



# MPI

## More of the Story

Timothy H. Kaiser, PH.D.  
[tkaiser2@nrel.gov](mailto:tkaiser2@nrel.gov)



**Slides at:**  
<https://github.com/timkphd/slides>



# Examples at

```
git clone https://github.com/timkphd/examples  
cd examples/mpi/mpi4py
```

Goal for today: quickly go over things but leave you with many well commented examples and at least one useful full example program.

# Outline

- Review
- Types
- Broadcast
- Wildcards
- Using Status and Probing
- Asynchronous Communication
- More Global communications
- Advanced topics
  - ~~"V"~~ operations
  - Communicators



# Outline: Advance examples

- Finite difference code
- Mixing mpi4py and C or Fortran
- Bag of tasks
- Passing a token

# Summary

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are:

- `INIT()` “not required”

- `comm=MPI.COMM_WORLD`

- `comm.Get_rank()`

- `comm.Get_size()`

- `comm.Send(buf, dest, tag=0)`

- `comm.Recv(buf, source=ANY_SOURCE, tag=ANY_TAG, Status status=None)`

- `MPI.Finalize()`



# Basic Send and Receive

```
#!/usr/bin/env python
# numpy is required
import numpy
from numpy import *

# mpi4py module
from mpi4py import MPI

# Initialize MPI and print out hello
comm=MPI.COMM_WORLD
myid=comm.Get_rank()
numprocs=comm.Get_size()
print("hello from ",myid," of ",numprocs)

# Tag identifies a message
mytag=1234

# Process 0 is going to send the data
mysource=0

# Process 1 is going to send the data
mydestination=1

# Sending a single value each time
count=1
for k in range(1,4):
    if myid == mysource:
# For the upper case calls we need to send/recv numpy arrays
        buffer=array(k+5678,"i")
# We are sending a integer, size is optional, to mydestination
        comm.Send([buffer, MPI.INT], dest=mydestination, tag=mytag)
        print("Python processor ",myid," sent ",buffer)

    if myid == mydestination:
# We are receiving an integer, size is optional, from mysource
        if(k == 1) : buffer=empty((1),"i")
        comm.Recv([buffer, MPI.INT], source=mysource, tag=mytag)
        print("Python processor ",myid," got ",buffer)

MPI.Finalize()
```

## P\_ex01.py , f\_ex01.f90

Blocking Send and Receive

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

## P\_ex01b.py

Blocking Send and Receive *Character Data*

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

Skip to #18

# Our Examples

## P\_ex00.py

Hello world

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- MPI.Finalize()

## P\_ex01.py , f\_ex01.f90

Blocking Send and Receive

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

## P\_ex01b.py

Blocking Send and Receive *Character Data*

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

## P\_ex02.py

Blocking Send and Receive with probe to find size

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Send()
- comm.probe()
- mystat.Get\_count()
- comm.Recv()
- MPI.Finalize

## P\_ex03.py

Nonblocking Send and Receive with wait

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.isend()
- comm.irecv()
- req.wait()
- MPI.Finalize



# Our Examples

## P\_ex03I.py

Nonblocking Send and Receive with wait

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Isend()
- comm.Irecv()
- req.wait()
- MPI.Finalize

## P\_ex04.py

Broadcast of an array of integers and a string

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.bcast()
- comm.Bcast()
- MPI.Finalize

## P\_ex05.py

This program shows how to use MPI\_Scatter and MPI\_Gather. Each processor gets different data from the root processor by way of mpi\_scatter. The data is summed and then sent back to the root processor using MPI\_Gather. The root processor then prints the global sum.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.bcast()
- comm.Scatter()
- comm.gather()
- comm.Gather()
- MPI.Finalize

# Our Examples

## [P\\_ex06.py](#)

This program shows how to use MPI\_Scatter and MPI\_Reduce. Each processor gets different data from the root processor by way of mpi\_scatter. The data is summed and then sent back to the root processor using MPI\_Reduce. The root processor then prints the global sum.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.bcast()
- comm.Scatter()
- `comm.reduce()`
- `comm.Reduce()`
- MPI.Finalize

## [P\\_ex07.py](#)

This program shows how to use MPI\_Alltoall. Each processor send/rec a different random number to/from other processors.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- `comm.Alltoall()`
- MPI.Finalize



# Our Examples

## P\_ex08.py

This program shows how to use MPI\_Gatherv. Each processor sends a different amount of data to the root processor. We use MPI\_Gather first to tell the root how much data is going to be sent.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.gather
- comm.Gatherv
- MPI.Finalize

## P\_ex09.py

This program shows how to use Alltoallv. Each processor gets amounts of data from each other processor. It is an extension to example P\_ex07.py. In mpi4py the displacement array can be calculated automatically from the rcounts array. We show how it would be done in "normal" MPI. See also P\_ex08.py for how this can be done

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Alltoall()
- comm.Alltoallv()
- MPI.Finalize

# Our Examples

## P\_ex10.py , ex10.in

Pass a "token" from one task to the next with a single task reading token from a file.

An extensive program. We first create a new communicator that contains every task except the zeroth one. This is done by first defining a group, `new_group`. We define `new_group` based on the group associated with `mpi_comm_world`, `old_group` and an array, `will_use`. `Will_use` contains a list of tasks to be included in the new group. It does not contain the zeroth one. The new communicator is `sub_comm_world`.

There are other ways to create communicator but this is one of the more general methods.

Next, we have break of the task not in the communicator to call the routine `get_input`. This routine will do input from a file `ex10.in`. The file contains a list of integers. The task will send the integer to the first task in the new communicator.

The remaining tasks which are port of `sub_comm_world` call the routine `pass_token`. `Pass_token` "just" receives a value from the previous processor and passes it on to the next.

There is a minor subtlety in `pass_token`. We are using both our new communicator and `MPI_COMM_WORLD`. The tasks that are port of the new communicator use it to pass data. We note that the task that is injecting values into the stream is not part of the new communicator so it must use `MPI_COMM_WORLD`. Thus we do a probe on `WORLD`, which is actually `MPI_COMM_WORLD` looking for a message. When we get it we send it on using the new communicator.

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `WORLD.Iprobe()`
- `comm.Get_group()`
- `old_group.Incl()`
- `comm.Create()`
- `new_group.Get_rank()`
- `MPI.Finalize`



# Our Examples

## [P\\_ex12.py](#)

This program shows how to use `mpi_comm_split`

`Split` will create a set of communicators. All of the tasks with the same value of `color` will be in the same communicator. In this case we get two sets one for odd tasks and one for even tasks. Note they have the same name on all tasks, `new_comm`, but there are actually two different values (sets of tasks) in each one.

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.Split`
- `comm.bcast`
- `MPI.Finalize`

## [P\\_ex13.py](#)

This program shows how to use `Scatterv`. Each processor gets a different amount of data from the root processor. We use `MPI_Gather` first to tell the root how much data is going to be sent.

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.gather`
- `comm.Scatterv`
- `MPI.Finalize`

# Our Examples

## [simple.py](#) , [flist](#)

This is a bag-of-tasks program. We define a manager task that distributes work to workers. Actually, the workers request input data. The manager sits in a loop calling `Iprobe` waiting for requests for work. In this case the manager reads input. The input is a list of file names. It will send a entry from the list as requested. When the worker is done processing it will request a new file name from the manager. This continues until the manager runs out of files to process. The manager subroutine is just "manager". The worker subroutine is "worker". It receives file names form the manager. The files in this case are outputs from an optics program tracking a laser beam as it propagates through the atmosphere. The workers read in the data and then create an image of the data by calling the routine `mkview.plotit`. This should worker with arbitrary 2d files except the size in `mkview.plotit` is currently hard coded to 64 x 64. We use the call to "Split" to create a seperate communicator for the workers. This is not important in this example but could be if you wanted multiple workers to work together.

To get the data...

```
curl http://hpc.mines.edu/examples/laser.tgz | tar -xz
```

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.gather`
- `comm.Send()`
- `comm.Recv()`
- `MPI.Status()`
- `comm.Iprobe()`
- `gotfrom=status.source`
- `MPI.Get_processor_name()`
- `MPI_COMM_WORLD.barrier()`
- `MPI.Finalize`



# Our Examples

File	Comment
<a href="#"><u>ccalc.c</u></a>	<b>parallel</b>
<a href="#"><u>stc_03.c</u></a>	<b>parallel</b>
<a href="#"><u>pcalc.py</u></a>	<b>parallel</b>
<a href="#"><u>stp_00.py</u></a>	<b>serial</b>
<a href="#"><u>stp.py</u></a>	<b>parallel</b>
<a href="#"><u>tiny.in</u></a>	<b>tiny input file</b>
<a href="#"><u>small.in</u></a>	<b>small input file</b>
<a href="#"><u>st.in</u></a>	<b>regular input file</b>

We have a finite difference model that will serve to demonstrate what a computational scientist needs to do to take advantage of Distributed Memory computers using MPI.

The model we are using is a two dimensional solution to a model problem for Ocean Circulation, the Stommel Model. It has Wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom and Coriolis force. Solution: intense crowding of streamlines towards the western boundary caused by the variation of the Coriolis parameter with latitude.

The python version, stp.py, follows this C version except it does a 1d decomposition.

The C version is 1500x faster than the python version.

pcalc.py and ccalc.c are similar except they create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. Thus we can have "C" do the heavy calculation and python do plotting.

# Our Examples

## [pwrite.py](#)

pwrite.py is a small MPI program designed to be run in conjunction with either ccalc.c or pcalc.py. If you are using mpiexec to launch your programs these might be launched together using one of the commands:

```
mpiexec -n 5 ./ccalc      : -n 1 ./pwrite.py < cut.in
mpiexec -n 5 ./pcalc.py : -n 1 ./pwrite.py < cut.in
```

pcalc.py and ccalc.c are versions of the finite difference program discussed above. pcalc.py and ccalc.c create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. The remaining task is pwrite.py.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- world.Get\_group()
- old\_group.Incl()
- world.Create()
- **world.barrier()**
- MPI.Finalize



# Our Examples

## write\_grid.py

write\_grid.py contains three procedures write\_each, write\_one, write\_extra, plot\_extra. These are different output routines for the finite difference code discussed above.

### write\_each

Each MPI task writes its portion of the grid in a separate file. Could be called from stp.py

### write\_one

Each MPI task sends its portion of the grid to a single task and it is written as a single file. Could be called from stp.py

### write\_extra

This could be called from the "extra" MPI task pwrite.py. This routine collects the data from all other tasks and prints it.

### plot\_extra

This could be called from the "extra" MPI task pwrite.py. This routine collects the data from all other tasks and plots it using mkview.py

Write\_one, write\_extra, and plot\_extra all work the same way. They collect data to a single task a line at a time using a combination of Gather and GatherV. For a give line, each processor tells the writing processor how much, if any of the line it holds using the Gather. The the Gatherv is used to actually transfer the data. For write\_one and write\_extra each line is printed as it is gathered. The routine plot\_extra collects the whole grid before plotting it. Write\_each opens a file with the name based on the task id. Each task writes its portion of the grid to its file.

- comm.Get\_rank()
- comm.Get\_size()
- comm.Gather()
- comm.Gatherv()

# MPI Types

- MPI has many different predefined data types
- Can be used in any communication operation



# Predefined types in C

C MPI Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

# Predefined types in Fortran

## Fortran MPI Types

MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	-
MPI_PACKED	-



# Predefined types in mpi4py (91)

AINT	DOUBLE_COMPLEX	INT_INT	SIGNED_LONG
BOOL	DOUBLE_INT	LB	SIGNED_LONG_LONG
BYTE	DOUBLE_PRECISION	LOGICAL	SIGNED_SHORT
CHAR	FLOAT	LOGICAL1	SINT16_T
CHARACTER	FLOAT_INT	LOGICAL2	SINT32_T
COMPLEX	F_BOOL	LOGICAL4	SINT64_T
COMPLEX16	F_COMPLEX	LOGICAL8	SINT8_T
COMPLEX32	F_DOUBLE	LONG	TWOINT
COMPLEX4	F_DOUBLE_COMPLEX	LONG_DOUBLE	UB
COMPLEX8	F_FLOAT	LONG_DOUBLE_INT	UINT16_T
COUNT	F_FLOAT_COMPLEX	LONG_INT	UINT32_T
CXX_BOOL	F_INT	LONG_LONG	UINT64_T
CXX_DOUBLE_COMPLEX	INT	OFFSET	UINT8_T
CXX_FLOAT_COMPLEX	INT16_T	PACKED	UNSIGNED
CXX_LONG_DOUBLE_COMPLEX	INT32_T	REAL	UNSIGNED_CHAR
C_BOOL	INT64_T	REAL16	UNSIGNED_INT
C_COMPLEX	INT8_T	REAL2	UNSIGNED_LONG
C_DOUBLE_COMPLEX	INTEGER	REAL4	UNSIGNED_LONG_LONG
C_FLOAT_COMPLEX	INTEGER1	REAL8	UNSIGNED_SHORT
C_LONG_DOUBLE_COMPLEX	INTEGER16	SHORT	WCHAR
DATATYPE_NULL	INTEGER2	SHORT_INT	_typedict
DOUBLE	INTEGER4	SIGNED_CHAR	_typedict_c
	INTEGER8	SIGNED_INT	_typedict_f

B.T.W. Rmpi has 3 normal types+ data frame

# MPI Broadcast call: MPI\_Bcast

- All nodes call MPI\_Bcast
- One node (root) sends a message all others receive the message
- **Bcast**(self, buf, int root=0)
- **bcast**(self, obj, int root=0)



# Broadcast

## P\_ex04.py

Broadcast of an array of integers and a string

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.bcast()`
- `comm.Bcast()`
- `MPI.Finalize`



# Wildcards

- Allow you to not necessarily specify a tag or source
- Example

```
MPI_Status status;  
int      buffer[5];  
int      error;  
error = MPI_Recv(&buffer[0], 5, MPI_INT,  
                MPI_ANY_SOURCE, MPI_ANY_TAG,  
                MPI_COMM_WORLD, &status);
```

- MPI\_ANY\_SOURCE and MPI\_ANY\_TAG are wild cards
- Status structure is used to get wildcard values



# Wildcards

- Allow you to not necessarily specify a tag or source
- Example

```
status = MPI.Status()  
  
comm.Recv([i, MPI.INT],  
          source=MPI.ANY_SOURCE,  
          tag=MPI.ANY_TAG,  
          status=mystat)
```

- MPI\_ANY\_SOURCE and MPI\_ANY\_TAG are wild cards
- Status object is used to get wildcard values

# Status

- The status parameter returns additional information for some MPI routines
  - Additional Error status information
  - Additional information with wildcard parameters
- C declaration : a predefined struct
  - **`MPI_Status status;`**
- Fortran declaration : an array is used instead
  - **`INTEGER STATUS(MPI_STATUS_SIZE)`**
- mpi4py: an class object
  - **`status=MPI.Status()`**



# Accessing status information mpi4py

```
class Status(builtins.object)
    Status

    Methods defined here:

    Get_count(...)
        Status.Get_count(self, Datatype datatype=BYTE)
        Get the number of *top level* elements

    Get_elements(...)
        Status.Get_elements(self, Datatype datatype)
        Get the number of basic elements in a datatype

    Get_error(...)
        Status.Get_error(self)
        Get message error

    Get_source(...)
        Status.Get_source(self)
        Get message source

    Get_tag(...)
        Status.Get_tag(self)
        Get message tag

    Is_cancelled(...)
        Status.Is_cancelled(self)
        Test to see if a request was cancelled
```

# MPI\_Probe

- MPI\_Probe allows incoming messages to be checked without actually receiving .
- The user can then decide how to receive the data.
- Useful when different action needs to be taken depending on the "who, what, and how much" information of the message.



# MPI4py "probe" part I

```
#!/usr/bin/env python3
# Program shows how to use probe and get_count to find the size
# of an incoming message
import numpy
from numpy import *
from mpi4py import MPI
import sys

# Initialize MPI and print out hello
comm=MPI.COMM_WORLD
myid=comm.Get_rank()
numprocs=comm.Get_size()
print("hello from ",myid," of ",numprocs)

# Tag identifies a message
mytag=123

# Process 0 is going to send the data
if myid == 0:
    i=array([1234,5678],"i")
    icount=2
    comm.Send([i, MPI.INT], dest=1, tag=mytag)
```

# MPI4py "probe" part 2

```
if myid == 1:
# We create a status to use in the probe command
    mystat=MPI.Status()
# Call probe to find out about the incoming message
    comm.probe(source=0, tag=mytag, status=mystat)
# We find out how big the incoming message is and allocate space
    icount=mystat.Get_count(MPI.INT)
    print("getting ", icount)
    i=empty((icount),"i")
# We are receiving an array of integers
    comm.Recv([i, MPI.INT], source=0, tag=mytag)
    print("i=",i)

MPI.Finalize()
```



# MPI\_Probe example

## [P\\_ex02.py](#)

Blocking Send and Receive with probe to find size

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.Send()`
- `comm.probe()`
- `mystat.Get_count()`
- `comm.Recv()`
- `MPI.Finalize`

# MPI\_BARRIER

- Blocks the caller until all members in the communicator have called it.
- Used as a synchronization tool.
- C
  - **MPI\_Barrier(comm )**
- Fortran
  - **Call MPI\_BARRIER(COMM, IERROR)**
- Parameter
  - Comm communicator (MPI\_COMM\_WORLD)



# MPI\_BARRIER

- Blocks the caller until all members in the communicator have called it.
- Used as a synchronization tool.
- mpi4py
  - `Barrier(self)`

# Asynchronous Communication

- Asynchronous send: send call returns immediately, send actually occurs later
- Asynchronous receive: receive call returns immediately. When received data is needed, call a wait subroutine
- Asynchronous communication used in attempt to overlap communication with computation (usually doesn't work)
- Can help prevent deadlock (not advised)



# Asynchronous Send with MPI\_Isend

- C
  - `MPI_Request request`
  - `int MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)`
- Fortran
  - Integer REQUEST
  - `MPI_ISEND(BUFFER, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)`
- Request is a new output Parameter
- Don't change data until communication is complete

# Asynchronous Send with MPI\_Isend

- mpi4py
  - **isend**(self, obj, int dest, int tag=0)
  - **Isend**(self, buf, int dest, int tag=0)
- They return a communication Request which is an object with various methods
- Don't change data until communication is complete



# Asynchronous Receive with MPI\_Irecv

- C

- `MPI_Request request;`
- `int MPI_Irecv(&buf, count, datatype, source, tag, comm, &request)`

- Fortran

- Integer request
- `MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)`

- Parameter Changes

- Request: communication request
- Status parameter is missing
- Don't use data until communication is complete

# Asynchronous Receive with MPI\_Irecv

- mpi4y
  - **Irecv**(self, buf, int source=ANY\_SOURCE, int tag=ANY\_TAG)
  - **irecv**(self, buf=None, int source=ANY\_SOURCE, int tag=ANY\_TAG)
- Parameter Changes
  - They return a communication Request which is an object with various methods
  - Status parameter is missing
- Don't use data until communication is complete



# MPI\_Wait used to complete communication

- Request from Isend or Irecv is input
- The completion of a send operation indicates that the sender is now free to update the data in the send buffer
- The completion of a receive operation indicates that the receive buffer contains the received message
- MPI\_Wait blocks until message specified by "request" completes

# MPI\_Wait used to complete communication

- C
  - `MPI_Request request;`
  - `MPI_Status status;`
  - `MPI_Wait(&request, &status)`
- Fortran
  - Integer request
  - Integer status(MPI\_STATUS\_SIZE)
  - `MPI_WAIT(REQUEST, STATUS, IERROR)`
- MPI\_Wait blocks until message specified by "request" completes



# MPI\_Wait used to complete communication

- Very different in mpi4py
- Wait is a method of the class object “Request”
- Where req is the Request returned by the Isend/Irecv the call is:
  - Irecv
    - req.wait()
    - req.Wait()
  - irecv()
    - buffer=req.wait()

# Asynchronous Send and Receive with MPI\_Wait used to complete communication

## [P\\_ex03.py](#)

Nonblocking Send and Receive with wait

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.isend()
- comm.irecv()
- req.wait()
- MPI.Finalize

## [P\\_ex03I.py](#)

Nonblocking Send and Receive with wait

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Isend()
- comm.Irecv()
- req.wait()
- MPI.Finalize



# MPI\_Test

- Similar to MPI\_Wait, but does not block
- Value of flags signifies whether a message has been delivered
- C
  - **int flag**
  - **int MPI\_Test(&request, &flag, &status)**
- Fortran
  - **LOGICAL FLAG**
  - **MPI\_TEST(REQUEST, FLAG, STATUS, IER)**

# Non blocking send example

```
call MPI_Isend (buffer,count,datatype,dest,  
               tag,comm, request, ierr)  
10 continue  
   Do other work ...  
  
call MPI_Test (request, flag, status, ierr)  
if (.not. flag) goto 10
```



# MPI\_Test

- Very different in mpi4py
- req.test returns a tuple (flag, buffer), works with both irecv and lrecv
- req.Test just returns a flag, works only with lrecv

## lrecv

```
while (not req.Test()) :  
    time.sleep(0.5)  
    print("dest", req.Test(), req.test()[0])  
  
print("processor ", destination, " got ", buffer)
```

```
...  
...  
dest False (False, None)  
dest False (False, None)  
source True (True, None)  
dest True (True, None)  
processor 1 got [5678]
```

## irecv

```
buffer=(False, None)  
while buffer == (False, None) :  
    buffer=req.test()  
    print("dest", req.Test(), buffer)  
    time.sleep(0.5)  
    buffer=buffer[1]  
  
print("processor ", destination, " got ", buffer)
```

```
dest False (False, None)  
dest False (False, None)  
source True (True, None)  
processor 0 sent 5678  
dest True (True, 5678)  
processor 1 got 5678
```

# Scatter Operation using MPI\_Scatter

- Similar to Broadcast but sends a section of an array to each processors

Data in an array on root node:

$A(0)$   $A(1)$   $A(2)$  . . .  $A(N-1)$

Goes to processors:

$P_0$   $P_1$   $P_2$  . . .  $P_{n-1}$



# MPI\_Scatter

- C

- `int MPI_Scatter(&sendbuf, sendcnts, sendtype, &recvbuf, recvcnts, recvtype, root, comm );`

- Fortran

- `MPI_Scatter(sendbuf, sendcnts, sendtype, recvbuf, recvcnts, recvtype, root, comm, ierror)`

- Parameters

- Sendbuf is an array of size (number processors\*sendcnts)
  - Sendcnts number of elements sent to each processor
  - Recvcnts number of elements obtained from the root processor
  - Recvbuf elements obtained from the root processor, may be an array

# MPI\_Scatter

- mpi4py
  - **scatter**(self, sendobj, int root=0)
  - **Scatter**(self, sendbuf, recvbuf, int root=0)
- Parameters
  - Sendbuf is an array of size (number processors\*sendcnts)
  - Sendcnts number of elements sent to each processor (not needed)
  - Recvcnts number of elements obtained from the root processor (not needed)
  - Recvbuf elements obtained from the root processor, may be an array



# Scatter Operation using MPI\_Scatter

- Scatter with Sendcnts = 2

Data in an array on root node:

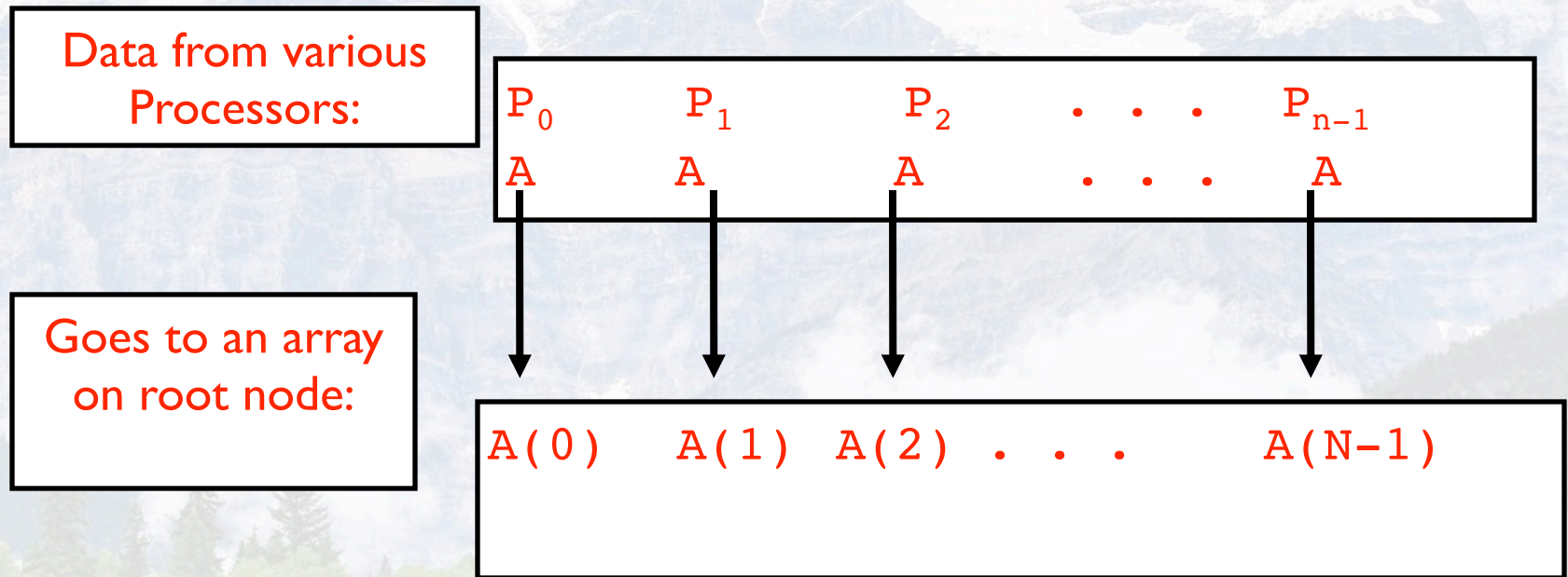
$A(0)$	$A(2)$	$A(4)$	$\dots$	$A(2N-2)$
$A(1)$	$A(3)$	$A(5)$	$\dots$	$A(2N-1)$

Goes to processors:

$P_0$	$P_1$	$P_2$	$\dots$	$P_{n-1}$
$B(0)$	$B(0)$	$B(0)$		$B(0)$
$B(1)$	$B(1)$	$B(1)$		$B(1)$

# Gather Operation using MPI\_Gather

- Used to collect data from all processors to the root, inverse of scatter
- Data is collected into an array on root processor





# MPI\_Gather

- C

- `int MPI_Gather(&sendbuf, sendcnts, sendtype, &recvbuf, recvcnts, recvtype, root, comm );`

- Fortran

- `MPI_Gather(sendbuf, sendcnts, sendtype, recvbuf, recvcnts, recvtype, root, comm, ierror)`

- Parameters

- Sendcnts # of elements sent from each processor
  - Sendbuf is an array of size sendcnts
  - Recvcnts # of elements obtained from each processor
  - Recvbuf of size Recvcnts\*number of processors

# MPI\_Gather

- mpi4py
  - **gather**(self, sendobj, int root=0)
  - **Gather**(self, sendbuf, recvbuf, int root=0)
- Parameters
  - Sendbuf is an array - can be size 1
  - Recvbuf elements obtained at the root processor - an array



# Scatter and Gather

## [P\\_ex05.py](#)

This program shows how to use MPI\_Scatter and MPI\_Gather. Each processor gets different data from the root processor by way of mpi\_scatter. The data is summed and then sent back to the root processor using MPI\_Gather. The root processor then prints the global sum.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.bcast()
- comm.Scatter()
- comm.gather()
- comm.Gather()
- MPI.Finalize

# Reduction Operations

- Used to combine partial results from all processors
- Result returned to root processor
- Several types of operations available
- Works on single elements and arrays



# MPI\_Reduce

- mpi4py
  - **reduce**(self, sendobj, op=**SUM**, int root=**0**)
  - **Reduce**(self, sendbuf, recvbuf, Op op=**SUM**, int root=**0**)
- Parameters
  - Like MPI\_Bcast, a root is specified.
  - Operation is a type of mathematical operation

# Operations for MPI\_Reduce

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location



# Global Sum with MPI\_Reduce

mpi4py

```
#each processor does a local sum
total=0
for i in range(0, count):
    total=total+myray[i]
print("myid=",myid,"total=",total)

#reduce back to the root and print
#reduce(self, sendobj, op=SUM, int root=0)
#Reduce(self, sendbuf, recvbuf, Op op=SUM, int root=0)
if lower :
    back_ray=comm.reduce(total)
else:
    back_ray=empty(2,"i") # Why does this need to be 2?
                        # Maybe to support the max_loc operation?
                        # However, MAXLOC does not work?
    comm.Reduce(total,back_ray,op=MPI.SUM,root=mpi_root)
if myid == mpi_root:
    print("results from all processors=",back_ray)
```

# Global Sum with MPI\_Reduce

## [P\\_ex06.py](#)

This program shows how to use MPI\_Scatter and MPI\_Reduce. Each processor gets different data from the root processor by way of mpi\_scatter. The data is summed and then sent back to the root processor using MPI\_Reduce. The root processor then prints the global sum.

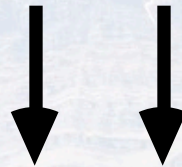
- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.bcast()
- comm.Scatter()
- `comm.reduce()`
- `comm.Reduce()`
- MPI.Finalize



# Global Sum with MPI\_Reduce

2d array spread across processors

	X(0)	X(1)	X(2)
NODE 0	A0	B0	C0
NODE 1	A1	B1	C1
NODE 2	A2	B2	C2



	X(0)	X(1)	X(2)
NODE 0	A0+A1+A2	B0+B1+B2	C0+C1+C2
NODE 1			
NODE 2			

# All Gather and All Reduce

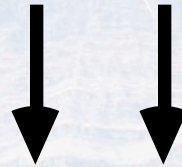
- Gather and Reduce come in an "ALL" variation
- Results are returned to all processors
- The root parameter is missing from the call
- Similar to a gather or reduce followed by a broadcast



# Global Sum with MPI\_AllReduce

2d array spread across processors

	X(0)	X(1)	X(2)
NODE 0	A0	B0	C0
NODE 1	A1	B1	C1
NODE 2	A2	B2	C2



	Y(0)	Y(1)	Y(2)
NODE 0	A0+A1+A2	B0+B1+B2	C0+C1+C2
NODE 1	A0+A1+A2	B0+B1+B2	C0+C1+C2
NODE 2	A0+A1+A2	B0+B1+B2	C0+C1+C2

# All to All communication with MPI\_Alltoall

- mpi4py
  - **alltoall**(self, sendobj)
  - **Alltoall**(self, sendbuf, recvbuf)
  - Parameters
- Each processor sends and receives the same amount of data to/from all others



# All to All with MPI\_Alltoall

- Parameters
  - Sendcnts # of elements sent to each processor
  - Sendbuf is an array of size sendcnts
  - Recvcnts # of elements obtained from each processor
  - Recvbuf of size Recvcnts\*number of processors
- Note that both send buffer and receive buffer must be an array of (size of the number of processors)\*N

# All to All with MPI\_Alltoall

## [P\\_ex07.py](#)

This program shows how to use MPI\_Alltoall. Each processor send/rec a different random number to/from other processors.

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- `comm.Alltoall()`
- MPI.Finalize



# Things Left

- “V” operations (Skipping for today)
- Communicators
- ~~Derived types~~ (Not as important in Python)
- Parallel IO
  - See simple example
  - <http://mpi4py.scipy.org/docs/usrman/tutorial.html#mpi-io>
  - Also there are hdf5 examples in the same repository in the pandas directory
- Real life examples
  - Finite Difference Code
  - Bag of tasks

# Communicators

- In “normal” MPI a communicator is a parameter in all MPI message passing routines
- In mpi4py a communicator is a class object that has message passing routines as methods
- A communicator is a collection of processors that can engage in communication
- MPI\_COMM\_WORLD is the default communicator that consists of all processors
- MPI allows you to create subsets of communicators



# Why Communicators?

- Isolate communication to a small number of processors
- Useful for creating libraries
- Different processors can work on different parts of the problem
- Useful for communicating with "nearest neighbors"

# MPI\_Comm\_create

- MPI\_Comm\_create creates a new communicator newcomm with group members defined by a group data structure.
- C
  - `int MPI_Comm_create(comm, group, &newcomm)`
- Fortran
  - Call `MPI_COMM_CREATE(comm, GROUP, NEWCOMM, IERROR)`
- mpi4py
  - `newcom=comm.Create(group)`
- How do you define a group?



# MPI\_Comm\_group

- Given a communicator, MPI\_Comm\_group returns in group associated with the input communicator
- C
  - `int MPI_Comm_group(comm, &group)`
- Fortran
  - `Call MPI_COMM_GROUP(COMM, GROUP, IERROR)`

# MPI\_Comm\_group

- Given a communicator, MPI\_Comm\_group returns in group associated with the input communicator
- mpi4py
  - `old_group=comm.Get_group()`
- As we have seen comm is an object. Get\_group is a method
- Groups “old\_group” is also an object with a collection of methods



# MPI\_Group\_incl

- MPI\_Group\_incl creates a group **new\_group** that consists of the n processes in **old\_group** with ranks rank[0],..., rank[n-1]
- C
  - `int MPI_Group_incl(group, n, &ranks, &new_group)`
- Fortran
  - `Call MPI_GROUP_INCL(GROUP, N, RANKS, NEW_GROUP, IERROR)`

# MPI\_Group\_incl

- Fortran
  - Call **MPI\_GROUP\_INCL(old\_GROUP, N, RANKS, NEW\_GROUP, IERROR)**
- Parameters
  - old\_group: your old group
  - N: number of elements in array ranks (and size of new\_group) (integer)
  - Ranks: ranks of processes in group to appear in new\_group (array of integers)
  - New\_group: new group derived from above, in the order defined by ranks



# MPI\_Group\_incl

- MPI\_Group\_incl creates a group **new\_group** that consists of the n processes in **old\_group** with ranks rank[0],..., rank[n-1]
- mpi4py
  - **new\_group**=**old\_group**.Incl(ranks)

# Create communicator...

```
# get our old group from MPI_COMM_WORLD
old_group=comm.Get_group()
```

```
# create a new group from the old group
# containing a subset of the processors
```

```
num_used=. . .
will_use=zeros(num_used,"i")
for ijk in range(0, num_used):
    will_use[ijk]=. . .
```

```
new_group=old_group.Incl(will_use)
```

```
# create the new communicator
sub_comm_world=comm.Create(new_group)
```



# Create communicator Example

[P\\_ex10.py](#) , [ex10.in](#)

Pass a "token" from one task to the next with a single task reading token from a file.

An extensive program. We first create a new communicator that contains every task except the zeroth one. This is done by first defining a group, `new_group`. We define `new_group` based on the group associated with `mpi_comm_world`, `old_group` and an array, `will_use`. `Will_use` contains a list of tasks to be included in the new group. It does not contain the zeroth one. The new communicator is `sub_comm_world`.

There are other ways to create communicator but this is one of the more general methods.

Next, we have break of the task not in the communicator to call the routine `get_input`. This routine will do input from a file `ex10.in`. The file contains a list of integers. The task will send the integer to the first task in the new communicator.

The remaining tasks which are port of `sub_comm_world` call the routine `pass_token`. `Pass_token` "just" receives a value from the previous processor and passes it on to the next.

There is a minor subtlety in `pass_token`. We are using both our new communicator and `MPI_COMM_WORLD`. The tasks that are port of the new communicator use it to pass data. We note that the task that is injecting values into the stream is not part of the new communicator so it must use `MPI_COMM_WORLD`. Thus we do a probe on `WORLD`, which is actually `MPI_COMM_WORLD` looking for a message. When we get it we send it on using the new communicator.

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `WORLD.Iprobe()`
- `comm.Get_group()`
- `old_group.Incl()`
- `comm.Create()`
- `new_group.Get_rank()`
- `MPI.Finalize`

# MPI\_Group\_excl

- MPI\_Group\_excl creates a group of processes **new\_group** that is obtained by deleting from **old\_group** those processes with ranks ranks[0], ... , ranks[n-1]



# MPI\_Comm\_split

- Provides a short cut method to create a collection of communicators
- All processors with the "same color" will be in the same communicator
- Index gives rank in new communicator
- Fortran
  - call MPI\_COMM\_SPLIT(**OLD\_COMM**, color, index, **NEW\_COMM**, mpi\_err)
- C
  - MPI\_Comm\_split(**OLD\_COMM**, color, index, &**NEW\_COMM**)

# MPI\_Comm\_split

- Provides a short cut method to create a collection of communicators
- All processors with the "same color" will be in the same communicator
- Index gives rank in new communicator
- mpi4py
  - `new_comm=old_comm.Split(color,index)`



# MPI\_Comm\_split

- Split odd and even processors into 2 communicators

```
Program comm_split
include "mpif.h"
Integer color,zero_one
call MPI_INIT( mpi_err )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numnodes, mpi_err )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, mpi_err )
color=mod(myid,2) !color is either 1 or 0
call MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,NEW_COMM,mpi_err)
call MPI_COMM_RANK( NEW_COMM, new_id, mpi_err )
call MPI_COMM_SIZE( NEW_COMM, new_nodes, mpi_err )
Zero_one = -1
If(new_id==0)Zero_one = color
Call MPI_Bcast(Zero_one,1,MPI_INTEGER,0, NEW_COMM,mpi_err)
If(zero_one==0)write(*,*)"part of even processor communicator"
If(zero_one==1)write(*,*)"part of odd processor communicator"
Write(*,*)"old_id=", myid, "new_id=", new_id
Call MPI_FINALIZE(mpi_err)
End program
```

# MPI\_Comm\_split

- Split odd and even processors into 2 communicators

## [P\\_ex12.py](#)

This program shows how to use `mpi_comm_split`

`Split` will create a set of communicators. All of the tasks with the same value of `color` will be in the same communicator. In this case we get two sets one for odd tasks and one for even tasks. Note they have the same name on all tasks, `new_comm`, but there are actually two different values (sets of tasks) in each one.

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.Split`
- `comm.bcast`
- `MPI.Finalize`



```
#!/usr/bin/env python3
#
# This program shows how to use mpi_comm_split
#
import numpy
from numpy import *
from mpi4py import MPI
import sys
```

```
. . .
. . .
```

```
# Get an integer representation of our color Note: this will be the same at select_val
color=col_to_int[color_str]

print("myid= %d      color integer = %d      color name = %s" %(myid,color,color_str))
```

```
. . .
. . .
```

```
new_comm=comm.Split(color,myid)
# Get the new id and number of tasks in the communicator
new_id=new_comm.Get_rank()
new_nodes=new_comm.Get_size()
```

## Mpi4py color/com.Split example

Sorry - I went a bit hog wild on this example  
here we just show the important parts

```
. . .  
. . .
```

```
# Here we do a bcast from the new roots to show we can use the communicator
```

```
bcast_val=None
```

```
if new_id == 0 : bcast_val=myid
```

```
bcast_val=new_comm.bcast(bcast_val)
```

```
# Iterate through the colors and each processor will print its information
```

```
for k in col_to_int.keys():
```

```
    if( k == color_str):
```

```
        print("new id is %d in the %s communicator or %d.%s \
```

```
original id is %d \
```

```
id bcast from root %d" %(new_id,k,new_id,k,myid,bcast_val))
```

```
MPI.Finalize()
```



# MPI\_Comm\_split example output

- Note, I have sorted the output

```
osage:mpi4py tkaiser$ mpiexec -n 6 ./P_ex12.py | sort
color to integer= {'blue ': 0, 'green ': 1, 'red ': 2, 'yellow': 3} and
integer to color= {0: 'blue ', 1: 'green ', 2: 'red ', 3: 'yellow'}
hello from 0 of 6
hello from 1 of 6
hello from 2 of 6
hello from 3 of 6
hello from 4 of 6
hello from 5 of 6
myid= 0    color integer = 0    color name = blue
myid= 1    color integer = 1    color name = green
myid= 2    color integer = 0    color name = blue
myid= 3    color integer = 1    color name = green
myid= 4    color integer = 0    color name = blue
myid= 5    color integer = 1    color name = green
new id is 0 in the blue communicator or 0.blue original id is 0 id bcast from root 0
new id is 0 in the green communicator or 0.green original id is 1 id bcast from root 1
new id is 1 in the blue communicator or 1.blue original id is 2 id bcast from root 0
new id is 1 in the green communicator or 1.green original id is 3 id bcast from root 1
new id is 2 in the blue communicator or 2.blue original id is 4 id bcast from root 0
new id is 2 in the green communicator or 2.green original id is 5 id bcast from root 1
osage:mpi4py tkaiser$
```