

mpi 学习日志(1):mpi 与 python

2016 年 07 月 18 日 16:50:20 [ljhandlwt](#) 阅读数: 4514 标签: [mpi](#) [mpi4py](#) [更多](#)

个人分类: [mpi 学习](#)

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51942708>

MPI 是什么?它用来干什么?

MPI 是信息传递接口(Message Passing Interface),简单来说就是一个用来实现进程通讯的库.它很多时候用于并行算法的设计.

下面我们先使用 windows 环境下 python 语言来了解 mpi 的使用.

mpi 在 python 的环境还是很好配置的.首先你得安装好 python 和 pip.然后直接用 pip 安装 mpi 在 python 的库,mpi4py.

```
C:\WINDOWS\system32\cmd.exe
```

```
C:\Users\Administrator>pip install mpi4py
```

然后我们需要安装 mpi,去 mpi 的官网找.网址 <http://www.mpich.org/downloads/>

Microsoft Windows	Microsoft MPI Team	[http]	1.0.3
-------------------	------------------------------------	------------------------	-------

下载得到的文件:

 MSMpiSetup.exe	2016/7/17 星期...	应用程序	5,184 KB
--	-----------------	------	----------

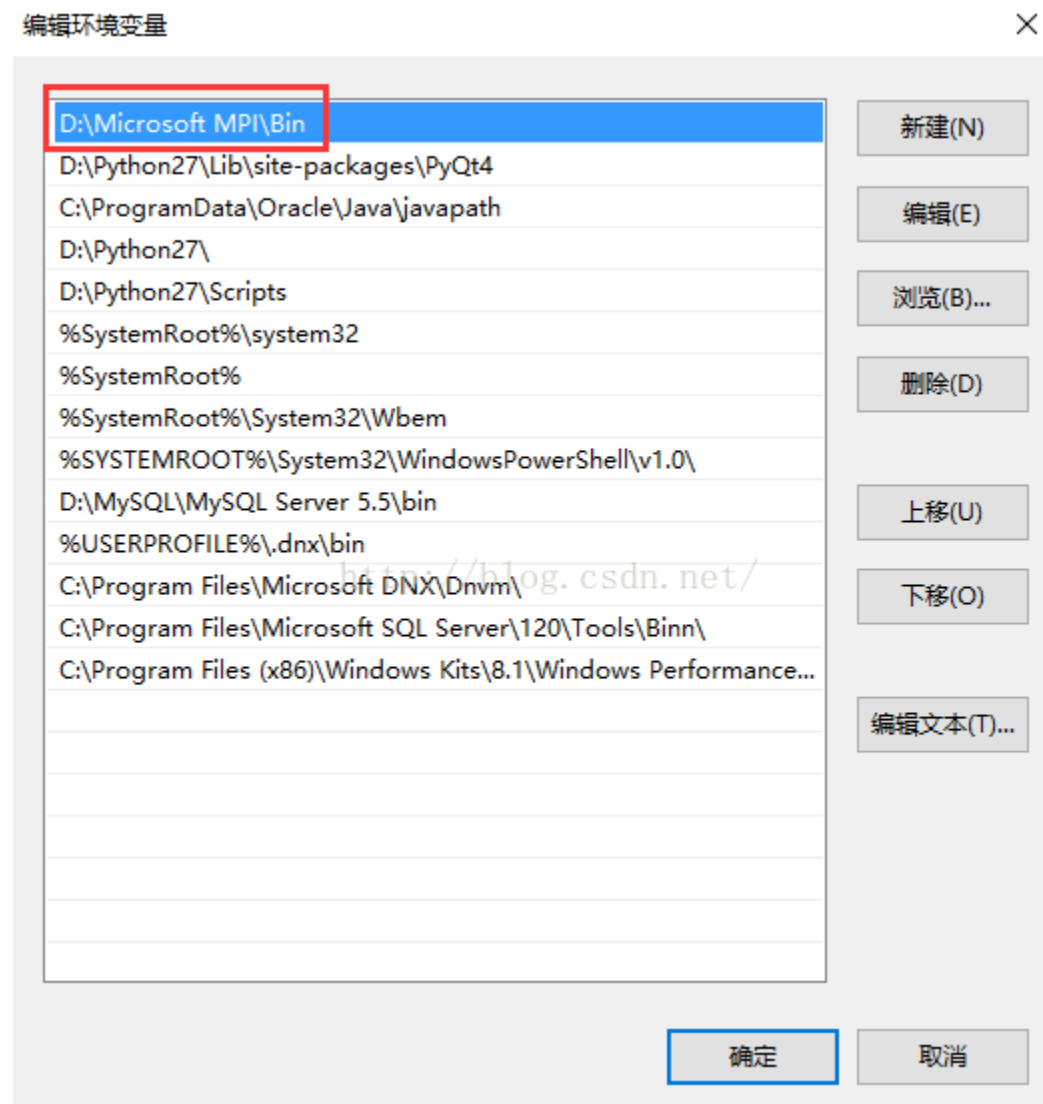
安装后在安装目录的 bin 文件夹里会有很重要的一个程序 mpiexec.exe:

电脑 > 软件 (D:) > Microsoft MPI > Bin

名称	修改日期	类型	大小
 mpiexec.exe	2016/6/13 星期...	应用程序	278 KB
 mpitrace.man	2016/6/13 星期...	MAN 文件	967 KB
 mspilaunchsvc.exe	2016/6/13 星期...	应用程序	28 KB
 smpd.exe	2016/6/13 星期...	应用程序	233 KB

为了方便运行程序,安装的时候,这个路径会自动添加到环境变量里.如果你打开命令行 `cmd`,输入 `mpiexec` 回车发现 `cmd` 没有找到这个程序,你最好手动添加这个环境变量.

环境变量的添加:我的电脑右键->属性->高级系统设置->高级->环境变量->系统变量里的 `path` 双击->新建,填入你的对应的路径(win10 以下的界面和这个不一样,具体百度)



环境变量设置好后,在 `cmd` 运行 `mpiexec` 会输出一些帮助信息.

```
C:\WINDOWS\system32\cmd.exe

C:\Users\Administrator>mpiexec
Microsoft MPI Startup Program [Version 7.1.12437.25]

Launches an application on multiple hosts.

Usage:

    mpiexec [options] executable [args] [ : [options] exe [args] : ... ]
    mpiexec -configfile <file name>

Common options:
    -n <num_processes>
    -env <env_var_name> <env_var_value>
    -wdir <working_directory>
    -hosts n host1 [m1] host2 [m2] ... hostn [mn]
    -cores <num_cores_per_host>
    -lines
    -debug [0-3]

Examples:

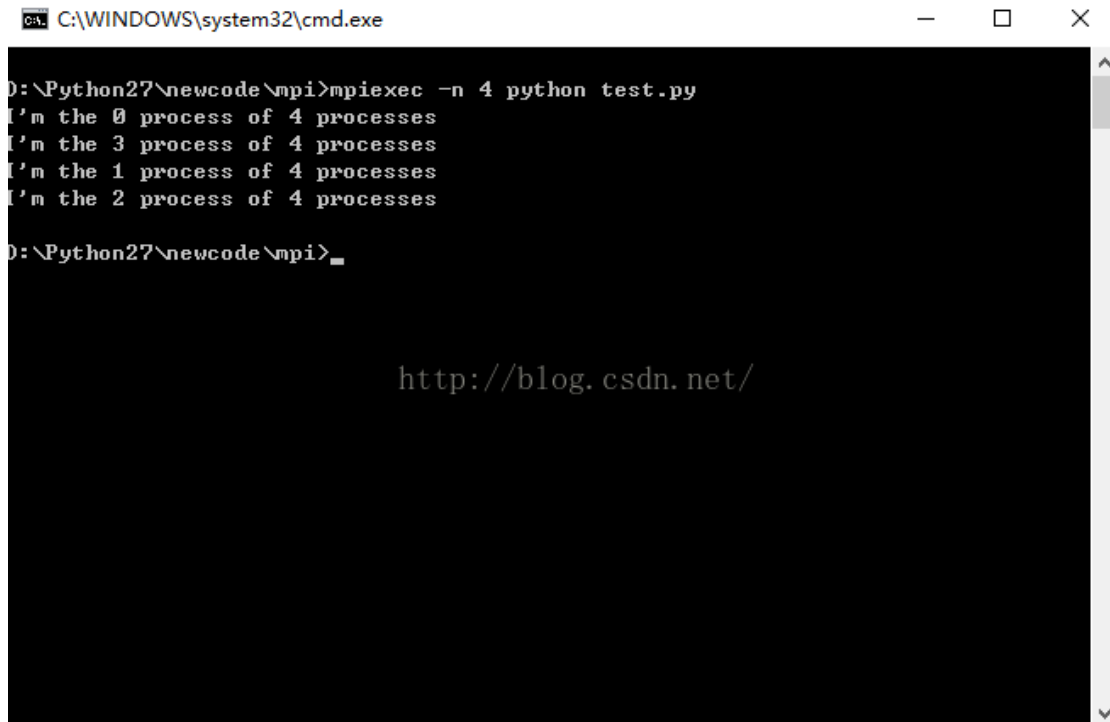
    mpiexec -n 4 pi.exe
    mpiexec -hosts 1 server1 master : -n 8 worker
```

PS:好像现在 windows 版本的 MPI 要跳转到 MS 的官网,而且下载到的是 MSMPI 而不是 MPICH2.,详情我也不太清楚.不过对于 py 上的开发,只要安装目录里有 mpiexec.exe 这个程序就行了.

安装完 mpi 和 mpi4py 之后,我们就可以编写代码了.代码样例是最简单的每个进程输出它的 rank(什么是 rank?下面解释).

```
1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  print "I'm the %d process of %d processes" % (comm_rank, comm_size)
9
```

代码很简单,运行命令和结果为下图:



我们先看命令,注意到运行命令不是简单的 `python xxx.py`,因此使用未配置好的 IDE 运行这个 `py` 文件可能导致运行失败.

在命令中,`-n` 表示进程数.`mpiexec` 在执行的时候会创建一组完全相同的进程,而这个进程组有多少个进程就是由这个参数`-n` 决定,图中例子为 4 个.

4 后面的参数实际上是要运行的一个 `exe`,由于 `py` 不是 `c/c++` 那样先编译出一个 `exe` 后运行它,而是动态解释.因此我们要运行的就是 `python.exe` 这个解释器,而这个 `python` 又带有参数 `test.py`,表示要运行的是 `test.py` 这个 `py` 文件.

看完命令再看代码.第 2 行是一个 `import` 语法,把 `MPI` 这个名字 `import` 进来.

第 4 行的 `comm` 是一个很重要的对象,之后的操作都围绕它来展开.

第 5 行通过 `comm` 对象的 `Get_rank()` 方法获取当前进程的 `rank`.什么是 `rank`,你当成是进程 ID 去理解吧.这里的 `rank` 是从 0 开始,不断加 1.第 6 行则是用 `Get_size` 方法获取这组进程的进程数.

可能你已经发现了,在代码里我只 `print` 了一行,但在实际运行却输出了 4 行,并且输出的东西还是不同的,并且是乱序的.

这其实就是 `mpiexec` 做的工作,它让这个 `test.py` 文件并行地运行了 4 次,而不需要像 `python` 的 `thread` 库那样在代码里先声明多少个线程

mpi 学习日志(2):mpi4py 与点对点通信

2016 年 07 月 18 日 17:56:17 [ljhandlwt](#) 阅读数: 1231

版权声明：本文为博主原创文章，未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51943523>

前文说到,mpi 是信息传递接口,因此信息传递是 mpi 的重点.而进程中的信息传递就是进程通信!今天我们将看看 mpi 创建的一组进程是怎么相互通信的.

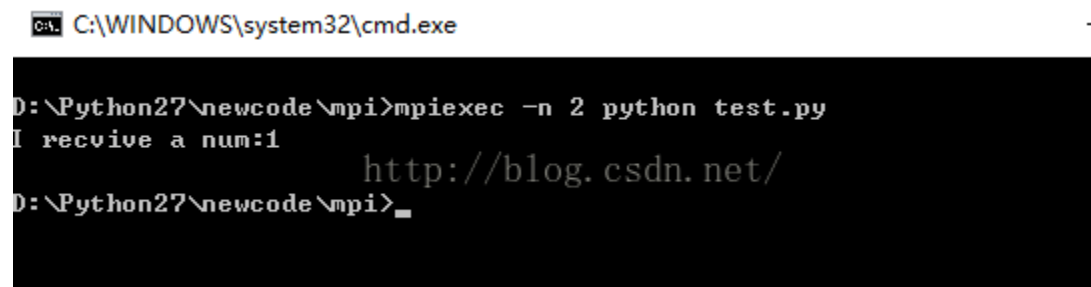
进程通信有许多种,点对点通信,广播,散播等.今天我们先学习点对点通信.

什么是点对点通信?其实就是最简单的进程 A 向进程 B 发送信息,而进程 B 向进程 A 接收信息.这是关于两个进程之间的通信.

代码:

```
1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  if comm_rank == 0:
9      data = 1
10     comm.send(data, dest=1)
11 else:
12     data = comm.recv(source=0)
13     print 'I recvive a num:%d' % data
14
```

运行结果:



```
C:\WINDOWS\system32\cmd.exe
D:\Python27\newcode\mpi>mpiexec -n 2 python test.py
I recvive a num:1
D:\Python27\newcode\mpi>
```

代码解释:

我们看见,进程 1 得到了进程 0 的数据,一个 int,值为 1.

使用到的一组方法是 comm 对象的 send 和 recv.

send 方法的第一个参数就是你要传送的数据,这个数据可以是一个 int,一个 float,还可以是列表,字典,甚至是 numpy 的 array 对象(你可以试一下)

send 方法还有一个 dest 参数,这就是你想发送的进程的 rank.

而 `recv` 方法在这里只使用了一个参数 `source`,就是你要指定的发送方进程的 `rank`.

阻塞 or 非阻塞?

一谈到消息传递,大家都会想到同步异步的问题,这就涉及到函数的阻塞性.那么,上面的 `send` 和 `recv` 方法是阻塞函数还是非阻塞函数?

`recv` 是阻塞函数,也就是说进程要收到发送方的数据,这个函数才返回.

而 `send` 是不确定的,也就是说它有时候是阻塞,有时候是非阻塞.当发送的数据不多的时候,`mpi` 会将数据存到一个系统缓冲区,然后马上进行 `send` 方法的返回.而当数据量很大超过缓冲区的大小的时候,`mpi` 需要等待接收方接收,然后把数据拷贝给接收方,再进行 `send` 方法的返回.

简单来说,数据量少->非阻塞,数据量大->阻塞.

小写 or 大写?

细心的同学还会发现,`comm` 对象除了 `send` 和 `recv` 方法,还有 `Send` 和 `Recv` 方法.一个字母小写和大写之差,到底意味着什么?

事实上,不止 `send` 和 `recv` 方法,以后学到的所有信息传递方法都会有这小写大写两个版本.

这里参考官方文档说法,这样区分是由于要传递的数据的性质差异.当我们要传递 `int`,`float`,`list`,`dict` 等 `python` 内置类型的数据的时候,我们使用小写的方法.而当使用 `buffer` 类型的数据的时候,我们要使用大写的方法.

`buffer` 类型是什么意思我还未理解,希望有人告诉我一下.

send 的多个版本:

事实上,除了大写小写的版本,`send` 还有不同的版本,这个不同是基于不同的发送策略的,而这些版本都有大小写之分.

`bsend`:缓冲模式,数据写入缓冲区,马上返回,用户必须确保缓冲区大小足够

`ssend`:同步模式,等接收方接收才返回

`rsend`:就绪模式,发送时必须确保接收方处于等待接收的状态,否则产生错误

`send`:标准模式(`bsend+ssend`)

`send` 实际上就是 `bsend` 和 `ssend` 的结合体.而 `rsend` 的意义我暂时不知道,一般在初学的时候,我们就使用标准模式 `send` 就好了.

另一种意义的阻塞 or 非阻塞

除了大小写之外,`mpi` 还为大部分的通信函数提供了阻塞和非阻塞这两种方式,非阻塞的在阻塞版本的前面加 `i` 或者 `I`.例如 `isend`,`ibsend`,`issend`,`irsend`.

不是说 `bsend` 就是非阻塞了吗,那 `ibsend` 又是什么意思?

这里的阻塞是针对信息的拷贝工作的.当使用 `bsend` 的时候,数据拷贝到缓冲区后,函数才返回.而使用 `ibsend` 的时候,拷贝的工作是交给 `mpi` 后台来完成,而在实际进行拷贝的时候,函数早已经返回了.

总结:

实际上,除了 `send`,其他版本的 `send` 我都没有用过,这些版本在这里也只是知识的拓展,以后如果有需要,会另外详细说明其原理和用法.

mpi 学习日志(3):mpi4py 与广播

2016 年 07 月 18 日 18:34:01 [ljhandlwt](#) 阅读数: 1056

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51944188>

前面我们学习了点对点通信,那是关于两个点的通信.今天,我们学习多个点之间的通信.首先,我们学习广播.

所谓广播,就是说某个进程要向其他所有进程发送数据,而不是单独某个进程.

显然,一次的广播可以等价为许多次的点对点通信,我们可以用学过的 `send` 和 `recv` 去实现广播.

代码:

```

1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  if comm_rank == 0:
9      data = [1,2,3]/blog.csdn.net/
10     for i in range(comm_size - 1):
11         comm.send(data, dest=i + 1)
12 else:
13     data = comm.recv(source=0)
14     print 'Process %d recvive' % comm_rank , data
15

```

运行结果:

```

C:\WINDOWS\system32\cmd.exe
D:\Python27\newcode\mpi>python test.py
Process 1 recvive [1, 2, 3]
Process 2 recvive [1, 2, 3]
Process 3 recvive [1, 2, 3]
Process 4 recvive [1, 2, 3]
D:\Python27\newcode\mpi> http://blog.csdn.net/

```

我们看见,5 个进程中,除了要广播的进程 0,后面 4 个进程都收到了进程 0 发送的一个列表.

效率?

但这样就完了吗?这样实现有什么问题?答案是效率.

用时间复杂度去说话,假设拷贝一份数据是 $O(1)$ 复杂度,有 n 个进程.那么后面 4 个进程的时间复杂度是 $O(1)$,而进程 0 的时间复杂度是 $O(n)$.

在单机上跑这 n 个进程好像这没什么所谓,CPU 始终在工作,时间复杂度也是 $O(n)$ 级别.但假如是 n 台机器分别跑这 n 个进程,那么问题就大了.

第 0 台机器始终在发送数据,而其他机器的大部分时间都在排队,等第 0 台机器往自己发送数据.这样的话,这堆机器要运行完这堆进程,需要 $O(n)$ 时间.这居然和一台机器所需时间一样!

bcast:

为了解决这个问题,我们可以像 p2p 那样做,有数据的机器都帮忙向没有数据的机器发送数据,这样的话时间复杂度是可以降低到 $O(\log n)$ 的!

要实现这样的算法,是很困难的,尤其是 mpi 下这些进程都不共享数据,难上加难.

幸好,mpi 帮我们实现了这个功能,我们只要使用一个统一的函数就行了,那就是 bcast

代码:

```
1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  if comm_rank == 0:
9      data = [1,2,3]
10     comm.bcast(data, root=0)
11 else:
12     data = comm.bcast(None, root=0)
13     print 'Process %d recvive' % comm_rank , data
14
```

运行结果:

C:\WINDOWS\system32\cmd.exe

```
D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
Process 1 recvive [1, 2, 3]
Process 4 recvive [1, 2, 3]
Process 2 recvive [1, 2, 3]
Process 3 recvive [1, 2, 3]

D:\Python27\newcode\mpi>_
```

可以看到,运行结果和前面的完全一样.

代码解释:

在代码里,新出现的函数就只有 bcast.

我们注意到,无论是广播者,还是被广播者,都是调用 **bcast** 函数,而不像点对点那样一个 **send** 另一个 **recv**.

在以后更多的通信方式里,都会像这样发送方和接收方使用同一个函数,只是参数可能不一样.

说到参数,发送方的第一个参数和 **send** 一样,也是要发送的数据.

后面还有一个 **root** 参数,这个参数接收方也有,也是实参一样.这个 **root** 是表示要广播的人是谁.这里就是我们的进程 0.

不同的是,接收方的第一个参数是 **none**.这个照写就是了,后面许多函数也是这样的.

返回值方面,我这里发送方没有接收返回值.事实上,发送方和接收方的返回值都是一样的,都是发送方要发送的数据.也就是说,广播者也会广播到自己

mpi 学习日志(4):mpi4py 与散播等

2016 年 07 月 18 日 19:42:20 [ljhandlwt](#) 阅读数: 869

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

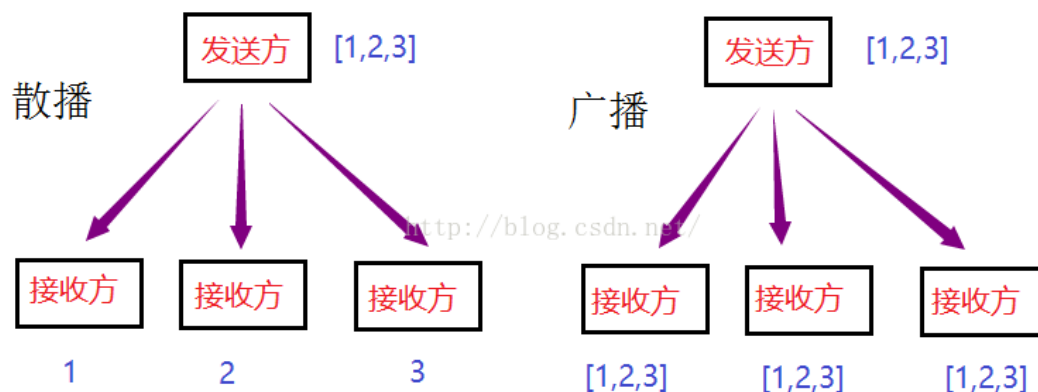
<https://blog.csdn.net/ljhandlwt/article/details/51944429>

今天要学习的是散播等多点通信.

散播:

什么是散播?就是发送方要把一堆数据发送出去,但接收方只接收其中的一个.

下图是一个散播和广播的对比.



同样的,散播也是能等价于多个点对点通信,也同样的有效率问题.

只是在考虑效率的时间,我们应该考虑建立通信的时间远大于数据拷贝的时间,比较好.

广播使用的函数是 `bcast`,而散播则是 `scatter`

代码:

```
1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  if comm_rank == 0:
9      data = [1,2,3,4,5]
10 else:
11     data = None
12
13 data = comm.scatter(data, root=0)
14 print 'Process %d recvive' % comm_rank , data
15
```

运行结果:

C:\WINDOWS\system32\cmd.exe

```
D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
Process 2 recvive 3
Process 0 recvive 1
Process 4 recvive 5
Process 1 recvive 2
Process 3 recvive 4
D:\Python27\newcode\mpi>
```

代码解释:

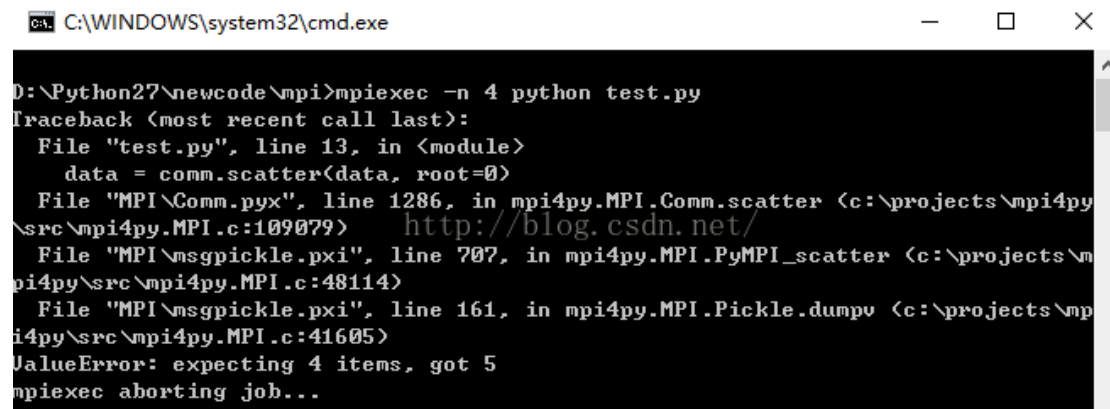
从代码可以看到,散播的函数和广播的参数是一样的,只是返回值不一样.

注意!散播的发送方也会接收到数据(这和上面的概念图有出入),因此我这里写法稍稍和广播的不一样.

从结果可以看到,进程 0 的列表[1,2,3,4,5],被 5 个进程各取了其中的一个元素.

同时我们注意到,散播里列表里元素的分发不是按进程 0 就分得第 0 个元素,进程 1 就第 1 个元素这样的.而是一种类似随机的打乱的分发策略.

有一点很关键的是,列表里元素的个数必须等于进程的个数.否则会出错,如下图我把进程数改成了 4.



```
Ca. C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 4 python test.py
Traceback (most recent call last):
  File "test.py", line 13, in <module>
    data = comm.scatter(data, root=0)
  File "MPI\Comm.pyx", line 1286, in mpi4py.MPI.Comm.scatter (c:\projects\mpi4py\src\mpi4py\MPI.c:109079) http://blog.csdn.net/
  File "MPI\msgpickle.pxi", line 707, in mpi4py.MPI.PyMPI_scatter (c:\projects\mpi4py\src\mpi4py\MPI.c:48114)
  File "MPI\msgpickle.pxi", line 161, in mpi4py.MPI.Pickle.dumpv (c:\projects\mpi4py\src\mpi4py\MPI.c:41605)
ValueError: expecting 4 items, got 5
mpiexec aborting job...
```

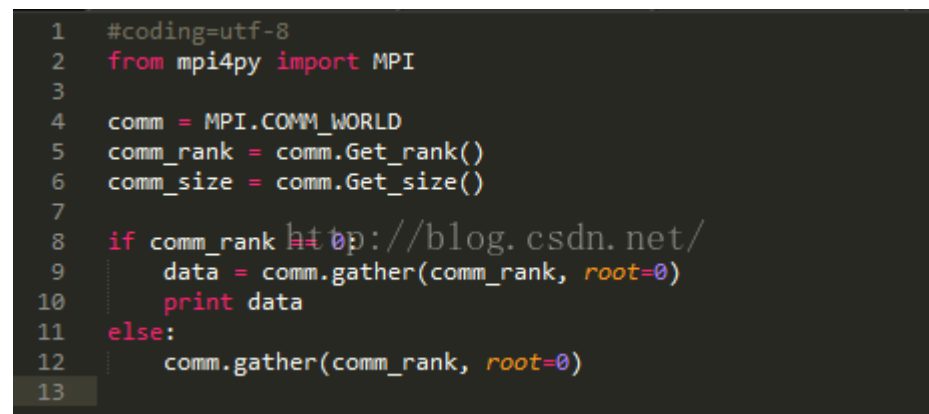
收集:

收集是散播的逆操作.

散播是 1 发多,而收集则是多发 1.

废话少说,直接上图.

代码:



```
1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  if comm_rank == 0: http://blog.csdn.net/
9      data = comm.gather(comm_rank, root=0)
10     print data
11 else:
12     comm.gather(comm_rank, root=0)
13
```

运行结果:

C:\WINDOWS\system32\cmd.exe

```
D:\Python27\newcode\mpi>mpiexec -n 5 python test.py  
[0, 1, 2, 3, 4]
```

```
D:\Python27\newcode\mpi>
```

reduce:

reduce 也是一种多点通信方式,它的作用机理和 python 的高阶函数 reduce 类似.

它相当于在收集的过程中不断地进行两元运算,最终在接收方那里只有一个值,而不是一个列表.

在这里我们尝试用 reduce 来求圆周率,怎么求,利用下面公式.

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

我们让每台机器用 reduce 的方式发送上面公式左边的每一项,就能在 $O(\log n)$ 的时间里求出和,作为 π 的近似值.

代码:

```
1  #coding=utf-8  
2  from mpi4py import MPI  
3  
4  comm = MPI.COMM_WORLD  
5  comm_rank = comm.Get_rank()  
6  comm_size = comm.Get_size()  
7  
8  k = (1.0 if comm_rank % 2 == 0 else -1.0) / (2 * comm_rank + 1)  
9  data = comm.reduce(k, root=0, op=MPI.SUM)  
10  
11 if comm_rank == 0:  
12     pi = data * 4  
13     print 'pi=%f' % pi  
14
```

运行结果:

```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 15 python test.py
pi=3.208186

D:\Python27\newcode\mpi>mpiexec -n 40 python test.py
pi=3.116597
http://blog.csdn.net/
D:\Python27\newcode\mpi>mpiexec -n 80 python test.py
pi=3.129093

D:\Python27\newcode\mpi>_
```

代码解释:

我使用了不同的进程数量去跑这个并行算法,可以看见,当 n 不断变大的时候,近似值会不断地靠近 π 的真实值.

不过要注意的是,单机的话在 win 下不要开太多进程,几百个进程这可不是开玩笑的...

在代码里,reduce 方法有一个 op 参数,它代表你要进行怎么的聚类,常用的有 sum,max,min 等等.

有一点注意的是,散播和 reduce 中发送方接收到的返回值,不是接收方最终得到的返回值,而是一个 none

mpi 学习日志(5):mpi4py 与多点通信续

2016 年 07 月 19 日 10:22:44 [ljhandlwt](#) 阅读数: 1136 标签: [mpi](#) [mpi4py](#) [更多](#)

个人分类: [mpi 学习](#)

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51952874>

在多点通信里我们已经学习了广播 bcast,散播 scatter,收集 gather,规约 reduce.

今天我们来简略看一些可能更为少用的多点通信.

1.allgather

简单来说就是收集+广播.

gather 中只有根进程会得到收集到的信息,而 allgather 则是所有进程都会得到收集到的信息,就相当于收集后再广播一次.

```

1  #coding=utf-8
2
3  from mpi4py import MPI
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7  comm_size = comm.Get_size()
8
9  data = comm.allgather(comm_rank)
10 print 'rank %d' % comm_rank, data
11

```

C:\WINDOWS\system32\cmd.exe

```

D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
rank 4 [0, 1, 2, 3, 4]
rank 3 [0, 1, 2, 3, 4]
rank 0 [0, 1, 2, 3, 4]
rank 1 [0, 1, 2, 3, 4]
rank 2 [0, 1, 2, 3, 4]

D:\Python27\newcode\mpi>

```

2.allreduce

reduce 与 allreduce 的关系,和 gather 与 allreduce 类似.

也就是说,allreduce 就是 reduce 之后得到的值会再广播一次,即 allreduce=reduce+bcast.

```

1  #coding=utf-8
2
3  from mpi4py import MPI
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7  comm_size = comm.Get_size()
8
9  data = comm.allreduce(comm_rank, op=MPI.SUM)
10 print comm_rank, data
11

```

```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
0 10
4 10
2 10
3 10
1 10

D:\Python27\newcode\mpi>
```

(PS:10=0+1+2+3+4)

3.alltoall

alltoall 实现的是转置的功能.

每个进程调用 alltoall,都会把一个列表传递出去,然后也是得到一个列表.并且这两个列表的长度都必须等于总进程数.

具体操作的时候,alltoall 会把第 i 个进程提供的列表的第 j 项,放到第 j 个进程返回的列表的第 i 项.

也就是相当于在一个 $n \times n$ 的矩阵 A 里,实现了 $\text{swap}(A_{ij}, A_{ji})$.这就是矩阵的转置嘛.

	第0项	第1项	第2项	第3项	第4项			第0项	第1项	第2项	第3项	第4项
进程0	0	0	0	0	0		进程0	0	1	2	3	4
进程1	1	1	1	1	1		进程1	0	1	2	3	4
进程2	2	2	2	2	2		进程2	0	1	2	3	4
进程3	3	3	3	3	3		进程3	0	1	2	3	4
进程4	4	4	4	4	4		进程4	0	1	2	3	4
		发送前							发送后			

```
1 #coding=utf-8
2
3 from mpi4py import MPI
4 from numpy import *
5
6 comm = MPI.COMM_WORLD
7 comm_rank = comm.Get_rank()
8 comm_size = comm.Get_size()
9
10 data = comm.alltoall([comm_rank] * comm_size)
11 print comm_rank, data
12
```



```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
4 [0, 1, 2, 3, 4]
0 [0, 1, 2, 3, 4]
2 [0, 1, 2, 3, 4]
1 [0, 1, 2, 3, 4]
3 [0, 1, 2, 3, 4]

D:\Python27\newcode\mpi>
```

4.scan

scan 的操作和 allreduce 类似,但又更复杂

scan 会把前 i 个收集的数据 reduce 成一个数据,返回发送给第 i 个进程.

下面的代码第 i 个进程会得到 0+1+2+3+....的前 i 项和.

```
1 #coding=utf-8
2
3 from mpi4py import MPI
4
5 comm = MPI.COMM_WORLD
6 comm_rank = comm.Get_rank()
7 comm_size = comm.Get_size()
8
9 data = comm.scan(comm_rank, op=MPI.SUM)
10 print comm_rank, data
11
```

```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
3 6
1 1
0 0
2 3
4 10

D:\Python27\newcode\mpi>
```

5.barrier

barrier 是一种全局同步,就是说全部进程进行同步.

当一个进程调用 `barrier` 的时候,它会被阻塞.

当所有进程都调用了 `barrier` 之后,`barrier` 会同时解除所有进程的阻塞.

相当于原本在跑道上跑得参差不齐的运动员,跑到起跑线上停下来等.当所有运动员都在起跑线上,他们才一起重新跑起来.

这就是所谓的全部进程进行同步.

```
1  #coding=utf-8
2
3  from mpi4py import MPI
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7  comm_size = comm.Get_size()
8
9  print comm_rank, 'begin'
10 comm.barrier()
11 print comm_rank, 'end'
12
```

C:\WINDOWS\system32\cmd.exe

```
D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
3 begin
3 end
2 begin
2 end
4 begin
4 end
1 begin
1 end
0 begin
0 end

D:\Python27\newcode\mpi>_
```

说是这么说,但运行起来发现并不是这回事.所有进程没有像期待那样先全部输出 `begin`,再全部输出 `end`,`barrier` 这个函数仿佛形同虚设.

其实这里问题不是在 `barrier`,而是在 `print`.

我们 OS 的 IO 是有缓冲的,一个数据要出现在屏幕上,简单来说是经过内存->标准 IO 文件->控制台屏幕.

而进程间不共享 IO 文件(后面会学到如何在 MPI 的进程里共享文件),共享控制台屏幕.

因此屏幕上语句的顺序依赖 OS 什么时候将 IO 文件里的内容推到屏幕上.

我们强制让内存->标准 IO 文件和标准 IO 文件->控制台屏幕这两步一起进行,也就是加上 flush 语句.

```
1  #coding=utf-8
2
3  from mpi4py import MPI
4  import sys
5
6  comm = MPI.COMM_WORLD
7  comm_rank = comm.Get_rank()
8  comm_size = comm.Get_size()
9
10 print comm_rank, 'begin'
11 sys.stdout.flush()
12 comm.barrier()
13 print comm_rank, 'end'
14
```

```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 5 python test.py
0 begin
1 begin
4 begin
3 begin
2 begin
3 end
1 end
0 end
2 end
4 end

D:\Python27\newcode\mpi>
```

果然,输出结果和我们的期待吻合了.

mpi 学习日志(6):mpi4py 与 sendrecv

2016 年 07 月 19 日 10:45:29 [ljhandlwt](#) 阅读数: 552

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51953643>

在点对点通信中,除了 send 和 recv 这两个最重要的函数,以及 send 和 recv 的各种版本的函数,还有一个可能常用的函数,那就是 sendrecv.

sendrecv 其实只是 send+recv,没有什么新的花招.只是使用 sendrecv 有一种函数调用的感觉.发送方像调用者,接收方像被调用者.

```

1  #coding=utf-8
2
3  from mpi4py import MPI
4  from numpy import *
5
6  comm = MPI.COMM_WORLD
7  comm_rank = comm.Get_rank()
8  comm_size = comm.Get_size()
9
10 if comm_rank != 0:
11     data = [1,2,3]
12     data = comm.sendrecv(data, dest=1)
13     print data
14 else:
15     data = comm.recv(source=0)
16     data = [x * 2 for x in data]
17     comm.send(data, dest=0)
18

```

C:\WINDOWS\system32\cmd.exe

```

D:\Python27\newcode\mpi>mpiexec -n 2 python test.py
[2, 4, 6]
D:\Python27\newcode\mpi>

```

上述代码,进程 1 把进程 0 发给它的列表里的每个元素都乘上 2,再返回这个新的列表.

mpi 学习日志(7):mpi4py 与通信子,通信组

2016 年 07 月 19 日 20:11:08 [ljhandlwt](#) 阅读数: 1398 标签: [mpi](#) [mpi4py](#) [更多](#)

个人分类: [mpi 学习](#)

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51958560>

从第一篇笔记开始,我们就使用了 `comm` 这个对象.

而且,几乎每次新学习的函数,都是这个 `comm` 的一个方法.

今天,我们就来重新认识 `comm`,它到底是什么?

1.通信子

`comm` 是一个通信子.什么是通信子?

通信子其实就是一个对讲机,每个进程都有一个对讲机.

通过对讲机,你可以和某个人对话,也可以对全部人发出广播.

2.通信组

警察有警察之间的对讲机,盗贼有盗贼之间的对讲机.

显然,警察的对讲机是不应该能和盗贼的对讲机通信的(不包括偷听).

这样的话,我们就有了通信组的概念.

警察之间形成一个通信组,盗贼之间形成一个通信组.

3.MPI 里的通信组

MPI 里有一个类 `Group`,它就是充当了通信组的角色.

当我们有一个对讲机 `comm` 的时候,我们通过 `Get_group()`的方法可以得到对应的 `Group` 对象.

`Group` 对象有什么用?它可以用来创建新的通信组!

4.MPI 里的默认通信子

我们一开始就用到了一个系统帮我们创建的全局通信子,就是 `COMM_WORLD`,它包含了所有的进程.

其实还有两个系统默认创建的通信子,一个是 `COMM_SELF`,另一个是 `COMM_NULL`.

`COMM_SELF` 仅仅包含了当前进程,而 `COMM_NULL` 则什么进程都没有包含.

在下面的代码中,我们将使用到 `COMM_NULL`.

5.创建新通信组

在实际开发中,我们往往不止需要一个全局的通信组,还需要很多小型的通信组,我们需要创建新通信组.

其实准确来说是创建新通信子,因为通信组只是一份名单,而通信需要的是对讲机!

某个系列的对讲机一定有对应的某份名单,而某份名单不一定有对应的对讲机.

要创建新通信子,我们的过程是这样的:

(1)获取一个旧的通信组(通常是利用 `COMM_WORLD` 获取)

(2)对这个通信组进行添加和删减,以达到我们的目标

(3)通过旧通信子的 **Create** 方法创建新通信子

下面我们会在 6 个进程中,让后 3 个进程单独创建一个新的通信组.

代码:

```
1  #coding=utf-8
2  from mpi4py import MPI
3
4  comm = MPI.COMM_WORLD
5  comm_rank = comm.Get_rank()
6  comm_size = comm.Get_size()
7
8  group = comm.Get_group()
9  group = group.Excl([0,1,2])g.csdn.net/
10
11  new_comm = comm.Create(group)
12
13  if new_comm != MPI.COMM_NULL:
14      print comm_rank, new_comm.Get_rank()
15
```

运行结果:



```
C:\WINDOWS\system32\cmd.exe
D:\Python27\newcode\mpi>mpiexec -n 6 python test.py
4 1
5 2
3 0
D:\Python27\newcode\mpi>_
```

代码解释:

第 8 行获取了全局通信子的通信组,第 9 行调用了 **Excl** 方法,得到了一个剔除了前 3 个进程的新通信组(具体的含义会在下一篇笔记说到).

然后我们用 **Create()** 方法,把 **group** 作为参数送进去,就会得到一个新的通信子 **new_comm**.

注意到,对于第 3,4,5 个进程来说,它们会得到新通信子.但是,对第 0,1,2 个进程来说,它们不应该得到新通信子.

那样的话,是否需要全部进程都执行 **Create()** 这个方法?

答案是一定要的.实际上这个方法有一个 **barrier** 的过程,也就是全局同步.

在 **comm** 这个通信子里的所有进程,必须都调用 **Create()**这个方法,MPI 才会生成并返回新的通信子.

不相信的话,你可以让某个进程不调用 **Create()**,结果是其他进程会被阻塞住.

那么新问题又到了,对第 0,1,2 个进程来说,它们得到的 **new_comm** 是个什么玩意?

没错,从代码第 13 行你已经发现了,它会是一个空通信子 **COMM_NULL**.

总的来说就是,旧通信子的所有进程都调用 **Create()**方法.对于在新通信组的进程会得到新通信子,而不在新通信组的进程会得到空通信子.

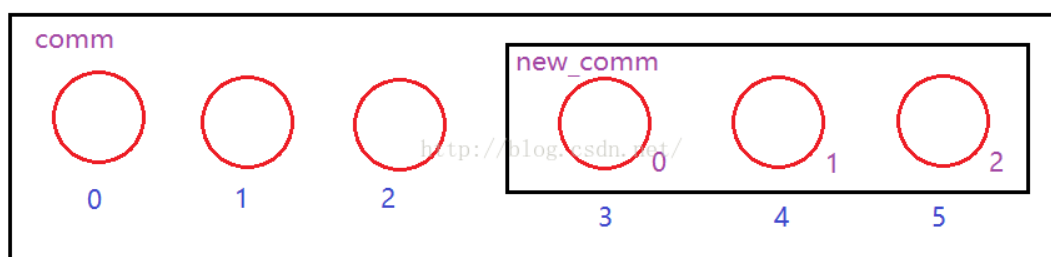
空通信子就是这样使用的,并且如果你像普通通信子一样调用它的方法,会抛出异常.

从输出结果可以看到新通信子对它的进程重新分配了 **rank**.

也就是说,对某个进程,它在不同的通信组会有不同的 **rank**,这也是把它称为 **rank** 而不是 **id** 的原因.

因此,在不同通信组进行通信的时候,这个进程使用的 **rank** 是不同的.这有点像,在不同的场合,人有不同的身份.

通信组示意图:



mpi 学习日志(8):mpi4py 与 Group 运算

2016 年 07 月 19 日 21:18:09 [ljhandlwt](#) 阅读数: 574

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51959425>

今天,我们来学习一下 **Group** 的各种增删运算.

1.Incl

incl 是挑选的意思.

你要把一个列表作为参数来调用 Incl,列表里放的是一些下标(注意不是 rank).

这些下标对应的进程会被挑选出来,形成一个新的 group,并返回这个 group.

PS:列表里放的是下标,不是 rank!!!

```
1 #coding=utf-8
2 from mpi4py import MPI
3
4 comm = MPI.COMM_WORLD
5 comm_rank = comm.Get_rank()
6 comm_size = comm.Get_size()
7
8 group = comm.Get_group()
9 group = group.Incl([0,1,4,5])
10 group = group.Incl([2,3])
11
12 new_comm = comm.Create(group)
13
14 if new_comm != MPI.COMM_NULL:
15     print comm_rank, new_comm.Get_rank()
16
```

```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 6 python test.py
4 0
5 1
http://blog.csdn.net/

D:\Python27\newcode\mpi>_
```



2.Excl

Incl 对应的就是 Excl,excl 是剔除的意思.

excl 接收的列表里的下标,会被剔除掉,然后返回这个剔除过的 group.(注意原 group 和 incl 一样是没有变化的).


```

8  group = comm.Get_group()
9  group = group.Excl([0,1])
10 group = group.Excl([0,1])
11 http://blog.csdn.net/
12 new_comm = comm.Create(group)

```

```

C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 6 python test.py
4 0
5 1      http://blog.csdn.net/

D:\Python27\newcode\mpi>_

```



3.Intersection

求交集,这个和集合 set 的 intersection 运算是一样的.

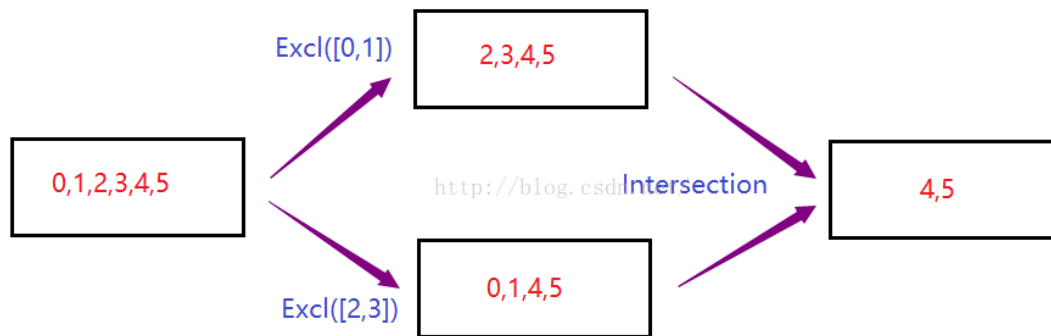
另外,还有一个函数 Intersect,好像它和 Intersection 是一样的...

还有注意的是,这些集合运算都是类方法.

```

8  group = comm.Get_group()
9  group1 = group.Excl([0,1])
10 group2 = group.Excl([2,3])
11 group3 = MPI_Group.Intersection(group1, group2)
12 http://blog.csdn.net/
13 new_comm = comm.Create(group3)

```



4.Union

求并集,用法和 Intersection 一样.

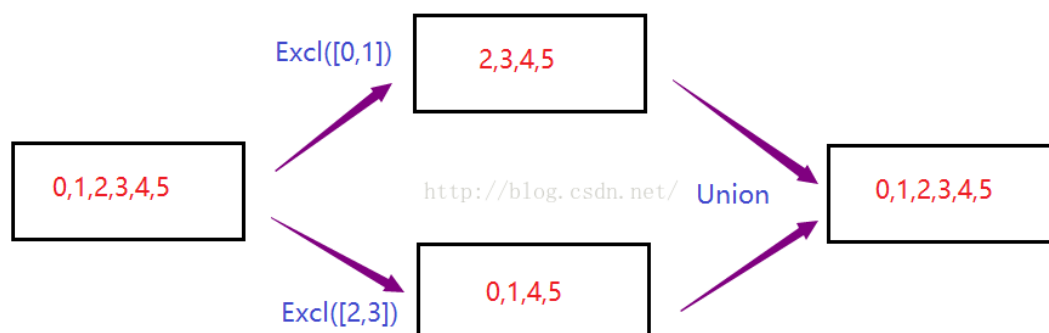
```
C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\mpi>mpiexec -n 6 python test.py
0 4
4 2
5 3
1 5
2 0
3 1

D:\Python27\newcode\mpi>
```

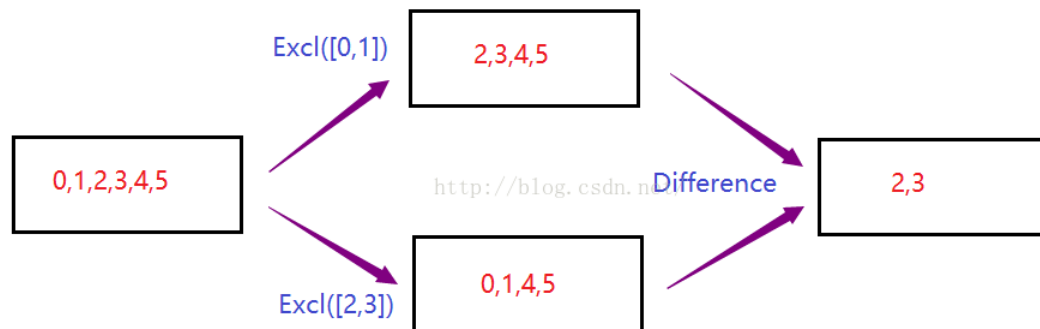
注意到,新通信组和旧通信组的成员是完全一样的,但是 rank 并不是按原来的顺序.

这说明新 rank 和旧 rank 没有必然联系.



5.Difference

求差集 A-B,同上.



mpi 学习日志(9):mpi4py 与 Split

2016 年 07 月 19 日 21:31:35 [ljhandlwt](#) 阅读数: 706

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51959663>

上两篇中,我们学习了如何创建新的通信组。

有时候,我们需要把进程分成若干组,它们各自形成新的通信组。

如果还是用 `Create()` 去实现的话,未免有点麻烦。

于是,`Split` 函数横空出世,帮我们解决这个麻烦。

`Split` 方法和 `Create` 一样,也是通信子的所有进程都要调用这个函数,同样也是有 `barrier` 的全局同步。

每个进程都要向 `Split` 提供一个 `color`,`color` 是一个 `int`.所有相同 `color` 的进程会形成新的组。

下面使用 `Split` 把 6 个进程分成[0,1],[2,3]和[4,5]三组.并且我们尝试在三个新组中,各自广播其颜色。

```

1  #coding=utf-8
2
3  from mpi4py import MPI
4  import sys
5
6  comm = MPI.COMM_WORLD
7  comm_rank = comm.Get_rank()
8  comm_size = comm.Get_size()
9
10 color = comm_rank / (comm_size / 3)
11
12 new_comm = comm.Split(color)
13
14 new_comm_rank = new_comm.Get_rank()
15 new_comm_size = new_comm.Get_size()
16
17 if new_comm_rank == 0:
18     data = new_comm.bcast(color, root=0)
19 else:
20     data = new_comm.bcast(None, root=0)
21
22 print 'old rank:%d new rank:%d data:%d' % (comm_rank, new_comm_rank, data)
23

```

C:\WINDOWS\system32\cmd.exe

```

D:\Python27\newcode\mpi>mpiexec -n 6 python test.py
old rank:0 new rank:0 data:0
old rank:4 new rank:0 data:2
old rank:5 new rank:1 data:2
old rank:1 new rank:1 data:0
old rank:3 new rank:1 data:1
old rank:2 new rank:0 data:1

D:\Python27\newcode\mpi>_

```

mpi 学习日志(10):mpi4py 实现简单并行矩阵乘法

2016 年 07 月 20 日 08:49:28 [ljhandlwt](#) 阅读数: 1267 标签: [mpi](#) [mpi4py](#) [更多](#)

个人分类: [mpi 学习](#)

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51964718>

除了之前利用 `gather` 求 π 之外,我们就没有写过 `mpi` 程序的实例。

今天我们就尝试用 `mpi` 去写一个简单的并行矩阵乘法,虽说是并行,但不是使用经典的分治去处理,而只是简单地每个进程计算一个格子的值。

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \\ 29 & 40 & 51 \end{bmatrix}$$

计算的式子是这样的,3*2 的矩阵 A 乘上 2*3 的矩阵 B,得到 3*3 的矩阵 C。

这样的话,我们就需要 9 个进程去计算每个格子的值.

问题分析:

我们假设一开始只有进程 0 拥有矩阵 A 和 B 的数据,也就是说进程 0 要把数据分发到各个进程.

但我们知道,每个格子并不需要整个矩阵,它只需要某行或某列即可.

具体来说,第 i 行第 j 列的格子,需要矩阵 A 的第 i 行和矩阵 B 的第 j 列.

按照这样分发的话,我们需要设计复杂的分发过程,而不是直接让进程 0 广播出去.

算法设计:

我们可以使用 Split,按行把进程分割成 3 个新的通信组,又按列分割成 3 个新通信组.

这样的话,每个进程都有 3 个通信子,全局通信子,行通信子,列通信子.

我们注意到,行通信组里的进程需要相同的行,列通信组里的进程需要相同的列.

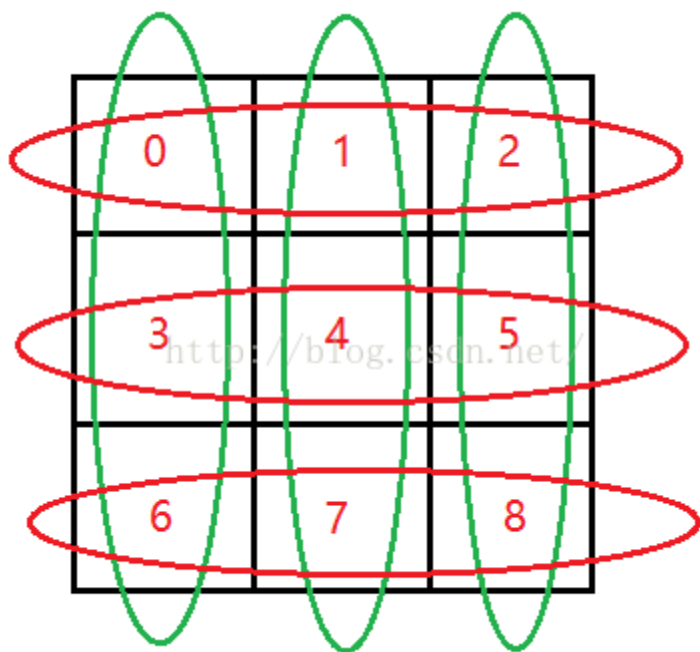
那么,只要我们行首和列首拥有对应的行和列,它就可以直接广播这个行和列.

怎么让行首和列首获得相应的行和列?很简单,用散播就行.

我们让进程 0 在第 0 列散播 A 的行,让进程 0 在第 0 行散播 B 的列,就行了!

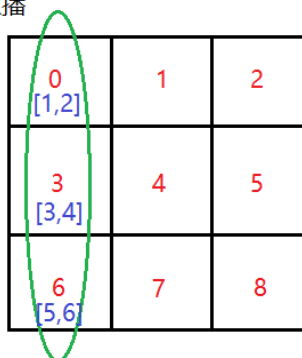
而对于结果,直接用 gather 收集起来.最后进程 0 用 numpy 的 array 的 reshape 方法改一下形状即可.

通信组示意图:



算法示意图:

第零列散播



每行广播

<http://blog.csdn.net/>




代码:

```

2  from mpi4py import MPI
3  from numpy import *
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7  comm_size = comm.Get_size()
8
9  #矩阵数据
10 if comm_rank == 0:
11     a = array([[1,2],[3,4],[5,6]])
12     b = array([[1,2,3],[4,5,6]])
13     b = b.transpose()
14 else:
15     a = None
16     b = None
17
18 #创建新通信子
19 color_row = comm_rank / 3
20 color_col = comm_rank % 3
21 comm_row = comm.Split(color_row)
22 comm_col = comm.Split(color_col)
23
24 #散播矩阵A(第零列) http://blog.csdn.net/
25 row = comm_col.scatter(a, root=0) if color_col == 0 else None
26 #广播行
27 row = comm_row.bcast(row, root=0)
28
29 #散播矩阵B(第零行)
30 col = comm_row.scatter(b, root=0) if color_row == 0 else None
31 #广播列
32 col = comm_col.bcast(col, root=0)
33
34 #求和
35 ret = sum([x * y for x, y in zip(row, col)])
36
37 #收集
38 c = comm.gather(ret, root=0)
39
40 #输出
41 if comm_rank == 0:
42     c = array(c).reshape(3, 3)
43     print c

```

运行结果:

 C:\WINDOWS\system32\cmd.exe

```

D:\Python27\newcode\mpi>mpiexec -n 9 python test.py
[[ 9 12 15]
 [19 26 33]
 [29 40 51]]
D:\Python27\newcode\mpi>

```

在写代码的时候,我们必须考虑每一个进程的行为,必要的情况下我们要分情况执行命令.这一点十分的关键.

比如在第 25 行那里,其实是有 3 类进程执行了不同的操作的.

对于进程 0,它向第零列散播矩阵 A.对于第零列的其他进程,它是散播的接收方.而对于其他进程,它仅仅是执行 `row=None` 的声明赋值语句.

还有一点注意的是,为了让矩阵 B 按列散播,我们需要 array 的 `transpose` 把 B 转置一下.

这个代码只是一时兴起写的,如果有更好的思路,欢迎大家指教指教

mpi 学习日志(11):mpi4py 与 Spawn(没法用 MSMPI 实现)

2016 年 07 月 21 日 11:12:01 [ljhandlwt](#) 阅读数: 1109

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/51980267>

上一篇我们用 `mpi` 实现了简单的并行矩阵乘法算法.

在这个算法中,有一个缺陷,进程数必须事先确定好.

没错,至今为止我们写过的 `mpi` 代码都是事先规定好进程数的.

那么,能不能动态创建进程?能不能在运行时,根据需要创建不同数目的进程?

答案当然是可以的!用 `Spawn` 函数就可以!

代码:

test.py(master)

```
1  #coding=utf-8
2  from mpi4py import MPI
3  from numpy import *
4  import sys
5
6  data = [1,2,3]
7  new_comm = MPI.COMM_SELF.Spawn(sys.executable, args=['test2.py'], maxprocs=3)
8  new_comm.bcast(data, root=0)
9
```

test2.py(slave)


```
1  #coding=utf-8
2  from mpi4py import MPI
3  from numpy import *
4  import sys
5
6  comm = MPI.Comm.Get_parent()
7  comm_rank = comm.Get_rank()
8  data = comm.bcast(None, root=0)
9  print comm_rank, data
10
```

代码解释:

我们第一次出现了两份代码!

test.py 是父进程的代码,而 test2.py 是子进程的代码.

当然,你可以让父进程和子进程使用同一份代码,只是你要像之前一样小心翼翼地处理不同角色的进程的行为.

test.py 的第七行是我们今天学习的重点,这里使用了我们今天要重点学习的 Spawn 函数.

我们发现,Spawn 方法的对象居然是 MPI.COMM_SELF,而不是一直使用的 COMM_WORLD.

我们还记得系统默认创建的三个通信子,COMM_WORLD,COMM_SELF,COMM_NULL.

COMM_WORLD 提供了一个全局通信的服务,COMM_NULL 用于 Create()时无效通信子的判断.

而 COMM_SELF,当时只是说这是一个只有自己的通信子,并没有说它有什么作用.

没错,它的作用就是在这里,它用来动态创建子进程.

Spawn 的第一个参数是 sys.executable,啥???

其实第一个参数是子进程的程序名,就是 xxx.exe.

而我们使用 python 来运行 mpi 并行程序,运行的不是某个编译好的 exe,而是 python 解释器.

这在第一篇就讲过了,因此这里其实填的是'python.exe'.

那 sys.executable 到底是什么?简单来说就是'python.exe',更复杂的理解自行百度.

Spawn 的 args 参数大家应该都清楚了,就是传入子进程的参数表.

对于 python.exe 来说,第一个参数就是要运行的 py 文件名,所以我们要把 test2.py 传进去,表示子进程要运行 test2.py 文件.

如果你要传更多的参数,可以写在那个列表里,只是要记住 py 文件名必须是第 0 个元素.

Spawn 的 maxprocs 参数,其实就是要创建的进程数.这里我创建了 3 个子进程.

Spawn 的返回值,当然是一个通信子.

它包括了父进程和所有的子进程.

并且,父进程在这个通信子的 rank 是 0(不确定).

父进程通过返回值得到了通信子,那子进程呢?子进程又没有调用 Spawn 函数.

答案是 test2.py 第 6 行的 MPI.COMM.Get_parent 函数,通过这个函数可以得到包含父进程和所有子进程的那一个通信组的通信子.

之后这个通信子的用法和前面的一样了.

我们这里父进程尝试把一个列表广播出去,子进程收到列表后会打印出来.

运行结果:

说了这么多,为什么没有看到运行结果呢?

因为运行不出来.....msmpi 不允许动态创建进程,或者说还没有实现.....

你硬要运行的话,只会得到一个 function not implemented 的异常.....

原因参考:<http://stackoverflow.com/questions/5214525/mpi-comm-spawn-fails-on-msmpi>



According to [this document](#), it is not supported, at least not in Windows HPC Server 2008.

2

From the document:



MS MPI includes a complete set of MPI2 functionality as implemented in MPICH2 **with the exception of dynamic process spawn** and publishing, which may be included in a later release.

[share](#) [improve this answer](#)

edited Mar 7 '11 at 16:59



Shawn Chin

42k ● 8 ● 93 ● 137

answered Mar 7 '11 at 1:07



Jeremiah Willcock

18.4k ● 2 ● 46 ● 68

mpi 学习日志(12):mpi4py 与需要 buf 的大写版本函数

2016 年 08 月 16 日 16:47:47 [ljhandlwt](#) 阅读数: 728

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/52222272>

在第二篇我们说过 `mpi` 里的函数很多都有大写和小写的版本,并且大写的版本是需要 `buffer` 的,但那时候并没有说明怎么使用和为什么要这样用,今天就让我填这个坑吧.

为什么?

这次先说为什么,为什么需要有大写版本的函数?

事实上我们是把因果颠倒了,在 `C` 语言里,`MPI` 只有大写版本的函数,没有小写版本的.

显然,关注效率的 `C` 语言当然会更喜欢你提供一个 `buffer`,然后拷贝复制,而不是通过返回值传递,这样可以避免很多无谓的数据复制.

但对于 `python` 来说,编程的效率更重要,于是才有了小写版本的函数.

其次,为什么我们还需要学习大写版本的函数?

因为有一些函数没有小写版本!(比如非阻塞型的广播)

这是一个很致命的原因.

怎么做?

`python` 有不少库能提供 `buffer` 型的数据类型,比如说 `numpy` 的 `array`.

下面我们使用 `numpy` 的 `array` 充当 `buffer`,尝试使用大写版本的 `Send` 和 `Recv` 来点对点通信.

代码:

```

1  #coding=utf-8
2  from mpi4py import MPI
3  import numpy
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7
8  if comm_rank == 0:
9      data = numpy.array([123,456,789], dtype=numpy.int)
10     comm.Send([data,MPI.INT], dest=1)
11 else:
12     buf = numpy.empty((3,), dtype=numpy.int)
13     comm.Recv([buf,MPI.INT], source=0)
14     print buf
15

```

运行结果:

```

D:\Python27\newcode\study\mpi>mpiexec -n 2 python 21.py
[123 456 789]

```

代码解释:

关于 numpy 的 array 的使用方法,请移步百度,这里不多讲.

代码里进程 0 创建了一个放有 3 个 int 的数组,并向进程 1 发送.
而进程 1 也创建了一个放有 3 个 int 的数组作为 buffer,通过 Recv 函数接收.

这里第一个重点是大写版本函数的用法和以前小写版本函数的区别.
首先,没有返回值,也就是接收数据不通过返回值,而是通过参数.
其次,发送和接收数据的格式不一样.你需要提供一个列表(或元组).
列表第 0 项放的是 buffer 的名字(相当于 C 里的指针),第 1 项放的是数据类型,
这里是 MPI.INT

事实上你还可以指定你要发送多少个数据(在 C 里这是必须的,指针本身不知道自己有多少个数据),格式是[buffer,count,type]

数据类型:

关于数据类型,这里有一个常用的简单对应表.

numpy. array	MPI	python
numpy. int	MPI. INT	int
numpy. float	MPI. DOUBLE	float
numpy. int64	MPI. INTEGER64	int

这里的 `int` 一般就是 4 个字节的 `int`,而 `float` 则是 8 个字节的 `double`.
由于 `buffer` 里的整数都是像 C 里的一样,有范围限制的,而 `python` 的 `int` 是没有的,因此使用的时候必须要小心溢出.

注意:

`python` 的列表 `list` 并不能充当 `buffer`,因为它本质不是一个 C 里的固定好类型的数组,而 `numpy` 的 `array` 则是这种类型固定的数组

mpi 学习日志(13):mpi4py 与非阻塞型函数

2016 年 08 月 16 日 18:02:44 [ljhandlwt](#) 阅读数: 487

版权声明: 本文为博主原创文章, 未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/52222878>

继续填坑.

前面第二篇也说过,很多函数有分阻塞型版本和非阻塞型版本,非阻塞型版本的名字比阻塞型的名字多一个 `i` 前缀.

那么,非阻塞型函数又是怎么回事?

我们知道,点对点通信里的 `recv` 函数是一个阻塞函数,也就是接收方要等发送方发送了信息,函数才能返回.

那么对应的,非阻塞型函数就是,不管发送方是否发送了信息,函数都马上返回,返回一个 `request` 对象.

我们可以通过 `request` 对象来测试发送方是否发送了信息.

`mpi4py` 并不能很好地实现小写版本的非阻塞函数,因此在这里我们习惯只要是非阻塞函数,我们都使用大写版本,即是需要 `buffer` 传递信息的版本.

代码:

```

1  #coding=utf-8
2  from mpi4py import MPI
3  import numpy
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7
8  if comm_rank == 0:
9      data = numpy.array([1,2,3], dtype=numpy.int)
10     request = comm.Isend([data,MPI.INT], dest=1)
11 else:
12     data = numpy.zeros(3, dtype=numpy.int)
13     request = comm.Irecv([data,MPI.INT], source=0)
14     cnt = 0
15     while not request.Test():
16         cnt += 1
17         if cnt % 1000 == 0:
18             print 'I wait for %d times' % cnt
19     print data
20

```

运行结果:

```

D:\Python27\newcode\study\mpi>mpiexec -n 2 python 21.py
I wait for 1000 times
I wait for 2000 times
I wait for 3000 times
I wait for 4000 times
I wait for 5000 times
[1 2 3]
D:\Python27\newcode\study\mpi>_

```

代码解释:

- 关于 buffer 型的大写版本函数的使用请参考上一篇,有一点奇怪的是使用 `numpy.empty` 创建数组时会出错,建议大家都用 `numpy.zeros` 创建全 0 的数组.
- 无论是 `Isend` 还是 `Irecv` 都会返回一个 `request` 对象,只是我这里没有让发送方也等待消息的发送.
- `request` 对象的 `Test()` 方法会返回一个 `bool` 值,表示信息是否已经完全发送/接收.
- `request` 对象除了 `Test()` 还有一个常用的 `Wait()` 方法,它会重新让进程阻塞地等待消息的发送.也就是说当 `Wait()` 返回了,消息也就已经完全发送/接收了.

我们将会在后面对 `request` 作更多的介绍.

mpi 学习日志(14):mpi4py 与 probe

2016 年 08 月 16 日 20:18:40 [ljhandlwt](#) 阅读数: 604 标签: [mpi](#) [mpi4py](#) [更多](#)

个人分类: [mpi 学习](#)

版权声明：本文为博主原创文章，未经博主允许不得转载。

<https://blog.csdn.net/ljhandlwt/article/details/52224355>

今天我们来学习一个颇重要的函数,probe.

probe 的中文意思是"探查",那么我们可能已经猜出这个函数的作用了,probe 函数是用来探

查当前进程是否收到消息的。

probe 函数也有大小写版本和阻塞非阻塞版本,合起来就有 4 个探查函数了。

不过,貌似大小写版本在使用是一样的,只是实现上不一样。(或者是我还未发现效果上的差别)

而阻塞和非阻塞版本则有差别,阻塞版本的 probe 必须在收到消息后才返回,返回一个 true。而非阻塞版本的 iprobe 则是马上返回一个 true 或者 false,true 表示有消息,false 则是没有消息。

无论是哪个版本的 probe 函数,它的参数表都是一样的。

第一个参数是 source,用来限制发送方,默认是 all。

第二个参数是 tag,用来限制 tag。我们好像未讨论过 tag,其实在之前的许多函数里都有 tag 参数,它是 int 类型,用来给消息打标签,来区分不同的消息。默认也是 all。

第三个参数是 status,它是一个 output 参数,就是说你给它传递一个空的 status 对象,函数返回后这个 status 对象会存放一些返回信息,例如 source 和 tag。默认是 none。

(status 对象还有很多的属性,以后可能会介绍)

老实说,一开始我没有想到这个 probe 函数有什么意义,它大概是实现一些很高级的同步问题。

后来我发现它能实现一些有趣的功能,比如上一篇说道,mpi4py 的 irecv 并不能很好地实现,而现在我们可以用 iprobe 和 recv 来实现 irecv。

实现 irecv 有什么好处,好处就是你可以随意地发送数据了,比如发送一个字典 dict,发送一个混杂着字符串和数字的列表 list。

代码:

```

1  #coding=utf-8
2  from mpi4py import MPI
3  import numpy
4
5  comm = MPI.COMM_WORLD
6  comm_rank = comm.Get_rank()
7
8  if comm_rank == 0:
9      d = {'mama':1,'papa':2}
10     comm.send(d, dest=1, tag=1)
11 else:
12     s = MPI.Status()
13     cnt = 0
14     while not comm.iprobe(source=0, tag=1, status=s):
15         #do something you like
16         cnt += 1
17         if cnt % 100 == 0:
18             print 'wait for %d times' % cnt
19     data = comm.recv(source=0, tag=1)
20     print data
21

```

运行结果:

```

C:\WINDOWS\system32\cmd.exe

D:\Python27\newcode\study\mpi>mpiexec -n 2 python 22.py
wait for 100 times
wait for 200 times
wait for 300 times
wait for 400 times
wait for 500 times
wait for 600 times
wait for 700 times
{'mama': 1, 'papa': 2}

D:\Python27\newcode\study\mpi>

```

代码解释:

这个代码的样子和上一篇的非阻塞 `lrecv` 是几乎一样的,并且我们在这里不需要被 `buffer` 限制,可以随心所欲.

不过有一点要注意的是,**probe** 函数只是针对点对点通信的探查,它并不能用来探查广播之类的多点通信.

要想非阻塞地广播,还是老老实实使用 `lbcst`.