

石邢越 16337208

张永东教授

高性能计算程序设计基础

2018 年 11 月 7 日星期三

### 实验 5: 使用笛卡尔拓扑结构通信子的 MPI 稠密矩阵-向量乘法

#### 1 实验目的

这次实验同实验 3、实验 4，继续研究矩阵-向量乘法的 MPI 算法。

本次实验尝试使用新的通信拓扑结构，即笛卡尔拓扑结构，来完成 MPI 并行化的稠密矩阵-向量乘法。

#### 2 实验要求

- 计算稠密矩阵-向量的乘积  $\mathbf{Ax} = \mathbf{y}$
- 矩阵  $\mathbf{A}$  使用二维网格划分，每个进程中的子矩阵大小为  $\frac{m}{\sqrt{p}} * \frac{n}{\sqrt{p}}$  ( $p$  为并行进程数)

- 矩阵  $\mathbf{A}$  中元素使用以下公式生成：

$$a_{ij} = i - 0.1 * j + 1$$

- 向量  $\mathbf{x}$  中元素使用以下公式生成：

$$x_i = 0.1 * i$$

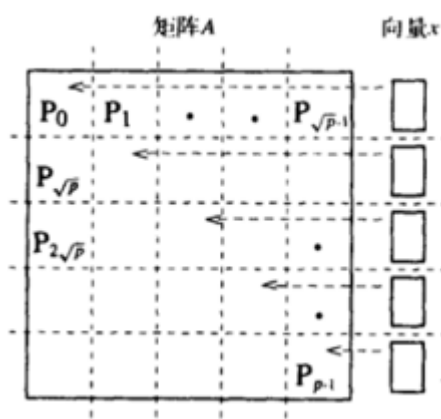
- 使用笛卡尔拓扑的 MPI 接口和集合通信规约函数 `MPI_Reduce` 实现并行化的稠密矩阵-向量乘法。

### 3 算法原理

#### 3.1 矩阵的二维划分

这里我要求了进程数  $p$  是一个完全平方数（若  $p$  非完全平方数，不便于各个进程上的工作均衡，没必要陷于这种情况），否则程序会认为这个  $p$  非法重新要求一个  $p$ 。

二维划分的方式见下图。矩阵  $A$  的大小为  $n * n$ ，向量  $x$  为  $n$  维向量。假设  $\sqrt{p} \mid n$ ，否则认为出现错误。



图表 1 矩阵  $A$  和向量  $x$  的划分方式

#### 3.2 如何得到矩阵 $A$ 和向量 $x$

本实验使用“按子矩阵”二维划分矩阵 $A$ 。

矩阵 $A$ 和向量 $x$ 都由进程 0 计算出来，之后二维划分矩阵 $A$ 并分发到其他各个进程中，并如图表 1 中所示划分向量 $x$ 并分发到各个进程中。分发使用 Scatter 函数。

### 3.3 笛卡尔拓扑结构的通信空间

默认地，MPI 将进程看成一维拓扑结构，并用线性顺序对进程进行编号。然而在这个实验中（以及许多其他的问题场景中），进程实际上是以二维拓扑结构组织的。这时，如果可以用符合二维拓扑结构的坐标来标识进程，将方便编程。

在 MPI 程序中最常用的通信结构是一维、二维或者更高维的网格，即笛卡尔拓扑结构（Cartesian topology）。MPI 提供与笛卡尔拓扑结构相关的一系列接口。

## 4 实验过程

### 4.1 创建通信子，并将通信子改成笛卡尔拓扑结构的

首先创建普通的通信子 `MPI_COMM_WORLD`，如代码 1。

```

1  Build_comms();
2  comm = MPI_COMM_WORLD;

```

代码1 创建通信子 `MPI_COMM_WORLD`

之后判断用户给定的进程数  $p$  是否是完全平方数，这里我限定了  $p$  需要

是完全平方数，否则会重新要求一个  $p$ 。判断过程如代码 2。

```

1  MPI_Comm_size(comm, &comm_sz);
2  int q = sqrt(comm_sz);
3  if (comm_sz != q*q) loc_ok = 0;
4  Check_for_error(loc_ok, "Build_comms", "comm_sz not a perfect square");
5  //若p是完全平方数 (p=q*q)
6  //则可以将原来的通信子改成笛卡尔拓扑结构的
7  //且拓扑结构是一个二维的q*q的网格
8  dim_sizes[0] = dim_sizes[1] = q;

```

代码2 判断  $p$  是否是完全平方数。若是，则得到笛卡尔网络的尺寸为  $q*q$ ，其中  $q=\sqrt{p}$

如果  $p$  符合要求，接下来即可将 `MPI_COMM_WORLD` 改成笛卡尔拓扑

结构的了！见代码 3。

```

1  //通信子MPI_COMM_WORLD现在的拓扑结构为q*q的二维网格
2  MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes, wrap_around, reorder, &grid_comm);
3  comm = grid_comm;
4  MPI_Comm_size(grid_comm, &comm_sz);
5  MPI_Comm_rank(grid_comm, &my_rank);
6
7
8  //每一行上的所有进程创建成一个通信子
9  //同一行上的进程在元素乘完，最后求和的时候需要相互通信
10 free_coords[0] = 0;
11 free_coords[1] = 1;
12 MPI_Cart_sub(grid_comm, free_coords, &row_comm);
13 MPI_Comm_size(row_comm, &row_comm_sz);
14 MPI_Comm_rank(row_comm, &my_row_rank);
15
16 //每一列上的所有进程创建成一个通信子
17 //同一列上的进程需要用到同一部分的向量x
18 free_coords[0] = 1;
19 free_coords[1] = 0;
20 MPI_Cart_sub(grid_comm, free_coords, &col_comm);
21 MPI_Comm_size(col_comm, &col_comm_sz);
22 MPI_Comm_rank(col_comm, &my_col_rank);
23
24 //记录哪些进程在对角线上
25 //之后需要将向量x发送给对角线上的元素
26 if (my_row_rank == my_col_rank)
27     diag = 1;
28 else
29     diag = 0;
30
31 //记录进程rank对应的二维坐标
32 MPI_Cart_coords(comm, my_rank, 2, coords);
   which_row_comm = coords[0];
   which_col_comm = coords[1];

```

代码3 让 `MPI_COMM_WORLD` 具有笛卡尔拓扑结构。之后同一列上的进程需要广播（Broadcast）同一部分的向量  $x$ ，所以把每一列创建成一个通信子。之后同一行上的乘积要加和（Reduce），所以把每一行创建成一个通信子。之后要将划分好的  $x$  发送给对角线上的进程（见图表1），所以这里我用变量 `diag` 标记了一个进程是否在对角线上。

## 4.2 进程 0 计算矩阵 $A$ 和向量 $x$ 并分发（Scatter）给各进程

这部分和之前的两次实验差不多，只不过之前是从文件中读取了矩阵  $A$  和向量  $x$  的数据，这次是用公式直接算出  $A$ ， $x$  中各元素的值，见代码 4，5。

```

1  if (my_rank == 0) {
2      //进程0中给矩阵A申请空间并根据公式计算出A中各元素的值
3      A = (double*)malloc(m*n*sizeof(double));
4      if (A == NULL) loc_ok = 0;
5      Check_for_error(loc_ok, "Read_matrix",
6                      "Can't allocate temporary matrix");
7
8      printf("Calculating elements of matrix %s...\n", prompt);
9      for (i = 0; i < m; i++)
10         for (j = 0; j < n; j++)
11             A[i*n+j] = i - 0.1*j + 1;
12
13     //将计算出的矩阵A分发给MPI_COMM_WORLD中的各进程
14     //注意此时的MPI_COMM_WORLD已经是二维的笛卡尔拓扑结构的了
15     MPI_Scatterv(A, distrib_counts, distrib_disps, submat_mpi_t,
16                 loc_A, loc_m*loc_n, MPI_DOUBLE, 0, comm);
17     free(A);
18 } else {
19     Check_for_error(loc_ok, "Read_matrix",
20                     "Can't allocate temporary matrix");
21     MPI_Scatterv(A, distrib_counts, distrib_disps, submat_mpi_t,
22                 loc_A, loc_m*loc_n, MPI_DOUBLE, 0, comm);
23 }

```

代码4 进程0 计算矩阵A 并分发 (Scatter) 给各进程

```

1  if (my_diag_rank == 0) {
2      vec = (double*)malloc(n*sizeof(double));
3      if (vec == NULL) loc_ok = 0;
4      Check_for_error(loc_ok, "Read_vector",
5                      "Can't allocate temporary vector");
6      printf("Calculating the elements of vector %s...\n", prompt);
7      for (i = 0; i < n; i++)
8          vec[i] = 0.1*i;
9      MPI_Scatter(vec, loc_n, MPI_DOUBLE,
10                 loc_vec, loc_n, MPI_DOUBLE, 0, diag_comm);
11     free(vec);
12 } else {
13     Check_for_error(loc_ok, "Read_vector",
14                     "Can't allocate temporary vector");
15     if (diag)
16         MPI_Scatter(vec, loc_n, MPI_DOUBLE,
17                     loc_vec, loc_n, MPI_DOUBLE, 0, diag_comm);
18 }

```

代码5 进程0 计算向量x 并分发 (Scatter) 给各进程

### 4.3 矩阵-向量乘法, 聚合 (Reduce) 得到最终结果

基于“按子矩阵划分”的矩阵-向量乘法，需要用到代码 3 中创建的“行通信

子”和“列通信子”。执行完代码 6，结果向量  $y$  已经得到了。

```

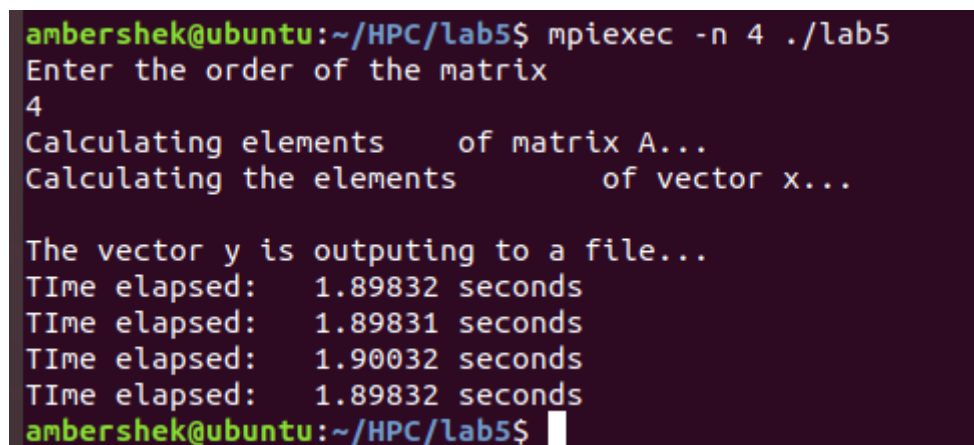
1 //通信子同一列上的进程需要用到同一部分的x
2 //在同一列所有进程构成的通信子上广播要用到的那一部分x
3 MPI_Bcast(loc_x, loc_n, MPI_DOUBLE, which_col_comm, col_comm);
4
5
6 //local的矩阵-向量乘法
7 for (loc_i = 0; loc_i < loc_m; loc_i++) {
8     sub_y[loc_i] = 0.0;
9     for (loc_j = 0; loc_j < loc_n; loc_j++)
10         sub_y[loc_i] += loc_A[loc_i*loc_n + loc_j]*loc_x[loc_j];
11 }
12
13 //通信子同一行上各个local矩阵-向量乘法的结果要加在一起
14 //用Reduce聚集这一行上所有local矩阵-向量乘法的结果
15 //每个reduce的结果是结果向量y中的一个元素
16 MPI_Reduce(sub_y, loc_y, loc_m, MPI_DOUBLE, MPI_SUM,
            which_row_comm, row_comm);

```

代码6 矩阵-向量乘法

## 5 结果分析

以  $p=4$ ， $A$  大小为  $4*4$  的情况为例，运行结果如图表 2。



```

ambershek@ubuntu:~/HPC/lab5$ mpiexec -n 4 ./lab5
Enter the order of the matrix
4
Calculating elements      of matrix A...
Calculating the elements      of vector x...

The vector y is outputting to a file...
Time elapsed: 1.89832 seconds
Time elapsed: 1.89831 seconds
Time elapsed: 1.90032 seconds
Time elapsed: 1.89832 seconds
ambershek@ubuntu:~/HPC/lab5$

```

图表2 运行结果样例

统计各种情况的运行的并行用时 $T_p$ ，得到表格 1（左列标题为矩阵 A 的尺寸 n，其他表格同）。

	1 进程	4 进程	16 进程	64 进程
1	0.003722			
2	0.000702	0.000592		
3	0.000501			
4	0.000939	0.000433	0.568664	
5	0.000346			
6	0.000345	0.000453		
7	0.000372			
8	0.000366	0.000431	0.388866	2.02215
9	0.000837			
10	0.000804	1.631044		
11	0.000756			
12	0.002674	0.000448	0.529115	
13	0.002701			
14	0.002672	0.000358		
15	0.001061			
16	0.002943	0.000427	0.468535	2.16436

表格 1  $T_p$  的比较

统计各种情况的运行的加速比 S，得到表格 2。

	1 进程	4 进程	16 进程	64 进程
1	1			
2	1	1.185588		
3	1			
4	1	2.167952	0.001651	
5	1			
6	1	0.762383		
7	1			
8	1	0.848921	0.000941	0.000181
9	1			
10	1	1.631044		
11	1			
12	1	5.972842	0.005054	
13	1			
14	1	7.472653		
15	1			
16	1	6.884546	0.006281	0.00136

表格 2 S 的比较



统计各种情况的运行的效率  $E$ ，得到表格 3。

	1 进程	4 进程	16 进程	64 进程
1	1.000000			
2	1.000000	0.296397		
3	1.000000			
4	1.000000	0.541988	0.000103	
5	1.000000			
6	1.000000	0.190596		
7	1.000000			
8	1.000000	0.212230	0.000059	0.000011
9	1.000000			
10	1.000000	0.407761		
11	1.000000			
12	1.000000	1.493210	0.000316	
13	1.000000			
14	1.000000	1.868163		
15	1.000000			
16	1.000000	1.721136	0.000393	0.000085

表格 3  $E$  的比较

## 6 分析讨论

从上面的表格 1, 2, 3 来看，进程数为 4 的时候加速效果最好。此时最大的加速比  $S$  达到了 7.472653，效率  $E$  达到了 1.868163。

随着进程数增加，加速效果变得越来越差，当进程数为 16 和 64 的时候得到的都是负加速比，这首先是因为进程间的通行骤增，其次也因为我的电脑实际上无法创建那么多的物理进程，所以实际并发工作的进程并没有那么多。

这次实验的重点是为通信子选择合适的拓扑结构。因为我们用的是“按子矩阵”划分矩阵  $A$ ，所以此时合适的拓扑结构就是二维的笛卡尔网格。用了这个拓扑结构后，可以方便地在整行、整列上进行集合通行。因为进程以二维网络，而不是一维线性的形式组织，整个程序设计的过程中对进程的操作是符合我们对“二维划分”算矩阵-向量乘的逻辑的。