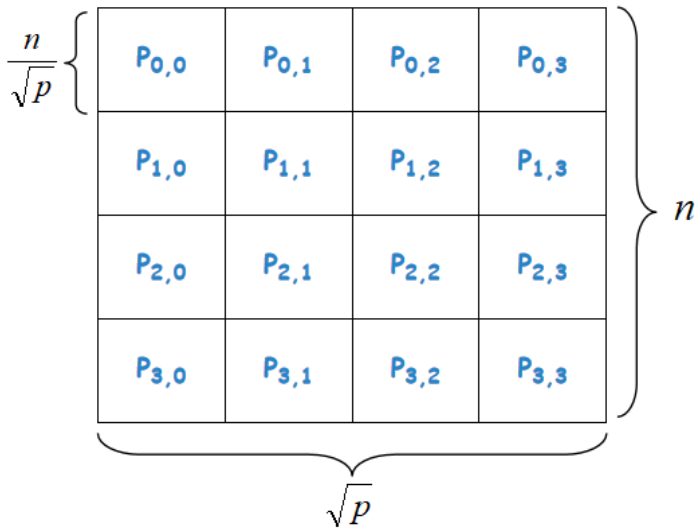
Project 3 并行矩阵乘法的不同通信模式实现

一、基本并行算法

任务是计算方阵A和B的乘积，A和B的形状都是 (n x n)

不妨设有p=q*q个处理器，处理器用元组 (i , j)来编号

我们将方阵 A 和 B （以及乘积C）划分成 (q x q) 个大小相同的子方阵，形状如下。



任务划分可以是：处理器 $P_{i,j}$ 负责计算最终乘积的第（i,j) 块$C_{i,j}$。

运算可以用下述的公式表示：

$$C_{i,j} = \sum_{k=0}^{q-1} A_{i,k} B_{k,j}$$

这意味着进程$P_{i,j}$ 需要获取到分块后A中的第 i 行的方块和B中的第j列的方块，如下图

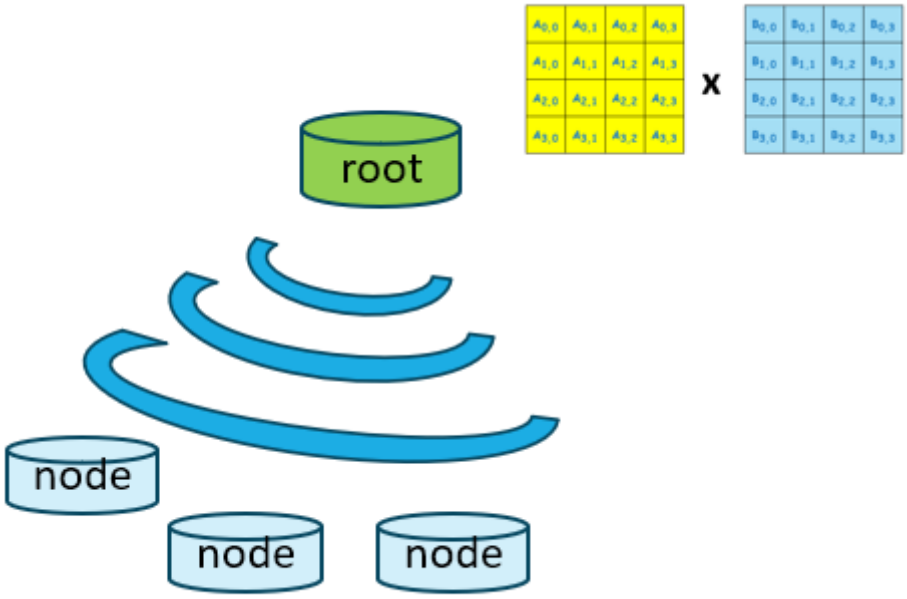
二、通信模式比较

在现实情况中，矩阵A和B的规模可能非常大，而且最开始只有一个处理器保存了A和B的数据，我们将这个进程称为root

现在需要考虑root处理器如何将数据分发给其它节点。

节点Node(i , j)至少需要获取A的第 i 行和B的第 j 列才能算出结果

1. BroadCast

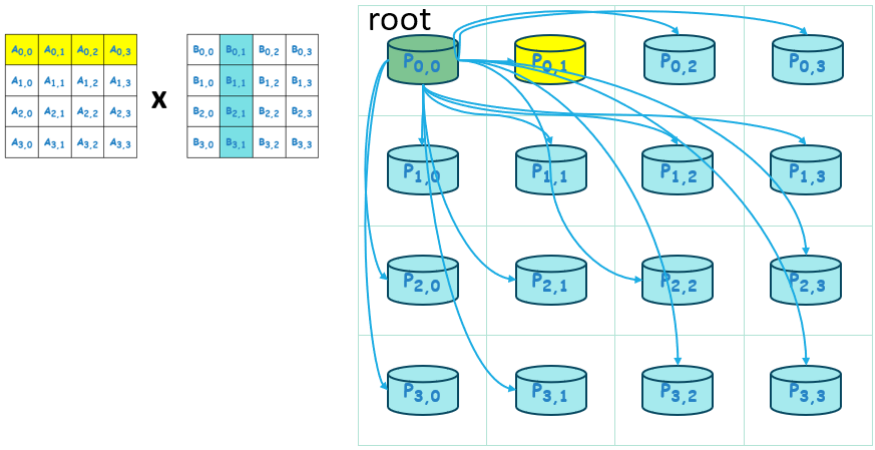


root直接将完整的A和B的数据广播给其它节点。

每个节点都能收到完整的A和B的数据，然后取自己需要的部分计算出C_{ij}

如果root和node以总线连接，那么root只需要发送一份数据，所有的Node就都可以接收。因此发送量是$2q^2$个Block（即一份A和B）

2. Scatter Strip



Scatter Strip是指 root节点给 node(i,j)发送其所需要用到的A的第i行和B的第j列（一行或者一列称为strip）

这种情况下每个节点所接受的数据正好是自己所需要的。

这种情况下root和每一个node相当于点到点通信，root总共发送了$2q^3$个Block

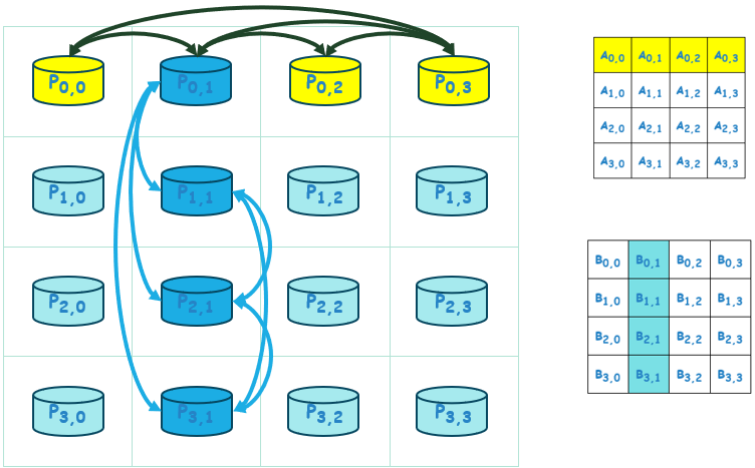
3. Block Exchange

上面的两种方法，数据发送任务都集中在root上，如果root的带宽不足，那么并行加速的性能将会遇到通信瓶颈。

因此我们可以减少 root 的数据发送量，root只给Node(i,j) 发送两个块$A_{ij}B_{ij}$

然后需要同一行的Node互相交换自己的A_{ij}，这样每一个node都可以还原出完整的一行

同一列的Node互相交换自己的B_{ij}，这样每一个node都可以还原出完整的一列



这种模式要求每一行每一列的Node可以互相通信，最后通信均匀地分散整个网络上

总的数据发送量为$2q^3$个Block

4. Cannon算法

以上的三种算法都要求Node(i,j)先收集好 第i行的A块和第j列的B块，然后再进行运算。

但是如果Node(i,j)处理器的存储空间不足以存储这完整的 $2q$ 个块，以上的算法都将失效。

Cannon算法可以解决存储空间不足的问题，只需要处理器可以保存两个完整的块

Cannon算法流程

将矩阵 A 和 B 分成 p 个方块 $A_{i,j}$ 和 $B_{i,j}$ $(0 \leq i,j \leq \sqrt{p}-1)$ ，每块大小为 $(\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}})$，并将它们分配给 $\sqrt{p} * \sqrt{p}$ 个处理器

$(P_{0,0}, P_{0,1}, \ldots, P_{\sqrt{p},\sqrt{p}})$ ，开始时处理器 $P_{i,j}$ 存放块 $A_{i,j}$ 和 $B_{i,j}$，负责计算C。

- 将块 $A_{i,j}, (0 \leq i,j \leq \sqrt{p})$ 向左循环移动 i 步；将块 $B_{i,j}, (0 \leq i,j \leq \sqrt{p})$ 向上循环移动 j 步；
- $P_{i,j}$执行矩阵乘法并将本地矩阵乘法结果相加； 将块 $A_{i,j}, (0 \leq i,j \leq \sqrt{p})$ 向左循环移动 1 步；将块 $B_{i,j}, (0 \leq i,j \leq \sqrt{p})$ 向上循环移动 1 步；
- 重复上面的步骤 $\sqrt{p}-1$次。

总的通信量是$2q^3$个Block

5. 不同通信模式比较

通信模式	**适用场景**	**特点**	**数据发送量/块**
Broadcast	Root和其它node以**总线**相连	Root发送量少	$2q^2$
Scatter Strip	Root和其它node**点到点**通信	通信集中在root	$2q^3$
Block Exchange	所有的node之间都**相互连接**	通信均匀分散在整个网络	$2q^3$
Cannon	Node之间相互连接，且每一个node的**存储空间不足**	需要每一个node的结构相似，处理速度相近，否则循环会发生阻塞	$2q^3$

三、 MPI实现

下面基于Python和MPI实现上述的不同通信模式

1. Matrix类

Matrix的值用一个python的二维列表来存放。

Matrix具有multiply和add方法，都是直接用for loop实现的。

后面如果涉及到求矩阵的积或者和，都会使用这两个方法。

use_np这个参数制定了是否使用numpy加速底层运算，如果为True，multiply和add方法都会使用numpy加速运算。

Matrix有一个split_blocks方法，输入q，会将原本的方阵分割成(qxq)个块，然后按照行优先的顺序添加到一个列表中，最后返回该列表

```
import random
from tqdm import tqdm
import numpy as np

class Matrix:
    def __init__(self,row,col=None,use_np=False):
        self.use_np = use_np
        if col:
            self.shape = (row,col)
        else:
            n = row
            self.shape = (n,n)
        self.matrix = [[0]*self.shape[1] for _ in range(self.shape[0])]

    def set_value(self,low=0,high=10):
        n = self.shape[0]
        for i in range(n):
            for j in range(n):
                self.matrix[i][j] = random.randint(low,high)

    def print(self):
        max_len = max(len(str(self.matrix[i][j])) for i in range(self.shape[0])
for j in range(self.shape[1]))
        row_format = "{:>" + str(max_len) + "}"

        for i in range(self.shape[0]):
            row_str = [row_format.format(self.matrix[i][j]) for j in
range(self.shape[1])]
            print(" ".join(row_str))

    def multiply(self, other_matrix):
        if self.shape[1] != other_matrix.shape[0]:
            print("Invalid matrix dimensions for multiplication")
            return None

        result_matrix = Matrix(self.shape[0],
other_matrix.shape[1],use_np=self.use_np)

        if self.use_np:
            A = np.array(self.matrix)
            B = np.array(other_matrix.matrix)
            Prod = np.dot(A,B)
            result_matrix.matrix = Prod.tolist()
            return result_matrix


        for i in tqdm(range(self.shape[0]),desc='multiply'):
        #for i in range(self.shape[0]):
            for j in range(other_matrix.shape[1]):
```

```
                for k in range(self.shape[1]):
                    result_matrix.matrix[i][j] += self.matrix[i][k] *
other_matrix.matrix[k][j]

        return result_matrix


    def add(self,other_matrix):
        if self.shape != other_matrix.shape:
            print('A.add(B) requires the same shape')
            return None

        sum = Matrix(self.shape[0],self.shape[1],use_np=self.use_np)

        if self.use_np:
            A = np.array(self.matrix)
            B = np.array(other_matrix.matrix)
            sum.matrix = (A + B).tolist()
            return sum

        for r in range(self.shape[0]):
            for c in range(self.shape[1]):
                sum.matrix[r][c] += self.matrix[r][c] + other_matrix.matrix[r]
[c]

        return sum


    def split_blocks(self,q):
        # matrix will be split into (q x q) blocks
        # p = q*q is the number of processes

        # only works for square matrix
        assert self.shape[0] == self.shape[1], "Only square matrix can be split
into blocks"

        block_size = self.shape[0]//q
        blocks = []
        for i in range(q):
            for j in range(q):

                block_ij = Matrix(block_size,use_np=self.use_np)
                for row in range(block_size):
                    block_ij.matrix[row] = self.matrix[i*block_size+row]
[j*block_size:(j+1)*block_size]

                blocks.append(block_ij)

        return blocks
```

初始化的时候， root进程 (rank=0) 会创建完整的矩阵A和矩阵B

```
    comm = MPI.COMM_WORLD
```

```
    rank = comm.Get_rank()
    num_proc = comm.Get_size()
    if rank == 0:
        # initialize A and B
        print(f'num_proc = {num_proc}')
        print(f'shape=({n},{n})')
        A = Matrix(n)
        A.set_value()
        B = Matrix(n)
        B.set_value()

    else:
        A = None
        B = None
```

2. 不同通信模式的实现

1. **Broadcast**

   代码见文件mm_bcast.py

   通过bcast，直接将A,B广播给所有的进程

```
    # Broadcast A,B to all processes
    A = comm.bcast(A,root=0)
    B = comm.bcast(B,root=0)
```

先获取当前进程的二维索引(i,j)， 然后从完整的A,B中取出自己需要的A_row_i和B_row_j

最后调用A_row_i.multiply（B_row_j）来计算局部积

```
    # get process index (i,j)
    i,j = 0,0
    q = int(num_proc**0.5)
    i = rank //q
    j = rank % q
    block_size = n//q

    # calculate block C(i,j) = \Sum_{k=0}^{q-1} A(i,k)*B(k,j)
    Cij = Matrix(block_size)

    mul_start = time.time()

    # get local strip A_row and B_col
    A_row_i = Matrix(block_size,n)
    A_row_i.matrix = A.matrix[i*block_size:(i+1)*block_size]
    B_col_j = Matrix(n,block_size)
    for r in range(n):
        B_col_j.matrix[r] = B.matrix[r][j*block_size:(j+1)*block_size]

    Cij = A_row_i.multiply(B_col_j)
```

最终通过gather来获取所有的块，以重构出完整的矩阵, 以下的所有通信模式都是这样收集最终结果，下面都会省略掉

```
    # Gather Cij from all processes
    C_blocks = comm.gather(Cij.matrix, root=0)
```

2. **Scatter Strip**

代码见文件mm_strip.py

root进程给node(i,j) 准备好A_row_i，和B_col_j

然后将元组(A_row_i, B_col_j)添加到scatter_buf

最后由root将 scatter_buf 里面的内容分发给各个子进程。

子进程从接收缓冲区获取A_row_i和B_col_j，然后调用A_row_i.multiply（B_row_j）来计算局部积 Cij

```
    if rank == 0:
        #-------prepare strips for scatter------
        scatter_buf = []
        for i in range(q):
            for j in range(q):
                A_row_i = Matrix(block_size,n)
                B_col_j = Matrix(n,block_size)
                A_row_i.matrix = A.matrix[i*block_size:(i+1)*block_size]
                for r in range(n):
                    B_col_j.matrix[r] = B.matrix[r][j*block_size:
(j+1)*block_size]
                scatter_buf.append((A_row_i,B_col_j))
    else:
        scatter_buf = None

    # -------scatter local region--------
    recv_buf = comm.scatter(scatter_buf,root=0)
    A_row_i,B_col_j = recv_buf
    # calculate local block Cij = A_i * B_j (A_i and B_j are strips)
    Cij = A_row_i.multiply(B_col_j)
```

3. **Block Exchange**

代码详见mm_block.py

首先由root进程给每个子进程准备对应的块 $A_{ij}B_{ij}$

然后通过scatter模式分发给对应的子进程。

```
    if rank == 0:
        #------- prepare blocks for scatter ------
        scatter_buf = []
        A_blocks = A.split_blocks(q)
```

```
        B_blocks = B.split_blocks(q)

        for i in range(q):
            for j in range(q):
                scatter_buf.append((A_blocks[i*q+j],B_blocks[i*q+j]))
    else:
        scatter_buf = None


    # -------scatter local block--------
    recv_buf = comm.scatter(scatter_buf,root=0)
```

子进程node(i,j)收到自己的局部块 $A_{ij}B_{ij}$ 后，先根据自己的行号 i 构建一个同行节点的通信分组row_comm，然后在该通信分组里面进行alltoall通信，交换各自的block，交换完成后每个节点都可复原出完整的一行A_row_i

同理可以和同一列的节点交换手中的B_{ij}，交换完成后每一个节点可以复原完整的一列B_col_i

```
    # get local block Aij Bij
    A_ij,B_ij = recv_buf
    # exchange Aij with other processes in the same row
    row_number = rank // q
    row_comm = comm.Split(row_number,rank)
    A_row_i = row_comm.alltoall([A_ij]*q)

    # exchange Bij with other processes in the same col
    col_number = rank % q
    col_comm = comm.Split(col_number,rank)
    B_col_j = col_comm.alltoall([B_ij]*q)
```

按照如下公式：

$$C_{i,j} = \sum_{k=0}^{q-1} A_{i,k}B_{k,j}$$

遍历完成C_{ij}的运算

```
    Cij = Matrix(block_size)
    for k in range(q):
        Cij = Cij.add(A_row_i[k].multiply(B_col_j[k]))
```

4. **Cannon**

代码详见mm_cannon.py

首先获取原始的分块，调用Matrix.split_blocks方法，获得一个一维列表，按照行优先的顺序存放了各个块

然后需要按照Cannon算法进行块的移位

将块 $A_{i,j},(0\leq i,j\leq\sqrt{p})$ 向左循环移动 i 步；将块 $B_{i,j},(0\leq i,j\leq\sqrt{p})$ 向上循环移动 j 步；

```
    if rank == 0:

        #------- prepare blocks for scatter ------
        scatter_buf = []
        A_blocks = A.split_blocks(q)
        B_blocks = B.split_blocks(q)

        #------- circular shift of A_blocks
        A_blocks_shift = [None]*num_proc
        for r in range(q):
            for c in range(q):
                c_shift = (c - r + q)%q
                A_blocks_shift[r*q + c_shift] = A_blocks[r*q+c]
        #------- circular shift of B_blocks
        B_blocks_shift = [None]*num_proc
        for c in range(q):
            for r in range(q):
                r_shift = (r - c + q)%q
                B_blocks_shift[r_shift*q + c] = B_blocks[r*q+c]


        for i in range(q):
            for j in range(q):

scatter_buf.append((A_blocks_shift[i*q+j],B_blocks_shift[i*q+j]))
    else:
        scatter_buf = None
```

首先由root向各个其它节点分发移动好了的局部块

然后所有处理器开始运算当前两个块的积，保存到Cij中，

然后将块 $A_{i,j},(0\leq i,j\leq\sqrt{p})$ 向左循环移动 1 步； 将块 $B_{i,j},(0\leq i,j\leq\sqrt{p})$ 向上循环移动 1 步；

（这里的移动是通过向相邻的处理器发送和接受块来做到的）

循环执行了q次，但是最后一次计算完后不需要再发送块，可以直接退出循环

```
    # -------scatter local block (Aij,Bij) --------
    recv_buf = comm.scatter(scatter_buf,root=0)


    # get local block Aij Bij
    A_ij,B_ij = recv_buf

    A_ik,B_kj = A_ij,B_ij
```

```
    Cij = Matrix(block_size)

    row = rank //q
    col = rank % q

    mm_start = time.time()
    for k in range(q):
        Cij = Cij.add(A_ik.multiply(B_kj))

        if k == q-1:
            break

        # send current A_ik to the left process and receive new A_ij from the
right
        dest_rank = row*q + (col - 1 + q)%q
        src_rank  = row*q + (col +1)%q
        comm.send(A_ik, dest_rank)
        A_ik = comm.recv(source=src_rank)
        # update A_ik with newly received block

        # send current B_kj to the upper process and receive new B_ij from below
        dest_rank = ((row -1 +q)%q)*q + col
        src_rank  = ((row+1)%q)*q + col
        comm.send(B_kj,dest_rank)
        B_kj =comm.recv(source=src_rank)
        # update B_kj with newly received block
```

注意到这里使用的通信是阻塞式的comm.send comm.recv，有可能导致死锁

可以用非阻塞式的通信 isend irecv 来避免死锁：

```
        # send current A_ik to the left process and receive new A_ij from the
right
        dest_rank = row*q + (col - 1 + q)%q
        src_rank  = row*q + (col +1)%q
        req1 = comm.isend(A_ik, dest_rank)
        req2 = comm.irecv(source=src_rank)
        req1.wait()
        A_ik = req2.wait()
        # update A_ik with newly received block

        # send current B_kj to the upper process and receive new B_ij from below
        dest_rank = ((row -1 +q)%q)*q + col
        src_rank  = ((row+1)%q)*q + col
        print(f'rank{rank}, try send B_kj to dest_rank={dest_rank}')
        req1 = comm.isend(B_kj,dest_rank)
        print(f'rank{rank}, try recv B_kj from src_rank={src_rank}')
        req2 =comm.irecv(source=src_rank)
        req1.wait()
        B_kj = req2.wait()
        # update B_kj with newly received block
```

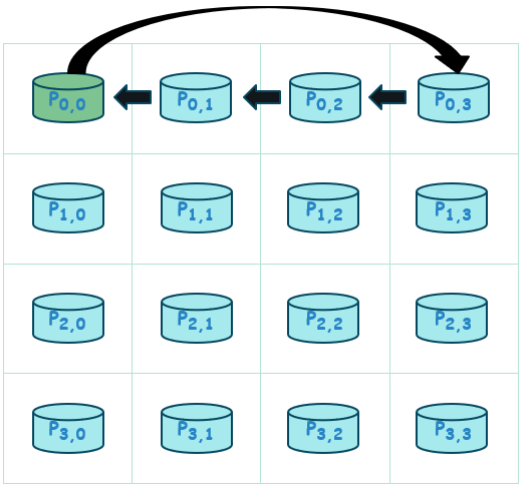
但是这种模式会遇到buffer的瓶颈：

当N = 300就会发生如下报错：超出了缓冲区

```
  File "mpi4py\MPI\Request.pyx", line 266, in mpi4py.MPI.Request.wait
  File "mpi4py\MPI\Request.pyx", line 266, in mpi4py.MPI.Request.wait
    A_ik = req2.wait()
  File "mpi4py\MPI\Request.pyx", line 266, in mpi4py.MPI.Request.wait
  File "mpi4py\MPI\msgpickle.pxi", line 450, in mpi4py.MPI.PyMPI_wait
  File "mpi4py\MPI\msgpickle.pxi", line 450, in mpi4py.MPI.PyMPI_wait
  File "mpi4py\MPI\msgpickle.pxi", line 450, in mpi4py.MPI.PyMPI_wait
mpi4py.MPI.Exception: Message truncated, error stack:
MPI_Wait(request=0x00000142294C0E18, status0x00000039E21EF588) failed
Message from rank 1 and tag 0 truncated; 80869 bytes received but buffer size is
32768
mpi4py.MPI.Exception: Message truncated, error stack:
MPI_Wait(request=0x000002169F0BF158, status0x000000E0CD5EEEB8) failed
Message from rank 2 and tag 0 truncated; 80869 bytes received but buffer size is
32768
mpi4py.MPI.Exception: Message truncated, error stack:
MPI_Wait(request=0x000001D8A618F458, status0x0000001D9A9EF1E8) failed
Message from rank 0 and tag 0 truncated; 80869 bytes received but buffer size is
32768
```

所以可以通过改变send和recv的顺序来打破同一行（同一列）之间的环：

对于同一行的处理器，它们需要循环传递手中的A块，如果顺序全部都是阻塞式的先发后收，会造成死锁



解决办法是，改变其中一个节点的收发顺序，打破上述的环。

下面的代码将每一行的第一个节点的行为改成先接受后发送，每一列同理。

```
        # send current A_ik to the left process and receive new A_ij from the
right
```

```
        dest_rank = row*q + (col - 1 + q)%q
        src_rank  = row*q + (col +1)%q
        if(col == 0):
            new_A_ik = comm.recv(source=src_rank)
            comm.send(A_ik, dest_rank)
            A_ik = new_A_ik
        else:
            comm.send(A_ik, dest_rank)
            A_ik = comm.recv(source=src_rank)
        # update A_ik with newly received block


        # send current B_kj to the upper process and receive new B_ij from below
        dest_rank = ((row -1 +q)%q)*q + col
        src_rank  = ((row+1)%q)*q + col
        if(row == 0):
            new_B_kj =comm.recv(source=src_rank)
            comm.send(B_kj,dest_rank)
            B_kj = new_B_kj
        else:
            comm.send(B_kj,dest_rank)
            B_kj =comm.recv(source=src_rank)
        # update B_kj with newly received block
```

3. 脚本使用介绍

提供了如下命令行参数:

- --N 矩阵的大小（行数）
- --v 是否显示A, B ,C（用于debug和演示）
- --np 是否使用numpy加速底层的运算

```
    parser = argparse.ArgumentParser()
    parser.add_argument('--N',type=int,default=8,help='size of the square
matrix')
    parser.add_argument('--v',default=False,action='store_true',help='whether to
print the matrix')
    parser.add_argument('--np',default=False,action='store_true',help='whether
to use numpy')
```

四、实验结果

为了计算并行加速比，主函数中还会由 root 执行一次串行的矩阵乘法 A.multiply(B)，并保存其计算时间。

最后显示串行和并行算法的运行时间和加速比。

为了检验并行算法的正确性，计算结果会由分块的结果重构，并和串行运算的结果相比较。

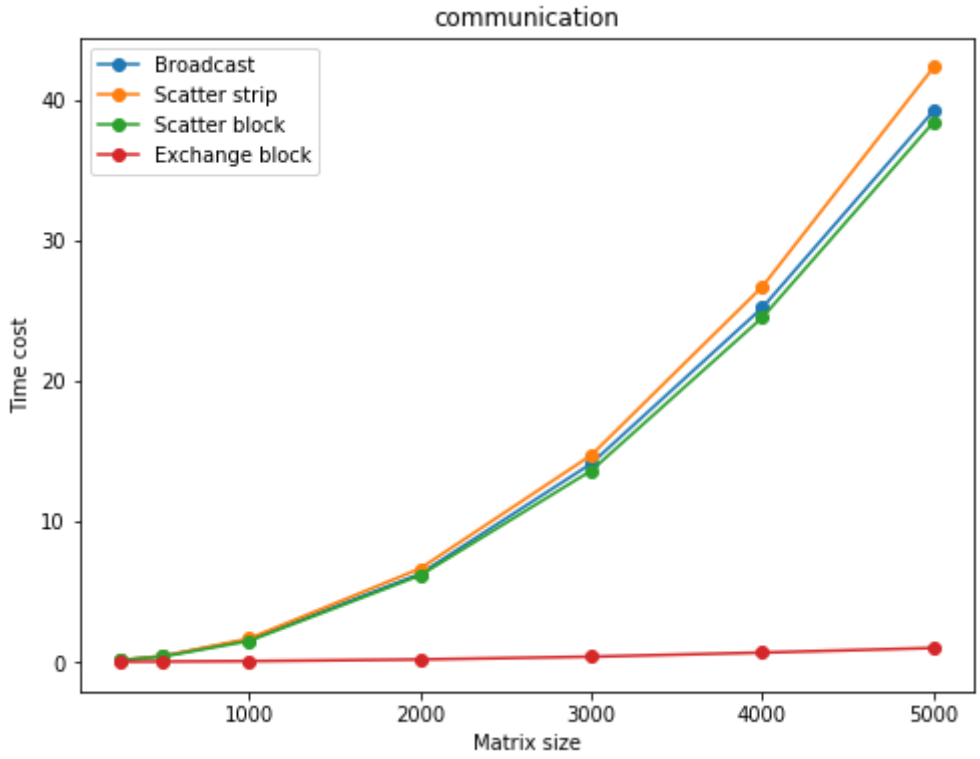
```
    # Reconstruct the final matrix on rank 0
    if rank == 0:
        C = Matrix(n)
        for i in range(q):
            for j in range(q):
                for row in range(i*block_size,(i+1)*block_size):
                    C.matrix[row][j*block_size:(j+1)*block_size] =
C_blocks[i*q+j][row%block_size]
        end = time.time()
        parallel_time = end - start
        start = time.time()
        seq_C = A.multiply(B)
        end = time.time()
        seq_time = end - start
        print(f'parallel:\t{parallel_time}s')
        print(f'sequential:\t{seq_time}s')
        print(f'C equal:{C.matrix == seq_C.matrix}')
        print(f'acceleration ratio:{seq_time/parallel_time}')
```

1. 通信成本

以下为9处理器模式下测得的通信时间（单位：s）

通信开销（s）	**250**	**500**	**1000**	**2000**	**3000**	**4000**	**5000**
BroadCast	0.105	0.405	1.57	6.26	14.1	24.9	39.1
Scatter Strip	0.115	**0.41**	**1.65**	**6.63**	**14.7**	**26.7**	**42.3**
Scatter Block	0.103	0.39	1.53	6.14	13.8	24.5	38.9
Block Exchange	0.0045	0.012	0.043	0.16	0.36	0.65	0.98

各类通信成本随矩阵尺寸的趋势图如下：



很明显，在本机中，MPI的各类通信模式的通信成本非常接近，但是scatter strip的通信成本比其它模式稍高一些。

Exchange block这种模式相比其它模式而言，通信成本非常低

2.并行加速比

按照前面的演示，不同模式的通信成本非常接近，因而其加速比应该十分类似。

下面在同一实验设置下比较不同模式的并行加速比。

设置：9个进程，矩阵大小：（300x300）

通信模式	运行时间	加速比
BroadCast	1.0204s	4.13
Scatter Strip	1.0059s	4.15
Block Exchange	1.0009s	4.35
Cannon	1.0174s	4.24

可以发现在本机上运行，MPI的不同通信模式的加速比在误差范围内可以认为是相同的。

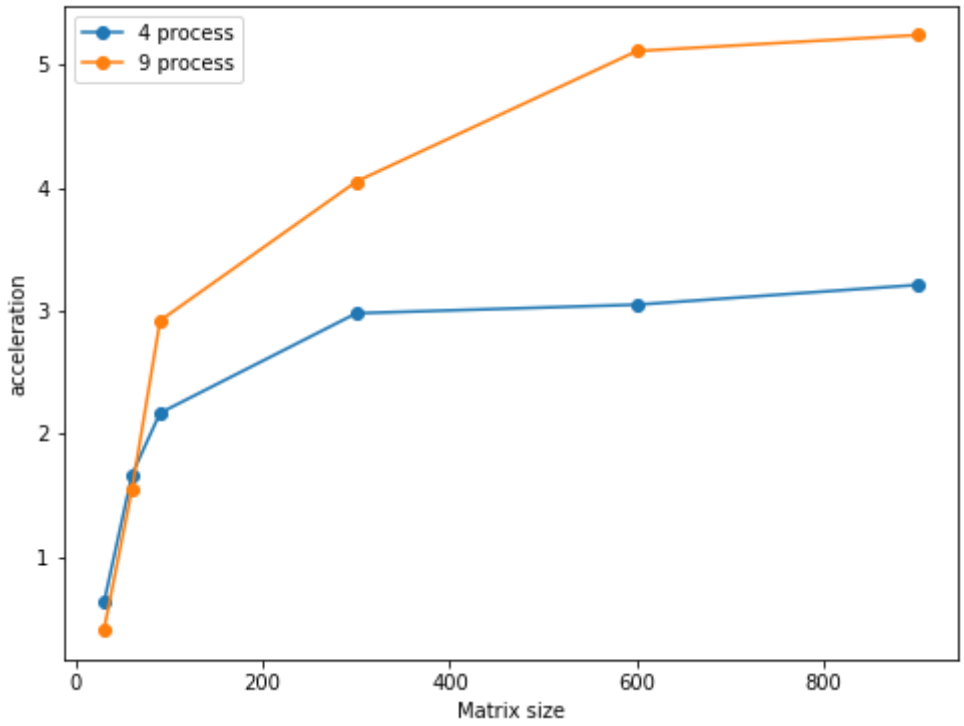
不同数据设置下的加速比（应用Cannon算法）

进程数/矩阵行数	30	60	90	300	600	900
4	0.63	1.67	2.17	2.98	3.05	3.21
9	0.41	1.55	2.92	4.05	5.11	5.24

趋势如下图，可以发现，通信成本在小规模的运算下不可忽略，但是大规模运算的时候，可以达到明显的加速

4核的情况下，加速比可以达到3以上

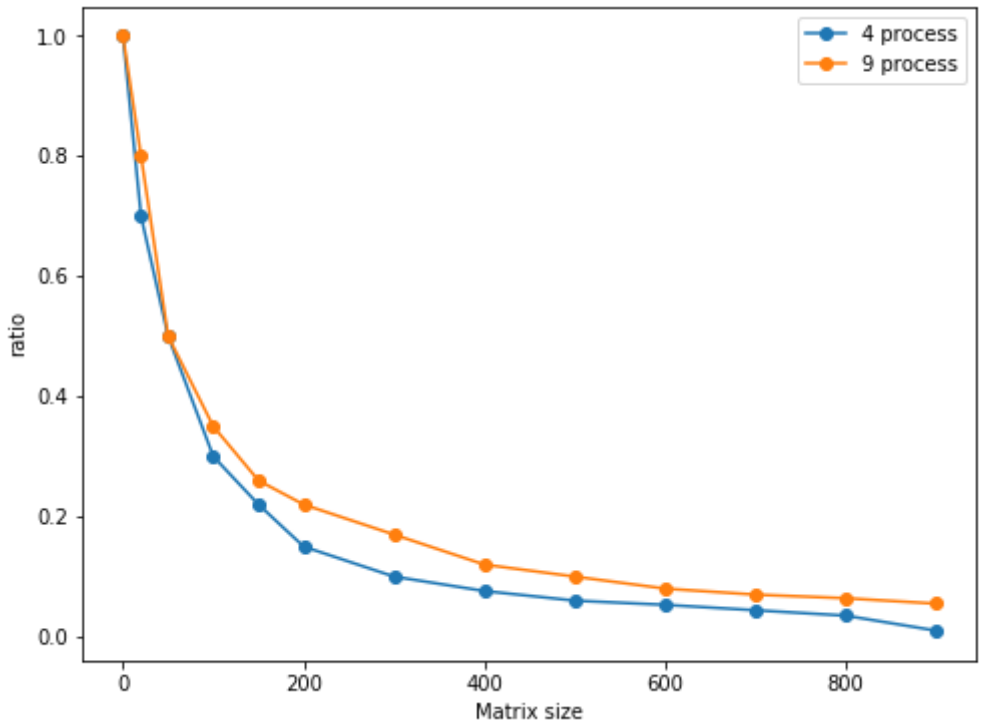
9核的情况下，加速比可以达到5以上



用Strip模式测得通信成本占比随数据规模的趋势如下：

通信成本占比指通信时间占并行算法总时间的比值

进程数/矩阵行数	20	50	100	150	200	300	400	500	600	700	800	900
4	0.7	0.5	0.3	0.22	0.15	0.10	0.076	0.06	0.053	0.044	0.035	0.01
9	0.8	0.5	0.35	0.26	0.22	0.17	0.12	0.10	0.08	0.07	0.064	0.055



五、总结与反思

本次实验用 MPI 的不同通信模式实现了并行矩阵乘法，分别称呼如下：

- Broadcast
- Scatter strip
- Block Exchange
- Cannon

在本机设置下，上述四种算法的性能差别不大，因为本机上四种通信模式的成本非常相似。

由于通信成本的占比随着数据规模的增大而降低，并行加速比会逐渐增大

- 9核情况可以达到超过5倍的加速比
- 4核情况可以达到超过3倍的加速比