Introduction to MPI and Domain decomposition

Ji-Hoon Kang, jhkang@kisti.re.kr

Korea Institute of Science and technology (KISTI)

Contents

Introduction to MPI

What is MPI??

- Message Passing Interface
- MPI is a library, not a language
- It is a library for inter-process communication and data exchange
- Use for Distributed Memory

- History
  - MPI-1 Standard (MPI Forum) : 1994
    - http://www.mcs.anl.gov/mpi/index.html
    - MPI-1.1(1995), MPI-1.2(1997)
  - MPI-2 Announce : 1997
    - http://www.mpi-forum.org/docs/docs.html
    - MPI-2.1(2008), MPI-2.2(2009)
  - MPI-3 Announce : 2012
    - http://www.mpi-forum.org/docs/docs.html

MPI (Message Passing Interface)

- **MPI is a library, not a language**
  - Subroutines handling inter-process communication and synchronization for programs running on parallel platforms
- **Consists of**
  - Library (subroutine or function)
  - Executable binary and running argument (Environment variables

```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char* argv[])
{
   /* Initialize the library */
   MPI_Init(&argc, &argv);

   printf("Hello world\n");

   /* Wrap it up. */
   MPI_Finalize();
}
```

Initialize MPI Library

Do some work!

Return the resources

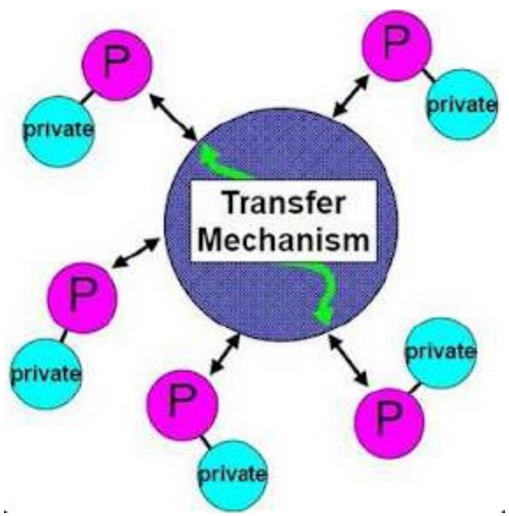
```
$ mpicc -o hello.x hello.c
$ mpirun -np 4 -hostfile hosts ./hello.x
```

MPI (Message Passing Interface)

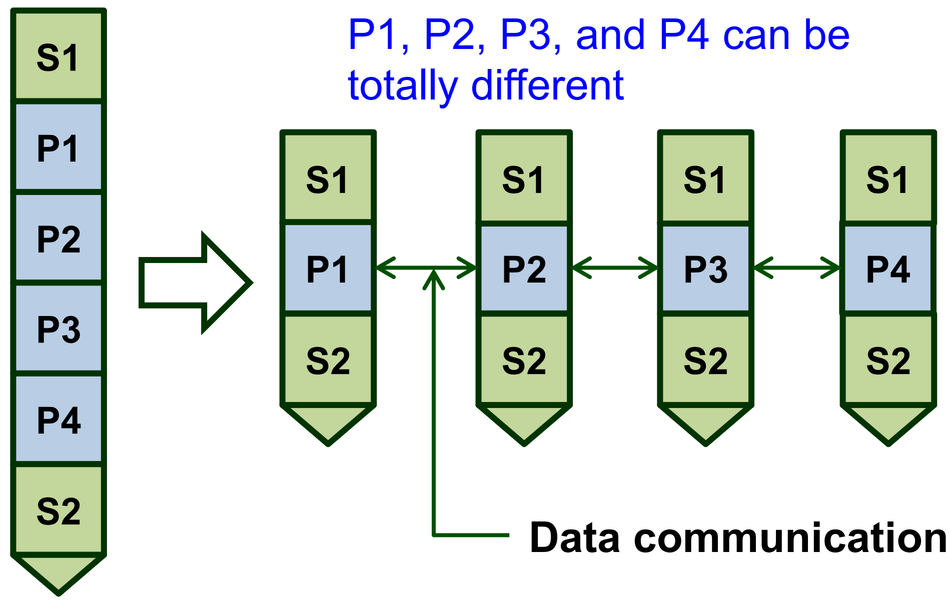
- **The use of MPI is never complicated**.
  - Six key functions are sufficient for any program, theoretically.
    → MPI_Init /MPI_Finalize
    → MPI_comm_size / MPI_comm_rank
    → MPI_send / MPI_recv
  - Hundreds+ additional functions that offer abstraction, performance portability and convenience for experts
- **The complicated one is parallelization method itself**
  - Domain decomposition scheme
  - Common program for different domains
  - Design of data transfer

MPI programming model

- **Message passing parallelism**



MPI: multi-processes



  - Process based
    - Independent processes runs on many multi-core processors and work together using their own memory and resources **through message-passing communication**.
  - Distributed memory model
    - Each process does its own work using its own memory and resources
    - In order to work together, data in memory are passed through communication
- **MPI only provides the tools of communication**

Parallel Models Compared

	MPI	Threads	OpenMP*
Portable	✓		✓
Scalable	✓	✓	✓
Performance Oriented	✓		✓
Supports Data Parallel	✓	✓	✓
Incremental Parallelism			✓
High Level			✓
Serial Code Intact			✓
Verifiable Correctness			✓
Distributed Memory	✓		

Common MPI Implementations

- **MPICH(Argonne National Laboratory)**
  - Most common MPI implementation
  - Derivatives
    - MPICH GM – Myrinet support (available from Myricom)
    - MVAPICH – infiniband support (available from Ohio State University)
    - Intel MPI – Version tuned to Intel Architecture systems
- **Open MPI(Indiana University/LANL)**
  - Contains many MPI 2.0 features
  - FT-MPI: University of Tennessee (Data types, process fault tolerance, high performance)
  - LA-MPI: Los Alamos (Pt-2-Pt, data fault-tolerance, high performance, thread safety)
  - LAM/MPI: Indiana University (Component architecture, dynamic processes)
  - PACX-MPI: HLRS - Stuttgart (dynamic processes, distributed environments, collectives)
- **Scali MPI Connect**
  - Provides native support for most high-end interconnects
- **MPI/Pro (MPI Software Technology)**

mpi4py

MPI for Python

- Python bindings for the Message Passing Interface (MPI) standard
- Allowing Python applications to exploit multiple processors
- Providing an object oriented interface resembling the MPI-2 C++ bindings
- Supporting P2P and collective communications.
- Handling python objects serialized with pickle module, as well as exposed to Python buffer interface of array data (e.g. NumPy arrays and built-in bytes/array/memory view objects).

MPI-2 bindings for C++ to Python

- Anyone using the standard C/C++ MPI bindings is able to use mpi4py module without need of learning a new interface.

MPI Basic Steps

- **Writing a program**
  - using “mpi.h” (or mpif.h for FORTRAN) and some essential function calls
  - Or import mpi4py

- **Compiling your program**
  - using a compilation script

- **Specify the machine file**
  - Making MPI hosts file
    - Use familiar editor : vi, emacs, gedit, etc…

```
$ cat  hosts
s0001
s0002
s0003
s0004
```

MPI_Init and MPI_Finalize

int MPI_Init(&argc, &argv)

- Subroutine of starting MPI
- Prepares the system for MPI execution

int MPI_Finalize()

- Return the error code when it is failed
- Subroutines of finalizing MPI

```
#include <stdio.h>
#include ”mpi.h”

int main (int argc, char* argv[])
{
    /* Initialize the library */
    MPI_Init(&argc, &argv);

    printf(”Hello world\n”);

    /* Wrap it up. */
    MPI_Finalize();
}
```

Library

Initialize MPI Library

Do some work!

Return the resources

LAB1 : "Hello, World" in MPI

- **Not required** `MPI_Init` **and** `MPI_Finalize`

```
from mpi4py import MPI

print("Hello World!")
```

```
$ mpirun  -np  4  python ./lab1_hello.py
Hello World!
Hello World!
Hello World!
$
```

Parallel Program Execution

Launch scenario for MPIRUN

- Find machine file(to know where to launch)
- Use SSH or RSH to execute a copy of the program on each node in machine file
- Once launched each copy establishes communications with local MPI lib (MPI_Init)
- Each copy ends MPI interaction with MPI_Finalize

Starting an MPI Program

1

Execute on node1:

```
$ mpirun -np 4 python ./lab1_hello.py
```

2

Check the MPI hostfile:

s0001
s0002
s0003
s0004

3

s0001

```
python ./lab1_hello.py
```

s0002

```
python ./lab1_hello.py
```

s0003

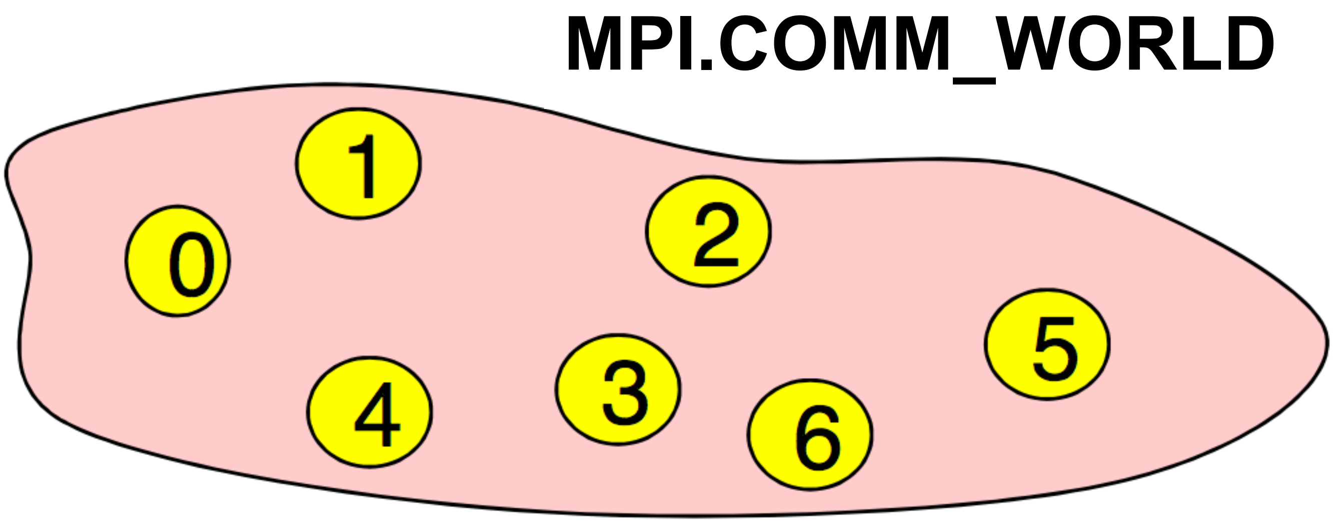
```
python ./lab1_hello.py
```

s0004

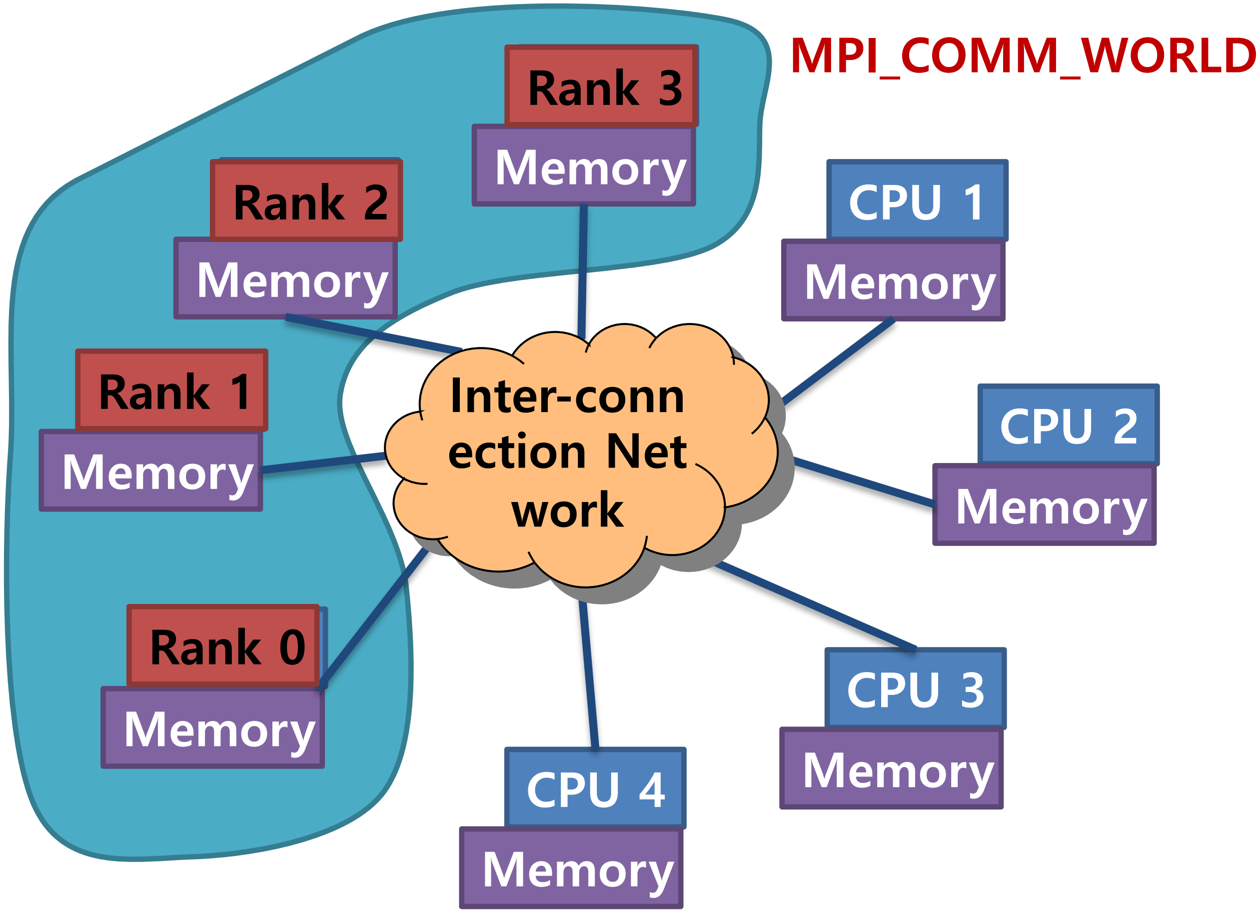
```
python ./lab1_hello.py
```

MPI communicator

- **A handle representing a group of processes that can communicate with each other(more about communicators later)**
- **All MPI communication calls have a communicator argument**
- **Most often you will use MPI.COMM_WORLD**
  - It is all of your processors.



MPI communicator – Real situation



Communicator size and rank

- **size = MPI.COMM_WORLD.Get_size() : How many we are?**
  - Subroutine which returns the number of MPI processes in the program (MPI.COMM_WORLD.size)
  - We run MPI program like "***mpirun –np m MY_PROGRAM***"
  - ***MY_PROGRAM*** runs on the ***m*** processes parallely.
  - Each MPI process can detect the value of ***m*** by calling this subroutine and returns it
- **rank = MPI.COMM_WORLD.Get_rank() : Who am I?**
  - Subroutines which returns the rank of the current MPI process calling this function (MPI.COMM_WORLD.rank)
  - Each MPI process in the COMM communicator can determine its own rank (***0 ~ m-1***), by calling this subroutine.
  - We can assign a task to a specific MPI process by using this rank number, **rank**.

```
if(rank.EQ.0) then
…
elseif(rank.EQ.1) then
…
endif
```

- **By calling Get_size() and Get_rank(), each MPI process can know the size of communicator(*m*) and its own rank(*0 ~ m-1*)**

Example : Starting MPI(I)

- **Calculate the $y=x^2+x+1$ at the point of x=0,1,2,3**
  - Use 4 MPI processes which calculate y at the point of x=0,1,2,3, respectively.
  - Calculate the value of y in each MPI process

- **Running 4 MPI processes using MPI executable and running argument**

mpirun –np m MY_PROGRAM

MPI executable

running argument –
create 4 MPI processes

Example : Starting MPI (II)

- **Serial program**

```
for x in range(4):
    y = x*x + x + 1
    print("The value of x*x+x+1 = {0}, x = {1}".format(y, x))
```

```
jihoon@V0841KJH:~/$ python3 x2.py
The value of x*x+x+1 = 1, x = 0
The value of x*x+x+1 = 3, x = 1
The value of x*x+x+1 = 7, x = 2
The value of x*x+x+1 = 13, x = 3
```

Example : Starting MPI (III)

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
myrank = comm.Get_rank() # myrank = comm.rank
nprocs = comm.Get_size() # nprocs = comm.size

if(myrank==0) :
    x=0.0
elif(myrank==1) :
    x=1.0
elif(myrank==2) :
    x=2.0
elif(myrank==3) :
    x=3.0
y=x*x+x+1
print("process{0} of {1} : the value of x*x+x+1 = {2}, x = {3}
".format(myrank,nprocs,y,x))
```

```
jihoon@V0841KJH:~/$ mpirun -np 4 python3 x2mpi.py
process 1 of 4 : the value of x*x+x+1 = 3.000000, x = 1.000000
process 2 of 4 : the value of x*x+x+1 = 7.000000, x = 2.000000
process 3 of 4 : the value of x*x+x+1 = 13.000000, x = 3.000000
process 0 of 4 : the value of x*x+x+1 = 1.000000, x = 0.000000
jihoon@V0841KJH:~/$
```

Example : Starting MPI (IV)

- **Better program**

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
myrank = comm.Get_rank() # myrank = comm.rank
nprocs = comm.Get_size() # nprocs = comm.size

x =myrank
y=x*x+x+1

print("process{0} of {1} : the value of x*x+x+1 = {2}, x = {3}".
format(myrank,nprocs,y,x))
```

Six key functions

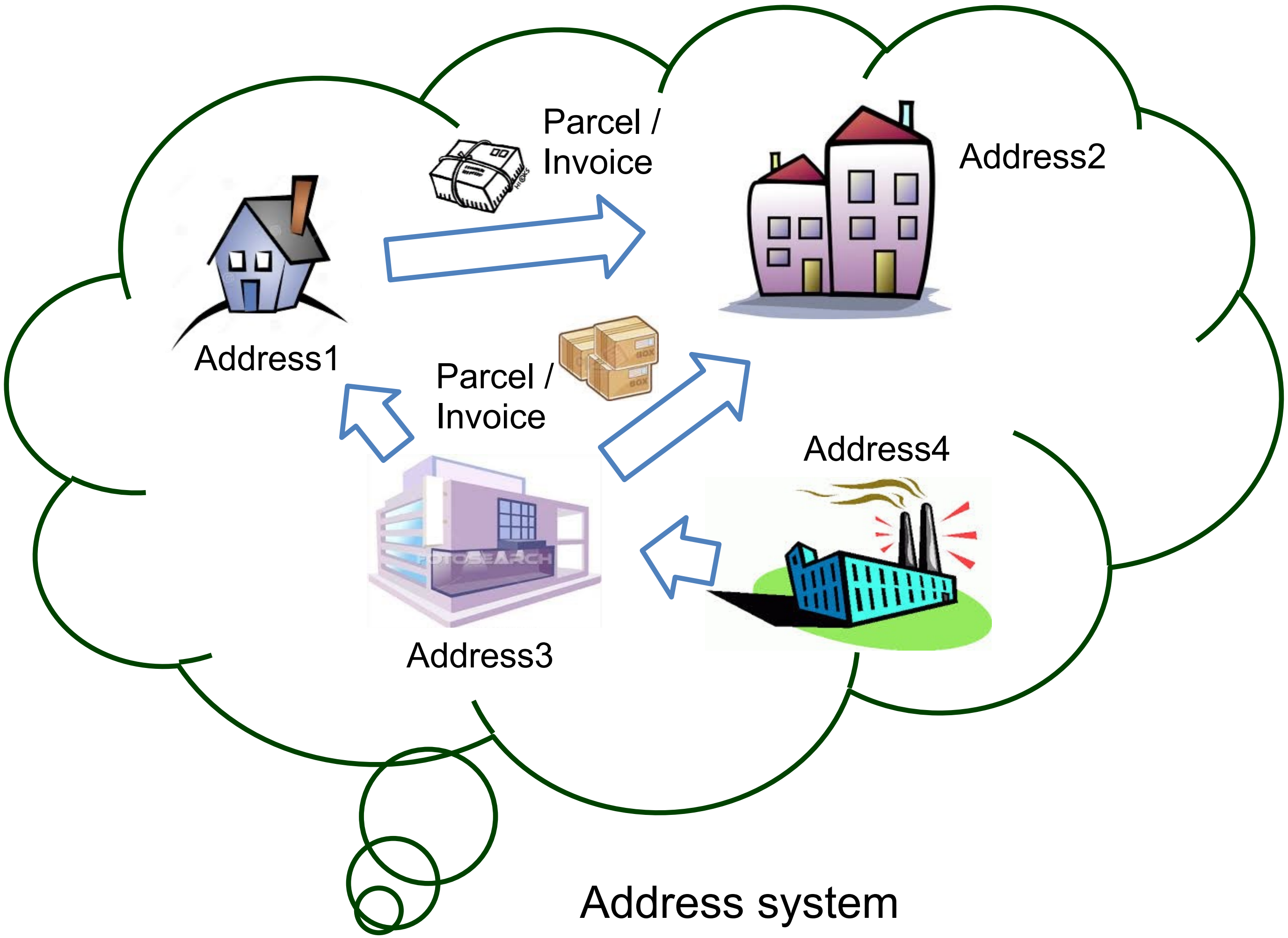
Function	Functions
~~MPI_INIT~~	~~Register communicator (address system)~~
~~MPI_FINALIZE~~	~~Destroy communicator~~
MPI.COMM_WORLD.Get_size()	Return communicator size (size of address site)
MPI.COMM_WORLD.Get_rank()	Return process number (address)
MPI.COMM_WORLD.Send()	Send data/message to target process Send: numpy array, send: python object
MPI.COMM_WORLD.Recv()	Recv data/message from source process Recv: numpy array, recv: python object

LAB2 : rank and size

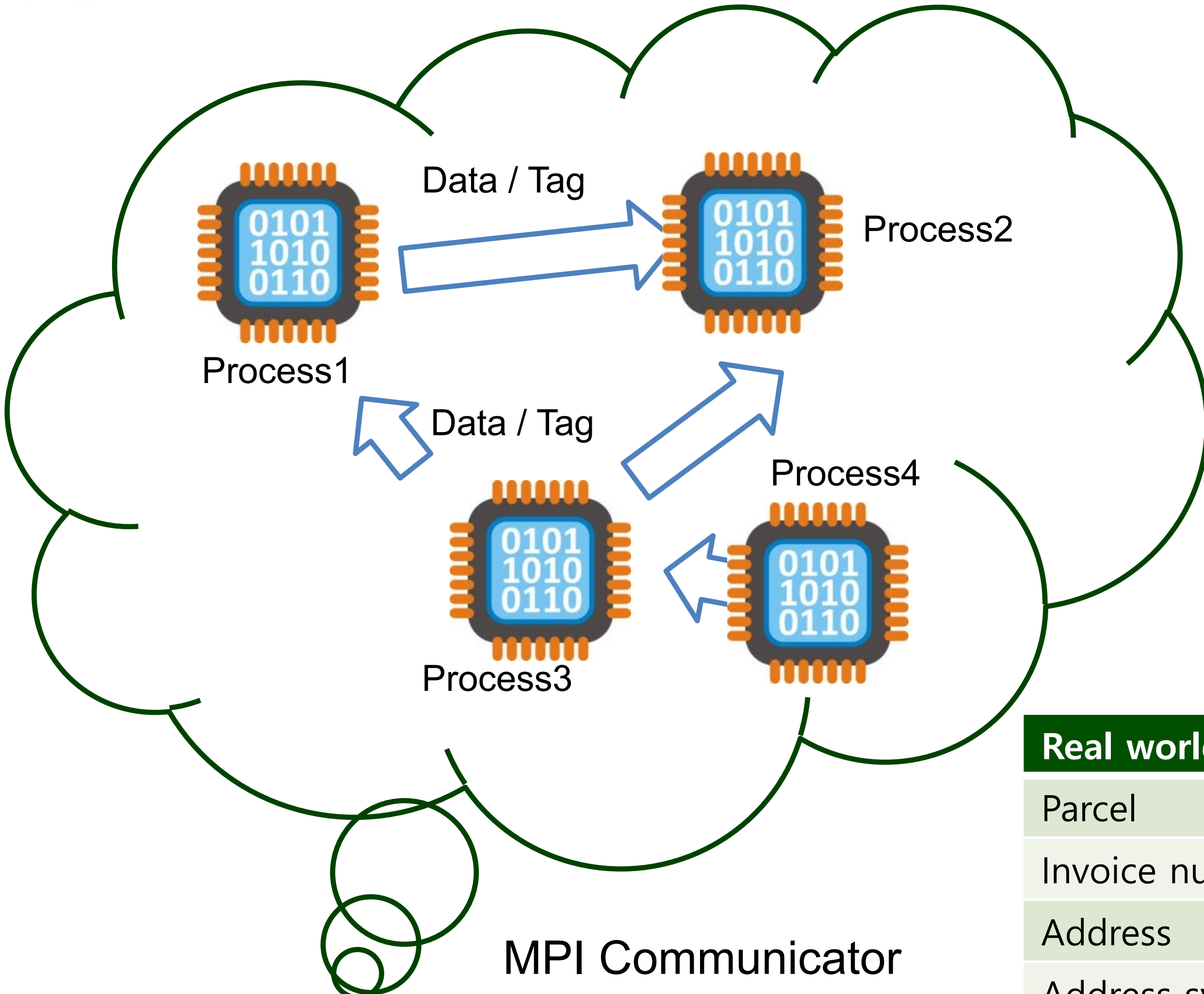
```
from mpi4py import MPI

comm = MPI.COMM_WORLD
myrank = comm.Get_rank() # myrank = comm.rank
nprocs = comm.Get_size() # nprocs = comm.size

ver, subver = MPI.Get_version()
if myrank == 0 :
    print("MPI Version {0}.{1}".format(ver, subver))

procName = MPI.Get_processor_name()

print("Hello World.(Process name={0}, nRank={1}, nProcs={2})". \
format(procName, myrank, nprocs))
```

Real world parcel delivery

- **Major component**
  - Parcel
  - Invoice number
  - Address
  - Address system
  - Signature of sender/receiver



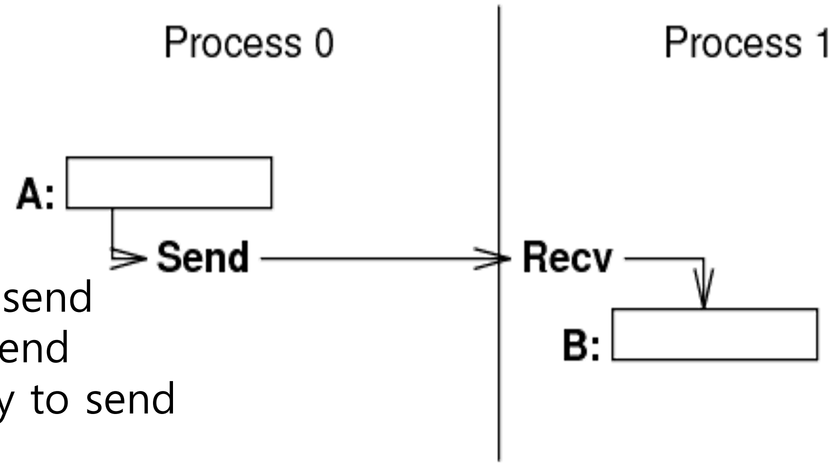
Comparison to MPI world



Real world	MPI world
Parcel	Data, Message
Invoice number	Tag
Address	MPI process
Address system	MPI Communicator
Signature	Status
Complain	Error

Sending and Receiving Message in MPI

- **Basic Message Passing Process**



•Where to send
•What to send
•How many to send

•Where to receive
•What to receive
•How many to receive

- **Message is divided into data and envelope**
  - Data
    - buffer
    - count
    - data type
  - Envelope
    - process identifier (source/destination rank)
    - message tag
    - communicator

MPI Data Types (1/2)

mpi4py data type	C Data Type
MPI.CHAR – 1 Byte character	signed char
MPI.SHORT – 2 Byte integer	signed short int
MPI.INT – 4 Byte integer	signed int
MPI.LONG – 4 Byte integer	signed long int
MPI.UNSIGNED_CHAR – 1 Byte u char	unsigned char
MPI.UNSIGNED_SHORT – 2 Byte u int	unsigned short int
MPI.UNSIGNED – 4 Byte u int	unsigned int
MPI.UNSIGNED_LONG– 4 Byte u int	unsigned long int
MPI.FLOAT – 4 Byte float point	float
MPI.DOUBLE – 8 Byte float point	double
MPI.LONG_DOUBLE- – 8 Byte float point	long double

MPI Data Types (2/2)

mpi4py data type	Fortran Data Type
MPI.INTEGER - 4 Byte Integer	INTEGER
MPI.REAL – 4 Byte floating point	REAL
MPI.DOUBLE_PRECISION – 8 Byte	DOUBLE PRECISION
MPI.COMPLEX – 4 Byte float real	COMPLEX
MPI.LOGICAL – 4 Byte logical	LOGICAL
MPI.CHARACTER – 1 Byte character	CHARACTER(1)

Message envelop

- **All message (or data) to send or receive carries verification information which is called message envelop consisting of,**

Source rank	=	Sender address
Destination rank	=	Receiver address
Tag	=	Invoice number
Communicator	=	Address system

Real world	MPI world
Parcel	Data, Message
Invoice number	Tag
Address	MPI process
Address system	MPI Communicator
Signature	Status

- **A message is received only if the arguments in MPI_Recv agree with the message envelop of an incoming message.**

Send process
Source rank

Send(buf, dest, tag)

Message with message envelop
- Source rank
- Destination rank
- Tag
- Communicator

Recv. process
destination rank

Recv(buf, source, tag, status)

Send & MPI_Recv

- **comm.Send(buf, dest, tag = 0)**
  - buf : initial address of send buffer (choice) (Parcel info.)
  - dest : rank of destination (integer) (Address)
  - tag : message tag (integer) (Invoice #)
  - comm : communicator (handle), usually MPI.COMM_WORLD (Address sys.)

- **comm.Recv(buf, source=ANY_SOURCE, tag=ANY_TAG, status=None)**
  - buf : initial address of receive buffer (choice)
  - source : rank of source (integer)
  - tag : message tag (integer)
  - status : status object (Status)
  - comm : communicator (handle), usually MPI.COMM_WORLD (Address sys.)

Information	Wildcard	Description
Source	ANY_SOURCE	Receive data from any source
Tag	ANY_TAG	Receive data from any tag

Recv - Status

Status Information

- Send Process (Rank)
- Tag
- Data size : Status.GET_COUNT()

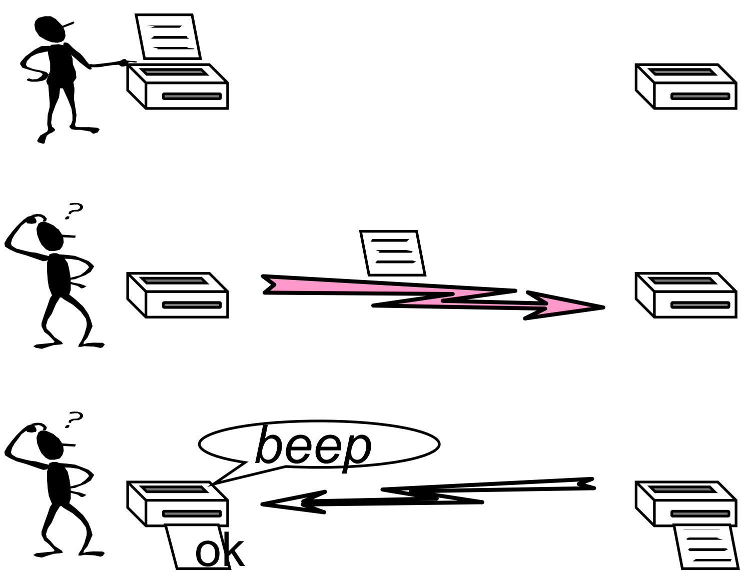
Information	Function	Member
source	Status.Get_source()	Status.source
tag	Status.Get_tag()	Status.tag
Error	Status.Get_error()	Status.error
count	Status.Get_count()	Status.count

Lab3: Send & Recv

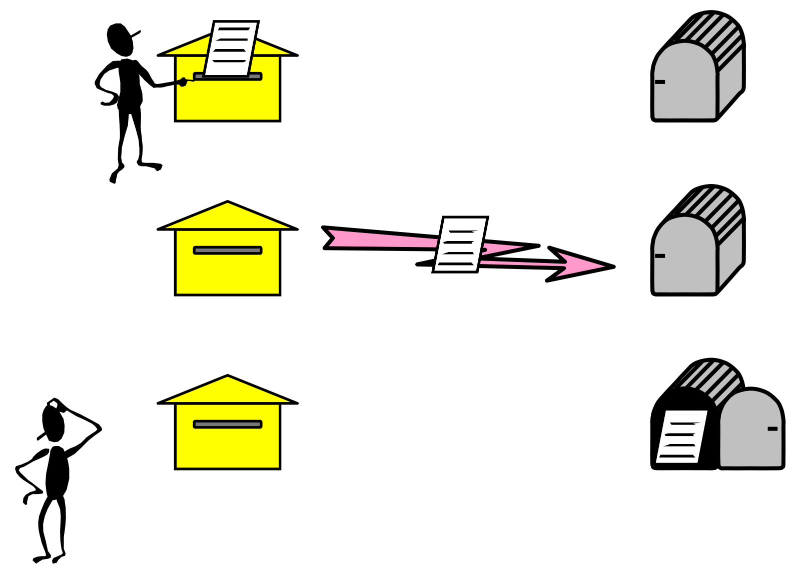
```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

rank = comm.Get_rank() # myrank = comm.rank

if rank == 0 :
   data = np.full(100, 3.0, dtype = float)
   comm.Send(data, dest = 1, tag = 55)

elif rank ==1 :
   value = np.empty(100, dtype = float)
   status = MPI.Status()
   comm.Recv(value, source = MPI.ANY_SOURCE, tag = 55, status = status)

   print("p{0} got data from processor {1}".format(rank, status.source))
   print("p{0} got {1} byte".format(rank, status.count))
   print("p{0} values(5) = {1}".format(rank, value[5]))
```

Synchronous vs Asynchronous comm.

- **Synchronous communication (Blocking comm.)**
  - A synchronous communication does not complete until the message has been received
  - Analogue to the beep or okay-sheet of a fax
- **Asynchronous communication (Non-blocking comm.)**
  - An asynchronous communication completes as soon as the message is on the way.
  - A post card or email



Synchronous communication



Asynchronous communication

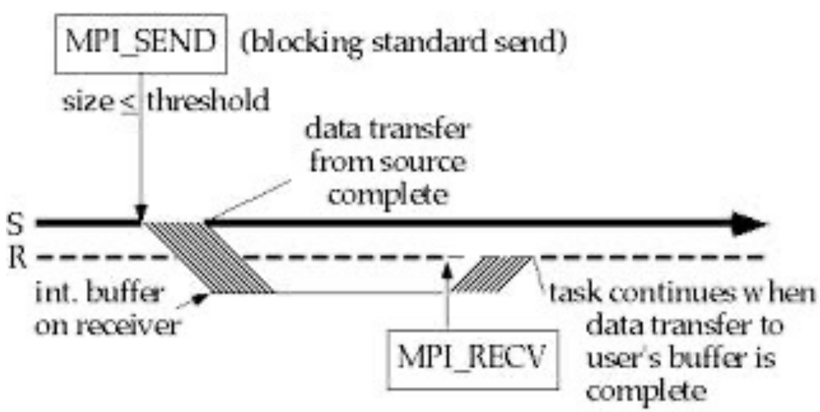
Communication mode

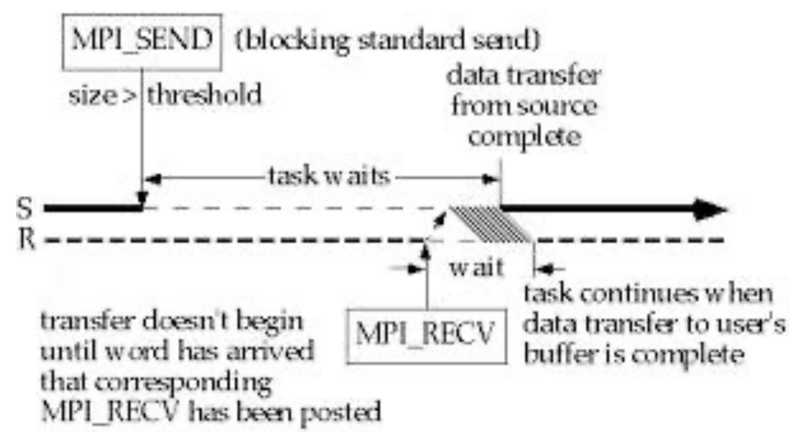
Mode	MPI Call Routine	
	Blocking	Non Blocking
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffer	MPI_BSEND	MPI_IBSEND
Standard	MPI_SEND	MPI_ISEND
Recv	MPI_RECV	MPI_IRECV

Avoiding deadlock (or hung)

MPI_SEND & MPI_RECV : Blocking communication

- MPI_SEND: the call does not return control to the calling program or routine, until the buffer containing the data to be copied unto the receiving process can be safely overwritten (This insures that the message being sent is not "corrupted" before the sending is complete)
- MPI_RECV: the call does not return control to the calling program until the data to be received has in fact been received.

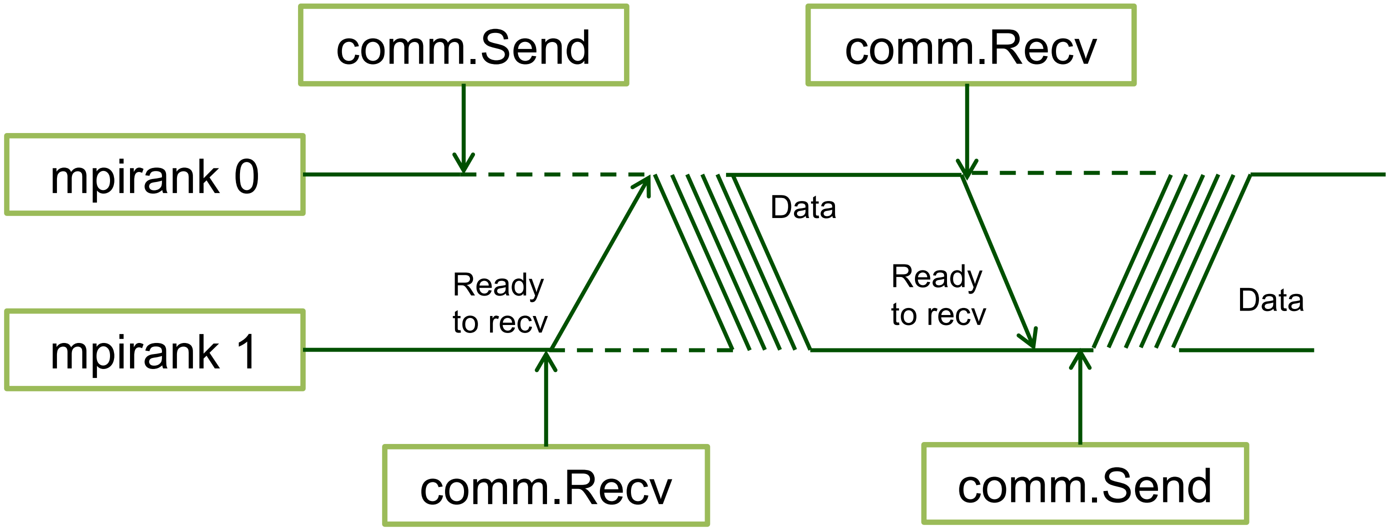




Deadlock can happen

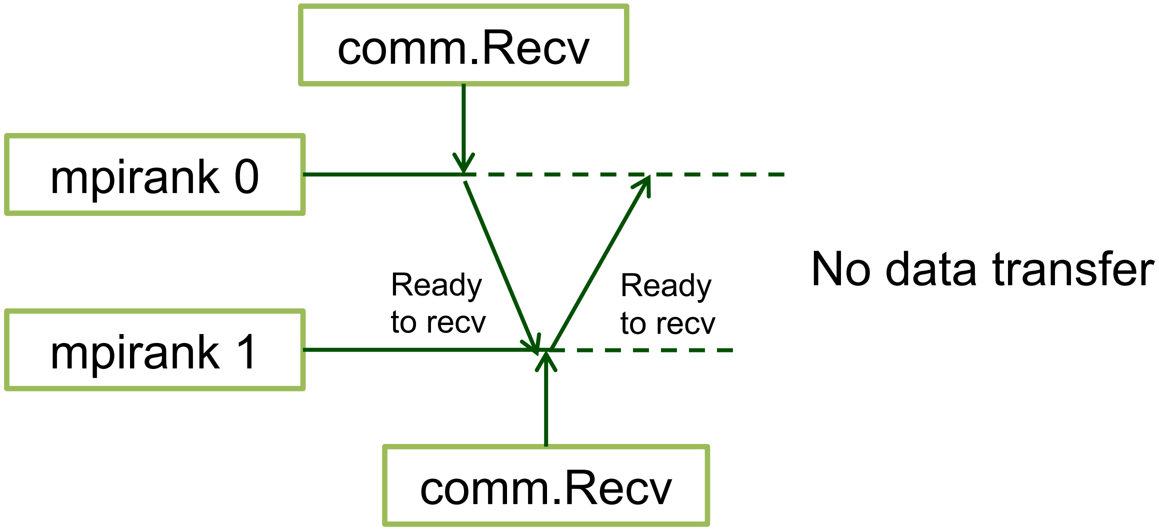
No deadlock

```
!--Exchange messages
if mpirank == 0 :
            comm.Send(a, 1, tag1)
            comm.Recv(b, 1, tag2)

elif mpirank == 1 :
            comm.Recv(a, 0, tag1)
            comm.Send(b, 0, tag2)
```



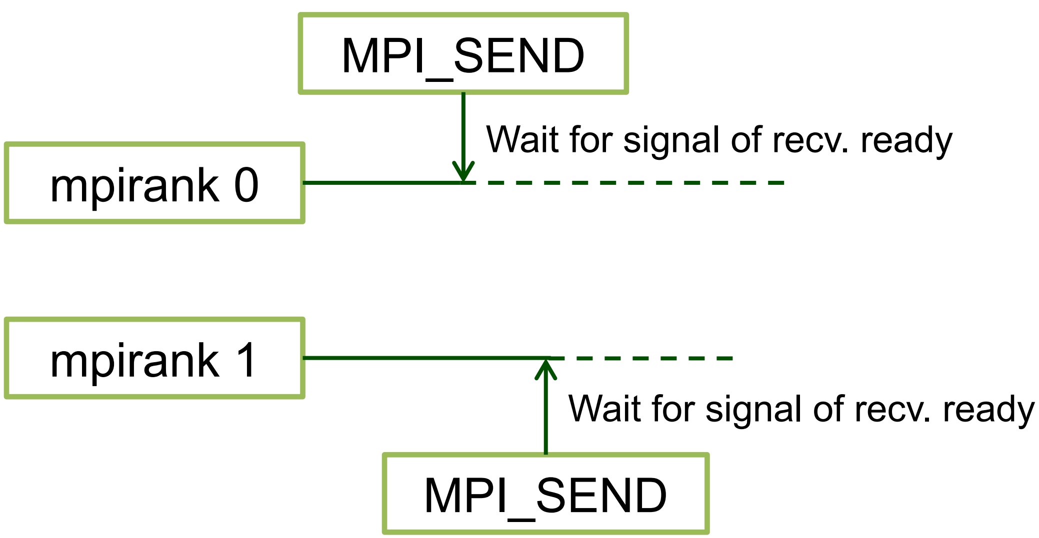
Unconditional deadlock

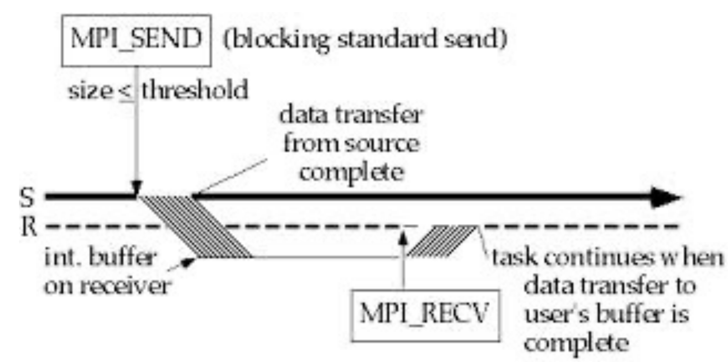
```
!--Exchange messages
if mpirank == 0 :
        comm.Recv(b, 1, tag2)
        comm.Send(a, 1, tag1)

elif mpirank == 1 :
        comm.Recv(a, 0, tag1)
        comm.Send(b, 0, tag2)
```



Conditional deadlock

```
!--Exchange messages
if mpirank == 0 :
        comm.Send(a, 1, tag1)
        comm.Recv(b, 1, tag2)

elif mpirank == 1 :
        comm.Send(b, 0, tag2)
        comm.Recv(a, 0, tag1)
```





Lab5 : deadlock

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

rank = comm.Get_rank() # myrank = comm.rank

buf_size = 256
a = np.ones(buf_size, dtype = int)
b = np.empty(buf_size, dtype = int)

if rank == 0 :
    comm.Send(a, dest = 1, tag = 11)
    comm.Recv(b, source = 1, tag = 55)

elif rank ==1 :
    comm.Send(a, dest = 0, tag = 55)
    comm.Recv(b, source = 0, tag = 11)
```

Ensuring a Program is Safe (No deadlock)

- Must work the same using comm.Send and comm.Ssend
  - comm.Ssend is synchronous mode send: it will *always* wait until the receive has been posted on the receiving end. Even if the message is small and can be buffered internally, it will still wait until the message has started to be received on the other side.
- Strategies for avoiding deadlock
  - pay attention to order of send/receive in communication operations
  - use synchronous or buffered mode communication
  - use comm.Sendrecv
  - use non-blocking communication

Derivative functions

- **comm.Sendrecv**
- **comm.Isend**
- **comm.Irecv**
- **comm.Wait**

- **Communication has three steps**
  1. Initialization : Posting send or recv
  2. Perform other job
     - Do communication and calculation at the same time
  3. Completion : Waiting or Testing

- **Easier to write dead-lock free code**
- **Reduce communication overhead**

comm.Sendrecv

- **Sendrecv(sendbuf, dest, sendtag=0, recvbuf=None, source=ANY_SOURCE, recvtag=ANY_TAG, status=None)**
  - sendbuf (BufSpec) –
  - dest (int) –
  - sendtag (int) –
  - recvbuf (BufSpec) –
  - source (int) –
  - recvtag (int) –
  - status (Optional[Status]) –

Lab7: Sendrecv

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

rank = comm.Get_rank()
size = comm.Get_size()

a = np.zeros(size, dtype = int)
b = np.zeros(size, dtype = int)

a[rank] = rank + 1

inext = rank + 1
iprev = rank - 1

if rank == 0 :
    iprev = size - 1
if rank == size - 1 :
    inext = 0

for i in range(size) :
    if rank == i :
        print('BEFORE : myrank={0}, A = {1}'.format(rank, a))

comm.Sendrecv(a, inext, 77, b, iprev, 77)
# b[rank] = comm.sendrecv(a[rank], inext, 77, None, iprev, 77)

for i in range(size) :
    if rank == i :
        print('AFTER  : myrank={0}, B = {1}'.format(rank, b))
```

Non-blocking communications

- **request = comm.Isend(…)**
  - request: request handle
- **request = comm.Irecv(…)**
  - request: request handle
- **request.Wait(*status=None*)**
- **MPI.Request.Waitall(*requests*, *statuses=None*)**

Lab4 : Isend & Irecv

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

rank = comm.Get_rank() # myrank = comm.rank

data  = np.zeros(100, dtype = float)
value = np.zeros(100, dtype = float)

req_list = []

if rank == 0 :
    for i in range(100) :
        data[i] = i * 100
        req_send = comm.Isend(data[i:i+1], dest = 1, tag = i)
        req_list.append(req_send)
elif rank == 1 :
    for i in range(100) :
        req_recv = comm.Irecv(value[i:i+1] , source = 0, tag = i)
        req_list.append(req_recv)

MPI.Request.Waitall(req_list)

if rank == 0 :
    print("data[99] = {0}\n".format(data[99]))
if rank == 1 :
    print("value[99] = {0}\n".format(value[99]))
```

Blocking and non-blocking communication

- **What is the difference between comm.Send and comm.Isend?**
  - *Send: the call does not return control to the calling program or routine, until the buffer containing the data to be copied unto the receiving process can be safely overwritten*
  - *Isend: the call returns control to the calling routine immediately after posting the send call, before it is safe to overwrite (or use) the buffer being sent.*

  ➔ It is free from deadlock

- **User should control the safe transfer of data by using comm.Wait**
  - Before the program is to use the sent/received buffer, a call to comm.Wait is necessary.
  - comm.Wait is a blocking routine. It does not return control to the calling routine until it is safe to re-use the buffer.

- **Non-blocking communication makes a big room for computation**
  - This allows the program to proceed with computations not involving the communication buffer, while the communication completes.

Non-blocking communication example

```
if (mpirank == 0) {
        req_send = comm.Isend(a,1,tag1)
        req_recv = comm.Irecv(b,1,tag2)
}
else if (mpirank == 1) {
        req_send = comm.Isend(b,0,tag2)
        req_recv = comm.Irecv(a,0,tag1)
}
    We can put calculation not using a or b


req_send.Wait()
req_send.Recv()
```

Assignment

1. Run lab1, 2, 3, 4, 5, 7, 8

2. Run the following code with different np and plot the execution time vs. number of np. Try with different N, 10000, 10000000 and so on.

```
from mpi4py import MPI
import numpy as np
comm= MPI.COMM_WORLD
N = 1000000
x = 2*np.random.rand(N) -1
y = 2*np.random.rand(N) -1
count = np.sum(np.where(x**2 + y**2 <= 1, 1, 0))
if comm.rank== 0:
        for i in range(1,comm.size):
                count += comm.recv()
        print(4*count/(N*comm.size))
else:
        comm.send(count, dest=0)
```