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 MPI 并行化实验报告

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 6 月 2 日

摘要

针对高斯消元法 (LU 分解) 的 MPI 并行化，采取不同的数据划分方式，并结合 OpenMP 以及 SSE/AVX。对每一种算法进行结果与原因分析并 Profiling。

关键字：LU 分解；MPI 编程；SSE/AVX；OpenMP

目录

一、 概述

(一) 问题描述

高斯消元法 (Gaussian Elimination) 是线性代数中的一个算法, 用于线性方程组求解、求矩阵的秩、求可逆方阵的逆矩阵。[1]

高斯消元法实际上就是最朴素的线性方程组消元, 其时间复杂度是 $O(n^3)$, 是一个在矩阵规模 N 较大时非常费时的算法, 且在高斯消元法的串行平凡算法中, 并非每一步都是必须的, 或者某几步是可以进行向量化的循环的, 使用 MPI 进行并行化的时候需要关注的重点是线程间信息的传递。

(二) 实验内容

针对高斯消去法 (LU 分解) 进行以下 MPI 并行化实现:

- 串行朴素 LU
- MPI 按块/循环划分 LU 消去并行处理
- MPI 结合 OpenMP 以及 SSE/AVX 进行并行处理
- MPI 流水线算法

并对各种并行化方法进行性能和参数上的比较分析。

(三) 实验环境

- 系统环境: MacOS 10.15 (64 位), 金山云主机
- 编译环境: mpich-3.0.4

二、 算法设计与实现

(一) 串行朴素 LU 实现

根据 Wiki 以及实验指导书, 串行朴素 LU 实际上就是将矩阵化为一个上三角矩阵, 分别对其实现按行和按列划分:

1. 第 k 步的时候从第 k+1 个元素开始除以第 k 个元素
2. 第 k 步的时候从第 k+1 个元素开始减去第 k 个元素与上一行第 k+1 个元素的乘积 (按行)
3. 第 k 步的时候从第 k+1 个元素开始减去第 k 个元素与上一列第 k+1 个元素的乘积 (按列)

此外由于系统平台从 Windows 变成了 Linux 所以时间测算使用了 Linux 的工具——gettimeofday。

串行朴素

```
1 // 同前
2 void naive_lu(float **Matrix, int N)
3 {
4     for(int k = 0; k<N; k++)
```

```
    {
        for(int j = k+1; j<N; j++)
            Matrix[k][j] = Matrix[k][j] / Matrix[k][k];
        Matrix[k][k] = 1.0;
        for(int i = k+1; i<N; i++){
            for(int j = k+1; j<N; j++)
                Matrix[i][j] = Matrix[i][j] - Matrix[i][k] * Matrix[k][j];
            Matrix[i][k] = 0;
        }
    }
}
// 计时
gettimeofday(&start , NULL); // 精确到微秒
naive_lu(A_copy, N);
gettimeofday(&end, NULL);
timersub(&end, &start , &diff);
timecost= diff.tv_sec + (1.0 * diff.tv_usec)/1000;
```

(二) MPI 按块划分消去并行处理

按块划分按照 LU 的特性, 假设 m 个 MPI 节点则给每一个节点分配 N/m 行的数据，对于第 i 个节点，其分配的范围为 [i*(N-N%m)/m,i*(N-N%m)/m+(N-N%m)/m-1], 而最后一个节点分配 [i*(N-N%m)/m,N-1] 行的内容。此外初始化矩阵等工作由 0 号节点实现, 同时 0 号节点将各行分配发送给各节点, 每个节点接收前面节点发送的除法结果进行消去并对自己负责行除法后再将结果传递给后面的节点，最后再将结果传给 0 号节点，由 0 号节点进行汇总得到最终消去上三角矩阵。算法复杂度 $O(N^3/n)$。n 为节点个数。实现如下

block-mpi

```
//mpi thread 实际节点操作
void mpi_thread(int rank)
{
        char *proc_name = new char[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(proc_name, &name_len);
        if(rank==0)//0号节点
        {
        int m = MPI::COMM_WORLD.Get_size();   // 总mpi节点数
        int _n = (N - N%m) / m;

                // 创建矩阵并赋初值
                float **A = new float*[N];
                for(int i=0; i<N; i++)
                        A[i] = new float[N];
        matrix_initialize(A, N, 100);
        // 拷贝用以计算和验证结果正确性
        float **A_copy = new float*[N];
        for(int i=0; i<N; i++)
```

```
            A_copy[i] = new float[N];

    copy_matrix(A_copy, A, N);
    //朴素算法用以验证结果
    naive_lu(A_copy, N);

    //计时开始
    start_time = MPI_Wtime();

    for(int i = _n; i<N; i++)
    {
        // 发送第i行到第dest节点
        int dest = i / _n;
        if(m == dest)
            dest--;
        if(dest != 0)
            MPI_Send(A[i], N, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
    }
    // 除法运算并将结果发给后面的节点对剩下的行做消去
    for(int k=0; k < _n; k++)
    {
        for(int j=k+1; j<N; j++)
            A[k][j] = A[k][j] / A[k][k];
        A[k][k] = 1.0;
        for(int dest=rank+1; dest<m; dest++){
            MPI_Send(A[k], N, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
        }
        int rows_left = _n-1 - k;
        if(rows_left <= 0)
            break;
        matrix_elimination(A, _n, A[k], N, k, k+1);
    }

    for(int i=_n; i<N; i++)
    {
        // 确定行来源节点进行处理
        int src = i/_n;
        if(src == m)
            src--;
        MPI_Recv(A[i], N, MPI_FLOAT, src, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    end_time = MPI_Wtime();
            show_matrix(A, N);
    cout << "N = "<<N<<". Number of Process: " << m << endl;
    cout << "time is " << (end_time - start_time)*1000 <<"ms"<< endl;
    if(!is_same(A, A_copy, N))
            cout << "wrong!!" << endl;
```

```
68          else
69                  cout << "correct" << endl;
70
71      }
72      //其他节点
73      else
74      {
75          int m = MPI::COMM_WORLD.Get_size();
76          int _n = (N - N%m) / m;
77          int begin_row, end_row;  // 起始结束行
78          begin_row = rank * _n;
79          if(rank == m-1)
80              end_row = N-1;
81          else
82              end_row = rank * _n + _n-1;
83          int matrix_size = end_row - begin_row + 1;
84          float **mpi_line = new float*[matrix_size];//存行内容
85          for(int i=0; i<matrix_size; i++)
86              mpi_line[i] = new float[N];
87          // 接收数据
88          for(int i=0; i<matrix_size; i++)
89          {
90              MPI_Recv(mpi_line[i], N, MPI_FLOAT,
91              0/*从0接收*/, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
92          }
93          // 消去
94          float *row_k = new float[N];
95          for(int k=0; k<begin_row; k++)
96          {
97              int src = k/_n;
98              // 接收除法后结果
99              MPI_Recv(row_k, N, MPI_FLOAT, src,
100             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
101             elimination(mpi_line, matrix_size, row_k, N, k);
102         }
103
104         // 除法并发送结果
105         for(int k=begin_row; k <= end_row; k++)
106         {
107             int i = k - begin_row;
108             for(int j=k+1; j<N; j++)
109                 mpi_line[i][j] = mpi_line[i][j] / mpi_line[i][k];
110             mpi_line[i][k] = 1.0;
111             //发给后面节点
112             if(rank != m-1){
113                 for(int dest=rank+1; dest<m; dest++){
114                     MPI_Send(mpi_line[i], N,
115                     MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
```

```
                }
            }
            int rows_left = end_row - k;
            if(rows_left <= 0)
                break;
            matrix_elimination(mpi_line,
            matrix_size, mpi_line[i], N, k, i+1);

        }
        for(int i=0; i<matrix_size; i++){
            MPI_Send(mpi_line[i], N,
            MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
        }
    }
}
//按块划分完整过程
int main(int argc, char ** argv)
{
    int rank;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    mpi_thread(rank);

    MPI_Finalize();
    return 0;
}
```

(三) MPI 循环划分消去并行处理

循环划分是将子矩阵循环分配给不同节点进行计算并最终发送回 0 号节点由其前向整合得到结果。

loop-mpi

```
//与块划分仅在节点的处理上有所不同
void mpi_thread(int rank)
{
        char *proc_name = new char[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(proc_name, &name_len);
    int m = MPI::COMM_WORLD.Get_size();   // 总mpi节点数
    int q = N % m;
    int _n = (N - q) / m;
    int matrix_size;
    if(rank < q && q != 0)//划分给节点的依据
        matrix_size = _n+1;
```

```
    else
        matrix_size = _n;

    // 该节点负责计算的行号及其内容
    float **mpi_line = new float*[matrix_size];
    for(int i=0; i<matrix_size; i++)
        mpi_line[i] = new float[N];
    float *row_k = new float[N];
        if(rank==0)
        {
                //  0线程创建矩阵并赋初值
                float **A = new float*[N];
                for(int i=0; i<N; i++)
                        A[i] = new float[N];
                matrix_initialize(A, N, 100);

        float **A_copy = new float*[N];
        for(int i=0; i<N; i++)
                        A_copy[i] = new float[N];

        copy_matrix(A_copy, A, N);
        //朴素算法验证结果
        naive_lu(A_copy, N);

        //计时开始
        start_time = MPI_Wtime();

        int count = 0;//计数行
        for(int i = 0; i<N; i++)
        {
            int dest = i % m;
            if(dest != rank)
                MPI_Send(A[i], N, MPI_FLOAT,
                 dest, 0, MPI_COMM_WORLD);
            else{
                // 自己负责的行移至mpi_line
                for(int j = 0; j<N; j++)
                    mpi_line[count][j] = A[i][j];
                count ++;
            }
        }
        for(int i=0; i<matrix_size; i++)
        {
            if(i==0){//接收
                for(int k=0; k<rank; k++){
                    int src = k%m;
                    MPI_Recv(row_k, N, MPI_FLOAT,
                     src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
                    // 消去右下
                    matrix_elimination(mpi_line, matrix_size, row_k, N, k, i);
                }
            }
            else{
                for(int j=1; j<=m-1;j++){
                    int k = m *(i-1) + j + rank;
                    int src = k%m;
                    MPI_Recv(row_k, N, MPI_FLOAT,
                    src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    // 消去
                    matrix_elimination(mpi_line, matrix_size, row_k, N, k, i);
                }
            }
            // 对负责行除法
            int k = m *i + rank;
            for(int j=k+1; j<N; j++)
                mpi_line[i][j] = mpi_line[i][j] / mpi_line[i][k];
            mpi_line[i][k] = 1.0;
            matrix_elimination(mpi_line, matrix_size, mpi_line[i], N, k, i+1);

            //对处理完内容发送给对应后续节点
            int end_row = N-1;
            int real_row = i*m + rank;
            if((i == matrix_size-1) && (end_row-m+2 <=real_row) &&(real_row <= end_row))
            {
                for(int row=real_row+1; row<=end_row; row++){
                        int dest = row % m;
                    if(dest != rank)
                        MPI_Send(mpi_line[i], N,
                        MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
                }
            }
            else
            {
                for(int dest=0; dest<m; dest++)
                    if(dest != rank)
                        MPI_Send(mpi_line[i], N, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
            }

        }
        //接收其他节点计算完结果
        count = 0;
        for(int i = 0; i<N; i++)
        {
            int src = i % m;
            if(rank != src){
                MPI_Recv(A[i], N, MPI_FLOAT, src, 0,
```

```
109                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
110             }
111             else{
112                 for(int j = 0; j<N; j++)
113                     A[i][j] =  mpi_line[count][j];
114                 count ++;
115             }
116         }
117 
118         end_time = MPI_Wtime();
119                 show_matrix(A, N);
120         cout << "N = "<<N<<". Number of Process: " << m << endl;
121         cout << "time is " << (end_time - start_time)*1000 <<"ms"<< endl;
122         if(!is_same(A, A_copy, N))
123                 cout << "wrong!!" << endl;
124         else
125                 cout << "correct" << endl;
126 
127         }
128     else
129     {
130         // 从第0节点接收原始数据
131         for(int i=0; i<matrix_size; i++)
132         {
133             MPI_Recv(mpi_line[i], N, MPI_FLOAT,
134             0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
135         }
136         for(int i=0; i<matrix_size; i++)
137         {
138             if(i==0){
139                 for(int k=0; k<rank; k++){
140                     int src = k%m;
141                     MPI_Recv(row_k, N, MPI_FLOAT, src,
142                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
143                     // 消去
144                     matrix_elimination(mpi_line, matrix_size, row_k, N, k, i);
145                 }
146             }
147             else{
148                 for(int j=1; j<=m-1;j++){
149                     int k = m *(i-1) + j + rank;
150                     int src = k%m;
151                     MPI_Recv(row_k, N, MPI_FLOAT, src,
152                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
153                     // 消去
154                     matrix_elimination(mpi_line, matrix_size, row_k, N, k, i);
155                 }
156             }
```

```
157
158             // 除法
159             int k = m *i + rank;
160             for(int j=k+1; j<N; j++)
161                 mpi_line[i][j] = mpi_line[i][j] / mpi_line[i][k];
162             mpi_line[i][k] = 1.0;
163             // 消去
164             matrix_elimination(mpi_line, matrix_size, mpi_line[i], N, k, i+1);
165             int end_row = N-1;
166             int real_row = i*m + rank;
167             if((i == matrix_size-1) && (end_row-m+2 <=real_row) &&(real_row <= end_row))
168             {
169                 for(int row=real_row+1; row<=end_row; row++){
170                     int dest = row % m;
171                 if(dest!=rank)
172                     MPI_Send(mpi_line[i], N,
173                     MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
174                 }
175             }
176             else
177             {
178                 for(int dest=0; dest<m; dest++)
179                     if(dest != rank)
180                     MPI_Send(mpi_line[i], N,
181                     MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
182             }
183
184         }
185         for(int i = 0; i<matrix_size; i++)
186         {
187             MPI_Send(mpi_line[i], N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
188         }
189     }
190 }
```

（四） OpenMP+MPI

OpenMP 与 MPI 的结合考虑的比较简单，在使用块划分的前提下仅对消去部分的循环进行 OpenMP 优化。同时为了成功调用 OpenMP 库，在编译的时候要加上-fopenmp 选项。

omp-mpi

```
1 //在各节点接收其他行除法结果后调用进行消去运算
2 void matrix_elimination(float **matrix, int n_row,
3 float *row_k, int dimension, int k, int row_No/**/)
4 {
5     #pragma omp parallel for num_threads(Threads)
6     for(int i=row_No; i<n_row; i++)
7     {
```

```
        for(int j=k+1; j<dimension; j++)
            matrix[i][j] = matrix[i][j] - matrix[i][k]*row_k[j];
        matrix[i][k] = 0.0;
    }
}
```

(五) SSE/AVX+MPI

SSE/AVX 的优化也主要体现在消去循环的部分上。实现方法与前面实验一致。此外编译时要对指令集进行声明-march=corei7-avx。复杂度 $O(N^3/n)$

sse/avx-mpi

```
//在各节点接收其他行除法结果后调用进行消去运算
//SSE
void matrix_elimination(float **matrix,
int n_row, float *row_k, int dimension, int k, int row_No/**/)
{
    int part_2 = (dimension-k-1)%4;
    for(int i=row_No; i<n_row; i++)
    {
        __m128 A_i_k = _mm_set_ps1(matrix[i][k]);
        for(int j=dimension-4; j>k; j-=4){
            __m128 A_k_j = _mm_loadu_ps(&row_k[j]);
            __m128 t = _mm_mul_ps(A_k_j, A_i_k);
            __m128 A_i_j = _mm_loadu_ps(matrix[i]+j);
            A_i_j = _mm_sub_ps(A_i_j, t);
             _mm_storeu_ps(matrix[i]+j, A_i_j);
        }
        for(int j = k+1; j<k+1+part_2; j++)
            matrix[i][j] = matrix[i][j] - matrix[i][k]*row_k[j];
        matrix[i][k] = 0.0;
    }
}
//AVX
void matrix_elimination(float **matrix,
int n_row, float *row_k, int dimension, int k, int row_No/**/)
{
    int part_2 = (dimension-k-1)%8;
    for(int i=row_No; i<n_row; i++)
    {
        __m256 A_i_k = _mm256_set1_ps(matrix[i][k]);
        for(int j=dimension-8; j>k; j-=8){
            __m256 A_k_j = _mm256_loadu_ps(&row_k[j]);
            __m256 t = _mm256_mul_ps(A_k_j, A_i_k);
            __m256 A_i_j = _mm256_loadu_ps(matrix[i]+j);
            A_i_j = _mm256_sub_ps(A_i_j, t);
             _mm256_storeu_ps(matrix[i]+j, A_i_j);
        }
```

```
        for(int j = k+1; j<k+1+part_2; j++)
            matrix[i][j] = matrix[i][j] - matrix[i][k]*row_k[j];
        matrix[i][k] = 0.0;
    }
}
```

（六） 流水线 +MPI

基于块划分的流水线算法和普通的块划分的区别在于一个节点负责行的除法运算完成之后，并不是将除法结果发送给下面的每一个节点而是下一个节点，同样当一个节点接收到上面节点发送过来的除法结果时，其对自己所负责的行进行消去操作，之后又将刚刚收到的除法结果发给下一节点。

pipeline-mpi

```
// 只在接收前面传递过来的除法结果处理时有区别
for(int k=0; k<begin_row; k++)
{
    int src = rank - 1;
    MPI_Recv(row_k, N, MPI_FLOAT, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    matrix_elimination(mpi_line, matrix_size, row_k, N, k, 0);
    // 流水线发送给下一个节点
    int dest = rank+1;
    if(dest <= m-1){
        MPI_Send(row_k, N, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
    }
}
```

三、 高精度计时分析算法复杂性

由于实验中可能出现种种误差，为尽量保证实验的公平性，作以下假设:

- Cache 对所有算法都是公平的，均为按行获取按行划分，仅有一例为按列划分以验证不同划分方式，每次换算法之前通过 reset 函数令初始矩阵一致，并验证结果正确性，在最终结果一致 (以串行朴素 LU 的计算结果为基准) 的情况下才认为计时是有效的
- 由于 N 较小的时候测出的计算时间也较小，误差较大，故采取多次测量取均值的方法确定较合理的性能测试结果，同时保证几种算法重复次数一致，减少误差
- 由于实验平台变为了 Linux 且为了公平，全部计时测试工具使用 MPI 计时。

本次实验对 4、8、16 个 MPI 节点进行了测试得到以下结果 根据全部测试数据绘制运行时间、加速比对比图、效率对比图，运行时间图可以看出各种方法对于朴素方法都节约了一半左右的运行时间，其中 AVX 在较大数据表现最佳，按循环划分效果不如预期,N<200 时，朴素算法更快一些，原因同前，其他方式或多或少有额外的创建节点分配开销等，这些开销大于加速的比重。

加速比对比图则可以看出各个算法在加速上的趋势基本一致，都在 N=1000 之后趋于平缓，最大加速比接近 8，但循环划分加速比却达不到 2。可见循环划分的方式在数据分配上有局限性。

效率对比图，基本上 SSE/AVX 的效率在 100% 以上，效果斐然。

Algo\ N	16	64	128	256	512	1024	2048
naive	0.0009	0.058651	0.476599	3.67212	28.5933	228.429	1831.17
block	1.1	1.65701	1.99389	3.88336	15.6291	106.502	719.279
loop	1.7	3.95012	5.005	12.4178	35.1026	179.31	1245.09
omp	3.5	8.99	5.565	12.1753	23.5963	120.23	780.01
sse	0.8	2.3942	3.105	3.38054	9.306	48.2638	356.248
avx	1.7	2.404	2.136	3.466	8.0376	39.5086	279.718
pipeline	0.9	1.67871	2.4268	5.106	18.5144	120.278	864.183

表 1: 性能测试结果 (4 节点)(单位:ms)

整体来看 AVX 优化下 MPI 并行化效果最佳，这与预期相吻合。而流水线由于传递判断较多所以性能测试在中等偏下的位置。

再结合对比不同节点的测试结果，可以发现节点数目的增加与加速比的上升不完全成比例，也是和前面随着 N 增大一样的趋势，先增大的较多而后趋于平缓，且本身就相对较快的 SSE 和 AVX 并行化，在节点数目增加以后提升的加速比相对更多。

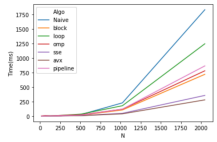

图 1: 四节点运行时间图

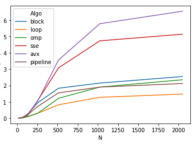

图 2: 四节点加速比

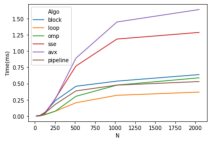

图 3: 四节点效率图

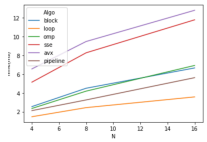

图 4: 节点加速比对比图

四、 总结

本次实验使用 MPI 对 LU 进行并行优化，探究了不同划分方式、不同并行优化方式对最终算法的性能影响。经过多次不同并行化方法的探索，仅使用 SIMD 的 SSE/AVX 优化效果反而是最好的，OpenMP 在大规模数据下的性能反而有所下滑可能是由于但由于其不够灵活，cache 缺失导致了性能的下降。但总的来说相较于平凡算法是很有效的，可以将 OpenMP 与 SIMD 优化相结合得到相对不错的结果。

参考文献

[1] 高斯消元法 [EB/OL].https://zh.wikipedia.org/wiki/高斯消去法

[2] https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE3,AVX&expand=3931

[3] https://www.openmp.org

[4] 课件 algo1-2,openmp4-12,mpi2-18