南 开 大 学

NanKai University

MPI 编程实验——以高斯消去为例

姓名：文静静

学号：1811507

年级：2018级

专业：计算机科学与技术

2021年6月8日

摘要

本文以高斯消去法为例，初始化生成矩阵元素值，使用 MPI 编程针对消去部分的两重循环按照不同任务划分方式（块划分和块循环划分）分别设计并实现 MPI 算法。并考虑将其与 OpenMp 以及 SIMD 算法结合，对各算法性能进行分析。

关键字：高斯消去　MPI　OpenMp　SIMD

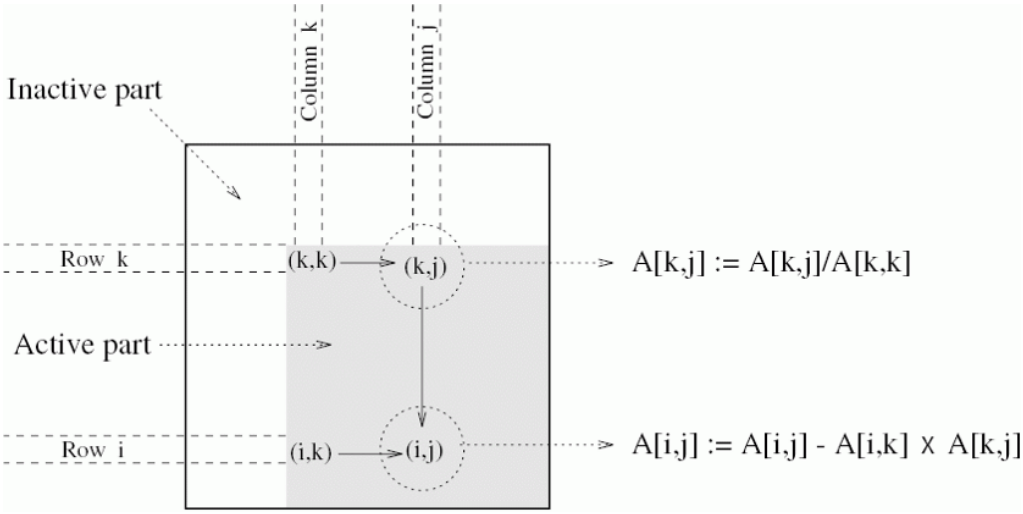
目录

1 绪论

本文以高斯消去法为例，初始化生成矩阵元素值，使用 MPI 编程针对消去部分的两重循环按照不同任务划分方式（块划分和块循环划分）分别设计并实现 MPI 算法。并考虑将其与 OpenMp 以及 SIMD 算法结合，对各算法性能进行分析。

2 实验选题

高斯消去的计算模式如图所示，在第 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。



高斯消去法（LU 分解）实现 MPI 算法，具体要求如下：

1. 算法方面的探讨：不同的任务分配策略 (块划分、循环块划分等)、流水线算法等，相应的复杂性分析 (并行时间、通信开销、加速比、效率等)。

2. 与 OpenMP 以及 SIMD(SSE/AVX) 算法结合。

3. 观测各算法运行时间的变化，对结果进行性能分析。借助 Vtune profiling 等工具分析算法过程中的同步开销和空闲等待等。

3 算法设计与编程

3.1 初始化

初始化系数矩阵：在使用高斯消去法的时候，有除法操作，为了避免出现除数为 inf 或者 nan 的情况，将系数矩阵初始为对角线元素全部为 1 ，其余元素全部为 2 的矩阵，以便运算和验证。（N为系数矩阵规模大小）

```
// 初始系数矩阵，为了方便将其设置成上三角矩阵
B = new float *[N];
for (int i = 0; i < N; i++)
{
    B[i] = new float[N];
}
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
        if (i == j)
            B[i][j] = 1;
        else
            B[i][j] = 2;
}
```

串行的高斯消去法代码如下：Linux 下使用 gettimeofday 计时。

```
B = new float *[N];
for (int i = 0; i < N; i++)
{
    B[i] = new float[N];
}
timeval tv_start, tv_end;
gettimeofday(&tv_start, NULL);
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
        if (i == j)
            B[i][j] = 1;
        else
            B[i][j] = 2;
```

```
}
for (int k = 0; k < N; k++)
{
    for (int j = k + 1; j < N; j++)
    {
        B[k][j] = B[k][j] / B[k][k];
    }
    B[k][k] = 1.0;
    for (int i = k + 1; i < N; i++)
    {
        for (int j = k + 1; j < N; j++)
        {
            B[i][j] = B[i][j] - B[i][k] * B[k][j];
        }
        B[i][k] = 0;
    }
}
gettimeofday(&tv_end, NULL);
cout << "Plain running time: " << 1000 * (tv_end.tv_sec - tv_start.
   tv_sec) + (tv_end.tv_usec - tv_start.tv_usec) / 1000 << "ms" << endl;
```

通过下列代码测试，如上初始的系数矩阵在高斯消去之后确实不会出现 inf 和 nan 的情况：

```
for(int i = 0; i < N; i++){
    for(int j = 0; j < N; j++){
        if(isinf(B[i][j]) || isnan(B[i][j])){
            cout << "error" << endl;
        }
    }
}
```

通过如下测试函数，确保每种算法的结果与平凡算法的误差不超过1e-4。

```
void comp(float **A, float **B)
{
    for (int i = 0; i < N; i++)
    {
```

```
        for (int j = i; j < N; j++)
        {
            if (fabs(A[i][j] - B[i][j]) > 1e-4)
            {
                cout << "error > 1e-4:";
                cout << fabs(A[i][j] - B[i][j]) << endl;
                return;
            }
        }
    }
    cout << "correct!" << endl;
}
```

3.2 按块划分

以 MPI 按一维块划分消去并行处理为例，假设 size 个 MPI 进程, 则给每一个进程分配 N /size 行的数据。rank 表示当前进程号，分配至各进程的子矩阵大小为 n * N，n = N / size，subA 用于存放子矩阵。初始化矩阵的工作由 0 号节点实现。

```
// 初始化
MPI_Init(&argc, &argv);
// rank 当前进程, size 进程数量
int rank, size;
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
double start_time = 0;
double end_time = 0;
float **A, **B;
// 0 号节点初始化矩阵
if (rank == 0)
{
    cout << "N = " << N << " Number of Process: " << size << endl;
    A = new float *[N];
    for (int i = 0; i < N; i++)
        A[i] = new float[N];
```

```
    start_time = MPI_Wtime();
    //初始化矩阵
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (i == j)
                A[i][j] = 1;
            else
                A[i][j] = 2;
        }
    }
}
else
{
    A = new float *[1];
    A[0] = new float[1];
}
```

0 号节点按照顺序将各行分配发送给各节点。

```
// 分配至各进程的子矩阵大小为n * N
int n = N / size;
float **subA = new float *[n];
for (int i = 0; i < n; i++)
    subA[i] = new float[N];
// 0 号节点划分
if (rank == 0)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < N; j++)
            subA[i][j] = A[i][j];
    for (int i = n; i < N; i++)
    {
        //块划分
        MPI_Send(&A[i][0], N, MPI_FLOAT, i / n, i % n, MPI_COMM_WORLD);
    }
    // cout << "subA:" << endl;
    // for (int i = 0; i < n; i++)
```

```
    // {
    //     for (int j = 0; j < N; j++)
    //         cout << subA[i][j] << " ";
    //     cout << endl;
    // }
}
else
{
    for (int i = 0; i < n; i++)
        MPI_Recv(&subA[i][0], N, MPI_FLOAT, 0, i, MPI_COMM_WORLD, &
            status);
}
```

外层一共有 N 次循环，当第 i 行作为首行时，需要由负责这一行的进程，即 i / n 号进程，先进行本行的除法，再将计算结果通过 MPI_Bcast 广播给其他进程，接收时也需要调用 MPI_Bcast，并检查 root 参数和自身 ID 号是否相同，若不同则接收。之后所有进程一起开始右下角矩阵的消去计算。

```
// 首行元素
float *line = new float[N];
for (int i = 0; i < N; i++)
{
    if (rank == i / n)
    {
        int row = i % n;
        for (int k = i + 1; k < N; k++)
        {
            subA[row][k] /= subA[row][i];
            line[k] = subA[row][k];
        }
        subA[row][i] = 1;
        line[i] = 1;
        // cout << "line "<< row <<" to be bcasted: " << endl;
        // for (int t = 0; t < N; t++)
        //     cout << line[t] << " ";
        // cout << endl;
    }
    MPI_Bcast(line, N, MPI_FLOAT, i / n, MPI_COMM_WORLD);
```

```
    for (int j = i + 1; j < N; j++)
    {
        if (rank * n + n < j)
            continue;
        // 右下角消去计算
        if (rank * n <= j && rank * n + n > j)
        {
            int row = j % n;
            for (int k = i + 1; k < N; k++)
                subA[row][k] = subA[row][k] - subA[row][i] * line[k];
            subA[row][i] = 0;
        }
    }
}
```

每个节点最后将结果传给 0 号节点，由 0 号节点进行汇总得到最终结果上三角矩阵。

```
// rank=0 的进程收集计算结果
if (rank == 0)
{
    for (int i = 1; i < size; i++)
    {
        for (int j = 0; j < n; j++)
        {
            MPI_Recv(&A[i * n + j][0], N, MPI_FLOAT, i, j,
                MPI_COMM_WORLD, &status);
        }
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = subA[i][j];
    end_time = MPI_Wtime();
    cout << "Running Time: " << (end_time - start_time) * 1000 << "ms"
        << endl;

    // 检验结果
    B = new float *[N];
    for (int i = 0; i < N; i++)
```

```
20      {
21          B[i] = new float[N];
22      }
23      timeval tv_start, tv_end;
24      gettimeofday(&tv_start, NULL);
25      for (int i = 0; i < N; i++)
26      {
27          for (int j = 0; j < N; j++)
28              if (i == j)
29                  B[i][j] = 1;
30              else
31                  B[i][j] = 2;
32      }
33      for (int k = 0; k < N; k++)
34      {
35          for (int j = k + 1; j < N; j++)
36          {
37              B[k][j] = B[k][j] / B[k][k];
38          }
39          B[k][k] = 1.0;
40          for (int i = k + 1; i < N; i++)
41          {
42              for (int j = k + 1; j < N; j++)
43              {
44                  B[i][j] = B[i][j] - B[i][k] * B[k][j];
45              }
46              B[i][k] = 0;
47          }
48      }
49      gettimeofday(&tv_end, NULL);
50      cout << "Plain running time: " << 1000 * (tv_end.tv_sec - tv_start.
            tv_sec) + (tv_end.tv_usec - tv_start.tv_usec) / 1000 << "ms" <<
            endl;
51      comp(A, B);
52  }
53  else
54  {
55      for (int i = 0; i < n; i++)
```

```
56          MPI_Send(&subA[i][0], N, MPI_FLOAT, 0, i, MPI_COMM_WORLD);
57  }
```

3.3 循环划分

使用一维块划分时，到消元部分前面的处理数据较多，后面处理的数据较少，因此会出现负载不均衡的情况，为了能让每个核处理的数据量能大致相似，可以使用循环块划分，比如一共4个进程，N = 8，那么 1 号进程负责第 1 行和第 5 行，2 号进程负责第 2 和第 6 行，等等。初始化和之前还是一样的，当 0 号节点给其他节点分配子矩阵时，跟块划分有所不同。

```
1   // 分配至各进程的子矩阵大小为n * N
2   int n = N / size;
3   float **subA = new float *[n];
4   for (int i = 0; i < n; i++)
5       subA[i] = new float[N];
6   // 0 号节点
7   if (rank == 0)
8   {
9       for (int i = 0; i < n; i++)
10          for (int j = 0; j < N; j++)
11              subA[i][j] = A[i * size][j];
12      for (int i = 0; i < N; i++)
13      {
14          // 循环块划分
15          if (i % size != 0)
16          // 对于每一行发送到 dest = i % size, 连续的行 tag = i / size + 1
17          // size 的整数倍的行由 rank=0 计算
18          MPI_Send(&A[i][0], N, MPI_FLOAT, i % size, i / size + 1,
                MPI_COMM_WORLD);
19      }
20      // cout << "subA:" << endl;
21      // for (int i = 0; i < n; i++)
22      // {
23      //     for (int j = 0; j < N; j++)
24      //         cout << subA[i][j] << " ";
25      //     cout << endl;
```

```
    // }
}
else
{
    // 0 号进程接收
    for (int i = 0; i < n; i++)
        MPI_Recv(&subA[i][0], N, MPI_FLOAT, 0, i + 1, MPI_COMM_WORLD, &
            status);
}
```

计算：

```
// 首行元素
float *line = new float[N];
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < size; j++)
    {
        // 进行到第个块的第行ij
        int row = i * size + j;
        if (rank == j)
        {
            // 先做除法运算并将结果广播给后面的节点，以便后面节点进行消去
            for (int k = row + 1; k < N; k++)
            {
                subA[i][k] = subA[i][k] / subA[i][row];
                line[k] = subA[i][k];
            }
            subA[i][row] = 1;
            line[row] = 1;
            // cout << "line "<< row <<" to be bcasted: " << endl;
            // for (int t = 0; t < N; t++)
            //     cout << line[t] << " ";
            // cout << endl;
        }
        MPI_Bcast(line, N, MPI_FLOAT, j, MPI_COMM_WORLD);
        // 消去计算右下角
        if (rank <= j)
        {
```

```
            for (int k = i + 1; k < n; k++)
            {
                for (int w = row + 1; w < N; w++)
                    subA[k][w] = subA[k][w] - line[w] * subA[k][row];
                subA[k][row] = 0;
            }
        }
        if (rank > j)
        {
            for (int k = i; k < n; k++)
            {
                for (int w = row + 1; w < N; w++)
                    subA[k][w] = subA[k][w] - line[w] * subA[k][row];
                subA[k][row] = 0;
            }
        }
    }
}
```

最后 0 号节点处理结果，同块划分一致。

3.4 结合 SIMD

初始化、子矩阵分配与计算结果收集的过程与上面相同。在分配完子矩阵之后，开始计算时，使用 __m128 类型或者 __m256 类型的变量进行计算。

使用 SSE：

```
// 首行元素
float *line = new float[N];
__m128 t1, t2, t3, t4;
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < size; j++)
    {
        // 进行到第个块的第行ij
        int row = i * size + j;
        if (rank == j)
        {
```

```
            // 先做除法运算并将结果广播给后面的节点，以便后面节点进行消去
            for (int k = row + 1; k < N; k++)
            {
                subA[i][k] = subA[i][k] / subA[i][row];
                line[k] = subA[i][k];
            }
            subA[i][row] = 1;
            line[row] = 1;
            // cout << "line "<< row <<" to be bcasted: " << endl;
            // for (int t = 0; t < N; t++)
            //     cout << line[t] << " ";
            // cout << endl;
        }
        MPI_Bcast(line, N, MPI_FLOAT, j, MPI_COMM_WORLD);
        // 消去计算右下角
        int remain = (N - row - 1) % 4;
        int kk = i;
        if (rank <= j)
        {
            kk = i + 1;
        }
        for (int k = kk; k < n; k++)
        {
            t1 = _mm_set1_ps(subA[k][row]);
            for (int w = row + 1; w < row + 1 + remain; w++)
                subA[k][w] = subA[k][w] - line[w] * subA[k][row];
            for(int w = row + 1 + remain; w < N; w += 4)
            {
                t2 = _mm_loadu_ps(&line[w]);
                t3 = _mm_loadu_ps(&subA[k][w]);
                t4 = _mm_mul_ps(t1, t2);
                t4 = _mm_sub_ps(t3, t4);
                _mm_storeu_ps(&subA[k][w], t4);
            }
            subA[k][row] = 0;
        }
    }
}
```

使用 AVX：

```
// 首行元素
float *line = new float[N];
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < size; j++)
    {
        // 进行到第个块的第行ij
        int row = i * size + j;
        if (rank == j)
        {
            // 先做除法运算并将结果广播给后面的节点，以便后面节点进行消去
            for (int k = row + 1; k < N; k++)
            {
                subA[i][k] = subA[i][k] / subA[i][row];
                line[k] = subA[i][k];
            }
            subA[i][row] = 1;
            line[row] = 1;
            // cout << "line "<< row <<" to be bcasted: " << endl;
            // for (int t = 0; t < N; t++)
            //     cout << line[t] << " ";
            // cout << endl;
        }
        MPI_Bcast(line, N, MPI_FLOAT, j, MPI_COMM_WORLD);
        // 消去计算右下角
        int remain = (N - row - 1) % 8;
        int kk = i;
        if (rank <= j)
        {
            kk = i + 1;
        }
        for (int k = kk; k < n; k++)
        {
            __m256 t1 = _mm256_set1_ps(subA[k][row]);
            for (int w = row + 1; w < row + 1 + remain; w++)
                subA[k][w] = subA[k][w] - line[w] * subA[k][row];
            for (int w = row + 1 + remain; w < N; w += 8)
```

```
38              {
39                  __m256 t2 = _mm256_loadu_ps(&line[w]);
40                  __m256 t3 = _mm256_loadu_ps(&subA[k][w]);
41                  __m256 t4 = _mm256_mul_ps(t1, t2);
42                  t4 = _mm256_sub_ps(t3, t4);
43                  _mm256_storeu_ps(&subA[k][w], t4);
44              }
45              subA[k][row] = 0;
46          }
47      }
48  }
```

3.5 结合 OpenMp

初始化、子矩阵分配与计算结果收集的过程与上面相同。为了避免 omp 频繁地创建销毁线程，在外层使用 omp parallel，在内层 for 循环使用 omp for，需要注意区分共享变量和私有变量。

```
1  float *line = new float[N];
2  for (int i = 0; i < n; i++)
3  {
4      for (int j = 0; j < size; j++)
5      {
6          // 进行到第个块的第行ij
7          int row = i * size + j;
8          if (rank == j)
9          {
10             // 先做除法运算并将结果广播给后面的节点，以便后面节点进行消去
11             for (int k = row + 1; k < N; k++)
12             {
13                 subA[i][k] = subA[i][k] / subA[i][row];
14                 line[k] = subA[i][k];
15             }
16             subA[i][row] = 1;
17             line[row] = 1;
18             // cout << "line "<< row <<" to be bcasted: " << endl;
19             // for (int t = 0; t < N; t++)
```

```
            //  cout << line[t] << " ";
            // cout << endl;
        }
        MPI_Bcast(line, N, MPI_FLOAT, j, MPI_COMM_WORLD);
        // 消去计算右下角
        int kk = i;
        if (rank <= j)
        {
            kk = i + 1;
        }
        #pragma omp parallel for num_threads(provided)
        for (int k = kk; k < n; k++)
        {
            for (int w = row + 1; w < N; w++)
            subA[k][w] = subA[k][w] - line[w] * subA[k][row];
            subA[k][row] = 0;
        }
    }
}
```

OpenMp 结合 SIMD：

```
// 首行元素
float *line = new float[N];
__m128 t1, t2, t3, t4;
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < size; j++)
    {
        // 进行到第个块的第行ij
        int row = i * size + j;
        if (rank == j)
        {
            // 先做除法运算并将结果广播给后面的节点，以便后面节点进行消去
            for (int k = row + 1; k < N; k++)
            {
                subA[i][k] = subA[i][k] / subA[i][row];
                line[k] = subA[i][k];
            }
```

```
18              subA[i][row] = 1;
19              line[row] = 1;
20              // cout << "line "<< row <<" to be bcasted: " << endl;
21              // for (int t = 0; t < N; t++)
22              //  cout << line[t] << " ";
23              // cout << endl;
24          }
25          MPI_Bcast(line, N, MPI_FLOAT, j, MPI_COMM_WORLD);
26          // 消去计算右下角
27          int remain = (N - row - 1) % 4;
28          int kk = i;
29          if (rank <= j)
30          {
31              kk = i + 1;
32          }
33          #pragma omp parallel for num_threads(provided)
34          for (int k = kk; k < n; k++)
35          {
36              t1 = _mm_set1_ps(subA[k][row]);
37              for (int w = row + 1; w < row + 1 + remain; w++)
38              subA[k][w] = subA[k][w] - line[w] * subA[k][row];
39              for (int w = row + 1 + remain; w < N; w += 4)
40              {
41                  t2 = _mm_loadu_ps(&line[w]);
42                  t3 = _mm_loadu_ps(&subA[k][w]);
43                  t4 = _mm_mul_ps(t1, t2);
44                  t4 = _mm_sub_ps(t3, t4);
45                  _mm_storeu_ps(&subA[k][w], t4);
46              }
47              subA[k][row] = 0;
48          }
49      }
50  }
```

4 编译运行

4.1 运行环境

MPI 编程实验在基于金山云的虚拟机集群上运行，虚拟机集群包括 1 个 master 节点和 32 个计算节点，每个计算节点 2 核，共 64 核可供计算。使用 lscpu 命令可以查看服务器的具体信息如下：

```
[sl8ll507@master ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel Xeon Processor (Cascadelake)
Stepping:              6
CPU MHz:               2593.958
BogoMIPS:              5187.91
Hypervisor vendor:     KVM
Virtualization type:   full
Lld cache:             32K
Lli cache:             32K
L2 cache:              1024K
L3 cache:              25344K
NUMA node0 CPU(s):     0,1
```

图 1: CPU 信息

4.2 编译命令

使用命令行 **mpic++ mpi.cpp** 将源码编译成可执行文件

如果是结合 AVX，使用命令行 **mpic++ mpi.cpp -march=corei7-avx**

如果是结合 OpenMp，使用命令行 **mpic++ mpi.cpp -fopenmp**

然后使用命令行 **pssh -h nodes mkdir -p /test 1>&2** 在分配节点上创建执行路径

使用命令行 **pscp.pssh -h nodes a.out /test 1>&2** 将管理节点上的可执行文件分发到各计算节点

使用命令行 **mpiexec -np 4 -f machinefile /home/s1811507/test/a.out** 在4个计算节点上运行对应 mpi 程序

5 结果分析

5.1 块划分与循环划分

得到块划分与循环划分的运行时间如下表：

规模	plain	block	loop_block
128	0.96	3.81517	9.21273
256	5.7608	6.97041	18.4205
512	42.423	25.104	42.8824
1024	340.918	153.196	134.97
2048	2742.09	1086.47	664.772
4096	23196.34	8467.48	4360.98

可得到如下加速比对比：

规模	block	loop_block
128	0.2516	0.1042
256	0.8265	0.3127
512	1.69	0.989
1024	2.225	2.526
2048	2.524	4.1249
4096	2.7395	5.319

由此可以得到加速比的折线图：

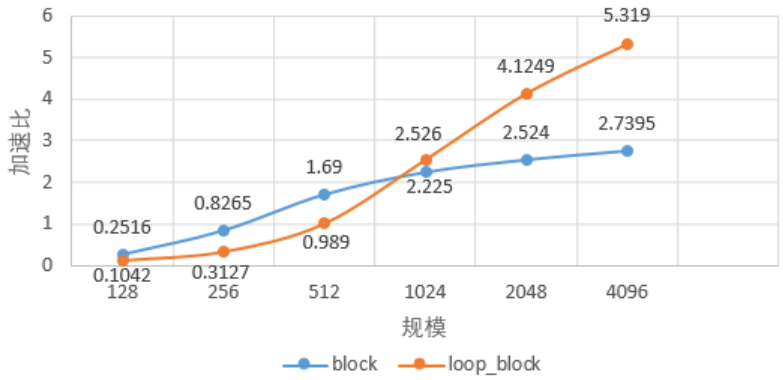


图 2: 块划分与循环划分的加速比折线图

我们可以发现，数据充足时，mpi 的加速效果很好，当规模等于 1024 的时候，块划分与循环划分的加速比都达到了 2 以上，当数据小于 512 的时候，循环划分相对于块划分来说并没有优势，这可能是因为数据不够多，尽管是按序划分，每个线程之间的空闲时间相差不多，并没有影响到性能，而且循环划分在处理子矩阵的时候有更多的判断开销。但是当数据大于 1024 时，块划分的加速效果相比于循环块划分差了很多，这是因为消元部分前面的处理数据较多（右上三角的部分），后面处理的数据较少（右下三角的部分），循环块划分能让每个核处理的数据量能大致相似，而采用块划分的话，首行元素越往下，闲置的节点数会越多，闲置的节点不进行计算反而还要保持同步通信，效率低下，因此加速比效果并不好。

5.2 节点数量

块划分与循环划分分别在节点数为 4 和 8 的时候的加速比如下：

规模	block+4	block+8	loop+4	loop+8
128	0. 2516	0. 1458	0. 1042	0. 0849
256	0. 8265	0. 5985	0. 3127	0. 261
512	1. 69	2. 082	0. 989	0. 982
1024	2. 225	3. 489	2. 526	3. 001
2048	2. 524	4. 447	4. 1249	6. 143
4096	2. 7395	4. 301	5. 319	6. 858

由此得到折线图如下：

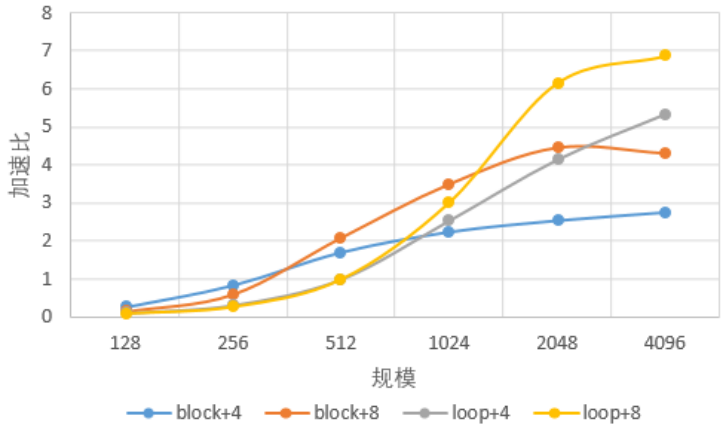


图 3: 不同节点数的加速比折线图

可以发现数据较大的时候 8 个节点的加速比显然大于 4 个节点的加速比，但是当数据较小的时候，比如 N=128、N=256 的时候，8 个节点的加速比并不如 4 个节点，是因为数据量不够而节点数又太多，每个节点的计算量很小，却花了很多时间进行同步通信，所以加速比反而下降了。

5.3 结合 SIMD

结合 SSE 与 AVX 的加速比如下：

规模	mpi	mpi+sse	mpi+avx
128	0. 1042	0. 108	0. 109
256	0. 3127	0. 3312	0. 3263
512	0. 989	1. 13	1. 053
1024	2. 526	3. 6001	3. 577
2048	4. 1249	7. 6386	7. 41356
4096	5. 319	12. 133	12. 3841

由此得到折线图如下：

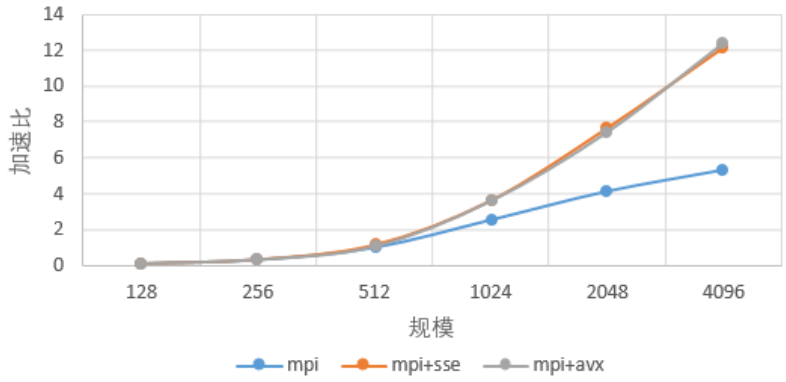


图 4: 结合 SIMD 的加速比折线图

结合了 SIMD 之后，加速比比普通 mpi 大了一倍，向量化对时间性能的改善十分明显！AVX 与 SSE 的效果差不多，可能是数据量还不够大，所以还没有显现 AVX 的优势。

5.4 结合 OpenMp

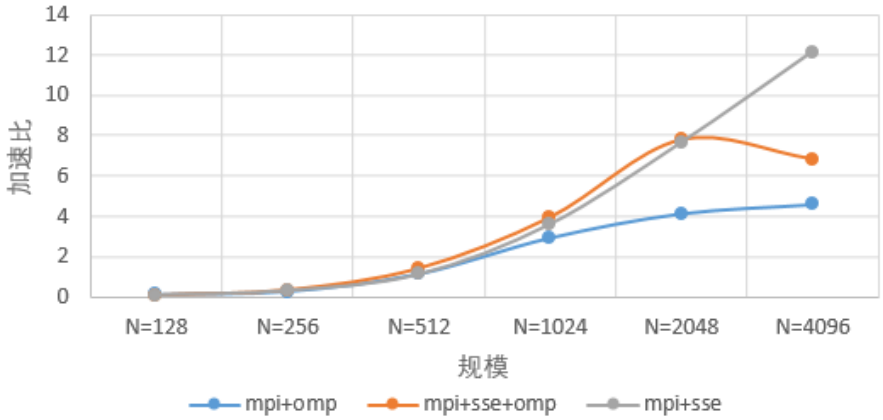
结合 OpenMP，设置每个节点有两个线程，使用的时间如下：

算法	plain	mpi+omp	mpi+sse+omp
N=128	0. 96	8. 01229	9. 0608
N=256	5. 7608	21. 2448	16. 1862
N=512	42. 423	37. 1349	29. 8769
N=1024	340. 918	115. 891	86. 4127
N=2048	2742. 09	660. 291	351. 819
N=4096	23196. 34	5027. 46	3385. 31

得到加速比如下：

算法	plain	mpi+omp	mpi+sse+omp
N=128	0. 96	0. 11982	0. 10595
N=256	5. 7608	0. 2712	0. 356
N=512	42. 423	1. 1424	1. 4199
N=1024	340. 918	2. 9417	3. 9452
N=2048	2742. 09	4. 1529	7. 794
N=4096	23196. 34	4. 6139	6. 852

结合 SSE 的加速比得到折线图如下：



结合了 OpenMp 和 SSE 的加速比比只结合 SSE 的加速比反而要小，因此我猜测，是因为数据量不足导致的各个线程不能被充分利用，反而多个线程的开销影响时间性能。

6 总结

这次实验使用 MPI 对高斯消去法进行并行优化，探究了不同划分方法、节点数量、结合 SIMD、结合 OpenMp 的算法效率，发现数据量很大的时候，结合 AVX 的效率更高，但是由于数据量的限制，并没有完全探究成功 OpenMp 对效率的提升，OpenMp 与 SSE 的结合反而不如 SSE 的效率。