

Communicating NumPy arrays

Xin Li

Viktor Rehnberg

Overview

- Numpy and buffer-like object
- Blocking communication
- Nonblocking communication
- Collective communication

Numpy and buffer-like object

NumPy

- Powerful N-dimensional arrays
- Optimized for performance
- Easy to use
- Open source

NumPy

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

NumPy

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

```
>>> b = a.T
```

```
>>> b
```

```
array([[ 0.,  4.,  8., 12.],  
       [ 1.,  5.,  9., 13.],  
       [ 2.,  6., 10., 14.],  
       [ 3.,  7., 11., 15.]])
```

NumPy

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

```
>>> c = a[0:2,0:2]
```

```
>>> c
```

```
array([[0., 1.],  
       [4., 5.]])
```

NumPy

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

```
>>> d = a[:,1].reshape(2,2)
```

```
>>> d
```

```
array([[ 1.,  5.],  
       [ 9., 13.]])
```


NumPy

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

```
>>> np.matmul(a, a.T)
```

```
array([[ 14.,  38.,  62.,  86.],  
       [ 38., 126., 214., 302.],  
       [ 62., 214., 366., 518.],  
       [ 86., 302., 518., 734.]])
```

NumPy with mpi4py

- Numpy arrays can be communicated by the all-lowercase methods like send, recv, bcast, etc.
- For best efficiency Numpy arrays can be communicated as **buffer-like objects**, using method with a leading uppercase letter.
 - Send, Recv
 - Isend, Irecv
 - Bcast, Reduce
 - Scatterv, Gatherv (more useful than Scatter/Gather)

Numpy array as buffer-like objects

- communication is fast
 - close to the speed of MPI communication in C
- less flexible
 - memory of the receiving buffer needs to be allocated
 - size of the sending buffer should not exceed that of the receiving buffer
 - mpi4py expects the buffer-like objects to have contiguous memory

Contiguous or non-contiguous?

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

Contiguous or non-contiguous?

```
>>> import numpy as np
```

```
>>> a = np.arange(16.).reshape(4,4)
```

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.],  
       [12., 13., 14., 15.]])
```

```
>>> a.flags
```

```
C_CONTIGUOUS : True
```

```
F_CONTIGUOUS : False
```

```
OWNDATA : False
```

```
...
```

Contiguous or non-contiguous?

```
>>> b = a.T  
>>> b.flags  
C_CONTIGUOUS : False  
F_CONTIGUOUS : True  
OWNDATA : False  
...
```

Contiguous or non-contiguous?

```
>>> b = a.T
>>> b.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  ...
```

```
>>> b[0,0] = 99.0
>>> b
array([[99.,  4.,  8., 12.],
       [ 1.,  5.,  9., 13.],
       [ 2.,  6., 10., 14.],
       [ 3.,  7., 11., 15.]])
```

Contiguous or non-contiguous?

```
>>> b = a.T
>>> b.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  ...
```

```
>>> b[0,0] = 99.0
>>> b
array([[99.,  4.,  8., 12.],
       [ 1.,  5.,  9., 13.],
       [ 2.,  6., 10., 14.],
       [ 3.,  7., 11., 15.]])
>>> a
array([[99.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```


Contiguous or non-contiguous?

```
>>> a  
array([[99., 1., 2., 3.],  
       [ 4., 5., 6., 7.],  
       [ 8., 9., 10., 11.],  
       [12., 13., 14., 15.]])
```

Contiguous or non-contiguous?

```
>>> a
array([[99., 1., 2., 3.],
       [ 4., 5., 6., 7.],
       [ 8., 9., 10., 11.],
       [12., 13., 14., 15.]])
```

```
>>> c = a[0:2, 0:2]
```

```
>>> c
array([[99., 1.],
       [ 4., 5.]])
```

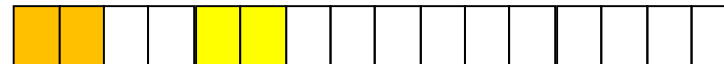
```
>>> c.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
```

Contiguous or non-contiguous?

```
>>> a  
array([[99., 1., 2., 3.],  
       [ 4., 5., 6., 7.],  
       [ 8., 9., 10., 11.],  
       [12., 13., 14., 15.]])
```



```
>>> c = a[0:2, 0:2]  
>>> c  
array([[99., 1.],  
       [ 4., 5.]])
```



```
>>> c.flags  
C_CONTIGUOUS : False  
F_CONTIGUOUS : False  
OWNDATA : False
```

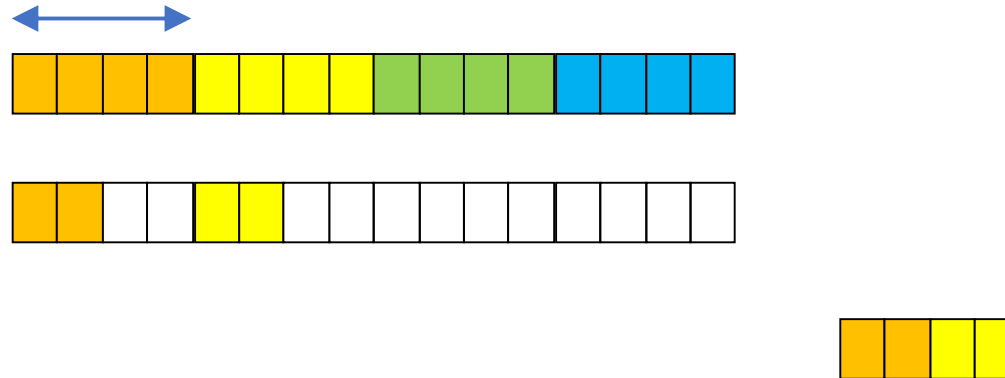
Contiguous or non-contiguous?

```
>>> d = c.copy()
>>> d.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
...
```

Contiguous or non-contiguous?

```
>>> d = c.copy()
>>> d.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
...
```

```
>>> a.strides
(32, 8)
>>> c.strides
(32, 8)
>>> d.strides
(16, 8)
```



Use NumPy array as buffer-like object

- The array itself, or a list or tuple with
 - 2 or 3 elements
 - 4 elements for the vector variants (Scatterv, Gatherv)
- data
- [data, MPI.DOUBLE]
- [data, n, MPI.DOUBLE]
- [data, count, displ, MPI.DOUBLE]

Blocking send/rcv

Send / Recv

- Syntax
 - `comm.Send(obj, dest=dest, tag=tag)`
 - `comm.Recv(obj, source=source, tag=rag)`
- Note
 - obj needs to be created prior to the communication
 - size of obj needs to be known before hand

Send / Recv

- example

```
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD

if comm.rank == 0:
    data = np.arange(4.)
    for i in range(1, comm.size):
        comm.Send(data, dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(4)
    comm.Recv(data, source=0, tag=comm.rank)
    print(f"Process {comm.rank} received data:", data)
```

Send / Recv

- output

Process 0 sent data: [0. 1. 2. 3.]

Process 0 sent data: [0. 1. 2. 3.]

Process 0 sent data: [0. 1. 2. 3.]

Process 1 received data: [0. 1. 2. 3.]

Process 2 received data: [0. 1. 2. 3.]

Process 3 received data: [0. 1. 2. 3.]

Send / Recv

- question
 - If the size of the numpy array is only known on the master process (rank 0), how do we send it to other processes (rank > 0)?

Send / Recv

- question
 - If the size of the numpy array is only known on the master process (rank 0), how do we send it to other processes (rank > 0)?
- solution
 - need to send the size of the array first
- exercise
 - rewrite the above example assuming that the size of array is not known on the non-master nodes (rank > 0).
 - for sending an integer you may use the all-lowercase send

Send / Recv

- exercise
 - what will happen if send and recv buffers are of different dtypes?

Send / Recv

- exercise
 - what will happen if you try to send an array with non-contiguous memory?
 - hint: this is how to create a simple array with non-contiguous memory
`data = np.arange(12.)[:,::2]`

Send / Recv

- exercise
 - what will happen if the receiving buffer is *larger* than the sent array?

```
if comm.rank == 0:
    data = np.arange(4.)
    for i in range(1, comm.size):
        comm.Send(data, dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(6)
    comm.Recv(data, source=0, tag=comm.rank)
    print(f"Process {comm.rank} received data:", data)
```

Send / Recv

- exercise
 - what will happen if the receiving buffer is *smaller* than the sent array?

```
if comm.rank == 0:
    data = np.arange(4.)
    for i in range(1, comm.size):
        comm.Send(data, dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(3)
    comm.Recv(data, source=0, tag=comm.rank)
    print(f"Process {comm.rank} received data:", data)
```


Send / Recv

- best practice: check status

```
if comm.rank == 0:
    data = np.arange(4.)
    for i in range(1, comm.size):
        comm.Send(data, dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(3)
    status = MPI.Status()
    comm.Recv(data, source=0, tag=comm.rank, status=status)
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data)
```

Send / Recv with buffer size

- use [data, n, MPI.DOUBLE] to specify the buffer

```
if comm.rank == 0:
    data = np.arange(4.)
    for i in range(1, comm.size):
        comm.Send([data, 2, MPI.DOUBLE], dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data[:2])
else:
    data = np.zeros(4)
    status = MPI.Status()
    comm.Recv([data, 2, MPI.DOUBLE], source=0, tag=comm.rank, status=status)
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data[:2])
```

Send / Recv with buffer size

- use [data, n, MPI.DOUBLE] to specify the buffer

```
if comm.rank == 0:
    data = np.arange(4.)
    for i in range(1, comm.size):
        comm.Send([data, 2, MPI.DOUBLE], dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data[:2])
else:
    data = np.zeros(4)
    status = MPI.Status()
    comm.Recv([data, 2, MPI.DOUBLE], source=0, tag=comm.rank, status=status)
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data[:2])
```

- NOTE: The size of buffer should never be larger than the size of the Numpy array.

Send / Recv with buffer size

- use array slicing

```
if comm.rank == 0:
    data = np.arange(10)
    for i in range(1, comm.size):
        comm.Send(data[2:6], dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(10)
    status = MPI.Status()
    comm.Recv(data[2:6], source=0, tag=comm.rank), status=status
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data[:2])
```

Send / Recv with 2D array

- exercise

- Send / Recv 2D array

```
if comm.rank == 0:
    data = np.arange(16).reshape(4, 4)
    for i in range(1, comm.size):
        comm.Send(data[2:6], dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(16).reshape(4, 4)
    status = MPI.Status()
    comm.Recv(data[2:6], source=0, tag=comm.rank, status=status)
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data[:2])
```

- Will it work if the receiving buffer is a 1D array?

Send / Recv with 2D array

- exercise

- What will happen if we send a 2D array with .T (transpose)?

```
if comm.rank == 0:
    data = np.arange(16).reshape(4, 4).T
    for i in range(1, comm.size):
        comm.Send(data[2:6], dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(16).reshape(4, 4)
    status = MPI.Status()
    comm.Recv(data[2:6], source=0, tag=comm.rank, status=status)
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data[:2])
```

Send / Recv with 2D array

- exercise
 - What will happen if we send a 2D array with `.T.copy()` (copy of transpose)?

```
if comm.rank == 0:
    data = np.arange(16).reshape(4, 4).T.copy()
    for i in range(1, comm.size):
        comm.Send(data[2:6], dest=i, tag=i)
        print(f"Process {comm.rank} sent data:", data)
else:
    data = np.zeros(16).reshape(4, 4)
    status = MPI.Status()
    comm.Recv(data[2:6], source=0, tag=comm.rank, status=status)
    if status.error:
        MPI.COMM_WORLD.Abort(status.error)
    print(f"Process {comm.rank} received data:", data[:2])
```

Non-blocking send/rcv

Isend / Irecv

- Syntax
 - `comm.Isend(obj, dest=dest, tag=tag)`
 - `comm.Irecv(obj, source=source, tag=rag)`
- Note
 - obj needs to be created prior to the communication
 - size of obj needs to be known before hand
 - A Request object is returned by Isend / Irecv
 - Use Wait method of the Request object
 - No Status involved

Isend / Irecv

- example

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

if comm.rank == 0:
    data = np.arange(4.)
    reqs = []
    for i in range(1, comm.size):
        reqs.append(comm.isend(data, dest=i, tag=i))
    for req in reqs:
        req.wait()
    print(f'Process {comm.rank} sent data:', data)
else:
    data = np.zeros(4)
    req = comm.irecv(data, source=0, tag=comm.rank)
    req.wait()
    print(f'Process {comm.rank} received data:', data)
```

Isend / Irecv

- example

```
from mpi4py import MPI
import numpy as np
```

```
comm = MPI.COMM_WORLD
```

```
if comm.rank == 0:
    data = np.arange(4.)
    reqs = []
    for i in range(1, comm.size):
        reqs.append(comm.isend(data, dest=i, tag=i))
    for req in reqs:
        req.wait()
    print(f'Process {comm.rank} sent data:', data)
else:
    data = np.zeros(4)
    req = comm.irecv(data, source=0, tag=comm.rank)
    req.wait()
    print(f'Process {comm.rank} received data:', data)
```

Isend / Irecv

- example

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

if comm.rank == 0:
    data = np.arange(4.)
    reqs = [comm.Isend(data, dest=i, tag=i) for i in range(1, comm.size)]
    for req in reqs:
        req.wait()
        print(f'Process {comm.rank} sent data:', data)
else:
    data = np.zeros(4)
    req = comm.Irecv(data, source=0, tag=comm.rank)
    req.wait()
    print(f'Process {comm.rank} received data:', data)
```

Isend / Irecv

- exercise:
 - what if the size of the receiving buffer is larger than the sent array?
 - what if the size of the receiving buffer is smaller than the sent array?
- send a 2D array
 - without transpose
 - with .T
 - with .T.copy()

Collectives

Bcast

- Syntax
 - `comm.Bcast(obj, root=root)`
- Note
 - obj needs to be created prior to the communication
 - size of obj needs to be known before hand
 - root can be non-zero (source of communication)

Bcast

```
from mpi4py import MPI
import numpy as np
```

```
comm = MPI.COMM_WORLD
```

```
if comm.rank == 0:
    data = np.arange(4.0)
else:
    data = np.zeros(4)
```

```
comm.Bcast(data, root=0)
```

```
print(f'Process {comm.rank} has data:', data)
```


Bcast

- exercise
 - what will happen if the size of the receiving buffer is *larger*?
 - what will happen if the size of the receiving buffer is *smaller*?

Scatterv

- vector variant of Scatter
- Syntax
 - `comm.Scatterv([sendbuf, count, displ, mpi_dtype], recvbuf, root=root)`
- Note
 - count is count per rank
 - displ by default calculated from count
 - recvbuf needs to be created prior to the communication
 - size of recvbuf needs to be known beforehand

Scatterv

- determine count

```
base, rem = divmod(sendbuf.size, comm.size)
```

```
# count: the size of each sub-task
```

```
count = np.full(shape=comm.size, fill_value=base))
```

```
count[:rem] += 1
```

Scatterv

- determine count

```
base, rem = divmod(sendbuf.size, comm.size)

# count: the size of each sub-task
count = np.full(shape=comm.size, fill_value=base))
count[:rem] += 1
```

- `sendbuf.size = 15; comm.size = 4`
 - `base = 3; rem = 3`
 - `count = np.array([4, 4, 4, 3])`

Scatterv

- determine count

```
base, rem = divmod(sendbuf.size, comm.size)
```

```
# count: the size of each sub-task
```

```
count = np.full(shape=comm.size, fill_value=base))
```

```
displ[:rem] += 1
```

- `sendbuf.size = 15; comm.size = 4`
 - `base = 3; rem = 3`
 - `count = np.array([4, 4, 4, 3])`

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

rank 0

rank 1

rank 2

rank 3

Scatterv

```
from mpi4py import MPI  
import numpy as np
```

```
comm = MPI.COMM_WORLD
```

Scatterv

```
if comm.rank == 0:  
    sendbuf = np.arange(15.0)  
  
    # count: the size of each sub-task  
    base, rem = divmod(sendbuf.size, comm.size)  
    count = np.full(shape=comm.size, base=rem)  
    count[:rem] += 1  
  
else:  
    sendbuf = None  
  
    # initialize count on worker processes  
    count = np.zeros(comm.size, dtype=int)
```

Scatterv

```
# broadcast count
```

```
comm.Bcast(count, root=0)
```

```
# initialize recvbuf on all processes
```

```
recvbuf = np.zeros(count[comm.rank])
```

```
comm.Scatterv([sendbuf, count], recvbuf, root=0)
```

```
print(f'After Scatterv, process {comm.rank} has data:', recvbuf)
```


Scatterv

- output

After Scatterv, process 0 has data: [0. 1. 2. 3.]

After Scatterv, process 2 has data: [8. 9. 10. 11.]

After Scatterv, process 1 has data: [4. 5. 6. 7.]

After Scatterv, process 3 has data: [12. 13. 14.]

Gatherv

- vector variant of Gather
- Syntax
 - `comm.Gatherv(sendbuf, [recvbuf, count, displ, mpi_dtype], root=root)`
- Note
 - count is count per rank
 - displ by default calculated from count
 - recvbuf needs to be created prior to the communication
 - size of recvbuf needs to be known before hand

Gatherv

- continue with the Scatterv code

```
sendbuf2 = recvbuf  
recvbuf2 = np.zeros(count.sum())
```

```
comm.Gatherv(sendbuf2, [recvbuf2, count], root=0)
```

```
if comm.rank == 0:  
    print('After Gatherv, process 0 has data:', recvbuf2)
```

Gatherv

- output

After Gatherv, process 0 has data:

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14.]
```

Reduce

- Syntax
 - `comm.Reduce(sendbuf, recvbuf, op=op, root=root)`
- Note
 - default op is `MPI.SUM`
 - root can be non-zero (target of Reduce)

Reduce

- continue with the Scatterv code

```
partial_sum = np.zeros(1)
partial_sum[0] = sum(recvbuf)
print(f'Partial sum on process {comm.rank} is:', partial_sum[0])
```

```
total_sum = np.zeros(1)
comm.Reduce(partial_sum, total_sum, op=MPI.SUM, root=0)
```

```
if comm.rank == 0:
    print('After Reduce, total sum on process 0 is:', total_sum[0])
```

Reduce

- output

Partial sum on process 3 is: 39.0

Partial sum on process 1 is: 22.0

Partial sum on process 2 is: 38.0

Partial sum on process 0 is: 6.0

After Reduce, total sum on process 0 is: 105.0

Reduce

- exercise
 - Use Reduce to compute the sum of numpy arrays

Reduce

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD

partial = np.arange(10.) * comm.rank
print(f'Before Reduce, partial on process {comm.rank} is:', partial)

total = np.zeros(10)
comm.Reduce(partial, total, op=MPI.SUM, root=0)
if comm.rank == 0:
    print(f'After Reduce, total on process {comm.rank} is:', total)
```

Reduce

Before Reduce, partial on process 0 is: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Before Reduce, partial on process 2 is: [0. 2. 4. 6. 8. 10. 12. 14. 16. 18.]

Before Reduce, partial on process 1 is: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]

Before Reduce, partial on process 3 is: [0. 3. 6. 9. 12. 15. 18. 21. 24. 27.]

After Reduce, total on process 0 is: [0. 6. 12. 18. 24. 30. 36. 42. 48. 54.]

Summary

Numpy array as buffer-like object in mpi4py

- fast communication
- less flexible code (need to deal with memory)
- blocking and nonblocking communication
 - Send, Recv, Isend, Irecv
- collective communication
 - Bcast, Scatterv, Gatherv, Reduce