

高性能计算与云计算

实 验 报 告

(2024-2025学年第一学期)

提交日期： 2024年12月 15 日

|  |  |  |  |
| --- | --- | --- | --- |
| 实验题目 | 实验2——MPI通信实验 | | |
| 学号 | 202230441138 | 姓名 | 黄鸿展 |
| 学院 | 计算机科学与工程学院 | 任课教师 | 何克晶 |
| 课程名称 | 高性能计算与云计算 | 课程编号 | 045101911 |

1. **实验目的**

1. 学习如何使用点对点通信指令

2. 了解几种发送和接收函数的优缺点

3. 了解在使用点对点指令时，哪些是常见的陷阱

4. 学习如何使用基本的集体通信指令

5. 学习如何使用 MPI 通信器

**二、实验过程**

**1. 问题陈述**

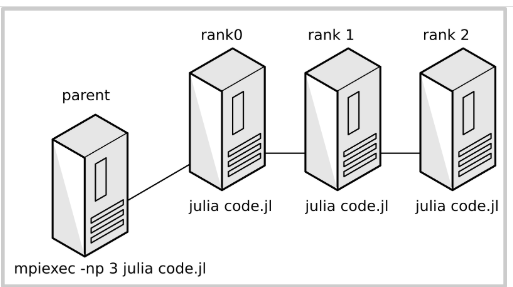
MPI概念：

MPI，即“Message Passing Interface”（消息传递接口），是一种在分布式存储系统中用于并行进程通信的标准化库协议。自20世纪90年代起，它已成为高性能计算领域分布式计算的标杆。这一标准有多种实现方式，包括OpenMPI、MPICH和IntelMPI等。通过Julia语言的MPI.jl包，可以轻松地将Julia语言与MPI绑定。MPI.jl为Julia提供了一个简洁的API，以便访问MPI的功能。用户可以通过MPI.COMM\_WORLD或者MPI.Comm\_rank(comm)等指令来获取进程信息，也可以使用ccall来调用C语言函数。MPI程序的执行必须从初始化开始，这标志着一个完整的MPI程序的启动。

|  |
| --- |
| using MPI MPI.Init() *# Your MPI program here* MPI.Finalize() *# Optional，也可以用Abort()提前终止程序*  *# HelloWorld*  using MPI MPI.Init() comm = MPI.COMM\_WORLD nranks = MPI.Comm\_size(comm) rank = MPI.Comm\_rank(comm) host = MPI.Get\_processor\_name() println("Hello from $host, I am process $rank of $nranks processes!") |

MPI进程：

MPI进程是由名为mpiexec的启动器创建的，这个启动器是MPI安装包的一部分。mpiexec接受应用程序作为参数，负责生成多个进程，并在这些进程上执行应用程序。这些应用程序通常会使用MPI的指令来实现进程间的通信。



我们可以用这种方法创建julia文件并运行：

|  |
| --- |
| code = raw""" using MPI MPI.Init() comm = MPI.COMM\_WORLD nranks = MPI.Comm\_size(comm) rank = MPI.Comm\_rank(comm) host = MPI.Get\_processor\_name() println("Hello from $host, I am process $rank of $nranks processes!") MPI.Finalize() """ filename = tempname()\*".jl" write(filename,code) filename *#在julia中输入* using MPI run(`$(mpiexec()) -np 4 julia --project=. $filename`); |

也可以使用quote在notebook中直接运行：

|  |
| --- |
| code = quote     using MPI     MPI.Init()     comm = MPI.COMM\_WORLD     nranks = MPI.Comm\_size(comm)     rank = MPI.Comm\_rank(comm)     host = MPI.Get\_processor\_name()     println("Hello from $host, I am process $rank of $nranks processes!")     MPI.Finalize() end run(`$(mpiexec()) -np 4 julia --project=. -e $code`); |

一个完整的MPI程序需要被包含在提供给Julia的源代码文件或模块中。较长的MPI程序通常会被分割成多个文件，并组织成Julia包的形式，随后由各个MPI进程进行加载。

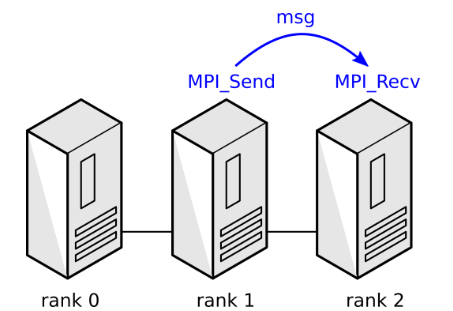
**2. MPI点对点通信**

MPI点对点通信 MPI提供了一系列的指令来支持进程间的任意点对点通信。这种通信是双向的，涉及一个发送进程和一个接收进程。以下是不同类型的指令，它们都用于从一个进程发送消息到另一个进程接收消息。

① MPI\_Send 和 MPI\_Recv：这是完整的（阻塞型）指令。

② MPI\_Isend 和 MPI\_Irecv：这是不完整的（非阻塞型）指令。

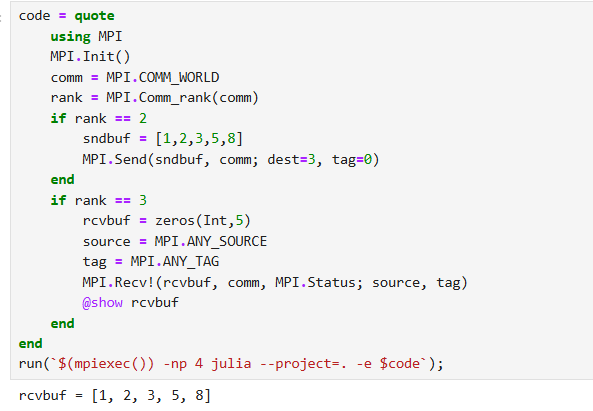
③ MPI\_Bsend、MPI\_Ssend 和 MPI\_Rsend：这些是高级通信模式。



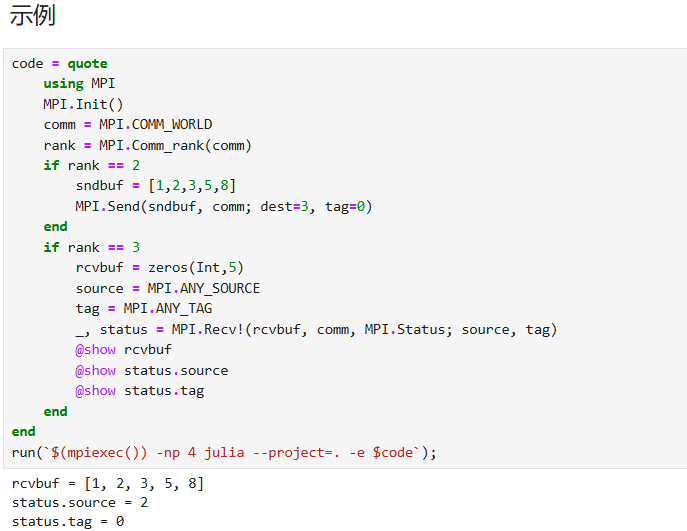
1. **完整操作**

|  |
| --- |
| *#In Julia* MPI.Send(sndbuf, comm; dest, tag) \_, status = MPI.Recv!(rcvbuf, comm, MPI.Status; source, tag) *#sndbuf 要发送的数据。* *#rcvbuf 用于存储接收数据的空间。* *#source 发送方的秩。* *#dest 接收方的秩。* *#tag 可能用于区分同一发送者发送给同一接收者的不同类型的信息,主题。* |

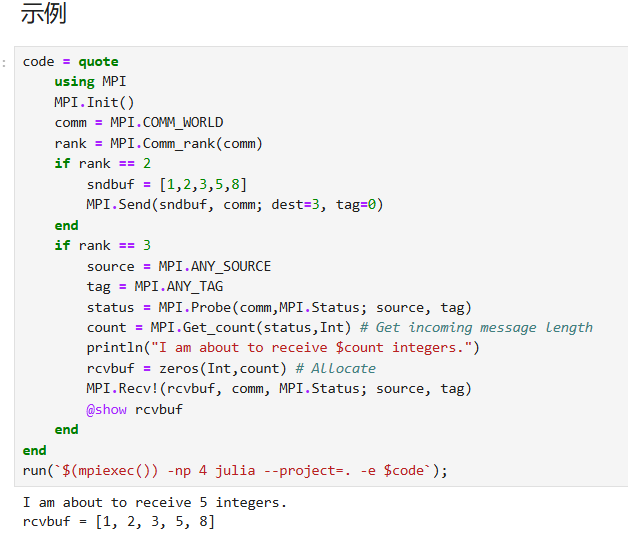
可以用source = MPI.ANY\_SOURCE，tag = MPI.ANY\_TAG接收任何来源/标记信息。如下例，从等级 2 向等级 3 发送 5 个整数：



在Recv中，可以用MPI\_STATUS查看信息来源：



可以通过MPI\_Probe了解接收信息的大小：



注意，其中的操作为完整操作，在函数返回后可以马上用其中的参数。

MPI\_Probe和MPI\_Recv是阻塞性操作，它们会暂停执行直到匹配的信息被发送。而MPI\_Send函数并不总是阻塞性的，但有时也可能引发死锁。对于小规模数据（例如n=1），MPI实现倾向于缓存这些小规模数据，并尽快从MPI\_Send返回。对于大规模数据（例如n=10000），MPI实现可能会选择等待匹配的接收操作，以减少数据复制。在调用MPI\_Send时，如果两个进程都被阻塞，且接收信息未被发布，可能会导致死锁。通过精心安排发送和接收的顺序，可以解决这些循环依赖问题。

|  |
| --- |
| code = quote     using MPI     MPI.Init()     comm = MPI.COMM\_WORLD     rank = MPI.Comm\_rank(comm)     n = 10000     sndbuf = fill(rank,n)     rcvbuf = zeros(Int,n)     if rank == 2         *# We first send and then receive*         MPI.Send(sndbuf, comm; dest=3, tag=0)         MPI.Recv!(rcvbuf, comm, MPI.Status; source=3, tag=0)     end     if rank == 3         *# We first receive and then send*         MPI.Recv!(rcvbuf, comm, MPI.Status; source=2, tag=0)         MPI.Send(sndbuf, comm; dest=2, tag=0)     end     @show (rcvbuf[1],rank) end run(`$(mpiexec()) -np 4 julia --project=. -e $code`); |

也可以使用，会为我们处理排序：

|  |
| --- |
| code = quote     using MPI     MPI.Init()     comm = MPI.COMM\_WORLD     rank = MPI.Comm\_rank(comm)     n = 10000     sndbuf = fill(rank,n)     rcvbuf = zeros(Int,n)     if rank == 2         MPI.Sendrecv!(sndbuf,rcvbuf, comm;dest=3,source=3,sendtag=0,recvtag=0)     end     if rank == 3         MPI.Sendrecv!(sndbuf,rcvbuf, comm;dest=2,source=2,sendtag=0,recvtag=0)     end     @show (rcvbuf[1],rank) end run(`$(mpiexec()) -np 4 julia --project=. -e $code`); |

**② 不完整操作**

如果作为用户希望决定何时存储消息或避免数据复制，可以使用通信模式。共有四种模式：常规模式、缓冲区模式、同步模式和准备模式，分别通过函数MPI\_Send、MPI\_Bsend、MPI\_Ssend和MPI\_Rsend实现。



函数MPI\_Isend和MPI\_Irecv中的"I"代表它们是非阻塞操作。这些函数执行完毕后，并不保证底层的通信操作已经彻底完成，通常需要配合使用MPI\_Wait函数来等待发送和/或接收操作的最终完成。它们的主要优势之一在于，能够实现通信与计算的并行执行，也就是所谓的延迟掩盖。在等待匹配的发送和数据传输的同时，可以执行本地计算工作。最终，通过调用MPI\_Wait来确保接收操作的完成。

|  |
| --- |
| code = quote     using MPI     MPI.Init()     work() = sum(rand(1000))     comm = MPI.COMM\_WORLD     rank = MPI.Comm\_rank(comm)     if rank == 2         sndbuf = [1,2,3,5,8]         request = MPI.Isend(sndbuf, comm; dest=3, tag=0)         work() *# Overlap local work with communication / synchronization*         MPI.Wait(request)     end     if rank == 3         rcvbuf = zeros(Int,5)         request = MPI.Irecv!(rcvbuf, comm; source=2, tag=0)         work()  *# Overlap local work with communication / synchronization*         MPI.Wait(request)         @show rcvbuf     end end run(`$(mpiexec()) -np 4 julia --project=. -e $code`); |

函数MPI\_Iprobe可以作为MPI\_Probe的替代，它不会等待匹配的发送信息，而是会立即返回。ismsg的值将指示是否有消息发送。如果存在发送，那么可以通过status对象来获取传入消息的详细信息。

**③ 点对点通信实现矩阵并行乘法&电话算法**

**Exercise 1**

使用MPI而非分布式计算来实现之前笔记本中的并行矩阵-矩阵乘法（算法3）。使用以下函数签名：function matmul\_mpi\_3!(C,A,B)。假设输入矩阵A和B只在0号进程中给出，其他进程获得的是空矩阵以节省内存。结果矩阵C仅应在0号进程上进行更新。按照入门教程中的说明，使用run函数从Julia REPL运行它。

设计算法如下：

|  |
| --- |
| function matmul\_mpi\_3!(C, A, B)     comm = MPI.COMM\_WORLD  *# 获取 MPI 通信器*     rank = MPI.Comm\_rank(comm)  *# 获取当前进程的 rank*     P = MPI.Comm\_size(comm)  *# 获取进程总数*     *# 主进程（rank == 0）负责分发矩阵 B*     if rank == 0         N = size(A, 1)  *# 获取矩阵 A 的行数，即矩阵 C 的行数*         B\_local = B  *# 主进程直接使用全局矩阵 B*         *# 将矩阵 B 发送给其他进程*         for dest in 1:(P - 1)             MPI.Send(B, comm; dest)         end     else         source = 0  *# 从主进程接收矩阵 B*         status = MPI.Probe(comm, MPI.Status; source)  *# 探测消息*         count = MPI.Get\_count(status, eltype(B))  *# 获取消息的大小*         N = Int(sqrt(count))  *# 计算矩阵 B 的维度*         B\_local = zeros(N, N)  *# 初始化局部矩阵 B*         MPI.Recv!(B\_local, comm; source)  *# 接收矩阵 B*     end     *# 计算每个进程处理的行数*     L = div(N, P)  *# 每个进程处理的行数*     A\_local = zeros(L, N)  *# 初始化局部矩阵 A*     *# 主进程（rank == 0）负责分发矩阵 A*     if rank == 0         lb = L \* rank + 1  *# 当前进程处理的行的起始索引*         ub = L \* (rank + 1)  *# 当前进程处理的行的结束索引*         A\_local[:, :] = view(A, lb:ub, :)  *# 提取当前进程负责的子矩阵 A*          *# 将矩阵 A 的子矩阵发送给其他进程*         for dest in 1:(P - 1)             lb = L \* dest + 1             ub = L \* (dest + 1)             MPI.Send(view(A, lb:ub, :), comm; dest)         end     else         source = 0  *# 从主进程接收矩阵 A*         MPI.Recv!(A\_local, comm; source)  *# 接收矩阵 A 的子矩阵*     end     *# 每个进程计算局部矩阵乘法*     C\_local = A\_local \* B\_local  *# 计算局部结果矩阵 C*     *# 主进程（rank == 0）负责收集所有进程的计算结果*     if rank == 0         lb = L \* rank + 1  *# 当前进程处理的行的起始索引*         ub = L \* (rank + 1)  *# 当前进程处理的行的结束索引*         C[lb:ub, :] = C\_local  *# 将局部结果写入全局矩阵 C*         *# 接收其他进程的计算结果*         for source in 1:(P - 1)             lb = L \* source + 1             ub = L \* (source + 1)             MPI.Recv!(view(C, lb:ub, :), comm; source)         end     else         dest = 0  *# 将局部结果发送给主进程*         MPI.Send(C\_local, comm; dest)     end     *# 返回最终的全局矩阵 C*     C end |

①获取MPI通信器comm，用于进程间通信，以及当前进程的rank和总进程数size1。

②确定矩阵的维度N，假设A和B是N x N的正方形矩阵。

③计算每个进程应该处理的行数rows\_per\_proc，即N除以进程数size1。

④为每个进程分配局部矩阵local\_A和local\_C，以及接收完整的矩阵B。

⑤如果当前进程是rank 0，则将矩阵A的行分发给其他进程。其他进程则接收。

⑥每个进程执行局部矩阵乘法，计算local\_C。

⑦如果当前进程是rank 0，则收集所有进程的局部结果并组装成完整的矩阵C。

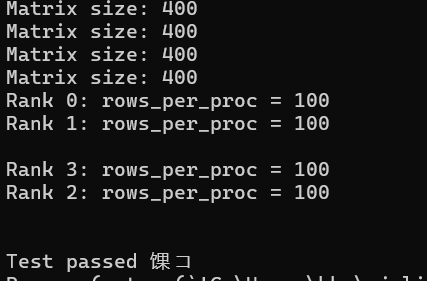
⑧同步所有进程，确保rank 0在检查结果之前已经收集到所有结果。

在julia REPL中输入

julia> using MPI

julia> run(`$(mpiexec()) -np 4 julia ex1.jl`)

运行结果：



**Exercise 2**

Rank 0 生成一条信息。Rank 0 将信息发送rank 1。等级 1 接收后，将信息递增 1，发送给等级 2。等级 2 接收信息，递增 1，发送给等级 3。而最后一级将信息发回第 0 级，完成一个循环。

以下是算法 2的主要执行步骤：

1. 工作进程从主进程接收相应的行 A[i,:] 和矩阵 B
2. 工作进程计算 A[i，:]行与 B 的乘积
3. 工作进程将 C[i,:]行的结果发回主进程

代码实现如下：

|  |
| --- |
| using MPI function main()     MPI.Init()     rank = MPI.Comm\_rank(MPI.COMM\_WORLD)     size = MPI.Comm\_size(MPI.COMM\_WORLD)     *# 初始化计数器*     counter = 0     if rank == 0         *# 等级 0 生成一条信息（整数）*         msg = 0         println("Rank $rank sending initial message: $msg")         MPI.Send(msg, mod(rank + 1, size), 0, MPI.COMM\_WORLD)     end     while true         *# 接收信息*         status = MPI.Probe(MPI.ANY\_SOURCE, MPI.ANY\_TAG, MPI.COMM\_WORLD)         (msg, \_) = MPI.Recv(Int, status.source, status.tag, MPI.COMM\_WORLD)         println("Rank $rank received message: $msg")         *# 增加计数器*         counter += 1         if rank == 0 && status.tag == 1             *# 如果等级 0 收到带有结束标记的消息，结束循环*             println("Rank $rank received final message: $msg")             break         end         *# 将信息递增 1*         msg += 1         *# 发送信息给下一个等级*         next\_rank = mod(rank + 1, size)         println("Rank $rank sending message: $msg to rank $next\_rank")         MPI.Send(msg, next\_rank, 0, MPI.COMM\_WORLD)         *# 检查是否达到 20 次通信*         if counter >= 20             if rank == 0                 println("Rank $rank: Stopping communication after $counter communications.")                 *# 发送结束标记给下一个等级*                 MPI.Send(msg, next\_rank, 1, MPI.COMM\_WORLD)             end              break  *# 停止通信*         end     end     MPI.Finalize() end main() |

①始化计数器：设置一个计数器counter，用于记录通信的次数。

②果当前进程是等级0（rank == 0），则生成一个初始消息（整数msg），并打印发送消息的信息。

③MPI.Send将消息发送给下一个进程（mod(rank + 1, size)），这里的mod函数确保消息在最后一个进程后能够循环回到第一个进程。

④无限循环接收和发送消息：

使用MPI.Probe探测是否有消息到达，并获取消息的状态（status）。

使用MPI.Recv接收消息，并打印接收到的消息。

计数器counter增加1。

如果等级0的进程收到标记为1的消息（结束标记），则打印信息并结束循环。

将接收到的消息msg递增1。发送给下一个进程。

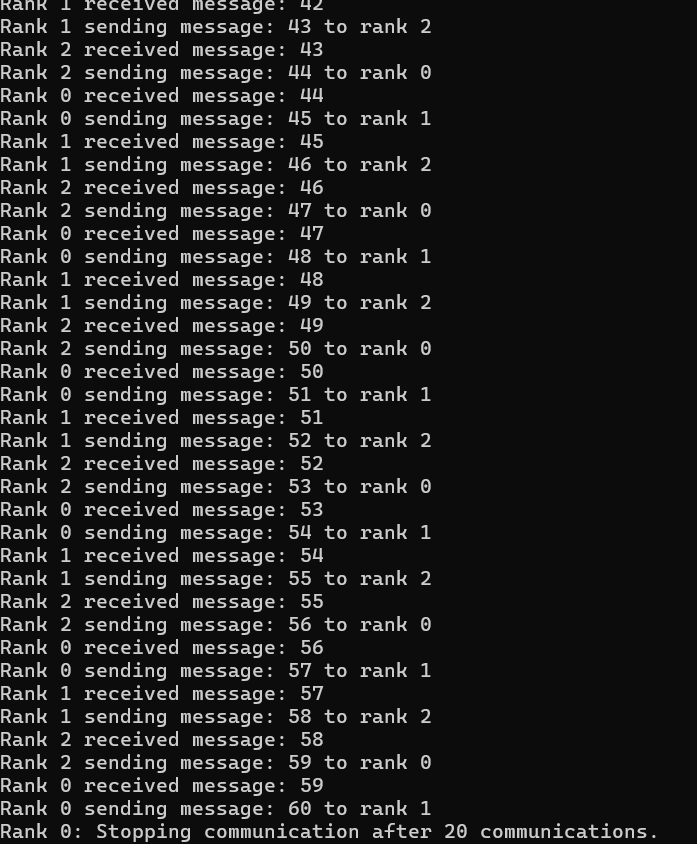
检查如果是等级0的进程且计数器达到20，发送结束标记给下个进程，结束循环。

在Julia REPL中输入：

Using MPI

run(`$(mpiexec()) -np 3 julia ex2.jl`)

结果如下：



**3. MPI集合通信**

MPI提供集体通信功能的原因主要包括以下几点：

简化使用：用户可以轻松地利用这些预定义的函数，无需自行实现复杂的通信逻辑。

提升性能：MPI库通常采用高效的算法实现，比如使用还原树算法来优化数据的收集和分发。

优化硬件利用：硬件供应商可以针对其特定的机器架构对MPI库进行优化，以实现更高效的MPI操作执行。

集体操作的关键特性包括：

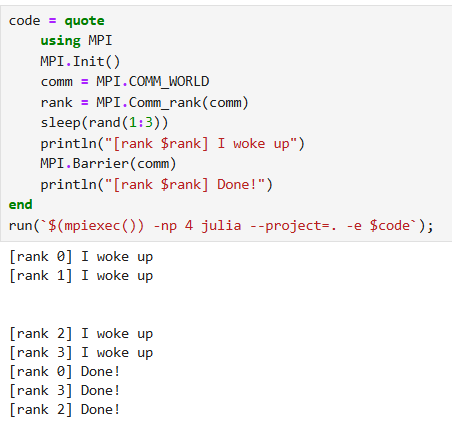
安全性：一旦集体操作的函数返回，就可以安全地使用和重置用于操作的缓冲区。

模式限制：集体指令通常只在标准模式下执行，例如MPI\_Send。

同步性：集体操作的完成并不保证其他进程也完成了它们的操作。

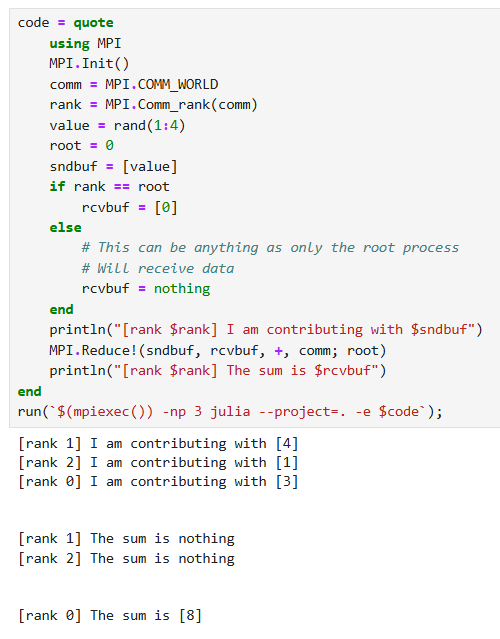
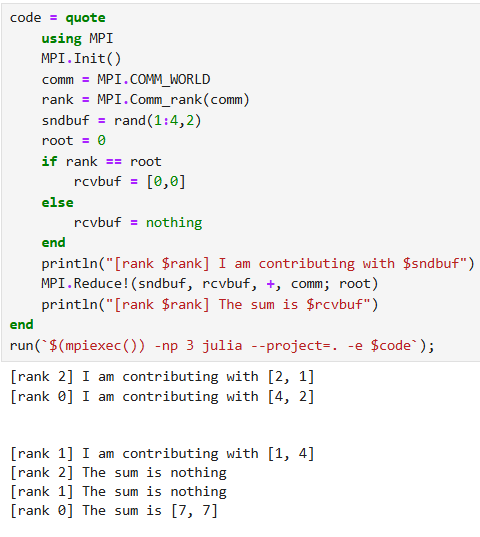
MPI中常见的集体通信操作包括：

MPI\_Barrier：这个操作用于同步一组进程。所有进程都会阻塞，直到所有进程都到达这个障碍点。它经常在循环的末尾被调用，以确保所有进程都完成了当前循环的迭代，然后才能继续执行下一个循环的迭代。



可以看到Barrier保证了在所有的进程都woke up前一直阻塞，所有进程同步工作。

MPI\_Reduce 函数是 MPI 集体通信操作之一，它的作用是将多个进程提供的数据值根据指定的归约（reduce）操作进行合并，最终的结果只由一个特定的进程——即根进程——接收。根进程可以是参与通信的任何进程，在 Julia 中，默认情况下根进程是等级为 0 的进程。

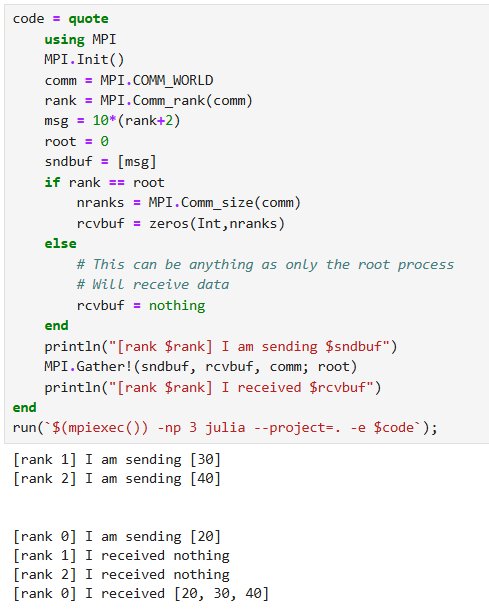
 

（简单进程加法的实现，缓冲区为矩阵时也可以传递多个数据）

MPI\_AllReduce：这是MPI\_Reduce的扩展版本，它允许所有参与通信的进程都能得到归约操作的结果。在MPI\_AllReduce操作中，每个进程首先应用归约操作（比如求和、取最大值等）到局部数据，然后将结果发送给根进程。根进程将所有进程发送的结果再次归约，并将最终结果广播给所有进程。这种操作常用于需要所有进程共享某个全局结果的场景，比如在并行算法中更新所有进程的共享变量。

MPI\_Gather：MPI\_Gather操作允许所有进程（包括根进程）将它们的消息发送给根进程。

在这个操作中，每个进程（包括根进程）都会发送数据给根进程，根进程则在其本地缓冲区中收集所有这些数据。这种操作适用于需要将分散在不同进程中的数据集中到一个进程中进行进一步处理的场景，比如数据汇总或者合并操作。



这段代码里可以看到，Gather汇集了所有进程发送的信息，通过root决定接收者。

同理，MPI\_AllGather可以让所有进程都得到结果。

MPI\_Gatherv：

MPI\_Gatherv是MPI\_Gather的变体，适用于每个进程发送的数据量可能不同的情况。

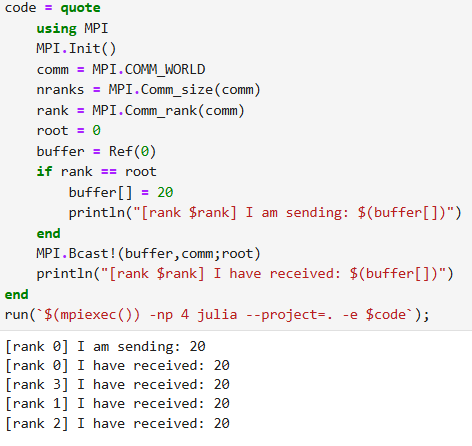
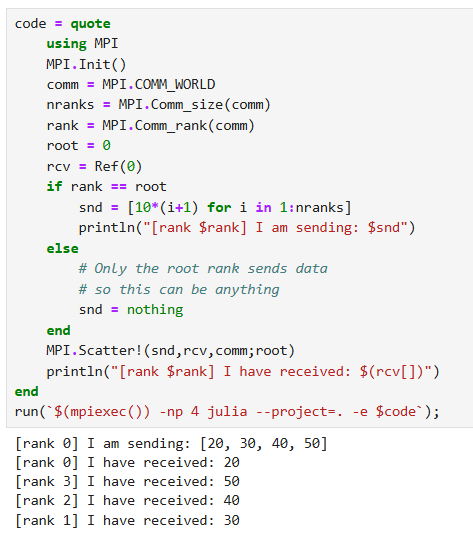
在MPI\_Gatherv操作中，每个进程向根进程发送不同数量的数据。根进程需要知道每个进程发送的数据量，并根据这些信息来正确地接收和存储数据。这种操作适用于需要收集不同大小数据的场景，比如处理不规则数据分布或者不同进程产生的数据量不一致的情况。

MPI\_Scatter：

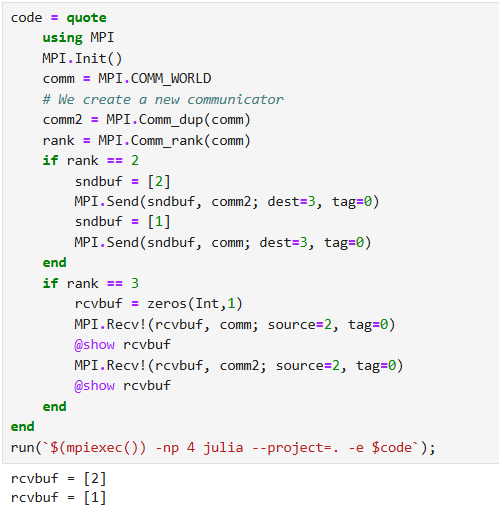
MPI\_Scatter是MPI\_Gather的逆操作，根进程将数据分发给所有其他进程。在MPI\_Scatter操作中，根进程拥有一个数据集合，并将这个集合分割成多个部分，每个部分发送给不同的进程。这种操作适用于初始化并行计算时，需要将数据分配给不同进程进行处理的场景。

MPI\_Bcast：

MPI\_Bcast（广播）操作与MPI\_Scatter类似，但根进程向所有进程发送相同的消息。在MPI\_Bcast操作中，根进程拥有一个数据项或数据集合，并将其发送给所有其他进程。这种操作适用于需要确保所有进程都有相同数据副本的场景，比如在并行计算中同步初始条件或者共享参数。

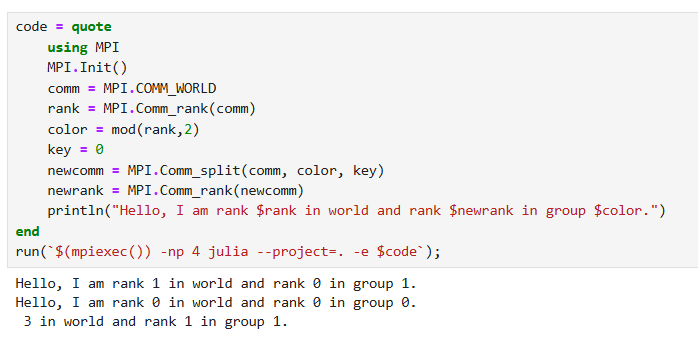


通信器：提供一个孤立的通信环境，定义一组进程。

MPI\_Comm\_dup：复制给定的通讯器，创建一个具有新通讯上下文的新通讯器。

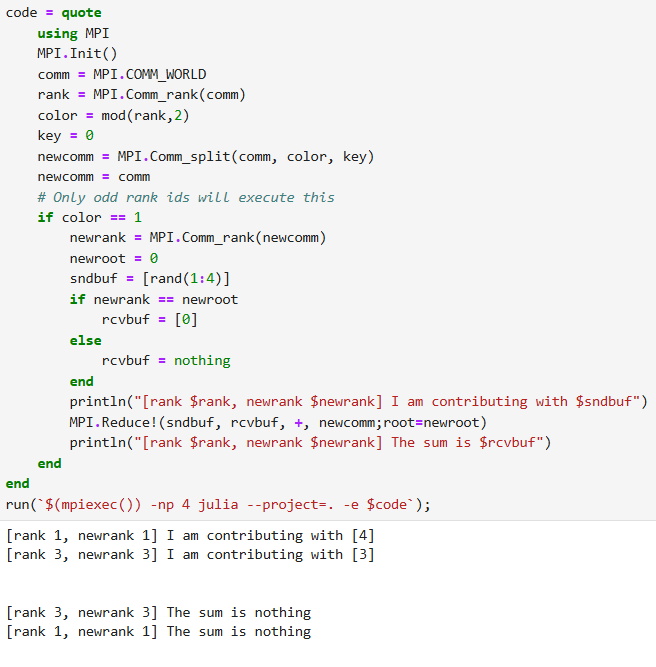
如果这里comm2=comm，那么2与1的顺序就会改变。

MPI\_Comm\_split：拆分给定的通信器。



color：所有具有相同颜色的进程将被分组到同一个新的通讯器中。

key ： 进程将根据关键字在新通讯器中进行排序，打破与旧通讯器中排序的联系。

上图可知，如果不使用新的通信器，那么就无法正常实现发送整数和到进程中。

**Exercise:** 使用 MPI 集合而不是点对点通信实现并行矩阵-矩阵乘法（算法 3）。即与上一个笔记本中的练习相同，只是使用不同的通信函数。

|  |
| --- |
| using MPI MPI.Init() function matmul\_mpi\_3!(C, A, B)     comm = MPI.COMM\_WORLD  *# 获取 MPI 通信器*     rank = MPI.Comm\_rank(comm)  *# 获取当前进程的 rank*     P = MPI.Comm\_size(comm)  *# 获取进程总数*     root = 0  *# 定义主进程的 rank*     if rank == root         N = size(A, 1)  *# 获取矩阵 A 的行数，即矩阵 C 的行数*         Nref = Ref(N)  *# 创建一个引用对象，用于广播 N*     else         Nref = Ref(0)  *# 其他进程初始化 Nref 为 0*     end     MPI.Bcast!(Nref, comm; root)      N = Nref[]      if rank == root         B\_local = B  *# 主进程直接使用全局矩阵 B*     else         B\_local = zeros(N, N)  *# 其他进程初始化局部矩阵 B*     end     MPI.Bcast!(B\_local, comm; root)  *# 广播矩阵 B 到所有进程*     *# 计算每个进程处理的列数*     L = div(N, P)      A\_local\_t = zeros(N, L)      A\_t = collect(transpose(A))      MPI.Scatter!(A\_t, A\_local\_t, comm; root)  *# 每个进程接收 A\_t 的一部分*     *# 每个进程计算局部矩阵乘法（转置形式）*     C\_local\_t = transpose(B\_local) \* A\_local\_t  *# 计算局部结果矩阵 C\_local（转置形式）*     C\_t = similar(C)  *# 初始化全局矩阵 C 的转置形式*     MPI.Gather!(C\_local\_t, C\_t, comm; root)  *# 收集所有进程的局部结果*     C .= transpose(C\_t)  *# 将转置后的结果还原为全局矩阵 C*     C end  function testit(load)     comm = MPI.COMM\_WORLD     rank = MPI.Comm\_rank(comm)     P = MPI.Comm\_size(comm)     N = load \* P     println("Matrix size: $N")     *# 在 rank 0 生成矩阵 A 和 B，并通过广播发送给所有进程*     if rank == 0         A = rand(Int64, N, N)  *# 生成 Int64 矩阵*         B = rand(Int64, N, N)  *# 生成 Int64 矩阵*     else         A = zeros(Int64, N, N)  *# 其他进程初始化空矩阵*         B = zeros(Int64, N, N)  *# 其他进程初始化空矩阵*     end     *# 广播 A 和 B 到所有进程*     MPI.Bcast!(A, 0, comm)     MPI.Bcast!(B, 0, comm)     *# 分配 C 为 Int64 矩阵*     C = zeros(eltype(A), N, N)     C = matmul\_mpi\_3!(C, A, B)  *# 调用函数并更新 C*     println("Rank $rank: A = $A")     *# 同步所有进程，确保 rank 0 在检查 C 之前已经收集到所有结果*     MPI.Barrier(comm)     if rank == 0         if !(C ≈ A \* B)             println("Test failed 😢")             println("A \* B = ", A \* B)             println("C = ", C)         else             println("Test passed 🥳")         end     end end testit(100) MPI.Finalize() |

①获取MPI通信器comm，以及当前进程的排名rank和总进程数size1。

②确定矩阵的维度N，假设A和B是N x N的正方形矩阵。

③计算每个进程应该处理的行数rows\_per\_proc，即N除以进程数size1。

④为每个进程分配局部矩阵local\_A和local\_C，以及接收完整的矩阵B。

⑤转置矩阵A并将其转换为连续的内存块A\_transpose。

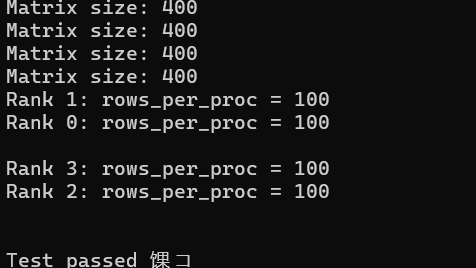
⑥使用MPI.Scatter!操作将转置后的矩阵A的行分发给所有进程。

⑦每个进程执行局部矩阵乘法，计算local\_C。

⑧使用MPI.Gather!操作收集所有进程的局部结果local\_C到rank 0进程的矩阵C中。

⑨对结果矩阵C进行转置，以确保最终结果的排布正确。

**结果如下：**

****

**三、总结**

我已经了解了使用 MPI 发送和接收消息的不同方法;还学习了如何使用MPI\_Status 、MPI\_Probe 和MPI\_Iprobe 获取有关的传入消息。，如果不正确遵循 MPI 操作的语义，就很容易编写出错误的程序。除此之外，MPI 还定义了涉及多个进程的操作，称为集体操作。

最后，我学习了 MPI 通信器。它有两个关键功能：隔离通信上下文和创建进程组。例如，在同一应用程序中结合使用 MPI 的不同库，以及在进程子集中使用集体操作时，它们都非常有用。