mpi4py and Numpy

Juan de Gracia, Xin Li
PDC - KTH, Stockholm

Overview

- Why NumPy with MPI?
- NumPy arrays as buffer-like objects
  - How to check contiguity with flags
  - How to check how memory allocation with strides
- Blocking point-to-point with Send and Recv
- Non-blocking point-to-point with Isend and Irecv
- Collectives I: Bcast
- Collectives II: uneven data with Scatterv and Gatherv
- Collectives III: Reduce

Introduction to NumPy and MPI

Introduction to NumPy and MPI

- NumPy:
  - High-performance numerical library.
  - Supports large arrays and vectorized operations.
- MPI:
  - Enables parallel programming for distributed systems.
  - Ideal for scalable computations.
- Together:
  - Communicate and process NumPy arrays efficiently in parallel

Advantages of Using NumPy with MPI

- **Performance:** Leverages both NumPy's speed and MPI's parallelism.
- **Ease of Use:** Direct integration with `mpi4py`.
- Example Applications:
  - Distributed matrix operations.
  - Large-scale scientific simulations.

Communication of NumPy Arrays

- **Buffer-Like Objects:** NumPy arrays can directly serve as MPI buffers.
- Some methods:
  - Blocking: `Send`, `Recv`
  - Non-blocking: `Isend`, `Irecv`
  - Collective: `Bcast`, `Scatterv`, `Gatherv`, `Reduce`

NumPy Arrays as Buffer-Like Objects

What are Buffer-Like objects

- Arrays used directly as send/receive buffers.
- Efficient and close to MPI's native communication speed.
- Example:
    - `comm.Send([data, MPI.DOUBLE], dest=1)`

Requirements for Numpy Buffers

- Must be **contiguous** in memory (MPI expects that)
- Receiving buffers must be **pre-allocated** and of the same size as the send buffer
- Example of how to allocate a buffer:
  - `recvbuf = np.zeros(data.size, dtype='float64')`
  - `comm.Recv(recvbuf, source=0)`

How to check contiguity with .flags

- NumPy flags displays if the array is contiguous in memory and according to which criteria.
- Flags to check:
    - `C_CONTIGUOUS` Row-major (C-Style)
    - `F_CONTIGUOUS` Column major (Fortran-Style)
- By default NumPY uses C-Style
- Try this out:
    - `a = np.random.rand(10,10)`
    - `a.flags`
    - `b = a.T`
    - `b.flags`
    - `c = b.copy()`
    - `c.flags`

How to check contiguity with .flags

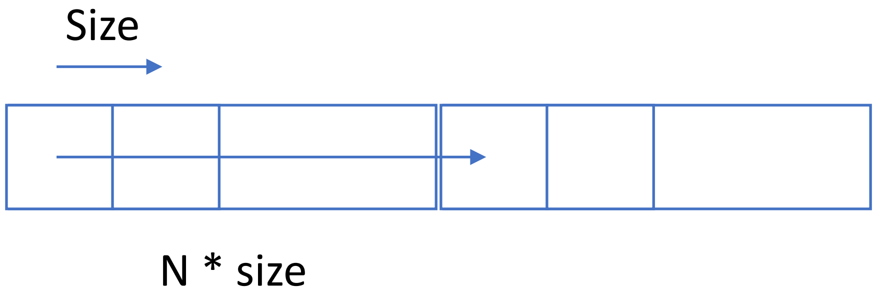
- NumPy flags displays if the array is contiguous in memory and according to which criteria.
- Flags to check:
  - `C_CONTIGUOUS` Row-major (C-Style)
  - `F_CONTIGUOUS` Column major (Fortran-Style)
- By default NumPY uses C-Style
- Try this out:
  - `a = np.random.rand(10,10)`
  - `a.flags # C_CONTIGUOUS = True`
  - `b = a.T`
  - `b.flags # F_CONTIGUOUS = True`
  - `c = b.copy()`
  - `c.flags # C_CONTIGUOUS = True`

How to check memory allocation with Strides

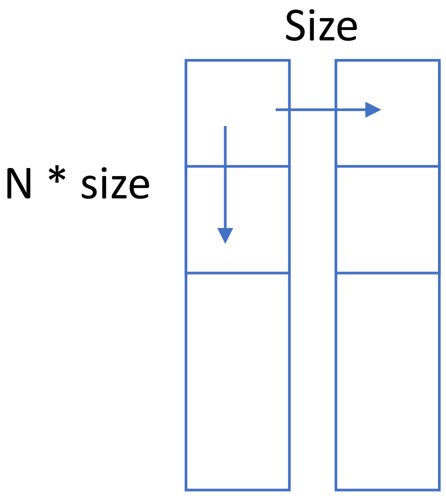
- What are strides?
  - Indicates how many **bytes** to move in each dimension to access the next element.
  - The method `a.strides` will return a tuple with:
    - For 1D arrays the size of the data type in bytes (1,2,4,8 = 8,16,32,64)
    - For 2D arrays (row stride, column stride)
      - C Style (row_stride > column_stride)
      - F Style (row_stride < column_stride)

Understanding C-Style vs F-Style

C Style = Row-major



Fortran-Style = Column major



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/recv

Send / Recv

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

Send / Recv

- example

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

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

if rank == 0:
    data = np.arange(10, dtype=np.float64)
    for i in range(1, size):
        comm.Send(data, dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(10, dtype=np.float64)
    comm.Recv(data, source=0, tag=rank)
    print(f"Rank {rank} received data from rank 0: {data}")

print(f"Rank {rank} has data: {data}")
```

Send / Recv

- output

```
Rank 0 sent data to rank 1
Rank 0 sent data to rank 2
Rank 0 sent data to rank 3
Rank 0 has data: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Rank 1 received data from rank 0: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Rank 1 has data: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Rank 2 received data from rank 0: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Rank 2 has data: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Rank 3 received data from rank 0: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Rank 3 has data: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

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 rank == 0:
    data = np.arange(10, dtype=np.float64)
    for i in range(1, size):
        comm.Send(data, dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(12, dtype=np.float64)
    comm.Recv(data, source=0, tag=rank)
    print(f"Rank {rank} received data from rank 0: {data}")

print(f"Rank {rank} has data: {data}")
```

Send / Recv

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

```
if rank == 0:
    data = np.arange(10, dtype=np.float64)
    for i in range(1, size):
        comm.Send(data, dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(9, dtype=np.float64)
    comm.Recv(data, source=0, tag=rank)
    print(f"Rank {rank} received data from rank 0: {data}")
```

Send / Recv

- best practice: check status

```
if rank == 0:
    data = np.arange(10, dtype=np.float64)
    for i in range(1, size):
        comm.Send(data, dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(9, dtype=np.float64)
    status = MPI.Status()
    comm.Recv(data, source=0, tag=rank, status=status)
    if status.error != 0:
        comm.Abort(status.error)

    print(f"Rank {rank} received data from rank 0: {data}")

print(f"Rank {rank} has data: {data}")
```

Send / Recv with buffer size

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

```
if rank == 0:
    data = np.arange(10, dtype=np.float64)
    for i in range(1, size):
        # Using [data, size, type] to send a buffer of data
        comm.Send([data, 2, MPI.DOUBLE], dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(10, dtype=np.float64)
    # Using [data, size, type] to receive a buffer of data
    comm.Recv([data, 2, MPI.DOUBLE], source=0, tag=rank)

    print(f"Rank {rank} received data from rank 0: {data[:2]}")

print(f"Rank {rank} has data: {data}")
```

Reminder: Standard send in C: MPI_Send

```
int MPI_Send(void* buf, int count, MPI_Datatype
 datatype, int dest, int tag, MPI_Comm comm)
```

- `buf:` address of send buffer
- `count:` number of elements to be sent
- `datatype:` date type of buffer (explained further down)
- `dest:` rank of receiver
- `tag:` message tag (put 0 if you don't need)
- `comm:` communicator

Send / Recv with buffer size

- use array slicing

```
if rank == 0:
    data = np.arange(10, dtype=np.float64)
    for i in range(1, size):
        # Using [data, size, type] to send a buffer of data
        comm.Send([data[2:5], 2, MPI.DOUBLE], dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(10, dtype=np.float64)
    # Using [data, size, type] to receive a buffer of data
    comm.Recv([data[2:5], 2, MPI.DOUBLE], source=0, tag=rank)

    print(f"Rank {rank} received data from rank 0: {data}")

print(f"Rank {rank} has data: {data}")
```

Send / Recv with 2D array

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

```
if rank == 0:
    data = np.arange(16, dtype=np.float64).reshape(4, 4)
    for i in range(1, size):
        comm.Send(data, dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(16, dtype=np.float64).reshape(4, 4)
    status = MPI.Status()
    comm.Recv(data, source=0, tag=rank)

    print(f"Rank {rank} received data from rank 0: {data}")

print(f"Rank {rank} has data: {data}")
```

Send / Recv with 2D array

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

```python
if rank == 0:
    data = np.arange(16, dtype=np.float64).reshape(4, 4).T.copy()
    for i in range(1, size):
        comm.Send(data, dest=i, tag=i)
        print(f"Rank 0 sent data to rank {i}")

else:
    data = np.empty(16, dtype=np.float64).reshape(4, 4)
    status = MPI.Status()
    comm.Recv(data, source=0, tag=rank)

    print(f"Rank {rank} received data from rank 0: {data}")

print(f"Rank {rank} has data: {data}")
```

Non-blocking send/recv

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

```python
# Example non blocking Isend/Irecv
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    data = np.arange(10, dtype=np.float64)
    reqs = []
    for i in range(1, size):
        reqs.append(comm.Isend(data, dest=i, tag=i))
    for req in reqs:
        req.Wait()
        print('process 0 sent:', data)
else:
    data = np.empty(10, dtype=np.float64)
    req = comm.Irecv(data, source=0, tag=rank)
    req.Wait()
    print('process', rank, 'received:', 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

```
# Example of Bcast

import numpy as np
from mpi4py import MPI

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

if rank == 0:
    data = np.arange(10, dtype='i')
else:
    data = np.empty(10, dtype='i')

comm.Bcast(data, root=0)

print("Rank: ", 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.DOUBLE], recvbuf, root=root)

- Note
  - [sendbuf, count, displ, MPI.DOUBLE] defines the sending buffer
  - recvbuf needs to be created prior to the communication
  - size of recvbuf needs to be known before hand
  - root can be non-zero (source of Scatterv)

Scatterv

```python
# Example of Scatterv and Gatherv
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    sendbuf = np.arange(100, dtype='i')
    # Determine the counts and displacements
    # for the scatter operation
    counts = np.full(size, 100//size, dtype='i')
    counts[:100%size] += 1
    displs = np.insert(np.cumsum(counts), 0, 0)[:-1]

    print('counts:', counts)
    print('displs:', displs)

else:
    sendbuf = None
    counts = np.empty(size, dtype='i')
    displs = None

# Scatter the data
# First we broadcast the counts to all processes
comm.Bcast(counts, root=0)

# Otherwise the recvbuf will have the same size as the sendbuf
recvbuf = np.empty(counts[rank], dtype='i')

comm.Scatterv([sendbuf, counts, displs, MPI.INT], recvbuf, root=0)

print('Rank:', rank, 'recvbuf:', recvbuf)
```

Scatterv

- output

```
counts: [25 25 25 25]
displs: [ 0 25 50 75]
Rank: 0 recvbuf: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
Rank: 1 recvbuf: [25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
Rank: 2 recvbuf: [50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74]
Rank: 3 recvbuf: [75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99]
```

Gatherv

- vector variant of Gather

- Syntax
  - comm.Gatherv(sendbuf, [recvbuf, count, displ, MPI.DOUBLE], root=root)
- Note
  - [recvbuf, count, displ, MPI.DOUBLE] defines the receiving buffer
  - recvbuf needs to be created prior to the communication
  - size of recvbuf needs to be known before hand
  - root can be non-zero (target of Gatherv)

Gatherv

```python
# Example of Scatterv and Gatherv
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    sendbuf = np.arange(100, dtype='i')
    counts = np.full(size, 100 // size, dtype='i')
    counts[:100 % size] += 1
    displs = np.zeros(size, dtype='i')
    displs[1:] = np.cumsum(counts[:-1])

    print('counts:', counts)
    print('displs:', displs)
else:
    sendbuf = None
    counts = np.empty(size, dtype='i')
    displs = np.empty(size, dtype='i')
```

```python
# Broadcast counts and displacements to all ranks
comm.Bcast(counts, root=0)
comm.Bcast(displs, root=0)

# Allocate recvbuf based on counts for the current rank
recvbuf = np.empty(counts[rank], dtype='i')

# Scatter the data
comm.Scatterv([sendbuf, counts, displs, MPI.INT], recvbuf, root=0)
print(f'Rank {rank} received data: {recvbuf}')

# Gather the data back
sendbuf2 = recvbuf
if rank == 0:
    recvbuf2 = np.empty(100, dtype='i')  # Full array to gather data
else:
    recvbuf2 = None

comm.Gatherv(sendbuf2, [recvbuf2, counts, displs, MPI.INT], root=0)

if rank == 0:
    print('Gathered data:', recvbuf2)
```

Gatherv

- output

```
Gathered data: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
92 93 94 95
 96 97 98 99]
```

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

```python
# Example of Scatterv and Reduce
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    sendbuf = np.arange(100, dtype='i')
    # Determine counts and displacements
    counts = np.full(size, 100 // size, dtype='i')
    counts[:100 % size] += 1
    displs = np.zeros(size, dtype='i')
    displs[1:] = np.cumsum(counts[:-1])

    print('counts:', counts)
    print('displs:', displs)
else:
    sendbuf = None
    counts = np.empty(size, dtype='i')
    displs = np.empty(size, dtype='i')
# Broadcast counts and displacements to all ranks
comm.Bcast(counts, root=0)
comm.Bcast(displs, root=0)
```

```python
# Allocate recvbuf based on counts for the current rank
recvbuf = np.empty(counts[rank], dtype='i')

# Scatter the data
comm.Scatterv([sendbuf, counts, displs, MPI.INT], recvbuf, root=0)
print(f'Rank {rank} received data: {recvbuf}')

# Compute the partial sum on each rank
partial_sum = np.sum(recvbuf)  # This is a scalar value

# Use Reduce to compute the total sum across all ranks
total_sum = comm.reduce(partial_sum, op=MPI.SUM, root=0)

if rank == 0:
    print('Total sum of all elements:', total_sum)
```

Reduce

- output

```
Rank 0 received data: [ 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24]
Rank 1 received data: [25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49]
Rank 2 received data: [50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73
 74]
Rank 3 received data: [75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98
 99]
Total sum of all elements: 4950
```

Reduce

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

Reduce

```python
# Simple example of Reduce
from mpi4py import MPI
import numpy as np

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

data = np.array([1, 2, 3, 4], dtype=np.float64) * (rank + 1)
result = np.empty(4, dtype=np.float64)

# Print partial data
print(f"Rank {rank} has data: {data}")

comm.Reduce(data, result, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Rank {rank} has data after reduce: {result}")
```

Reduce

```
Rank 1 has data: [2. 4. 6. 8.]
Rank 2 has data: [ 3.  6.  9. 12.]
Rank 3 has data: [ 4.  8. 12. 16.]
Rank 0 has data: [1. 2. 3. 4.]
Rank 0 has data after reduce: [10. 20. 30. 40.]
```

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