

并行程序设计

柯有华

2019.4.16

MPI是什么

- **消息传递并程序序设计**

- 指用户必须通过显式地发送和接收消息来实现处理机间的数据交换。
- 在这种并行编程中，每个并行进程均有自己独立的地址空间，相互之间访问不能直接进行，必须通过显式的消息传递来实现。
- 这种编程方式是大规模并行处理机（MPP）和机群（Cluster）采用的主要编程方式。

- **并行计算粒度大，特别适合于大规模可扩展并行算法**

- 由于消息传递程序设计要求用户很好地分解问题,组织不同进程间的数据交换,并行计算粒度大,特别适合于大规模可扩展并行算法.

- **消息传递是当前并行计算领域的一个非常重要的并程序序设计方式**

mpi4py安装

- 1、下载 [Microsoft MPI v10.0](#)的exe文件，然后进行安装。找到其bin文件夹，一般在C:\Program Files\Microsoft MPI\Bin\，将此地址添加进环境变量。在cmd下命令： mpiexec，若未报错则安装成功。
- 2、python安装mpi4py包： pip install mpi4py。一定要先1后2，不然可能会出错。

mpi4py安装测试

```
from mpi4py import MPI #test文件
comm = MPI.COMM_WORLD
rank = comm.rank
size=comm.size
print("process %d of %d:%s" %(rank,size,'Hello world!'))
```

MPI 提供了下列函数来回答这些问题:

- 用MPI.COMM_WORLD表示我们的程序的交流组
- 用**comm.size**获得进程个数p
- 用**comm.rank** 获得进程的一个叫**rank**的值, 该**rank**值为0到p-1间的整数,相当于进程的ID

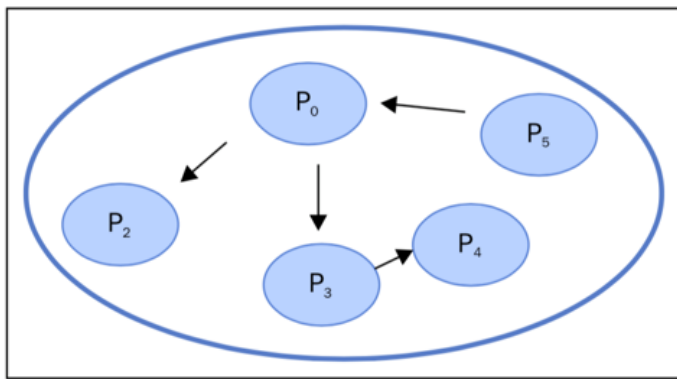
运行mpi程序

```
(tensorflow-gpu) C:\Users\USER\mpi\MPI4PY\多线程和并行\mpi4py>mpiexec -n 5 python test.py  
process 1 of 5:Hello world!  
process 2 of 5:Hello world!  
process 3 of 5:Hello world!  
process 4 of 5:Hello world!  
process 0 of 5:Hello world!
```

其中mpiexec -n 后面接的是我们要运行的进程数，即我们的comm.size

设计MPI程序

- 在写MPI程序时，我们常需要知道以下两个问题的答案：
 - 我是谁：我是哪一个进程
 - 我在干嘛：给每个进程分配相应的任务

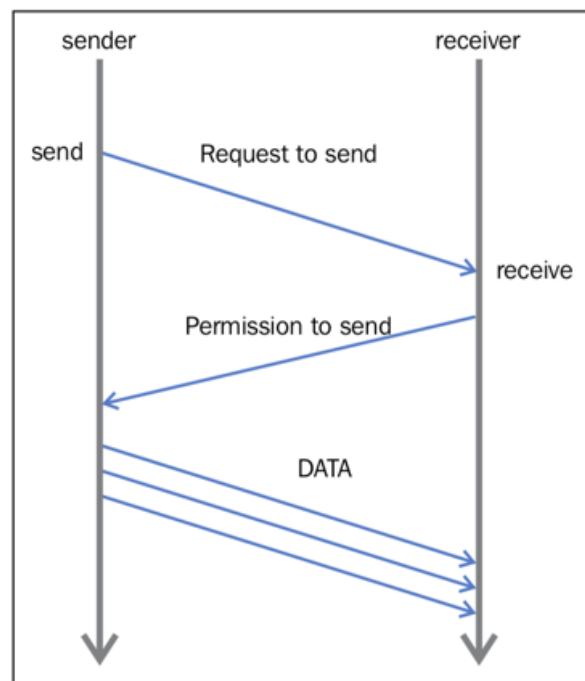


An example of communication between processes in MPI.COMM_WORLD

点到点通信

MPI提供的最实用的一个特性是点对点通讯。

两个不同的进程之间可以通过点对点通讯交换数据：一个进程是接收者，一个进程是发送者。



The send/receive transmission protocol

阻塞通信

- 阻塞通信是指消息发送方的 send 调用需要接收方的 recv 调用的配合才可完成。即在发送的消息信封和数据被安全地“保存”起来之前，send 函数的调用不会返回。标准模式的阻塞 send 调用要求有接收进程的 recv 调用配合则发送的顺序与接收顺序严格匹配
- 下面是 mpi4py 中用于标准阻塞点到点通信的方法接口：
 - 1、`send(self, obj, int dest, int tag)` #obj要可被pickle化
`recv(self, buf=None, int source=ANY_SOURCE, int tag)`
 - 2、`Send(self, buf, int dest, int tag)` #buf为类数组的大数据
`Recv(self, buf, int source=ANY_SOURCE, int tag)`

阻塞通信

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    print('process %d sends %s' % (rank, data))
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print('process %d receives %s' % (rank, data))
```

运行结果为：

process 0 sends {'a': 7, 'b': 3.14}

process 1 receives {'a': 7, 'b': 3.14}

阻塞通信

```
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
count = 10
send_buf = np.arange(count, dtype='i')
recv_buf = np.zeros(count, dtype='i')
if rank == 0:
    comm.Send([send_buf, count, MPI.INT], dest=1, tag=11)
    comm.Recv([recv_buf, count, MPI.INT], source=1, tag=22)
elif rank == 1:
    comm.Recv([recv_buf, count, MPI.INT], source=0, tag=11)
    comm.Send([send_buf, count, MPI.INT], dest=0, tag=22)
print('process %d receives %s' % (rank, recv_buf))
```

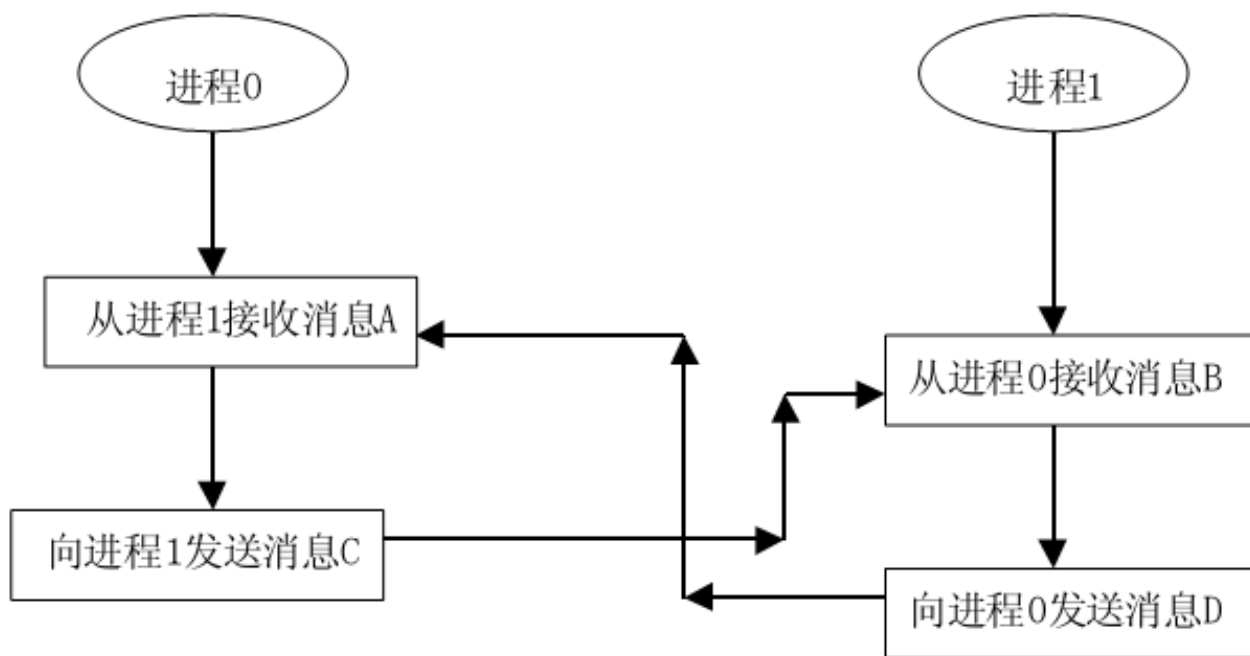
运行结果:

process 0 receives [0 1 2 3 4 5 6 7 8 9]

process 1 receives [0 1 2 3 4 5 6 7 8 9]

deadlock

发送和接收是成对出现的，忽略这个原则 很可能会产生死锁



非阻塞通信

- 用户发送缓冲区的重用:
 - 非阻塞的发送：仅当调用了有关结束该发送的语句后才能重用发送缓冲区，否则将导致错误；对于接收方，与此相同，仅当确认该接收请求已完成后才能使用。**所以对于非阻塞操作，要先调用等待MPI_Wait()或测试MPI_Test()函数来结束或判断该请求，然后再向缓冲区中写入新内容或读取新内容。**

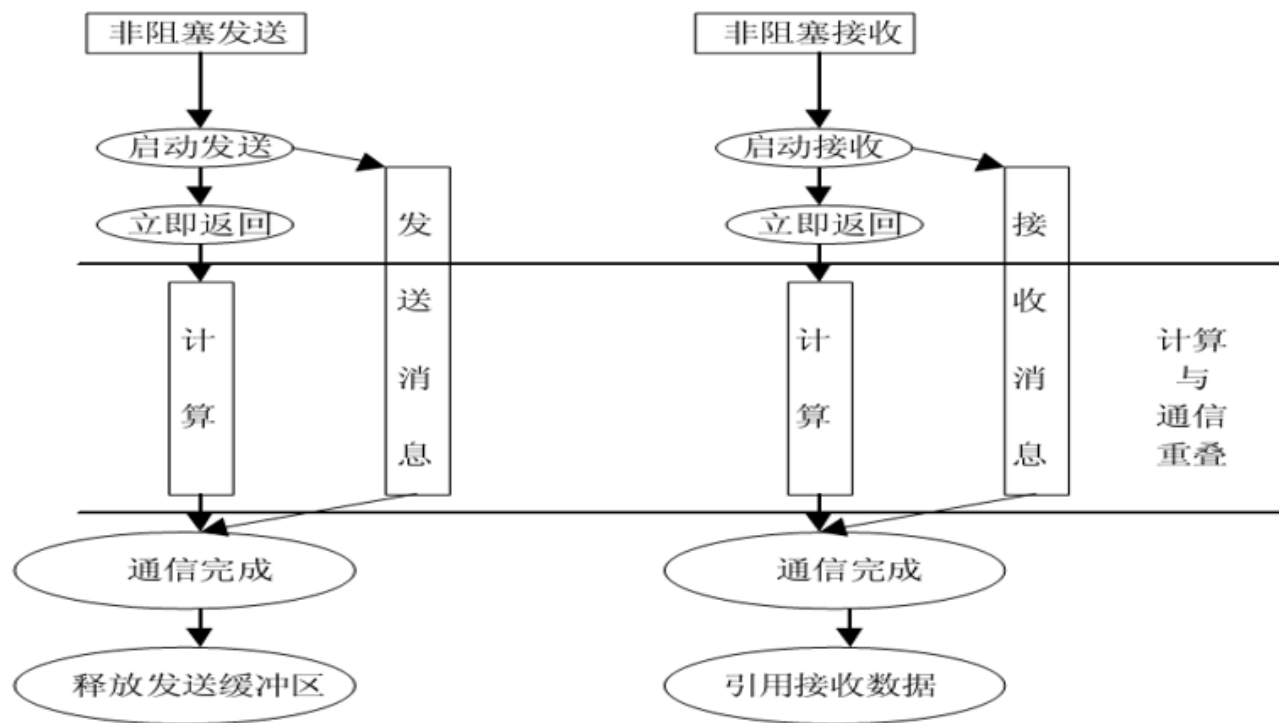
想象在一辆载满乘客的大巴上：

- 1、司机过程中定时询问每个乘客是否到达目的地，若有人说到了，那么司机停车，乘客下车。（类似阻塞式）
- 2. 每个人告诉售票员自己的目的地，然后睡觉，司机只和售票员交互，到了某个点由售票员通知乘客下车。（类似非阻塞）

很显然，每个人要到达某个目的地可以认为是一个线程，司机可以认为是 CPU 。在阻塞式里面，每个线程需要不断的轮询，上下文切换，以达到找到目的地的结果。而在非阻塞方式里，每个乘客（线程）都在睡觉（休眠），只在真正外部环境准备好了才唤醒，这样的唤醒肯定不会阻塞。

非阻塞通信

非阻塞标准发送和接收



非阻塞通信

- 非重复非阻塞的标准通信模式是与[阻塞的标准通信模式](#)相对应的，其通信方法（MPI.Comm 类的方法）接口有一个前缀修饰 I/i，如下：

1、isend(self, obj, int dest, int tag=0)

irecv(self, int source=ANY_SOURCE, int tag=ANY_TAG)

2、lsend(self, buf, int dest, int tag=0)

lrecv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG)

非阻塞通信

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
send_obj = {'a': [1, 2.4, 'abc', -2.3+3.4j],
            'b': {2, 3, 4}}
if rank == 0:
    send_req = comm.isend(send_obj, dest=1, tag=11)
    send_req.wait()
    print('process %d sends %s' % (rank, send_obj))
elif rank == 1:
    recv_req = comm.irecv(source=0, tag=11)
    recv_obj = recv_req.wait()
    print('process %d receives %s' % (rank, recv_obj))
```

运行结果:

process 0 sends {'a': [1, 2.4, 'abc', (-2.3+3.4j)], 'b': set([2, 3, 4])}

process 1 receives {'a': [1, 2.4, 'abc', (-2.3+3.4j)], 'b': set([2, 3, 4])}

非阻塞通信

```
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
count = 10
send_buf = np.arange(count, dtype='i')
recv_buf = np.zeros(count, dtype='i')
if rank == 0:
    send_req = comm.Isend(send_buf, dest=1, tag=11)
    send_req.Wait()
    print('process %d sends %s' % (rank, send_buf))
elif rank == 1:
    recv_req = comm.Irecv(recv_buf, source=0, tag=11)
    recv_req.Wait()
    print('process %d receives %s' % (rank, recv_buf))
```

运行结果：

process 0 sends [0 1 2 3 4 5 6 7 8 9]

process 1 receives [0 1 2 3 4 5 6 7 8 9]

小结

分类		发送	接收	说明
阻塞通信		Send/send	Recv/recv Irecv/irecv Recv_init	如果接收动作使用了 Irecv, Recv_init, 则使用 MPI.Request 对象及其 Wait/wait, Waitall/waitall, Waitany/waitany, Waitsome, Test/test, Testall/testall, Testany/testany, Testsome 方法进行测试
		Bsend/bsend		
		Rsend		
		Ssend/ssend		
非阻塞通信	非重复	Isend/isend	Recv/recv Irecv/irecv Recv_init	需用到 MPI.Request 对象, 以及其 Wait/wait, Waitall/waitall, Waitany/waitany, Waitsome, Test/test, Testall/testall, Testany/testany, Testsome 方法进行测试与运行
		Ibsend/ibsend		
		Irsend		
		Issend/issend		
	重复	Send_init	Recv/recv Irecv/irecv Recv_init	需用到 MPI.Prerequest 对象, 结合其 Start, Startall 方法, 以及 MPI.Request 对象及其 Wait/wait, Waitall/waitall, Waitany/waitany, Waitsome, Test/test, Testall/testall, Testany/testany, Testsome 方法进行测试与运行
		Bsend_init		
		Rsend_init		
		Ssend_init		

MPI:求PI值

利用下面求和公式

$$\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4}$$

我们可以取个很大的整数N，如2**26，以及相应的进程总数p

定义一个函数value=calculate_part(start, step):

Step表示每个进程分配到的数的个数即：N//p

Start表示求和的起始位置：rank*step

$$\text{value} = \sum_{i=\text{rank}*\text{step}}^{\text{step}*(\text{rank}+1)-1} \frac{1}{2*i+1} * (-1)^i$$

每个rank(除主进程0)都求一个其相应的value,并将其send给主进程0，最后在主进程0输出最终的和。

```

from mpi4py import MPI
import time
import numpy as np
def calculate_part(start, step):
    sum=0.0
    flag=1
    for i in range(0,step):
        if(start % 2 != 0):
            flag=-1
        else:
            flag=1
        sum+=flag * (1/(2*start+1))
        start +=1
    return float(sum)

```

```

if __name__ == '__main__':
    N=2**26
    result=0
    t1=MPI.Wtime()
    step = N // size
    start = rank * step
    value=calculate_part(start,step)
    if rank == 0:
        result += value
        for i in range(1,size):
            value = comm.recv(source=i, tag=0)
            result += value
        print('PI is : ',result * 4)
        print('time cost is', MPI.Wtime() - t1, 's')
    else:
        comm.send(value, dest=0, tag=0)

```



```

(tensorflow-gpu) C:\Users\USER\mpi\MPI4PY\多线程和并行\mpi4py>mpiexec -n 10 python compute_pi.py
PI is : 3.1415926386886146
time cost is 11.999848533461773 s

```

串行:

the pi value is: 3.1415926386888584
the time is: 22.141777276992798

集合通信

Collective Communication

- 特点
 - 通信空间中的所有进程都参与通信操作
 - 每一个进程都需要调用该操作函数
- 一到多
- 多到一
- 同步

集合通信

Collective Communication



All:表示结果到**所有**进程.

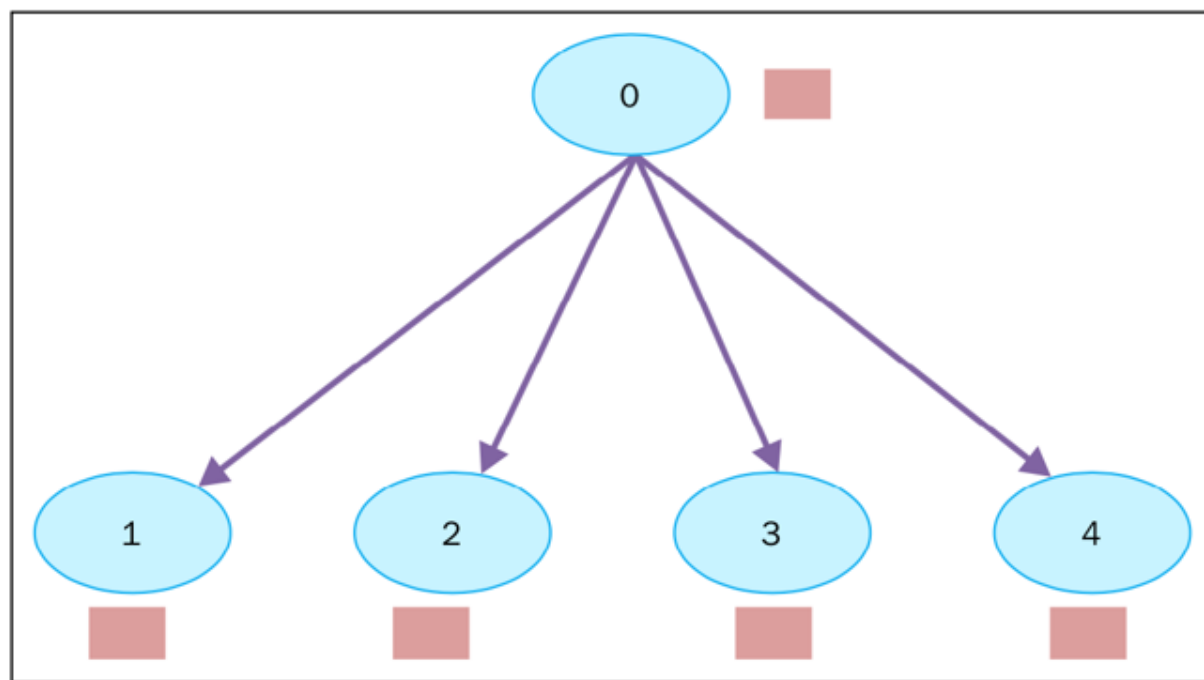
V:Variety,被操作的数据对象和操作更为灵活.

MPI 集合通信函数

类型	函数	功能
数据移动	MPI_Bcast	一到多, 数据广播
	MPI_Gather	多到一, 数据汇合
	MPI_Gatherv	MPI_Gather的一般形式
	MPI_Allgather	MPI_Gather的一般形式
	MPI_Allgatherv	MPI_Allgather的一般形式
	MPI_Scatter	一到多, 数据分散
	MPI_Scatterv	MPI_Scatter的一般形式
	MPI_Alltoall	多到多, 置换数据(全互换)
	MPI_Alltoallv	MPI_Alltoall的一般形式
数据聚集	MPI_Reduce	多到一, 数据归约
	MPI_Allreduce	上者的一般形式, 结果在所有进程
	MPI_Reduce_scatter	结果scatter到各个进程
	MPI_Scan	前缀操作
同步	MPI_Barrier	同步操作

广播(Broadcast)

在并行代码的开发中，我们会经常发现需要在多个进程间共享某个变量运行时的值，或操作多个进程提供的变量：



广播(Broadcast)

```
buf = comm.bcast(data_to_share, rank_of_root_process)
```

data_to_share表示我们要广播的数据, rank_of_root_process表示我们在哪个rank进行广播。

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
if rank == 0:
    variable_to_share = 100
else:
    variable_to_share = None
```

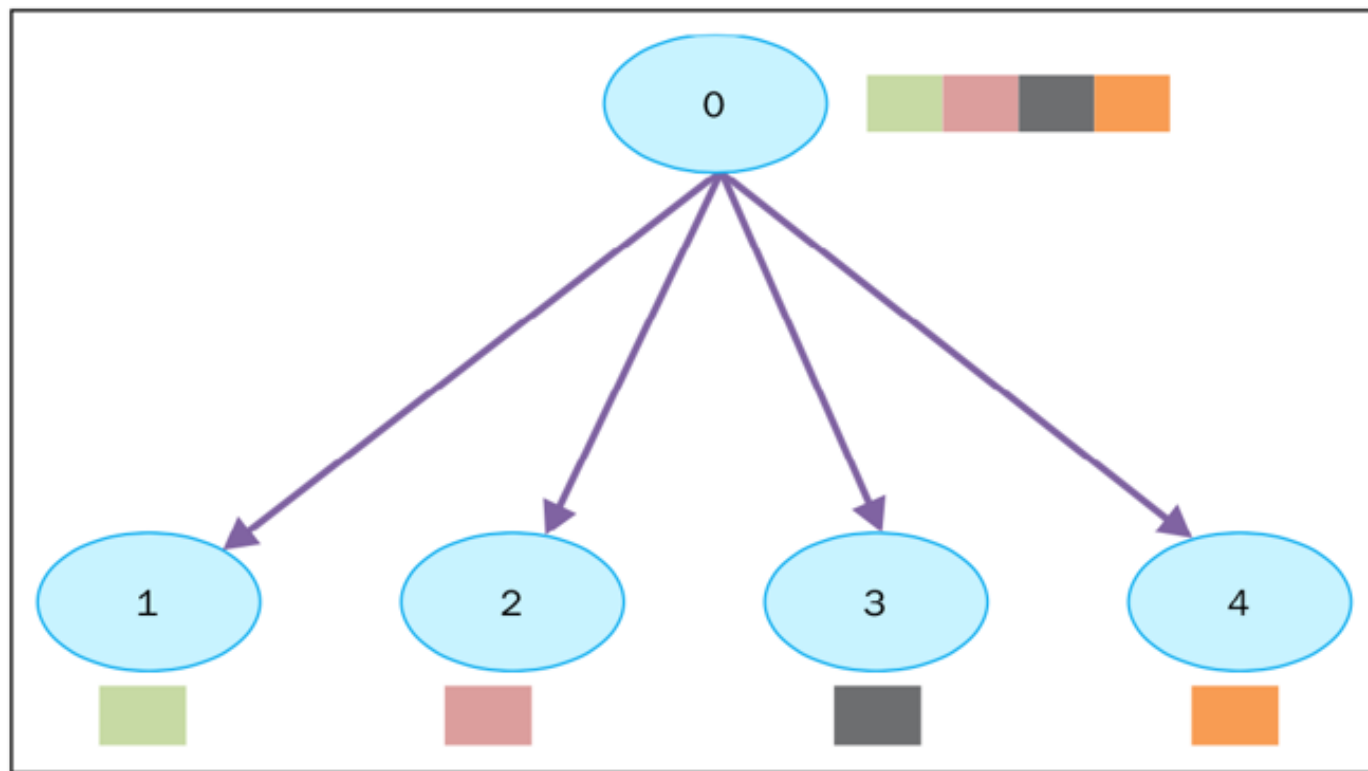
```
variable_to_share = comm.bcast(variable_to_share, root=0)
```

```
print("process = %d" %rank + " variable shared = %d " %variable_to_share)
```

```
process = 0 variable shared = 100
process = 8 variable shared = 100
process = 2 variable shared = 100
process = 3 variable shared = 100
process = 4 variable shared = 100
process = 5 variable shared = 100
process = 9 variable shared = 100
process = 6 variable shared = 100
process = 1 variable shared = 100
process = 7 variable shared = 100
```

发散(scatter)

scatter函数和广播很像，但是有一个很大的不同，`comm.bcast` 将相同的数据发送给所有在监听的进程，`comm.scatter` 可以将数据放在数组中，发送给不同的进程。下图展示scatter的功能：



Scattering data from process 0 to processes 1, 2, 3, 4

发散(scatter)

`recvbuf = comm.scatter(sendbuf, rank_of_root_process)`

Sendbuf一般是个array, 它的维数要和comm.size一样大

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
```

```
rank = comm.Get_rank()
```

```
if rank == 0:
```

```
    array_to_share = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
else:
```

```
    array_to_share = None
```

```
recvbuf = comm.scatter(array_to_share, root=0)
```

```
print("process = %d" %rank + " recvbuf = %d " %recvbuf)
```

process = 0 variable shared = 1

process = 4 variable shared = 5

process = 6 variable shared = 7

process = 2 variable shared = 3

process = 5 variable shared = 6

process = 3 variable shared = 4

process = 7 variable shared = 8

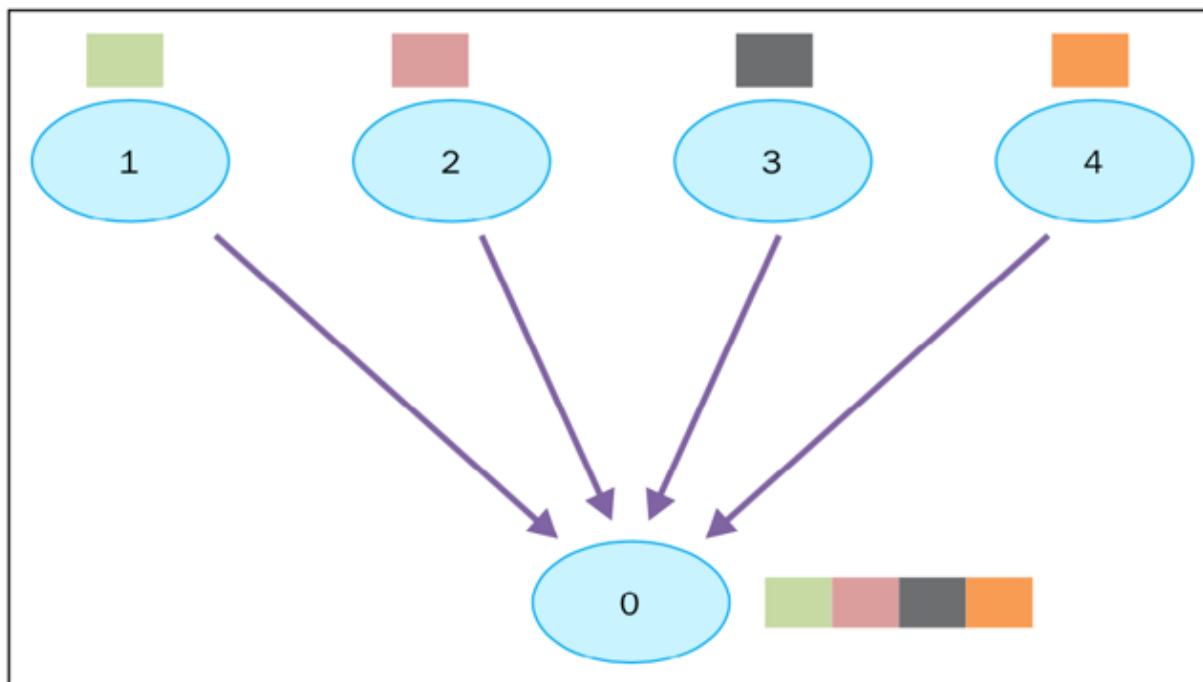
process = 1 variable shared = 2

process = 8 variable shared = 9

process = 9 variable shared = 10

收集(gather)

gather 函数基本上是反向的 scatter，即手机所有进程发送向root进程的数据。

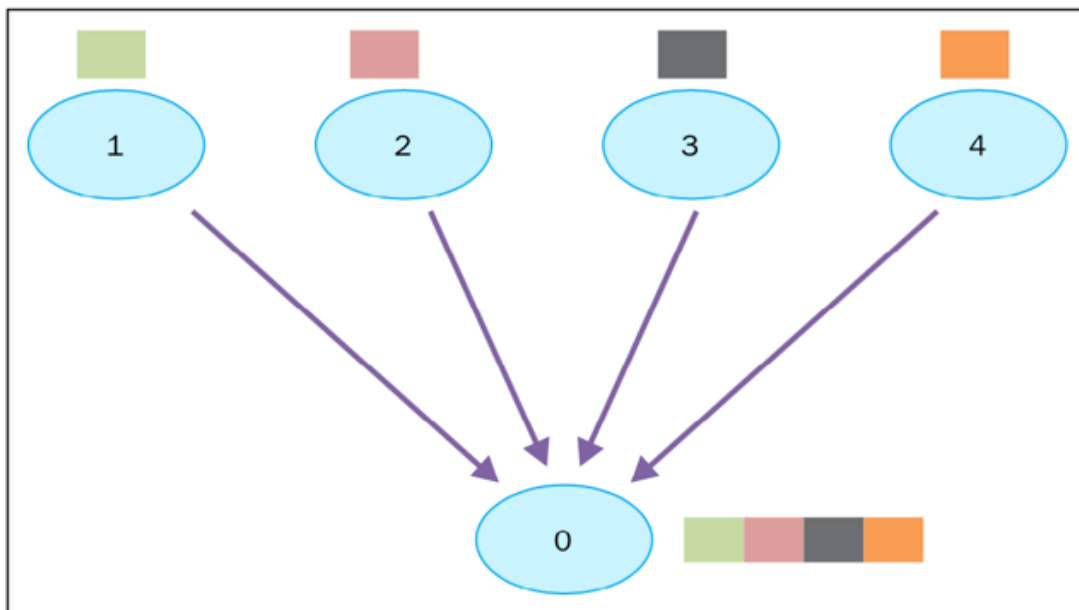


Gathering data from processes 1, 2, 3, 4

收集(gather)

```
recvbuf = comm.gather(sendbuf, rank_of_root_process)
```

这里,sendbuf 是要发送的数据,rank_of_root_process 代表要接收数据进程.recvbuf是一个维数和comm.size一样大的array.



Gathering data from processes 1, 2, 3, 4

Reduce

同 comm.gather 一样， comm.Reduce 接收一个数组， 每一个元素是一个进程的输入， 然后返回一个数组， 每一个元素是进程的输出， 返回给 root 进程。输出的元素中包含了简化的结果。

comm.Reduce(sendbuf, recvbuf, rank_of_root_process, op = type_of_reduction_operation)
sendbuf, recvbuf 的长度要一样

常见op:

MPI.MAX : 返回最大的元素

MPI.MIN : 返回最小的元素

MPI.SUM : 对所有元素相加

MPI.PROD : 对所有元素相乘

MPI.LAND : 对所有元素进行逻辑操作

MPI.MAXLOC : 返回最大值， 以及拥有它的进程

MPI.MINLOC : 返回最小值， 以及拥有它的进程

Reduce

```
import numpy
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.size
rank = comm.rank
array_size = 3
recvdata = numpy.zeros(array_size, dtype=numpy.int)
senddata = (rank+1)*numpy.arange(size, dtype=numpy.int)
print("process %s sending %s " % (rank , senddata))
comm.Reduce(senddata, recvdata, root=0, op=MPI.SUM)
print('on task', rank, 'after Reduce:  data = ', recvdata)
```



process 2 sending [0 3 6] on task 2
after Reduce: data = [0 0 0]
process 1 sending [0 2 4] on task 1
after Reduce: data = [0 0 0]
process 0 sending [0 1 2] on task 0
after Reduce: data = [0 6 12]

其他集体通信

`comm.Alltoall(sendbuf, recvbuf) :`

`Comm.allgather(self, sendobj)`

`Comm.Allgather(self, sendbuf, recvbuf)`

`Comm.Allgatherv(self, sendbuf, recvbuf)`

`Comm.allreduce(self, sendobj, op=SUM)`

`Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)`

allgather

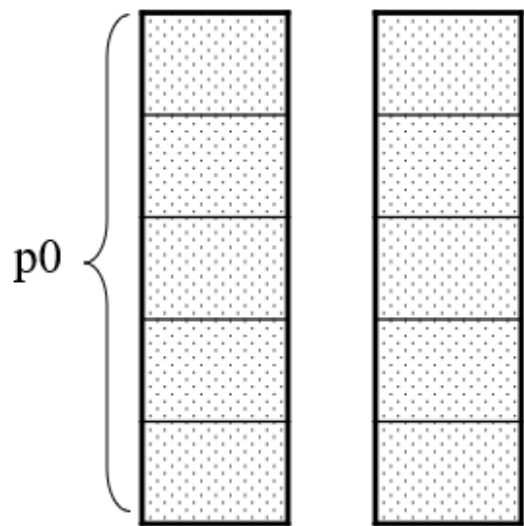
- `from mpi4py import MPI`
- `comm=MPI.COMM_WORLD`
- `rank=comm.rank`
- `b=comm.allgather(rank)`
- `print('my rank is %d,res is' %rank,b)`



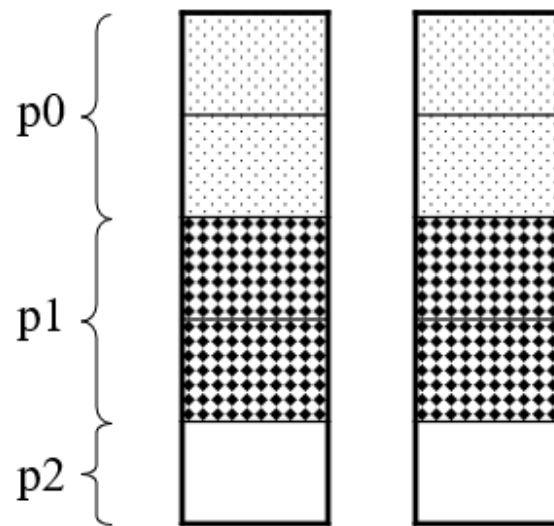
```
my rank is 9,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 5,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 1,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 7,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 3,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 0,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 2,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 4,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 8,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
my rank is 6,res is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

MPI: 向量点乘

$$c = \sum_{i=0}^{n-1} a_i \cdot b_i$$



$$c = \sum_{j=0}^{n/p} \sum_{i=0}^{n_j-1} a_i \cdot b_i$$




```
from mpi4py import MPI
import numpy as np
```

```
N=2**20
```

```
x=[1,2,3]*(N//3)
```

```
y=[1,-2,1]*(N//3)
```

```
def dot_mul(x,y)->'float':
```

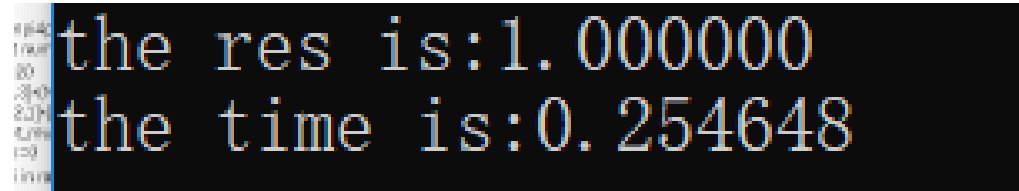
```
    sum=0
```

```
    for i in range(len(x)):
```

```
        sum+=x[i]*y[i]
```

```
    return float(sum)
```

```
if __name__=='__main__':
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    t0=MPI.Wtime()
    res=np.zeros(1)
step=N//size
tmp_x=x[rank*step:(rank+1)*step]
tmp_y=y[rank*step:(rank+1)*step]
value=dot_mul(tmp_x,tmp_y)
value=value*np.ones(1)
comm.Reduce(value,res,root=0,op=MPI.SUM)
    if rank==0:
        print('the res is:%f'%res)
        print('the time is:%f'%(MPI.Wtime()-t0))
```



```
the res is:1.000000
the time is:0.254648
```

MPI:矩阵相乘

$$C=A*B$$

思路一：

$$\sum_{i=0}^{p-1} A_i * B$$

按行将矩阵A分成 p 份，scatter给每个进程 i 。将矩阵B广播给每个进程 i ，scatter给每个进程 i ，每个进程 i 都计算一次 $A_i * B$ ，最后将每个进程的结果gather起来，再进行矩阵的vstack。

MPI:矩阵相乘

```
if __name__ == '__main__':
```

```
    comm = MPI.COMM_WORLD
```

```
    rank = comm.Get_rank()
```

```
    size = comm.Get_size()
```

```
    t1=MPI.Wtime()
```

```
    if rank==0:#设置0进程为主进程
```

```
        A,B=gen_data(2000,1000,200)#生成矩阵A, B
```

```
        group=A.shape[0]//size
```

```
        tmp=[]#将矩阵A进行切片
```

```
        for i in range(size-1):
```

```
            tmp.append(A[group*i:group*(i+1),:])
```

```
        tmp.append(A[group*(size-1):,:])
```

串行
时间



```
recv_A=comm.scatter(tmp,root=0)
```

```
recv_B=comm.bcast(B,root=0)
```

```
tmp_c=matrix_mul(recv_A,recv_B)
```

```
result=comm.gather(tmp_c,root=0)
```

```
if rank==0:
```

```
    result=np.vstack(result)#将最后矩阵进行重排
```

```
    t2=MPI.Wtime()
```

```
    print(result.shape)
```

```
    print('the time is:',t2-t1)
```



```
(tensorflow-gpu) C:\Users\USER\mpi\mpi4py_1>mpiexec -n 10 python matrix_mul.py
my rank is 1
the time is: 1.1726407060123165
my rank is 2
the time is: 1.1251288532239414
my rank is 3
the time is: 1.3322463298154616
my rank is 4
the time is: 1.3852020179620013
my rank is 5
the time is: 1.2169653615219431
my rank is 6
the time is: 1.4549299584596156
my rank is 7
the time is: 1.302828541587587
my rank is 8
the time is: 1.459910434210542
my rank is 9
the time is: 1.2497354668921616
my rank is 0
the time is: 1.3497537282091798
the time is: 1.349906901412396
```

```
In [1]: runfile('C:/Users/USER/mpi/mpi4py_1/matrix_mul.py', wdir='C:/
Users/USER/mpi/mpi4py_1')
the time is 1.5301620960235596
```

MPI:矩阵相乘

思路二：Cannon算法

算法原理 Cannon 算法(Cannon's Algorithm)是一种存储有效的算法。为了使两矩阵的下标满足相乘的要求,它和上一节的并行分块乘法不同,不是阵列的各行和各列施行多到多播送,而是有目的地在各行和各列施行循环移位,从而使处理器的总存储要求可以降下来。照例,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 $(n/\sqrt{p}) \times (n/\sqrt{p})$, 并将它们分配给 $\sqrt{p} \times \sqrt{p}$ 个处理器 ($P_{0,0}, P_{0,1}, \dots, P_{\sqrt{p}-1, \sqrt{p}-1}$)。开始时处理器 $P_{i,j}$ 存放有块 $A_{i,j}$ 和块 $B_{i,j}$, 并负责计算块 $C_{i,j}$, 然后算法开始执行:

① 将块 $A_{i,j}$ ($0 \leq i, j < \sqrt{p}$) 向左循环移动 i 步;

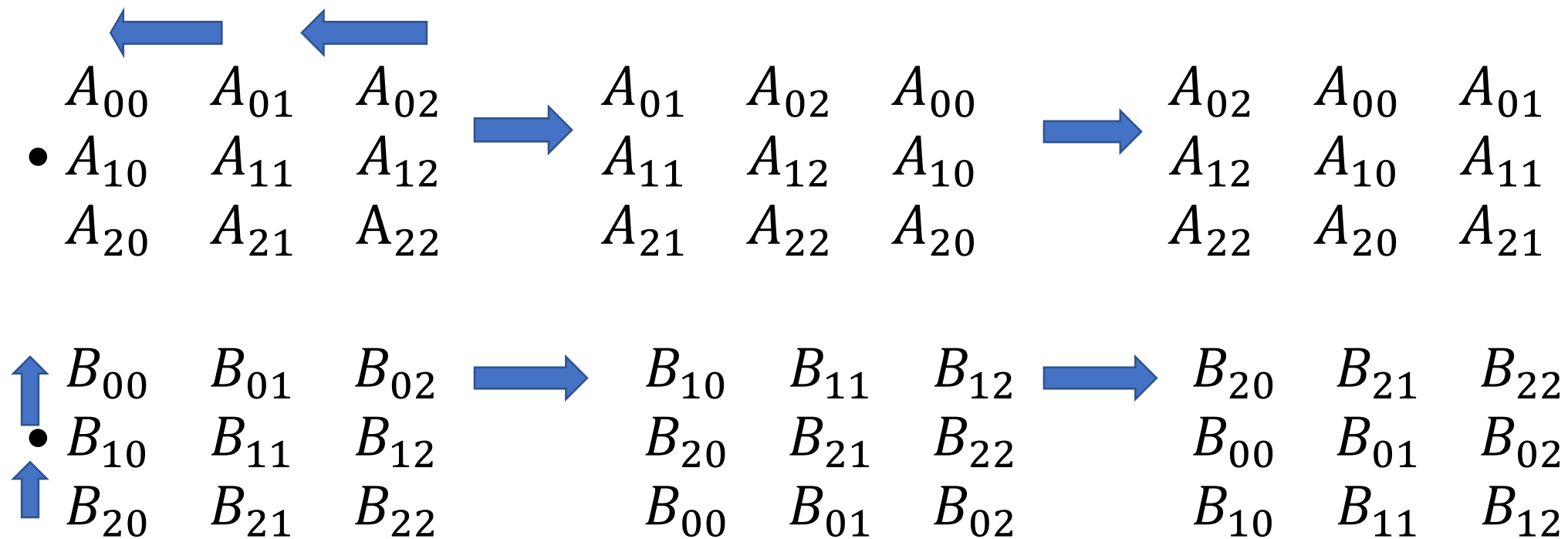
将块 $B_{i,j}$ ($0 \leq i, j < \sqrt{p}$) 向上循环移动 j 步;

② $P_{i,j}$ 执行乘-加运算;

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

将块 $B_{i,j}$ ($0 \leq i, j < \sqrt{p}$) 向上循环移动 1 步;

③ 重复第②步, 在 $P_{i,j}$ 中共执行 \sqrt{p} 次乘-加运算和 \sqrt{p} 次块 $A_{i,j}$ 和 $B_{i,j}$ 的循环单步移位。



这里我们取的 $p=9$ ，A,B都分解成 $\sqrt{p} \times \sqrt{p}=3 \times 3$ 的矩阵块，我们需要移动 $\sqrt{p} - 1 = 2$ 次，若 $C = A * B$ ，我们可以看到：

$$C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20}$$

MPI进程拓扑

在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型。据问题需要，进程经常被排列成二维、三维网格乃至更复杂的图结构上。

MPI 支持笛卡尔拓扑（Cartesian topology）和图拓扑（graph topology）两种。

Cartesian topology

- `Create_cart(self, dims, periods=None, bool reorder=False)`

`dims` 是长度为维数 `ndims` 的整型数组，指出各维的进程数，
`periods` 可取值 `None`, `True`, `False` 或长度为 `ndims` 的布尔数组，
指出各维是否周期性循环，默认值 `None` 表示各维都不循环，
`True/False` 表示各维都循环/都不循环。

布尔型的 `reorder` 指出进程在新创建的通信子组内是否进行重排序，
默认值为 `False`,

Cartesian topology

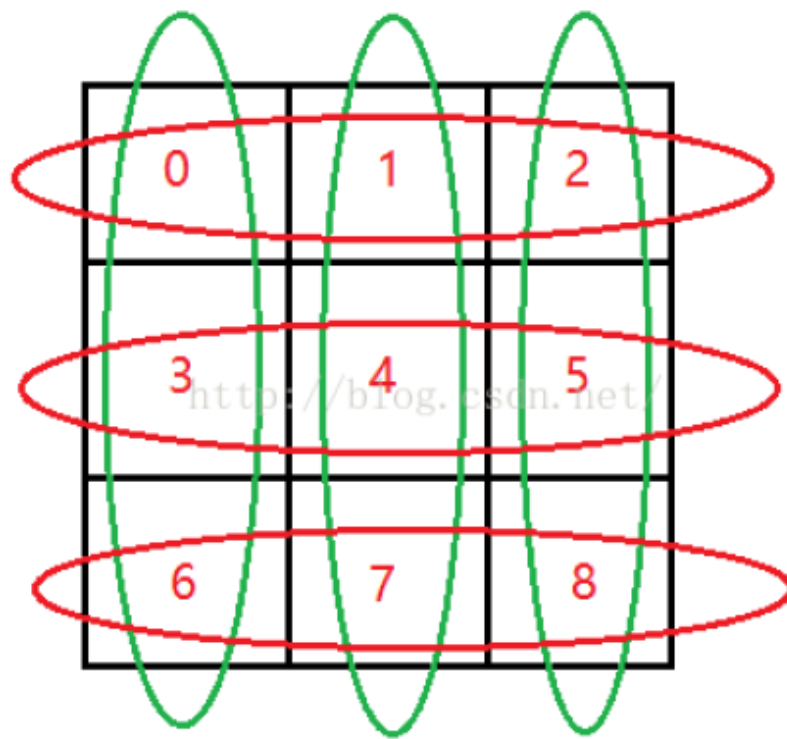
- `Create_cart([3,2],[True,False])`

period = True period = True

-----+-----+-----+-----
0,0 (0) 0,1 (1) period = False
-----+-----+-----+-----
1,0 (2) 1,1 (3) period = False
-----+-----+-----+-----
2,0 (4) 2,1 (5) period = False
-----+-----+-----+-----
(0) (1)

MPI:矩阵相乘

- 建立下面通讯组



```
if rank==0:
    A,B=gen_data(N,size)#将生成的矩阵进行切片
else:
    A,B=None,None
    my_A=comm.scatter(A,root=0)
    my_B=comm.scatter(B,root=0)
    mpi_rows = int(np.sqrt(comm.size))
    ccomm = comm.Create_cart((mpi_rows, mpi_rows), periods=(True, True))
    my_mpi_row, my_mpi_col = ccomm.Get_coords(ccomm.rank)
    my_C = np.zeros_like(my_A)
    tile_A=np.empty_like(my_A)#建立两个buf区
    tile_B=np.empty_like(my_A)
    req = [None, None, None, None]
```

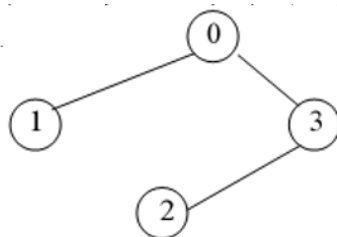
```
rows=[[mpi_rows*i+j for j in range(mpi_rows)] for i in range(mpi_rows)]
cols=[[i+3*j for j in range(mpi_rows)] for i in range(mpi_rows)]
for r in range(1,mpi_rows):
    req[EAST] = ccomm.Isend(tile_A , dest=rows[my_mpi_row][(my_mpi_col-r)%mpi_rows])
    req[WEST] = ccomm.Irecv(tile_A_, source=rows[my_mpi_row][(my_mpi_col+r)%mpi_rows])
    req[SOUTH] = ccomm.Isend(tile_B, dest=cols[my_mpi_col][(my_mpi_row-r)%mpi_rows])
    req[NORTH]=ccomm.Irecv(tile_B_,source=cols[my_mpi_col][(my_mpi_row+r)%mpi_rows])
    req[0].Waitall(req)
    if rank==0:
        print(rank,tile_A_,tile_B_)
        my_C+=np.dot(tile_A_,tile_B_)
comm.barrier()
my_C+=np.dot(my_A,my_B)
res=comm.gather(my_C,root=0)#最后将矩阵拼接起来
```

图拓扑

Create_graph(self, index, edges, bool reorder=False)

index: 存储节点信息

edges: 表示边的信息



结点、度数、边的对应关系

结点编号	结点度数	该结点连接的其它结点
0	2	1, 3
1	1	0
2	1	3
3	2	0, 2

图拓扑的定义参数

结点总数	结点的顺序累计度数	按结点顺序的边列表
nnodes = 4	index = 2, 3, 4, 6	edges = 1, 3, 0, 3, 0, 2

kmeans算法设计

并行设计

初始化：若有 n 个点要聚类，设置进程数 p 为我们要聚类的中心点个数，将前 p 个点作为每个进程 i 的中心点 c_i

并行：

- 1、每个进程都计算各个点到 c_i 的距离 $\text{dist} = [c_{i_0}, c_{i_1} \dots c_{i_n}]$
- 2、将每个进程的 dist 数组Reduce一下，我们就获得了每个点到离它最近的中心点距离，即为 min_dist ，再将 min_dist 广播给每个进程
- 3、每个进程都将其 dist 数组和 min_dist 进行比对，获得属于它的点，更新 c_i ，将每个进程的 c_i 收集起来再进行广播
- 4、重复1，2，3直到达到收敛条件

while True:

dist = []

min_dist = numpy.zeros(num_points)

for point in data:#dist记录每个rank到其他点的欧式距离

dist.append(eucl_distance(initial[rank], point))

temp_dist = numpy.array(dist)

comm.Reduce(temp_dist, min_dist, op = MPI.MIN)

comm.Barrier()

if rank == 0:

min_dist = min_dist.tolist()#numpy数据类型变list

recv_min_dist = comm.bcast(min_dist, root = 0)

comm.Barrier()

cluster = []

for i in range(len(recv_min_dist)):

if recv_min_dist[i] == dist[i]:

cluster.append(data[i])#表示该点到center的距离就是最小的

```
for j in range(dimensions):
    if(len(cluster) != 0):
        center_val[j] = center_val[j] / len(cluster)#即每个center_val的中心坐标
    center = comm.gather(center_val, root = 0)
    comm.Barrier()
    if rank == 0:
        compare_val = compare_center(initial, center, dimensions, size, cutoff)
        if compare_val == size:
            print('my rank is %d'% rank,center)
            print("Execution time %s seconds" % (time.time() - start_time))
initial = comm.bcast(center, root = 0)#广播新center
    comm.Barrier()
if break_val == size:#中止条件
    break
```

Dijkstra并行算法设计

串行算法主体:

顶点集S表示已经找到最短路径的点集合

Q表示尚未找到最短路径的点

while Q is not an empty set // *Dijkstra*算法主体

u := **Extract_Min**(Q) 此处可并行

S.append(u)

for each edge outgoing from u as (u,v)

if $d[v] > d[u] + w(u,v)$ // 拓展边

$d[v] := d[u] + w(u,v)$ // 更新路径长度到更小的那个和值。



for v in S:

for d in Q:

find_min($d[v] + \text{adj_mat}[v][d]$)

Dijkstra并行算法设计

- 1、初始化： p 为进程数，将邻接矩阵按列切片成 p 份，scatter 给每个进程，每个进程对应一个全为 0 的数组 a ，长度为其对应的分配到的节点数，其对应位置为 0 表示此点在 Q 中，为 1 表示此点在 S 中。
- 2、并行化：每个进程计算其节点到源点的距离的局部最小值，和其对应的节点，进行 allgather，进行排序后找到全局最小值。更新 S, d, a ，将 S, d 进行广播。
- 3、重复 2，直到达到中止条件

Dijkstra并行算法设计

```
while True:
```

```
    min=float('inf')
```

```
    for i in seen:
```

```
        for j in range(len(final)):
```

```
            tmp=dist[i]+mat[i][j]
```

```
            if dist[i]+mat[i][j]<min and final[j]==0:
```

```
                min=dist[i]+mat[i][j]
```

```
                local=rank*(m//size)+j
```

```
                p=j
```

```
            tmp=[min,local,p,rank]#min表示该rank局部最小点，local表示在全局的位置，p表示其在该rank的位置
```

```
            tmps=comm.allgather(tmp)
```

```
            comm.Barrier()
```

```
            tmps=sorted(tmps,key=lambda d:d[0])
```

```
            value=tmps[0][0]
```

```
            dist_local=tmps[0][1]
```

```
            final_local=tmps[0][2]
```

```
            rank_local=tmps[0][3]
```

```
if rank==rank_local:#更新
```

```
    dist[dist_local]=value
```

```
    final[final_local]=1
```

```
    seen.append(dist_local)
```

```
seen=comm.bcast(seen,root=rank_local)
```

```
dist=comm.bcast(dist,root=rank_local)
```

```
comm.barrier()
```

```
if len(seen)==m:
```

```
    print('the time is',MPI.Wtime()-t1)
```

```
    print('our result is',dist,seen)
```

```
    break
```

参考资料

- 1、 <https://www.jianshu.com/u/59ef38a1d84b>
- 2、 https://python-parallel-programming-cookbook.readthedocs.io/zh_CN/latest/chapter3/11_Using_the_mpi_py_Python_module.html