MPI (Message Passing Interface)

All it means is that an application passes messages among processes in order to perform a task. This model works out quite well in practice for parallel applications. For example, a master process might assign work to slave processes by passing them a message that describes the work. Another example is a parallel merge sorting application that sorts data locally on processes and passes results to neighboring processes to merge sorted lists. Almost any parallel application can be expressed with the message passing model.

The 2nd task is to learn MPI. In this task we will be using the mpi4py module.

mpi4py - Point-to-Point Communication. In addition, for quick understanding, please see in the following link,


- **MPI Explanation**

MPI for Python supports convenient, _pickle_-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

- **Communication of generic Python objects**

You have to use all-lowercase methods (of the Comm class), like send(), recv(), bcast(). An object to be sent is passed as a parameter to the communication call, and the received object is simply the return value.

The isend() and irecv() methods return Request instances; completion of these methods can be managed using the test() and wait() methods of the Request class.

The recv() and irecv() methods may be passed a buffer object that can be repeatedly used to receive messages avoiding internal memory allocation. This buffer must be sufficiently large to accommodate the transmitted messages; hence, any buffer passed to recv() or irecv() must be at least as long as the pickled data transmitted to the receiver.

Collective calls like scatter(), gather(), allgather(), alltoall() expect a single value or a sequence of Comm.size elements at the root or all process. They return a single value, a list of Comm.size elements, or None.
• Communication of buffer-like objects

You have to use method names starting with an upper-case letter (of the Comm class), like Send(), Recv(), Bcast(), Scatter(), Gather().

In general, buffer arguments to these calls must be explicitly specified by using a 2/3-list/tuple like [data, MPI.DOUBLE], or [data, count, MPI.DOUBLE] (the former one uses the byte-size of data and the extent of the MPI datatype to define count).

Automatic MPI datatype discovery for NumPy arrays and PEP-3118 buffers is supported, but limited to basic C types (all C/C99-native signed/unsigned integral types and single/double precision real/complex floating types) and availability of matching datatypes in the underlying MPI implementation. In this case, the buffer-provider object can be passed directly as a buffer argument, the count and MPI datatype will be inferred.

Running mpi4py within ipyparallel environment

MPI jobs are able to run both in distributed memory as well as on top of shared memory architectures.

To start running MPI jobs in Jupyter Notebook in parallel, the first thing is to

Start a number of engines:

--> This can be done under the Tab "IPython Clusters".

--> Start 2-8 engines of type "MPI". (Possibly "default" might work as well...)

Now you are ready to start running the ipyparallel code. (Note that for now we are running ipyparallel only with one purpose in mind -- to be able to run MPI jobs. Ipyparallel itself has a lot of additional usage possibilities, but currently we do not consider this functionality.)

import ipyparallel

cluster = ipyparallel.Client(profile='MPI') # or profile='default'

print('profile:', cluster.profile)

print("IDs:", cluster.ids) # Print process id numbers

OutPut:

profile: MPI

IDs: [0, 1, 2, 3, 4, 5, 6, 7]

Now it should be ready to run MPI commands:
from mpi4py import MPI

mpi_rank=MPI.COMM_WORLD.Get_rank()

mpi_size = MPI.COMM_WORLD.Get_size()

print("Process %d out of %d started.
" % (mpi_rank, mpi_size))

Output:

[stdout:0]

Process 0 out of 8 started.

[stdout:1]

Process 1 out of 8 started.

[stdout:2]

..

...

[stdout:7]

Process 7 out of 8 started.

After analyzing the above examples, reader supposed to be perform few tasks, which will be follows,

Task to Perform

The purpose of this task is to play around with the basic MPI commands in given examples and do corresponding coding exercises in meanwhile.

Python Objects

from mpi4py import MPI

comm = MPI.COMM_WORLD

rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
comm.send(data, dest=1, tag=11)

elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(data)

Output:
[stdout:1] {'a': 7, 'b': 3.14}

Task 2.1 {5 points}

Write a function called \texttt{return\_2nd (source, dest, tag, items)}, which

- sends a message with the given \texttt{tag} containing the \texttt{items-object} of type \texttt{list} or \texttt{tuple} -- consisting of at least two items -- from \texttt{source} process to the process \texttt{dest}. The function will send the second item from the received object back to the sender in a message tagged with \texttt{tag+1}.
- On the \texttt{dest} process the function returns the received object \texttt{items};
- On the \texttt{source} process the function returns the item that was sent back to it from \texttt{dest};
- On rest of the processes the function returns None.

from nose.tools import assert_equal
from mpi4py import MPI

comm = MPI.COMM_WORLD

rank_m = comm.Get_rank()

size_m = comm.Get_size()

if size_m <= 1:
    print('Start at least 2 engines!')
else:

    source=0; dest=1; items=[0.1,'Yes',0.001]
    data=return_2nd(source,dest,111,items)

    if rank_m == source:
        assert_equal(data,'Yes')

    elif rank_m == dest:

Python objects with non-blocking communication

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
    print(data)
```

**Task 2.2 {3 points}**
Modify your function from **Task 1.1** into a function **return_2nd_nonblocking()**, which is using non-blocking communication!

```python
from nose.tools import assert_equal
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank_m = comm.Get_rank()
size_m = comm.Get_size()
if size_m <= 1:
    print('Start at least 2 engines!')
```
else:
    source=1; dest=2; items=[0.1,'Y',0.001]
    data=return_2nd_nonblocking(source,dest,222,items)
    if rank_m == source:
        assert_equal(data,'Y')
    elif rank_m == dest:
        assert_equal(data,items)
    else:
        assert_equal(data,None)

NumPy arrays

from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(30, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)  # NOTE the Capital letters in Send and Recv here!
elif rank == 1:
    data = numpy.empty(30, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
    print(data)
# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
data = numpy.empty(100, dtype=numpy.float64)

comm.Recv(data, source=0, tag=13)

print(data)

Task 2.3 {5 points}

Write a function called `return_every_2nd_number(source, dest, tag, data)`, which

- sends a message with the given `tag` containing the `data-object` of type `numpy.array` -- consisting of at least two array elements -- from `source` process to the process `dest`. On the `dest` process the function will send back every second array element to the original source process in a message tagged with `tag+1`.
- On the `dest` process the function returns the received object `data`
- On the `source` process the function returns the values that were sent back to it from `dest`
- On rest of the processes the function returns None.

NB! Beware! If trying to send with `comm.Send` an array with noncontiguous memory data[::2] - the kernel might get stalled.
- do send `data[::2].copy()` instead (as this will repack the array into memory without gaps...)

from nose.tools import assert_equal
from mpi4py import MPI

comm = MPI.COMM_WORLD

rank_m = comm.Get_rank()
size_m = comm.Get_size()

if size_m <= 1:
    print('Start at least 2 engines!')
else:
    source=0; dest=1; arr = numpy.random.random(5)
    returned_arr=return_every_2nd_number(source=source,dest=dest,tag=333,data=arr)
    if rank_m == source:
        assert_equal(arr[2],returned_arr[1]) # check just here...
    elif rank_m == dest: # just check that the length of the array is correct?
assert_equal(len(arr),5)

else:
    assert_equal(returned_arr,None)

Task 2.4 {15 points}

Parallel matrix-vector operation

- The process with rank=0 generates an $n \times m$ matrix $A$ with random floating-point values and a vector $x$ of length $m$.
- Write a function $A_{x \_par}(A,x)$ which does the following:
  - The matrix $A$ gets distributed to $A_{\_loc}$ row-by-row to all processes in a round-robin manner (including the process with rank=0), i.e.
    - row 0 goes to process 0,
    - row 1 goes to process 1,
    - ...
    - row $n$ goes to process $n \mod mpi\_size$
  - Vector $x$ gets sent separately to all processes.
  - Now $A \times x$ operation is performed in parallel and the result is appropriately placed back to the process with rank=0, which gives it as a return value.
  - Processes with rank $\neq$ 0 return None

Use only Point-to-point communication in this solution!

from nose.tools import assert_equal
from mpi4py import MPI

comm = MPI.COMM_WORLD

rank_m = comm.Get_rank()

size_m = comm.Get_size()

if size_m <= 1:
    print('Start at least 2 engines!')
else:
    source=0; dest=1; arr = numpy.random.random(5)

n=12; m=7

A=numpy.random.random((n,m))

x=numpy.random.random(m)

y=Ax_par(A,x)

yy=numpy.dot(A,x)

if rank_m == 0:
    numpy.testing.assert_allclose(yy,y)
Before submitting the assignment, please write your comments below!

Comments like the following are very welcome:

- Was it interesting to work with (or rather boring)?
  - In the latter case, please suggest some constructive ideas on improvement...!
- Was the assignment too easy / too difficult / on appropriate level of complexity for you?
- What did you learn?
- Are the awarded points in balance with the time you spent on different parts?
- Any ideas you want to share for the future on how to improve the worksheet in any aspect?

Also, we would be very thankful, if you could contribute on:

- Any ideas you can suggest for next problems to work out for future classes (either within this year's course or for next year)? Special thanks to those who gave some good suggestions from last time!

... and you earn {1 point} for this as well :-)

ANSWER.

And the very last thing before submitting -- approximately, how many hours did you spend on this assignment (including in-class activities)?

... and you earn another {1 point} for this as well :-)

ANSWER.