mat[k][k] = 1.0;
            //            向之后的进程发送消息
            for (int p = rank + 1; p < num_proc; p++)
                MPI_Send(&mat[k], N, MPI_FLOAT, p, 2, MPI_COMM_WORLD);
        }
        else
        {
            int cur_p = k / block;
            //            当所处行属于当前进程前一进程的任务，需接收消息
            if (cur_p < rank)
                MPI_Recv(&mat[k], N, MPI_FLOAT, cur_p, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        }
        for (int i = begin; i < end && i < N; i++)
        {
            if (i >= k + 1)
            {
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0.0;
            }
        }
    }
}
```

MPI 块划分 id 为 0 执行

```
    if (rank == 0)
    {
        init_mat(mat);
  QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&head);
        //        在0号进程进行任务划分
        for (int i = 1; i < num_proc; i++)
        {
            if (i != num_proc - 1)
```

```
            {
                for (int j = 0; j < block; j++)
                    MPI_Send(&mat[i * block + j], N, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
            }
            else
            {
                for (int j = 0; j < block + remain; j++)
                    MPI_Send(&mat[i * block + j], N, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
            }
        }
        eliminate(mat, rank, num_proc);
        //        处理完0号进程自己的任务后需接收其他进程处理之后的结果
        for (int i = 1; i < num_proc; i++)
        {
            if (i != num_proc - 1)
            {
                for (int j = 0; j < block; j++)
                    MPI_Recv(&mat[i * block + j], N, MPI_FLOAT, i, 1,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            else
            {
                for (int j = 0; j < block + remain; j++)
                    MPI_Recv(&mat[i * block + j], N, MPI_FLOAT, i, 1,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }
        QueryPerformanceCounter((LARGE_INTEGER*)&tail);
        //  cout <<"静态"<<(tail-head)*1000.0/freq<<"ms"<<endl;
        cout << "mpishijian          "<<(tail - head) * 1000.0 / freq << "
            ";
        print_mat(mat);
    }
```

MPI 块划分 id 不为 0 执行

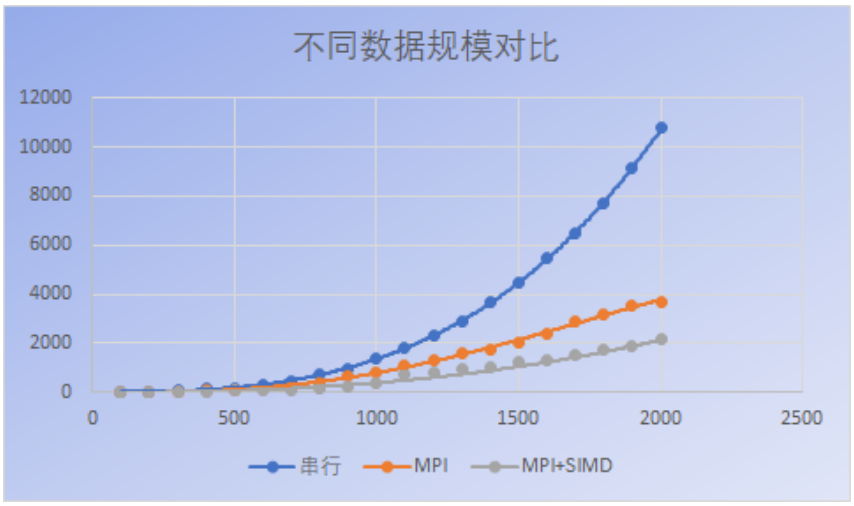
```
    //        非0号进程先接收任务
        if (rank != num_proc - 1)
        {
            for (int j = 0; j < block; j++)
                MPI_Recv(&mat[rank * block + j], N, MPI_FLOAT, 0, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        else
        {
            for (int j = 0; j < block + remain; j++)
                MPI_Recv(&mat[rank * block + j], N, MPI_FLOAT, 0, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
```

```
        eliminate(mat, rank, num_proc);
        //          处理完后向零号进程返回结果
        if (rank != num_proc - 1)
        {
            for (int j = 0; j < block; j++)
                MPI_Send(&mat[rank * block + j], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
        }
        else
        {
            for (int j = 0; j < block + remain; j++)
                MPI_Send(&mat[rank * block + j], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
        }
```

而加入 SIMD 的话，只需要按照之前 SIMD 的思路，在进行消去的过程中，进行 SSE 优化，使用 SSE 语言，进行优化，此处不再进行代码展示。完整代码在github

2.2 不同规模下结果分析

数据规模	串行/ms	MPI/ms	MPI+SIMD/ms
100	1.3139	2.7522	2.8172
200	11.6317	9.666	6.0645
300	80.8055	24.4053	14.5795
400	141.789	66.8381	26.4507
500	192.091	107.97	58.7845
600	301.722	181.89	81.1587
700	477.348	253.946	108.907
800	698.812	387.507	191.763
900	963.811	630.371	215.234
1000	1355.28	832.078	343.83
1100	1831.78	1068.73	702.141
1200	2284.96	1297.93	787.199
1300	2904.35	1585.37	954.662
1400	3648.33	1720.99	1055.39
1500	4490.82	2018.42	1208.35
1600	5470.1	2367.25	1337.26
1700	6502.8	2920.14	1549.75
1800	7704.91	3199.64	1756.91
1900	9132.01	3506.89	1854.42
2000	10767.6	3709.56	2140.76

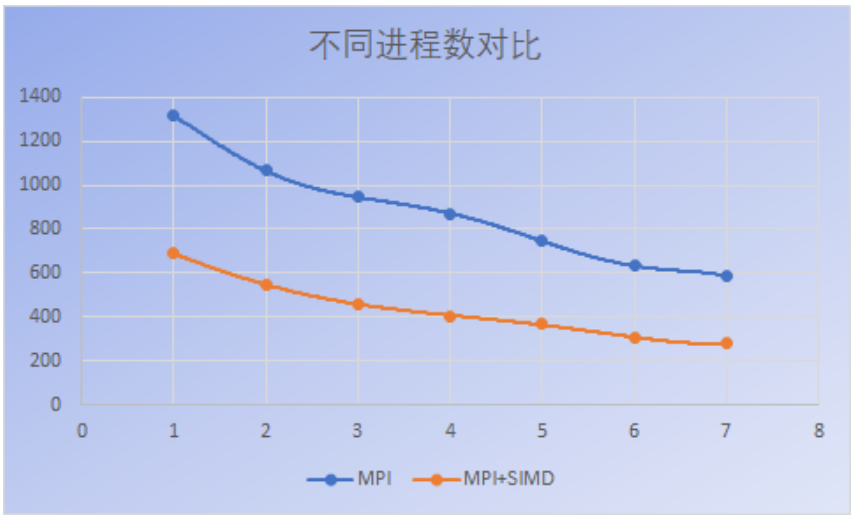


此结果是设置进程数为 4 时得到的，上图和上表中可以看到整体上大致符合预期，设置 4 个进程在数据规模较大时基本实现了 2 倍多的优化效果，但是在数据规模较小的时候优化效果并没有很好，可能是因为通信开销比较大，在数据规模较小时，优化效果体现的不明显。

加入 SIMD 优化后，经过之前的实验结果，我们可以得到 SIMD 优化可以达到 2 倍多的优化效果，查看这个表中数据，在数据规模较小时，确实可以达到大概 2 倍的优化，但是当数据规模增大时，这个优化效果就没有那么明显，叠加效果没有单独使用 SIMD 效果好。

2.3 不同线程数下结果分析

进程数	MPI/ms	MPI+SIMD/ms
1	1313.34	688.153
2	1064.57	546.267
3	944.561	463.096
4	872.017	402.911
5	746.109	368.373
6	632.957	307.373
7	586.913	279.197



基于上述实验，这里也是选择了矩阵规模为 1000 进行实验测试，对比串行时间，可以看到进程数为 1 的情况下，就相当于是串行，MPI+SIMD 实验就是只进行了 SIMD 优化，这个优化效果跟只进行 SIMD 优化是相似的大概达到了两倍，随着进程数的增大，优化效果在不断增加，但是结果并没有想理论上的那么优异，设置的进程数的优化效果并不很好，甚至连基本的倍数都没有达到，可能是因为过程中的通信消耗有些大，严重影响了 MPI 的优化。可以看到整体呈现出来的规律和预期一致，随着进程数的减小，运行时间也在不断增长，且在进程数较大时，这种减小不是很明显。

2.4 流水线划分

源码 github 链接戳这里。下面是一些核心代码。

MPI 流水线算法任务划分

```
    void eliminate(float mat[][N], int rank, int num_proc)
{
    int seg = task * num_proc;
    //    计算当前进程的前一进程及下一进程
    int pre_proc = (rank + (num_proc - 1)) % num_proc;
    int next_proc = (rank + 1) % num_proc;
    for (int k = 0; k < N; k++)
    {
        //        判断当前行是否是自己的任务
        if (int((k % seg) / task) == rank)
        {
            for (int j = k + 1; j < N; j++)
                mat[k][j] = mat[k][j] / mat[k][k];
            mat[k][k] = 1.0;
            //            处理完自己的任务后向下一进程发送消息
            MPI_Send(&mat[k], N, MPI_FLOAT, next_proc, 2, MPI_COMM_WORLD);
        }
        else
        {
            //            如果当前行不是当前进程的任务，则接收前一进程的消息
            MPI_Recv(&mat[k], N, MPI_FLOAT, pre_proc, 2,
```

```
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            //            如果当前行不是下一进程的任务，需将消息进行传递
            if (int((k % seg) / task) != next_proc)
                MPI_Send(&mat[k], N, MPI_FLOAT, next_proc, 2, MPI_COMM_WORLD);
        }
        for (int i = k + 1; i < N; i++)
        {
            if (int((i % seg) / task) == rank)
            {
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0.0;
            }
        }
    }
}
```

MPI 流水线 id 为 0 执行

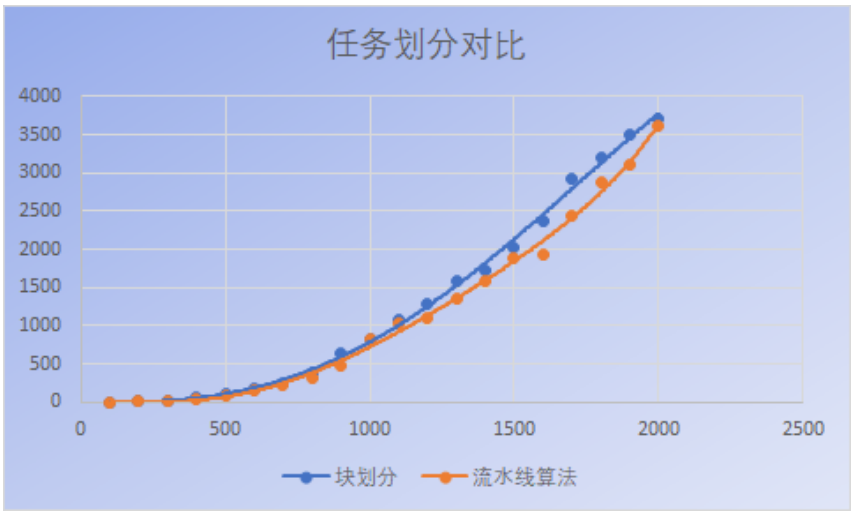
```
     int seg = task * num_proc;
    if (rank == 0)
    {
        init_mat(mat);
        QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
        QueryPerformanceCounter((LARGE_INTEGER*)&head);
        //        在0号进程进行任务划分
        for (int i = 0; i < N; i++)
        {
            int flag = (i % seg) / task;
            if (flag == rank)
                continue;
            else
                MPI_Send(&mat[i], N, MPI_FLOAT, flag, 0, MPI_COMM_WORLD);
        }
        eliminate(mat, rank, num_proc);
        //        处理完0号进程自己的任务后需接收其他进程处理之后的结果
        for (int i = 0; i < N; i++)
        {
            int flag = (i % seg) / task;
            if (flag == rank)
                continue;
            else
                MPI_Recv(&mat[i], N, MPI_FLOAT, flag, 1, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        }
        QueryPerformanceCounter((LARGE_INTEGER*)&tail);
        //  cout <<"静态"<<(tail-head)*1000.0/freq<<"ms"<<endl;
        cout << "mpishijian     " << (tail - head) * 1000.0 / freq << " ";
        //print_mat(mat);
```

```
    }
```

MPI 流水线 id 不为 0 执行

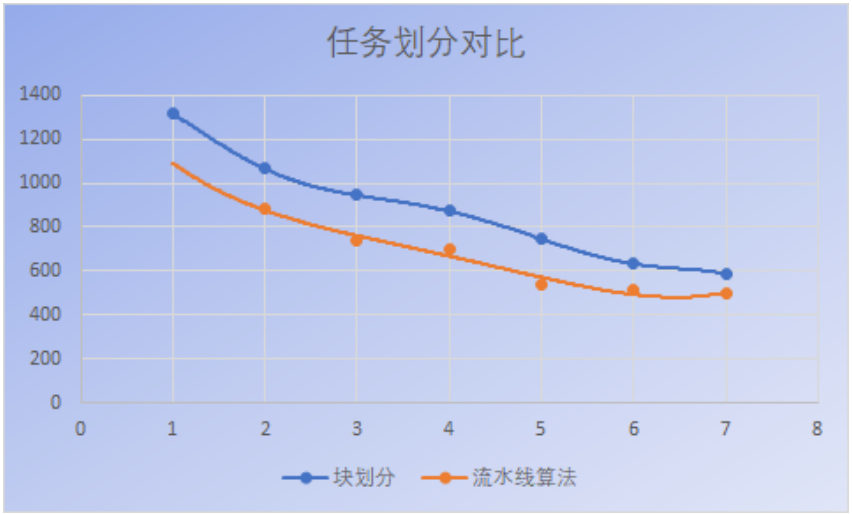
```
    //          非0号进程先接收任务
        for (int i = task * rank; i < N; i += seg)
        {
            for (int j = 0; j < task && i + j < N; j++)
                MPI_Recv(&mat[i + j], N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        }
        eliminate(mat, rank, num_proc);
        //          处理完后向零号进程返回结果
        for (int i = task * rank; i < N; i += seg)
        {
            for (int j = 0; j < task && i + j < N; j++)
                MPI_Send(&mat[i + j], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
        }
```

数据规模	块划分 MPI/ms	流水线算法 MPI/ms
100	2.7522	3.3037
200	9.666	9.3231
300	24.4053	25.3843
400	66.8381	43.07
500	107.97	76.2018
600	181.89	145.623
700	253.946	231.87
800	387.507	318.093
900	630.371	471.615
1000	832.078	826.578
1100	1068.73	1035.98
1200	1297.93	1091.78
1300	1585.37	1365.07
1400	1720.99	1578.56
1500	2018.42	1881.76
1600	2367.25	1943.93
1700	2920.14	2443.79
1800	3199.64	2887.19
1900	3506.89	3099.88
2000	3709.56	3609.75



这是固定进程数为 4 得出的结果，根据实验结果我们可以看到基本与预期一致，使用流水线算法相较于块划分来比有一定的优化，且随着数据规模的增大，这个优化在上升，这是因为数据规模变大，消息传递就会变多，这样流水线的优势就体现出来了，下面看一下不同线程时的流水线算法的优化效果。

进程数	块划分 MPI/ms	流水线 MPI/ms
1	1313.34	/
2	1064.57	880.071
3	944.561	743.226
4	872.017	703.171
5	746.109	537.01
6	632.957	511.237
7	586.913	500.239



从上述结果可以看出来，使用流水线算法，对于不同进程均有一定的优化效果，随着进程数的增加，这个优化效果在减小。

3 ARM 平台

这次试验进行了 ARM 平台的编码，但是最后在鲲鹏服务器上没有分配到节点，先把代码放在这里

4 总结

通过本次实验，将课堂上学到的 MPI 只是应用于实践，对 MPI 的理解更深刻了，发现理论和实践还是有一定距离的，在代码编写过程中，消息传递最开始不太确定要怎么写，怎么给各进程分配任务，后来查找资料解决了相关问题，以及时间函数放置的位置，本来以为进程只执行 MPI_Init 和 MPI_Finalize 之间的，但后来发现每个进程都执行，这两个函数只是说明之前和知乎不能有 MPI 函数，并没有销毁进程，这个后来找到了进行类串行的解决方法。感觉 MPI 编程和之前的感觉不太一样，这个对人物的划分会更复杂一点，要有严格的分割和消息传递。