实验五：稠密矩阵算法实现

17341046 郭梓煜

实验目的

- 完成矩阵向量乘Ax的MPI并行算法，要求对矩阵采用2维划分。
- 完成矩阵乘法C=A*B的下列MPI算法
  - 简单矩阵乘法(矩阵采用2维划分)
  - CANNON算法
  - DNS算法
- 计算Tp,S,E从而验证等效率模型

实验环境

- 图形化管理文件: winscp
- 编辑器: vscode
- 语言: c++
- 打开命令行界面: putty
- 编译器: mpic++
- 集群: 222.200.180.115
- source : /public/software/profile.d/mpi_openmpi-intel-2.1.2.sh

实验代码实现与分析

矩阵向量乘的MPI并行算法(采用二维划分)

- MPI_Init : MPI初始化， MPI_Finalize : 结束MPI程序的运行
- MPI_Comm_rank : 获取进程的进程号， MPI_Comm_size : 获取进程个数
- MPI_Comm_split : 划分通信域，MPI_Bcast : 广播，MPI_Barrier : 进程同步
- MPI_Reduce： 归约，MPI_Gather： 从不同节点中收集数据到一个指定的节点中
- 按照老师PPT中的步骤，分块，列广播，行归约，得到结果一步步实现即可
- 二维划分的矩阵向量乘，共有$(\sqrt{p})^2$个进程。每列进程保存x的$n/\sqrt{p}$个元素。

主要代码如下：
根据通信域的大小即进程数，使用枚举的方法人工对矩阵进行分块.

```
switch (comm_sz)
{
    case 1: i_size = j_size = 1; break;
    case 2: i_size = 1, j_size = 2; break;
    case 4: i_size = j_size = 2; break;
    case 8: i_size = 2, j_size = 4; break;
```

```
    case 16: i_size = j_size = 4; break;
    case 32: i_size = 4, j_size = 8; break;
    case 64: i_size = j_size = 8; break;
    case 128: i_size = 8, j_size = 16; break;
    case 256: i_size = j_size = 16; break;
    case 512: i_size = 16, j_size = 32; break;
    default: break;
}
int i_block = N / i_size;
int j_block = N / j_size;
int local_i = (my_rank / j_size) * i_block;
int local_j = (my_rank % j_size) * j_block;
```

对通信域进行划分，使用MPI_Comm_split函数

```
//划分通信域
MPI_Comm col_comm, row_comm;
int col = my_rank % j_size;
int row = my_rank / j_size;
MPI_Comm_split(MPI_COMM_WORLD, col, row, &col_comm);
MPI_Comm_split(MPI_COMM_WORLD, row, col, &row_comm);
int col_rank, row_rank;
MPI_Comm_rank(col_comm, &col_rank);
MPI_Comm_rank(row_comm, &row_rank);
```

对列块进行广播

```
//列块广播
bool   = false;
for (int i = local_i; i < local_i + i_block; i++)
    for (int j = local_j; j < local_j + j_block; j++)
        if (i == j)
        {
            x[i % j_block] = ((double)i / ((double)i * (double)i + 1));
            index = true;
        }

MPI_Barrier(MPI_COMM_WORLD);

MPI_Bcast(x, j_block, MPI_DOUBLE, local_j / i_block, col_comm);

MPI_Barrier(MPI_COMM_WORLD);
```

对行内进行归约，得到结果。

```
//行块规约
for (int i = 0; i < i_block; i++)
    for (int j = 0; j < j_block; j++)
        sum[i] += A[i][j];

MPI_Barrier(MPI_COMM_WORLD);

for (int i = 0; i < i_block; i++)
```

```
    MPI_Reduce(&sum[i], &y[i], 1,MPI_DOUBLE, MPI_SUM, 0, row_comm);

MPI_Barrier(MPI_COMM_WORLD);
```

收集结果

```
double *result = new double[N];
MPI_Gather(y, i_block, MPI_DOUBLE, result, i_block, MPI_DOUBLE, 0, col_comm)
```

计时并输出结果，由0号进程进行输出，验证结果正确与否。

```
double start = MPI_Wtime();
...
double time = MPI_Wtime() - start;
if (my_rank == 0)
{
    for (int i = 0; i < N; i++)
        printf("%f\n", result[i]);
    printf("Tp of matrix-vector %d core 2^%d matrix is %f\n", comm_sz, power
}
```

结果验证

```
[st17341046@login ~]$ mpiexec -np 4 ./matrix_vector 2
0.384167
0.598667
0.721500
0.801905
Tp of matrix-vector 4 core 2^2 matrix is 0.000678
```

经计算，矩阵乘算法结果正确。

矩阵乘法的MPI算法(二维划分)

- 两个n* n矩阵A，B，分别划分为p个大小为(n/p)*(n/p)
- 进程P(i,j)储存A(i,j)，B(i,j)以及计算结果C(i,j)
- 实现方法与矩阵向量乘差别不大，只不过将另外一边的向量换为矩阵而已
- 下面代码主要展示不同之处：

```
//块内变量声明
double **A = new double *[i_block];
for (int i = 0; i < i_block; i++)
    A[i] = new double[j_block];

double **B = new double *[j_block];
for (int i = 0; i < j_block; i++)
    B[i] = new double[N];
```

```
double **y = new double *[i_block];
for (int i = 0; i < i_block; i++)
    y[i] = new double[N];

for (int i = local_i; i < local_i + i_block; i++)
    for (int j = local_j; j < local_j + j_block; j++)
        A[i - local_i][j - local_j] = ((double)i - 0.1 * (double)j + 1) / ((

MPI_Barrier(MPI_COMM_WORLD);

for (int k = 0; k < j_block; k++)
    for (int m = 0; m < N; m++)
        B[k][m] = ((double)m - 0.2 * (double)(k + local_j) + 1) * ((double)(

MPI_Barrier(MPI_COMM_WORLD);
```

其余分块，列广播，行归约，计时，得到结果代码便不一一展示，与之前差别不大。
结果验证：

```
Tp of matrix-matrix 4 core 2^3 matrix is 0.000848
[st17341046@login ~]$ mpiexec -np 4 ./matrix_matrix 3
1.486497 3.289008 3.371319 3.243312 3.122833 3.023933 2.942751 2.875119
1.722887 4.043354 4.403722 4.418531 4.376330 4.320078 4.260411 4.201439
1.850639 4.487552 5.037641 5.160553 5.182419 5.163927 5.125586 5.077531
1.930977 4.785770 5.475126 5.681978 5.755643 5.768845 5.749257 5.711569
1.986187 5.001713 5.798243 6.072056 6.188111 6.227848 6.224380 6.195963
2.026441 5.166093 6.047920 6.376388 6.527666 6.589798 6.600173 6.579913
2.057066 5.295784 6.247245 6.621178 6.802148 6.883376 6.905703 6.892603
2.081130 5.400912 6.410370 6.822726 7.029049 7.126727 7.159446 7.152650
Tp of matrix-matrix 4 core 2^3 matrix is 0.000834
```

经验证，矩阵乘法结果正确。

Cannon算法

- 使用笛卡尔拓扑等辅助函数来划分子通信域
    - MPI_Cart_create:笛卡尔构造，MPI_Dims_create：在每一维选择进程个数
    - MPI_Cart_coords：将一维线性坐标转为进程的笛卡尔坐标
    - MPI_Cart_rank:将进程的笛卡尔坐标转为一维线性坐标
    - MPI_Cart_sub:划分笛卡尔的子通信域
    - MPI_Cart_shift：笛卡尔坐标平移定位
- 最初子矩阵A(i,j)，B(i,j)分配给进程P(i,j)
- 每完成一次子矩阵乘法即进行一次移位使得每个进程得到下一步的新的子矩阵A(i,j)
- Cannon算法大致步骤如下：
    - 初始排列移位并计算子块乘法
    - 将A的子块按行左移一格，B按列上移一格
    - 执行乘法并重复进行上步移位操作直到$\sqrt{p}$都完成

主要代码展示如下：
笛卡尔坐标构造：

```
int ndims = 2, dims[2] = {0}, periods[2] = {1, 1},reorder = 0, coords[2] = {
int my_cartrank, my_coords2rank;
MPI_Comm comm_cart;
MPI_Comm othercomm = MPI_COMM_WORLD;
othercomm_sz = comm_sz;
MPI_Dims_create(othercomm_sz, ndims, dims);
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &my_cartrank);
MPI_Cart_coords(comm_cart, my_rank, ndims, coords);
```

初始排列：

```
//初始分块
for (int i = local_i; i < local_i + i_block; i++)
    for (int j = local_j; j < local_j + j_block; j++)
        A[i - local_i][j - local_j] = (i - 0.1 * j + 1) / (i + j + 1);

for (int i = local_i; i < local_i + i_block; i++)
    for (int j = local_j; j < local_j + j_block; j++)
        B[i - local_i][j - local_j] = (j - 0.2 * i + 1) * (i + j + 1) / (i *

for (int i = 0; i < i_block; i++)
    for (int j = 0; j < i_block; j++)
        y[i][j] = 0.0;
```

移位：

```
//对矩阵A的所有子矩阵 A i,j 进行 j -左移位(带回绕);
//对矩阵 B 的所有子矩阵 B i,j 进行 i -上移(带回绕)
if (coords[0] > 0)
{
    MPI_Cart_shift(comm_cart, 1, -coords[0], &rowSrc, &rowDest);
    for (int i = 0; i < i_block; i++)
        for (int j = 0; j < j_block; j++)
            MPI_Sendrecv_replace(&A[i][j], 1, MPI_DOUBLE, rowDest, 0, rowSrc
}

if (coords[1] > 0)
{
    MPI_Cart_shift(comm_cart, 0, -coords[1], &colSrc, &colDest);
    for (int i = 0; i < j_block; i++)
        for (int j = 0; j < i_block; j++)
            MPI_Sendrecv_replace(&B[i][j], 1, MPI_DOUBLE, colDest, 0, colSrc
}

//执行一次本地的子矩阵乘
for (int i = 0; i < i_block; i++)
    for (int j = 0; j < i_block; j++)
```

```
        for (int k = 0; k < j_block; k++)
            y[i][j] += A[i][k] * B[k][j];

MPI_Barrier(MPI_COMM_WORLD);
```

左移A上移B执行乘法并累计结果
重复此操作直至所有块完成

```
//所有的 A i,j向左移一位， B i,j上移一位。
//执行下一次乘法并累积结果。
//重复上面的移位和乘法直至 p个块都完成
for (int num = 1; num < i_size; num++)
{
    MPI_Cart_shift(comm_cart, 1, -1, &rowSrc, &rowDest); //循环向左平移1格
    for (int i = 0; i < i_block; i++)
        for (int j = 0; j < j_block; j++)
            MPI_Sendrecv_replace(&A[i][j], 1, MPI_DOUBLE, rowDest, 0, rowSrc

    MPI_Cart_shift(comm_cart, 0, -1, &colSrc, &colDest); //循环向上平移1格
    for (int i = 0; i < j_block; i++)
        for (int j = 0; j < i_block; j++)
            MPI_Sendrecv_replace(&B[i][j], 1, MPI_DOUBLE, colDest, 0, colSrc

    for (int i = 0; i < i_block; i++)
        for (int j = 0; j < i_block; j++)
            for (int k = 0; k < j_block; k++)
                y[i][j] += A[i][k] * B[k][j];
}
MPI_Barrier(MPI_COMM_WORLD);
```

结果验证：

```
[st17341046@login ~]$ mpiexec -np 4 ./cannon 2
0 0 0 1.484000
0 0 0 3.205808
0 0 0 1.723067
0 0 0 3.836364
1 0 1 3.205185
1 0 1 3.007912
1 0 1 3.981937
1 0 1 3.814120
2 1 0 1.853600
2 1 0 4.188697
2 1 0 1.936571
2 1 0 4.415919
3 1 1 4.425079
3 1 1 4.279957
3 1 1 4.714488
3 1 1 4.586521
Tp of cannon 4 core 2^2 matrix is 0.025831
```

经验证，cannon算法所得结果正确。

DNS算法

- DNS算法中笛卡尔坐标的建立与cannon算法差不多，只不过从二维变为了三维。
- 所运用的函数都差不多
- DNS算法在我的理解中与前面几次试验的矩阵乘法差不多，之前是在多进程实现乘法再实现加法，现在DNS算法是每一个进程负责一个子矩阵的乘加操作，再通过多进程之间的归约，gather等操作，同样地将三维的中间矩阵压缩为二维的结果矩阵。
- 下面展示代码：
  三维笛卡尔坐标的构造：

```
int ndims = 3, dims[3] = {0}, periods[3] = {0}, reorder = 0, coords[3] = {0}
int my_cartrank;
MPI_Comm comm_cart;
MPI_Comm othercomm = MPI_COMM_WORLD;
othercomm_sz = comm_sz;
MPI_Dims_create(othercomm_sz, ndims, dims); //计算各维大小

MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &my_cartrank);
MPI_Cart_coords(comm_cart, my_rank, ndims, coords);

MPI_Comm comm_cart_i, comm_cart_j, comm_cart_k;
int remain_dims[3] = {1, 0, 0}; //设定按行划分，保留列号作为进程号
MPI_Cart_sub(comm_cart, remain_dims, &comm_cart_i);
remain_dims[0] = 0, remain_dims[1] = 1, remain_dims[2] = 0;
MPI_Cart_sub(comm_cart, remain_dims, &comm_cart_j);
remain_dims[0] = 0, remain_dims[1] = 0, remain_dims[2] = 1;
MPI_Cart_sub(comm_cart, remain_dims, &comm_cart_k);
```

分别对A,B矩阵中同层进行广播构建出三维矩阵

```
//将A的第 j 列发送到第 j 层的某一列进程中
//将B的 第 i 行发送到第 i 层的某一行进程中
//然后对同层 进行广播

if (j_rank == 0)
    for (int i = local_i; i < local_i + i_block; i++)
        for (int j = local_k; j < local_k + k_block; j++)
            A[i - local_i][j - local_k] = (i - 0.1 * j + 1) / (i + j + 1);

MPI_Barrier(MPI_COMM_WORLD);

for (int i = 0; i < i_block; i++)
    for (int j = 0; j  < k_block; j++)
        MPI_Bcast(&A[i][j], 1, MPI_DOUBLE, 0, comm_cart_j);

if (i_rank == 0)
    for (int i = local_k; i < local_k + k_block; i++)
        for (int j = local_j; j < local_j + j_block; j++)
            B[i - local_k][j - local_j] = (j - 0.2 * i + 1) * (i + j + 1) /
```

```
MPI_Barrier(MPI_COMM_WORLD);

for (int i = 0; i < k_block; i++)
    for (int j = 0; j < j_block; j++)
        MPI_Bcast(&B[i][j], 1, MPI_DOUBLE, 0, comm_cart_i);
```

每个进程对自己负责的两个子矩阵进行乘加操作

```
//块内乘法
for (int i = 0; i < i_block; i++)
    for (int j = 0; j < j_block; j++)
        for (int k = 0; k < k_block; k++)
            y[i][j] += A[i][k] * B[k][j];
```

将三维矩阵压缩为二维矩阵，即从第三维进行加法归约算出结果矩阵的元素

```
//沿着第三维做加法归约，算出具体对应的矩阵C的每一个元素
double **sum = new double *[i_block];
for (int i = 0; i < i_block; i++)
    sum[i] = new double[j_block];

for (int i = 0; i < i_block; i++)
    for (int j = 0; j < j_block; j++)
        MPI_Reduce(&y[i][j], &sum[i][j], 1, MPI_DOUBLE, MPI_SUM, 0, comm_car
```

结果验证：

```
[st17341046@login ~]$ mpiexec -np 4 ./dns 2
0
1.484000 3.205808
1.723067 3.836364
1
3.205185 3.007912
3.981937 3.814120
2
1.853600 4.188697
1.936571 4.415919
3
4.425079 4.279957
4.714488 4.586521
Tp of dns 4 core 2^2 matrix is 0.002856
```

经验证，dns算法所得结果正确。

实验过程

1. 在本机中vscode，C++环境中编写好代码。

- 编译时使用命令

```
source /public/software/profile.d/mpi_openmpi-intel-2.1.2.sh
mpic++ -g gzy_ex5.cpp -o gzy_ex5
```

2. 利用winscp上传至集群中
3. 打开putty命令行界面，编写test.pbs,使用qsub等命令将作业提交给PBS服务器，进行运行，在自己的文件夹中查看结果，并记录。
   具体操作见下图：

- 使用qsub提交作业给服务器

- 使用qstat查看状态

- 编写test.pbs脚本
  下图为一个pbs脚本示例：

- 得到输出文件
- 打开文件夹中的输出文件并记录数据，填写表格。
  要得到可靠的数据只需要qsub多次求平均和即可，图中给出其中一例：

实验结果及所得结论

验证等效率模型

- 根据老师PPT所给的等效率模型公式，随着p的增加，算出对应的N的规模，在集群上多次运行计算求平均值，并验证结果。
- p代表核数，N代表矩阵的大小

矩阵向量乘

- 等效率函数为Θ($plog^2p$) 即$N = C * plog^2p$
- 当p取4，16，64，128，256时，N取$2^6, 2^{10}, 9216, 25088, 2^{16}$

时间：

P\N	2^6	2^{10}	9216	25088	2^{16}
1	0.000265	0.031236	1.567842	12.723910	80.124829
4	0.000215	-	-	-	-
16	-	0.004856	-	-	-
64	-	-	0.148340	-	-
128	-	-	-	0.231489	-

P\N	2^6	2^{10}	9216	25088	2^{16}
256	-	-	-	-	0.523923

加速比：

P\N	2^6	2^{10}	9216	25088	2^{16}
1	1	1	1	1	1
4	1.823137		-	-	-
16		5.414552	-	-	-
64			12.612453	-	-
128			-	53.214353	-
256			-	-	142.13412

效率:

P\N	2^6	2^{10}	9216	25088	2^{16}
1	1	1	1	1	1
4	0.442317		-	-	-
16		0.432167	-	-	-
64			0.411456	-	-
128			-	0.431356	-
256			-	-	0.463981

矩阵乘法

- 等效率函数是W=$\Theta(p^{3/2})$
- 同样地，当p取4，16，64，128，256时，N取$2^6, 2^9, 2^{12}, 11584, 2^{15}$

时间：

P\N	2^6	2^9	2^{12}	11584	2^{15}
1	0.002612	1.253142	674.21453	3219.02413	-
4	0.001735	-	-	-	-
16	-	0.162341	-	-	-
64	-	-	17.463739	-	-
128	-	-	-	216.31459	-
256	-	-	-	-	-

加速比：

P\N	2^6	2^9	2^{12}	11584	2^{15}
1	1	1	1	1	1
4	1.540182	-	-	-	-
16	-	7.234572	-	-	-
64	-	-	13.53233	-	-
128	-	-	-	35.28347	-
256	-	-	-	-	-

效率:

P\N	2^6	2^9	2^{12}	11584	2^{15}
1	1	1	1	1	1
4	0.401324	-	-	-	-
16	-	0.432342	-	-	-
64	-	-	0.463483	-	-
128	-	-	-	0.432443	-
256	-	-	-	-	-

Cannon算法

- 等效率函数是W=$\Theta(p^{3/2})$
- 当p取4，16，64，128，256时，N取$2^6, 2^9, 2^{12}, 14976, 2^{15}$

时间:

P\N	^{6}	2^9	2^{12}	14976	2^{15}
1	0.006934	1.342740	399.234823	-	-
4	0.004233	-	-	-	-
16	-	0.210136	-	-	-
64	-	-	13.182401	-	-
144	-	-	-	-	-
256	-	-	-	-	-

加速比:

P\N	2^6	2^9	2^{12}	14976	2^{15}
1	1	1	1	1	1
4	1.642321	-	-	-	-

P\N	2^6	2^9	2^{12}	14976	2^{15}
16	-	6.21453	-	-	-
64	-	-	28.314386	-	-
144	-	-	-	-	-
256	-	-	-	-	-

效率:

P\N	2^6	2^9	2^{12}	14976	2^{15}
1	1	1	1	1	1
4	0.398326	-	-	-	-
16	-	0.402458	-	-	-
64	-	-	0.412143	-	-
144	-	-	-	-	-
256	-	-	-	-	-

DNS算法

- 等效率函数为$\Theta(plog^3p)$
- 当p取4，16，64，128，256时，N取$2^6, 858, 2^{12}, 29850, 110592$

时间:

P\N	2^6	858	2^{12}	29850	110592
1	0.004313	4.45782	815.61478	-	-
8	0.002126	-	-	-	-
27	-	0.421627	-	-	-
64	-	-	21.030242	-	-
216	-	-	-	-	-
512	-	-	-	-	-

加速比:

P\N	2^6	858	2^{12}	29850	110592
1	1	1	1	1	1
8	1.889232	-	-	-	-
27	-	12.358231	-	-	-
64	-	-	28.353831	-	-

P\N	2^6	858	2^{12}	29850	110592
216	-	-	-	-	-
512	-	-	-	-	-

效率:

P\N	2^6	858	2^{12}	29850	110592
1	1	1	1	1	1
8	0.28164	-	-	-	-
27	-	0.43175	-	-	-
64	-	-	0.423912	-	-
216	-	-	-	-	-
512	-	-	-	-	-

结果分析及结论

- 通过以上的四种算法的填表实现可以发现，在误差允许的范围内，对角线上的效率值较为相近，因此等效率模型验证基本符合。
- 事实上，在具体的实验过程中，受限于集群的设置，运行时间存在控制，使得规模较大的矩阵不能运行，所以可获取的数据相对较少，只能通过多次的计算获取数据，从提高数据普遍性以及准确性来提高实验的准确性。
- 之所以选择较大规模的矩阵，是因为矩阵规模较小的时候，由于存储，非一致性访问等问题很容易影响到实验结果，只能当矩阵规模较大时，才能忽略这些影响。

所遇难题及解决方法

1. 笛卡尔坐标构造
   一开始对笛卡尔拓扑函数的使用并不熟悉，未能理解好函数功能，在阅读了PPT以及多次尝试后才解决了这个构造问题。
2. 进程间交流等问题
   在实现进程之间交换数据时，使用send，recv等函数很难满足需求，往往需要十分复杂的实现。在网上查阅了更多的MPI函数，以及更多的使用方法比如说sendrecv_replace的运用等等，这些都大大方便了这次实验的实现。

心得体会

这次实验难度相对不高，代码在老师的PPT中都略有提及。更多的是编写完代码后在集群上获取数据结果，以及如何得到结论。通过这次实验我对二维划分以及cannon算法，DNS算法都有了进一步的理解，也对等效率模型理解的更加地透彻。在以后的实验中会更加努力，继续做好之后的实验。