



南開大學
Nankai University

计算机学院
并行程序设计第 7 次作业

高斯消去法的 MPI 并行化

姓名：丁屹
学号：2013280
专业：计算机科学与技术

2022 年 6 月 23 日

目录

1 问题描述	2
2 算法设计	3
2.1 测试用例的确定	3
2.2 实验环境和相关配置	3
2.3 默认平凡算法	3
2.4 MPI 并行化算法	4
2.5 MPI 并行化算法结合 OpenMP 多线程库	4
2.6 MPI 并行化算法结合 NEON 指令优化	5
3 算法分析	6
3.1 时间复杂度分析	6
3.2 运行时间分析	7

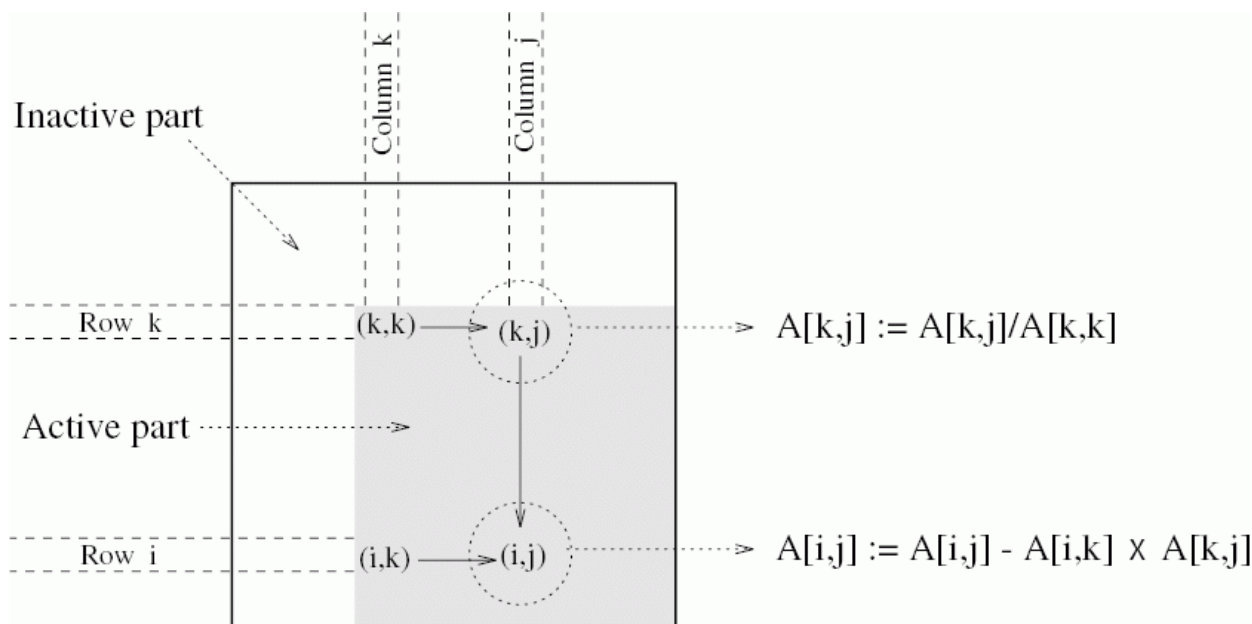


图 1.1: 高斯消去法示意图

1 问题描述

高斯消去的计算模式如图 1.1 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，串行算法如下面伪代码所示。

Algorithm 1 普通高斯消元算法伪代码

```

1: function LU
2:   for  $k := 0$  to  $n$  do
3:     for  $j := k + 1$  to  $n$  do
4:        $A[k, j] := A[k, j]/A[k, k]$ 
5:     end for
6:      $A[k, k] := 1.0$ 
7:     for  $i := k + 1$  to  $n$  do
8:       for  $j := k + 1$  to  $n$  do
9:          $A[i, j] := A[i, j] - A[i, k] * A[k, j]$ 
10:      end for
11:       $A[i, k] := 0$ 
12:    end for
13:  end for
14: end function

```

观察高斯消去算法，注意到伪代码第 4, 5 行第一个内嵌循环中的 $A[k, j] := A[k, j]/A[k, k]$ 以及伪代码第 8 9 10 行双层 *for* 循环中的 $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$ 都是可以进行向量化的循环。可以通过 MPI 对这两步进行并行优化。

2 算法设计

源码链接: <https://github.com/ArcanusNEO/Parallel-Programming/tree/master/7>

2.1 测试用例的确定

由于测试数据集较大, 不便于各个平台同步, 所以采用固定随机数种子为 12345687 的 mt19937 随机数生成器。经过实验发现不同规模下, 所有元素独立生成, 限制大小在 $[0, 100]$, 能够生成可以被正确消元的矩阵。

代码如下:

```
1 uniform_real_distribution<float> dist(0, 100);
2 mt19937 mt(12345687);
3 int n;
4 istream iss(argv[1]);
5 iss >> n;
6 cout << n << endl;
7 for (int i = 1; i <= n; ++i)
8     for (int j = 1; j <= n; ++j) cout << dist(mt) << " \n"[j == n];
```

2.2 实验环境和相关配置

实验在本地 Arch Linux x86_64 平台和华为鲲鹏服务器平台完成, 使用 cmake 构建项目, 均开启 O3 加速;

对于本地的 x86 环境, 安装了 OpenMPI 库, 使用的 CPU 是 8 核心 16 线程, 足以应付 4 核 MPI 并行和 4 线程 OpenMP 叠加。

2.3 默认平凡算法

使用一维数组模拟矩阵, 避免改变矩阵大小时第二维不方便调整、必须设成最大值的问题, 可以减少 cache 失效;

使用 `#define matrix(i, j) arr[(i) * n + (j)]` 宏, 增强可读性;

```
1 void func(int& ans, float arr[], int n) {
2     for (int k = 0; k < n; ++k) {
3         for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
4         matrix(k, k) = 1.0;
5         for (int i = k + 1; i < n; ++i) {
6             for (int j = k + 1; j < n; ++j)
7                 matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
8             matrix(i, k) = 0;
9         }
10    }
11 }
```

2.4 MPI 并行化算法

采用了按照行划分的分割方法，每个进程分配到固定的行，某行除法工作由负责的进程计算，计算完成后全局同步。如果矩阵的秩不能被进程数整除，则将余数行划分给最后一个进程；如果矩阵的秩小于进程数，则最后一个进程会承担所有工作，因此需要避免这种情况。

```

1 void func(int& ans, float arr[], int n) {
2     int comm_sz;
3     int my_rank;
4     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
5     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
6
7     MPI_Bcast(arr, n * n, MPI_FLOAT, 0, MPI_COMM_WORLD);
8
9     int block_sz = n / comm_sz;
10    int row_begin = block_sz * my_rank;
11    int row_end = (my_rank + 1 == comm_sz ? n : row_begin + block_sz);
12
13    for (int k = 0; k < n; ++k) {
14        if (row_begin <= k && k < row_end) {
15            for (int j = k + 1; j < n; ++j) matrix(k, j) /= matrix(k, k);
16            matrix(k, k) = 1.0;
17        }
18        int bc_rank = comm_sz - 1;
19        if (block_sz && k / block_sz < bc_rank) bc_rank = k / block_sz;
20        MPI_Bcast(prow(k), n, MPI_FLOAT, bc_rank, MPI_COMM_WORLD);
21        for (int i = max(row_begin, k + 1); i < row_end; ++i) {
22            for (int j = k + 1; j < n; ++j)
23                matrix(i, j) -= matrix(i, k) * matrix(k, j);
24            matrix(i, k) = 0;
25        }
26    }
27 }

```

2.5 MPI 并行化算法结合 OpenMP 多线程库

MPI 是非共享内存的多进程模型，OpenMP 是共享内存的多线程模型，可以混合使用，在除法和消去阶段都可以进一步划分循环：除法部分按列划分，消去部分动态按行划分。需要注意的是，在 MPI 通信过程中只能单线程调用 MPI_Bcast 函数，在通信前需要设置 barrier 保证所有线程都完成了除法阶段，在通信后也需要设置 barrier 保证所有线程等待同步行结束再执行消去。

```

1 #define THREADS 4
2
3 #define matrix(i, j) (arr[(i) * (n) + (j)])
4 #define pmatrix(i, j) (arr + ((i) * (n) + (j)))
5 #define prow(i) (pmatrix(i, 0))
6
7 void func(int& ans, float arr[], int n) {

```

```

8  int comm_sz;
9  int my_rank;
10 MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
11 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12
13 MPI_Bcast(arr, n * n, MPI_FLOAT, 0, MPI_COMM_WORLD);
14
15 int block_sz = n / comm_sz;
16 int row_begin = block_sz * my_rank;
17 int row_end = (my_rank + 1 == comm_sz ? n : row_begin + block_sz);
18
19 int i, j, k;
20 int bc_rank;
21 float tmp;
22 #pragma omp parallel num_threads(THREADS), private(i, j, k, tmp, bc_rank)
23 for (k = 0; k < n; ++k) {
24     if (row_begin <= k && k < row_end) {
25         tmp = matrix(k, k);
26 #pragma omp for
27         for (j = k + 1; j < n; ++j) matrix(k, j) /= tmp;
28         matrix(k, k) = 1.0;
29     }
30     bc_rank = comm_sz - 1;
31     if (block_sz && k / block_sz < bc_rank) bc_rank = k / block_sz;
32 #pragma omp barrier
33     if (omp_get_thread_num() == 0)
34         MPI_Bcast(prow(k), n, MPI_FLOAT, bc_rank, MPI_COMM_WORLD);
35 #pragma omp barrier
36 #pragma omp for
37     for (i = max(row_begin, k + 1); i < row_end; ++i) {
38         tmp = matrix(i, k);
39         for (j = k + 1; j < n; ++j) matrix(i, j) -= tmp * matrix(k, j);
40         matrix(i, k) = 0;
41     }
42 }
43 }

```

2.6 MPI 并行化算法结合 NEON 指令优化

NEON 优化作用于循环的 CPU 核心内并行化, 由于 O3 级别的优化可能已经引入了 SIMD, 效果不一定明显。

```

1 #define matrix(i, j) (arr[(i) * (n) + (j)])
2 #define pmatrix(i, j) (arr + ((i) * (n) + (j)))
3 #define prow(i) (pmatrix(i, 0))
4
5 void func(int& ans, float arr[], int n) {
6     int comm_sz;

```

```

7  int my_rank;
8  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
9  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
10
11  MPI_Bcast(arr, n * n, MPI_FLOAT, 0, MPI_COMM_WORLD);
12
13  int block_sz = n / comm_sz;
14  int row_begin = block_sz * my_rank;
15  int row_end = (my_rank + 1 == comm_sz ? n : row_begin + block_sz);
16
17  for (int k = 0; k < n; ++k) {
18      int j;
19      if (row_begin <= k && k < row_end) {
20          auto vt = vdupq_n_f32(matrix(k, k));
21          for (j = k + 1; j + 4 <= n; j += 4) {
22              auto va = vld1q_f32(pmatrix(k, j));
23              va = vdivq_f32(va, vt);
24              vst1q_f32(pmatrix(k, j), va);
25          }
26          for (; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
27          matrix(k, k) = 1.0;
28      }
29      int bc_rank = comm_sz - 1;
30      if (block_sz && k / block_sz < bc_rank) bc_rank = k / block_sz;
31      MPI_Bcast(prow(k), n, MPI_FLOAT, bc_rank, MPI_COMM_WORLD);
32      for (int i = max(row_begin, k + 1); i < row_end; ++i) {
33          auto vaik = vdupq_n_f32(matrix(i, k));
34          for (j = k + 1; j + 4 <= n; j += 4) {
35              auto vakj = vld1q_f32(pmatrix(k, j));
36              auto vaij = vld1q_f32(pmatrix(i, j));
37              auto vx = vmulq_f32(vakj, vaik);
38              vaij = vsubq_f32(vaij, vx);
39              vst1q_f32(pmatrix(i, j), vaij);
40          }
41          for (; j < n; ++j) matrix(i, j) -= matrix(i, k) * matrix(k, j);
42          matrix(i, k) = 0;
43      }
44  }
45 }

```

3 算法分析

3.1 时间复杂度分析

按行划分的 MPI 算法分成 k 个步骤, 第 k 行的除法部分计算次数为 $n-k-1$, 设 MPI 节点数为 N , 则并行消去部分的计算时间为 $\frac{(n-k-1) \cdot n}{N}$, 则计算部分的时间复杂度为 $\sum_{k=0}^{n-1} \frac{(n-k-1) \cdot n}{N} \sim O(\frac{n^3}{N})$ 。由于使用了 MPI_Bcast 通信, 单次通信时间复杂度为 $O(n \log N)$, 则总通信时间复杂度为 $\sum_{k=0}^{n-1} n \log N \sim$

$O(n^2 \log N)$ ，所以总时间复杂度为 $O(\frac{n^3}{N})$

加入 OpenMP 的算法计算部分时间复杂度可以继续除以线程数 T ，得到时间复杂度 $O(\frac{n^3}{N \cdot T})$ ；对于 NEON 优化的算法，类似的除以向量长度常数。

3.2 运行时间分析