Python in High Performance Programming

March 21 – 22, 2022

CSC – IT Center for Science Ltd., Espoo

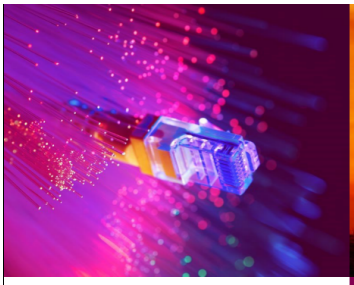
Jussi Enkovaara
Martti Louhivuori

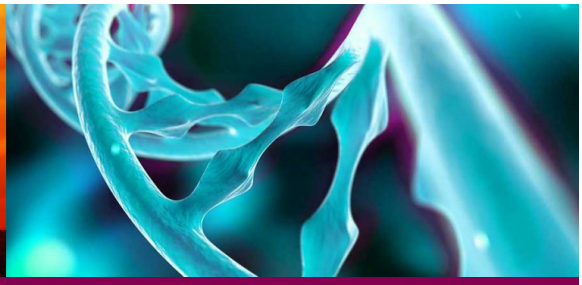
CSC
Training for Brilliant Minds

PRACE

Course schedule

	Mon	Tue
9.00-9.45	NumPy – fast array interface to Python	Using MPI with Python
9.45-10.00	Break	
10.00-10.45	Advanced NumPy topics	Point-to-point communication
10.45-11.00	Break	
11.00-11.45	Performance analysis	Collective communication
12.00-13.00	Lunch break	
13.00-16.00	Hands-on exercises	Hands-on exercises
16.00-16.15	Wrap-up	Wrap-up





Python and High-Performance Computing

CSC – Finnish expertise in ICT for research, education and public administration

Efficiency

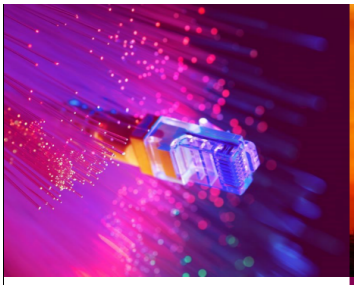
- Python is an interpreted language
  - no pre-compiled binaries, all code is translated on-the-fly to machine instructions
  - byte-code as a middle step which may be stored (.pyc)
- All objects are dynamic in Python
  - nothing is fixed == optimisation nightmare
  - lot of overhead from metadata
- Flexibility is good, but comes with a cost!

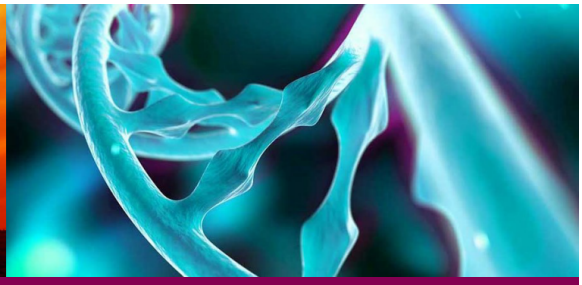
Improving Python performance

- Array based computations with NumPy
- Using extended Cython programming language
- Embed compiled code in a Python program
  - C/C++, Fortran
- Utilize parallel processing

Parallelisation strategies for Python

- Global Interpreter Lock (GIL)
  - CPython's memory management is not thread-safe
  - no threads possible, except for I/O etc.
  - affects overall performance of threading
- Process-based "threading" with multiprocessing
  - fork independent processes that have a limited way to communicate
- **Message-passing** is the Way to Go to achieve true parallelism in Python





Numpy basics


CSC – Finnish expertise in ICT for research, education and public administration

Numpy – fast array interface

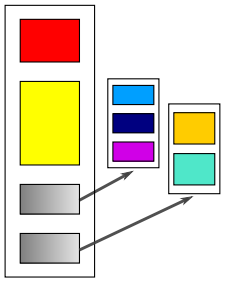
- Standard Python is not well suitable for numerical computations
  - lists are very flexible but also slow to process in numerical computations
- Numpy adds a new **array** data type
  - static, multidimensional
  - fast processing of arrays
  - tools for linear algebra, random numbers, *etc.*

Numpy arrays

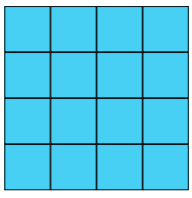
- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed

Python list vs. NumPy array

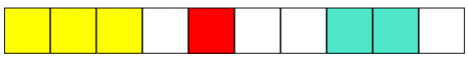
Python list



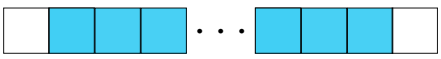
NumPy array



Memory layout



Memory layout



Creating numpy arrays

From a list:

```
>>> import numpy
>>> a = numpy.array((1, 2, 3, 4), float)
>>> a
array([ 1., 2., 3., 4.])

>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = numpy.array(list1, complex)
>>> mat
array([[ 1.+0.j, 2.+0.j, 3.+0.j],
       [ 4.+0.j, 5.+0.j, 6.+0.j]])

>>> mat.shape
(2, 3)

>>> mat.size
6
```


Helper functions for creating arrays: 1

- arange and `linspace` can generate ranges of numbers:

```
>>> a = numpy.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> b = numpy.arange(0.1, 0.2, 0.02)
>>> b
array([0.1 , 0.12, 0.14, 0.16, 0.18])

>>> c = numpy.linspace(-4.5, 4.5, 5)
>>> c
array([-4.5 , -2.25, 0. , 2.25, 4.5 ])
```

Helper functions for creating arrays: 2

- array with given shape initialized to `zeros`, `ones` or arbitrary value (`full`):

```
>>> a = numpy.zeros((4, 6), float)
>>> a.shape
(4, 6)

>>> b = numpy.ones((2, 4))
>>> b
array([[ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.]])

>>> c = numpy.full((2, 3), 4.2)
>>> c
array([[4.2, 4.2, 4.2],
       [4.2, 4.2, 4.2]])
```

- Empty array (no values assigned) with `empty`

Helper functions for creating arrays: 3

- Similar arrays as an existing one with `zeros_like`, `ones_like`, `full_like` and `empty_like`:

```
>>> a = numpy.zeros((4, 6), float)
>>> b = numpy.empty_like(a)
>>> c = numpy.ones_like(a)
>>> d = numpy.full_like(a, 9.1)
```

Non-numeric data

- NumPy supports also storing non-numerical data e.g. strings (largest element determines the item size)

```
>>> a = numpy.array(['foo', 'foo-bar'])
>>> a
array(['foo', 'foo-bar'], dtype='|U7')
```

- Character arrays can, however, be sometimes useful

```
>>> dna = 'AAAGTCTGAC'
>>> a = numpy.array(dna, dtype='c')
>>> a
array([b'A', b'A', b'A', b'G', b'T', b'C', b'T', b'G', b'A', b'C'],
      dtype='|S1')
```


Accessing arrays

- Simple indexing:

```
>>> mat = numpy.array([[1, 2, 3], [4, 5, 6]])
>>> mat[0,2]
3

>>> mat[1,-2]
5
```

- Slicing:

```
>>> a = numpy.arange(10)
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])

>>> a[:-1]
array([0, 1, 2, 3, 4, 5, 6, 7, 8])

>>> a[1:3] = -1
>>> a
array([0, -1, -1, 3, 4, 5, 6, 7, 8, 9])

>>> a[1:7:2]
array([1, 3, 5])
```

Slicing of arrays in multiple dimensions

- Multidimensional arrays can be sliced along multiple dimensions
- Values can be assigned to only part of the array

```
>>> a = numpy.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[ 0., 0., 0., 0.],
   [ 0., 2., 2., 0.],
   [ 0., 2., 2., 0.],
   [ 0., 0., 0., 0.]])
```


Views and copies of arrays

- Simple assignment creates references to arrays
- Slicing creates "views" to the arrays
- Use copy() for real copying of arrays

```
a = numpy.arange(10)
b = a               # reference, changing values in b changes a
b = a.copy()        # true copy

c = a[1:4]          # view, changing c changes elements [1:4] of a
c = a[1:4].copy()   # true copy of subarray
```

Array manipulation

- reshape : change the shape of array

```
>>> mat = numpy.array([[1, 2, 3], [4, 5, 6]])
>>> mat
array([[1, 2, 3],
       [4, 5, 6]])

>>> mat.reshape(3,2)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

- ravel : flatten array to 1-d

```
>>> mat.ravel()
array([1, 2, 3, 4, 5, 6])
```


Array manipulation

- concatenate : join arrays together

```
>>> mat1 = numpy.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = numpy.array([[7, 8, 9], [10, 11, 12]])
>>> numpy.concatenate((mat1, mat2))
array([[ 1, 2, 3],
       [ 4, 5, 6],
       [ 7, 8, 9],
       [10, 11, 12]])

>>> numpy.concatenate((mat1, mat2), axis=1)
array([[ 1, 2, 3,  7,  8,  9],
       [ 4, 5, 6, 10, 11, 12]])
```

- split : split array to N pieces

```
>>> numpy.split(mat1, 3, axis=1)
[array([[1], [4]]), array([[2], [5]]), array([[3], [6]])]
```

Array operations

- Most operations for numpy arrays are done element-wise (+, -, *, /, **)

```
>>> a = numpy.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2., 4., 6.])

>>> a + b
array([ 3., 4., 5.])

>>> a * a
array([ 1., 4., 9.])
```


Array operations

- Numpy has special functions which can work with array arguments (sin, cos, exp, sqrt, log, ...)

```
>>> import numpy, math
>>> a = numpy.linspace(-math.pi, math.pi, 8)
>>> a
array([-3.14159265, -2.24399475, -1.34639685, -0.44879895,
        0.44879895,  1.34639685,  2.24399475,  3.14159265])

>>> numpy.sin(a)
array([ -1.22464680e-16, -7.81831482e-01, -9.74927912e-01,
        -4.33883739e-01,  4.33883739e-01,  9.74927912e-01,
         7.81831482e-01,  1.22464680e-16])

>>> math.sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: only length-1 arrays can be converted to Python scalars
```

Summary

- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays

Numpy tools

- NumPy contains ready to use tools for many common tasks
- I/O
- Random numbers
- Polynomials
- Linear algebra

I/O with Numpy

- Numpy provides functions for reading data from file and for writing data into the files
- Simple text files
  - `numpy.loadtxt`
  - `numpy.savetxt`
  - Data in regular column layout
  - Can deal with comments and different column delimiters

Random numbers

- The module `numpy.random` provides several functions for constructing random arrays
  - `random`: uniform random numbers
  - `normal`: normal distribution
  - `choice`: random sample from given array
  - ...

```
>>> import numpy.random as rnd
>>> rnd.random((2,2))
array([[ 0.02909142, 0.90848 ],
       [ 0.9471314 , 0.31424393]])

>>> rnd.choice(numpy.arange(4), 10)
array([0, 1, 1, 2, 1, 1, 2, 0, 2, 3])
```

Polynomials

- Polynomial is defined by an array of coefficients p

  $p(x, N) = p[0]x^{N-1} + p[1]x^{N-2} + \ldots + p[N-1]$
- For example:
  - Least square fitting: `numpy.polyfit`
  - Evaluating polynomials: `numpy.polyval`
  - Roots of polynomial: `numpy.roots`

```
>>> x = numpy.linspace(-4, 4, 7)
>>> y = x**2 + rnd.random(x.shape)
>>>
>>> p = numpy.polyfit(x, y, 2)
>>> p
array([ 0.96869003, -0.01157275, 0.69352514])
```

Processing math: 100%

Linear algebra

- Numpy can calculate matrix and vector products efficiently: `dot`, `vdot`, ...
- Eigenproblems: `linalg.eig`, `linalg.eigvals`, ...
- Linear systems and matrix inversion: `linalg.solve`, `linalg.inv`

```
>>> A = numpy.array(((2, 1), (1, 3)))
>>> B = numpy.array(((-2, 4.2), (4.2, 6)))
>>> C = numpy.dot(A, B)
>>>
>>> b = numpy.array((1, 2))
>>> numpy.linalg.solve(C, b) # solve C x = b
array([ 0.04453441, 0.06882591])
```

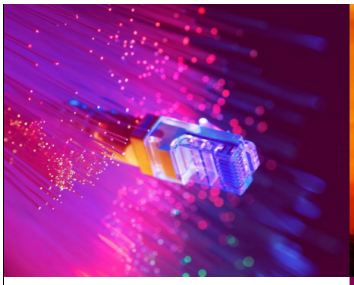
Processing math: 100%

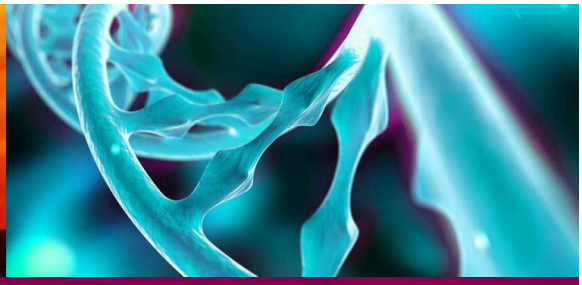
Linear algebra

- Normally, NumPy utilises high performance libraries in linear algebra operations
- Example: matrix multiplication C = A * B matrix dimension 1000
  - pure python: 522.30 s
  - naive C: 1.50 s
  - numpy.dot: 0.04 s
  - library call from C: 0.04 s

Summary

- NumPy contains many useful tools for numerical computing
- Scipy includes even wider set of utilities for scientific computing
- Further useful packages
  - Visualization: matplotlib, Mayavi
  - Data analysis: pandas



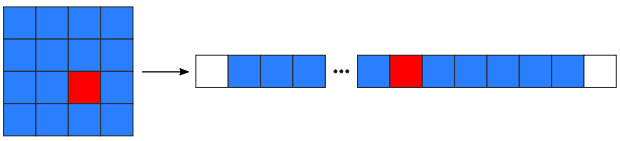


Advanced concepts in NumPy


CSC – Finnish expertise in ICT for research, education and public administration

Anatomy of NumPy array

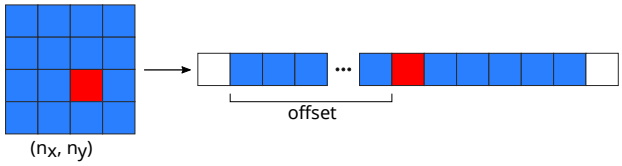
- **ndarray** type is made of
  - one dimensional contiguous block of memory (raw data)
  - indexing scheme: how to locate an element
  - data type descriptor: how to interpret an element



NumPy indexing

- There are many possible ways of arranging items of N-dimensional array in a 1-dimensional block
- NumPy uses **striding** where N-dimensional index $(n_0, n_1, \ldots, n_{N-1})$ corresponds to offset from the beginning of 1-dimensional block

$$\text{offset} = \sum_{k=0}^{N-1} s_k n_k, \; s_k \text{ is stride in dimension } k$$



ndarray attributes

`a = numpy.array(...)`

`a.flags`
various information about memory layout

`a.strides`
bytes to step in each dimension when traversing

`a.itemsize`
size of one array element in bytes

`a.data`
Python buffer object pointing to start of arrays data

`a.__array_interface__`
Python internal interface

Advanced indexing

- Numpy arrays can be indexed also with other arrays (integer or boolean)

```
>>> x = numpy.arange(10,1,-1)
>>> x
array([10, 9, 8, 7, 6, 5, 4, 3, 2])

>>> x[numpy.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

- Boolean "mask" arrays

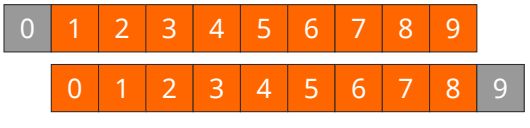
```
>>> m = x > 7
>>> m
array([ True, True, True, False, False, ...

>>> x[m]
array([10, 9, 8])
```

- Advanced indexing creates copies of arrays

Vectorized operations

- `for` loops in Python are slow
- Use "vectorized" operations when possible
- Example: difference
  - for loop is ~80 times slower!

```
# brute force using a for loop
arr = numpy.arange(1000)
dif = numpy.zeros(999, int)
for i in range(1, len(arr)):
    dif[i-1] = arr[i] - arr[i-1]

# vectorized operation
arr = numpy.arange(1000)
dif = arr[1:] - arr[:-1]
```



Broadcasting

- If array shapes are different, the smaller array may be broadcasted into a larger shape

```
>>> from numpy import array
>>> a = array([[1,2],[3,4],[5,6]], float)
>>> a
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.]])

>>> b = array([[7,11]], float)
>>> b
array([[ 7., 11.]])

>>> a * b
array([[ 7., 22.],
       [ 21., 44.],
       [ 35., 66.]])
```


Broadcasting

- Example: calculate distances from a given point

```
# array containing 3d coordinates for 100 points
points = numpy.random.random((100, 3))
origin = numpy.array((1.0, 2.2, -2.2))
dists = (points - origin)**2
dists = numpy.sqrt(numpy.sum(dists, axis=1))

# find the most distant point
i = numpy.argmax(dists)
print(points[i])
```

Temporary arrays

- In complex expressions, NumPy stores intermediate values in temporary arrays
- Memory consumption can be higher than expected

```
a = numpy.random.random((1024, 1024, 50))
b = numpy.random.random((1024, 1024, 50))

# two temporary arrays will be created
c = 2.0 * a - 4.5 * b

# three temporary arrays will be created due to unnecessary parenthesis
c = (2.0 * a - 4.5 * b) + 1.1 * (numpy.sin(a) + numpy.cos(b))
```

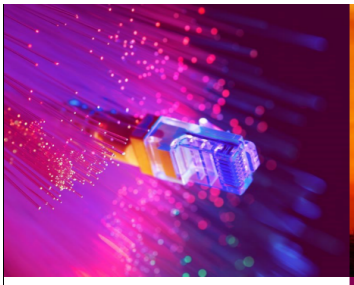

Temporary arrays

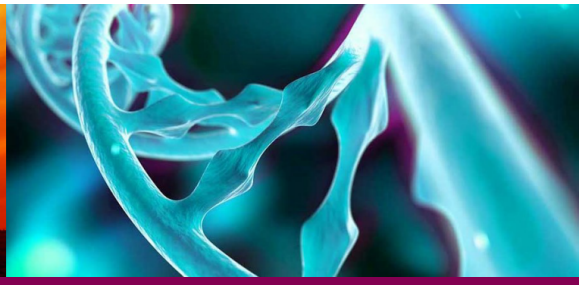
- Broadcasting approaches can lead also to hidden temporary arrays
- Example: pairwise distance of **M** points in 3 dimensions
  - Input data is M x 3 array
  - Output is M x M array containing the distance between points i and j
  - There is a temporary 1000 x 1000 x 3 array

```
X = numpy.random.random((1000, 3))
D = numpy.sqrt(((X[:, numpy.newaxis, :] - X) ** 2).sum(axis=-1))
```

Summary

- NumPy arrays consist internally of contiguous data block and strides to describe dimensions and shape
- Vectorization improves performance
- Arrays can be broadcasted into same shapes
- Expression evaluation can lead into temporary arrays





Performance analysis

CSC – Finnish expertise in ICT for research, education and public administration

Performance measurement

Measuring application performance

- Correctness is the most import factor in any application
  - Premature optimization is the root of all evil!
- Before starting to optimize application, one should measure where time is spent
  - Typically 90 % of time is spent in 10 % of application
- Mind the algorithm!
  - Recursive calculation of Fibonacci numbers

	Speedup
Pure Python	1
Pure C	126
Pure Python (better algorithm)	24e6

Measuring application performance

- Applications own timers
- **timeit** module
- **cProfile** module
- Full fedged profiling tools: TAU, Intel Vtune, Python Tools for Visual Studio …

Measuring application performance

- Python **time** module can be used for measuring time spent in specific part of the program
  - `time.perf_counter()` : include time spent in other processes
  - `time.process_time()` : only time in current process

```
import time

t0 = time.process_time()
for n in range(niter):
    heavy_calculation()
t1 = time.process_time()

print('Time spent in heavy calculation', t1-t0)
```


timeit module

- Easy timing of small bits of Python code
- Tries to avoid common pitfalls in measuring execution times
- Command line interface and Python interface
- `%timeit` magic in IPython

```
In [1]: from mymodule import func
In [2]: %timeit func()

10 loops, best of 3: 433 msec per loop
```

```
$ python -m timeit -s "from mymodule import func" "func()"

10 loops, best of 3: 433 msec per loop
```

cProfile

- Execution profile of Python program
  - Time spent in different parts of the program
  - Call graphs
- Python API:
- Profiling whole program from command line

```
import cProfile
...

# profile statement and save results to a file func.prof
cProfile.run('func()', 'func.prof')
```

```
$ python -m cProfile -o myprof.prof myprogram.py
```


Investigating profile with pstats

- Printing execution time of selected functions
- Sorting by function name, time, cumulative time, ...
- Python module interface and interactive browser

```
In [1]: from pstats import Stats
In [2]: p = Stats('myprof.prof')
In [3]: p.strip_dirs()
In [4]: p.sort_stats('time')
In [5]: p.print_stats(5)

Mon Oct 12 10:11:00 2016 my.prof
...
```

```
$ python -m pstats myprof.prof

Welcome to the profile statistics
% strip
% sort time
% stats 5

Mon Oct 12 10:11:00 2016 my.prof
...
```

Summary

- Python has various built-in tools for measuring application performance
- **time** module
- **timeit** module
- **cProfile** and **pstats** modules

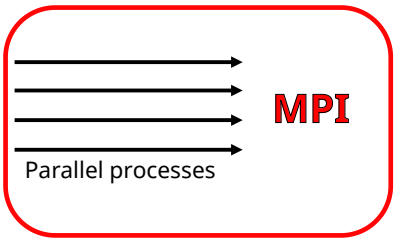
MPI for Python

Message Passing Interface

Message passing interface

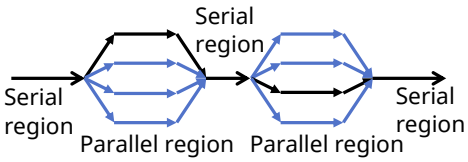
- MPI is an application programming interface (API) for communication between separate processes
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from PC's to supercomputers
  - the most widely used approach for distributed parallel computing
- MPI is flexible and comprehensive
  - large (over 300 procedures)
  - concise (often only 6 procedures are needed)
- MPI standard defines C and Fortran interfaces
  - MPI for Python (mpi4py) provides an unofficial Python interface

Processes and threads



Process

- Independent execution units
- Have their own state information and *own memory* address space



Thread

- A single process may contain multiple threads
- Have their own state information, but *share* the *same memory* address space

Execution model

- MPI program is launched as a set of *independent*, *identical processes*
  - execute the same program code and instructions
  - can reside in different nodes (or even in different computers)
- The way to launch a MPI program depends on the system
  - mpiexec, mpirun, srun, aprun, ...
  - mpiexec/mpirun in training class
  - srun on puhti.csc.fi

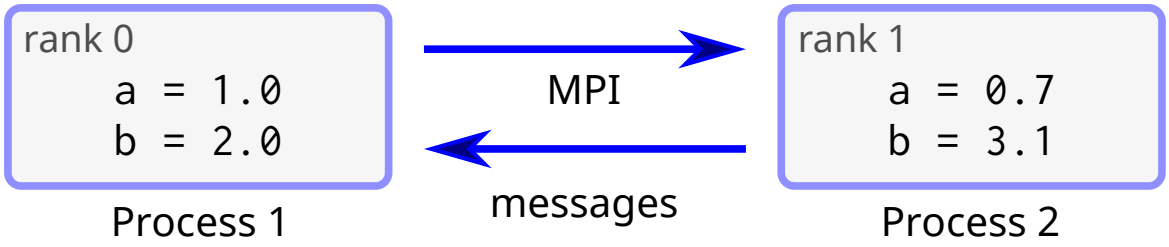
MPI rank

- Rank: ID number given to a process
  - it is possible to query for rank
  - processes can perform different tasks based on their rank

```
if (rank == 0):
    # do something
elif (rank == 1):
    # do something else
else:
    # all other processes do something different
```

Data model

- Each MPI process has its own *separate* memory space, i.e. all variables and data structures are *local* to the process
- Processes can exchange data by sending and receiving messages



MPI communicator

- Communicator: a group containing all the processes that will participate in communication
  - in mpi4py most MPI calls are implemented as methods of a communicator object
  - `MPI_COMM_WORLD` contains all processes (`MPI.COMM_WORLD` in mpi4py)
  - user can define custom communicators

Routines in MPI for Python

- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes
- Synchronization between processes
- Communicator creation and manipulation
- Advanced features (e.g. user defined datatypes, one-sided communication and parallel I/O)

Getting started

- Basic methods of communicator object
  - `Get_size()` Number of processes in communicator
  - `Get_rank()` rank of this process

```
from mpi4py import MPI

comm = MPI.COMM_WORLD # communicator object containing all processes

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

print("I am rank %d in group of %d processes" % (rank, size))
```

Running an example program

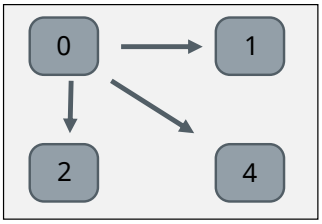
```
$ mpiexec -n 4 python3 hello.py

I am rank 2 in group of 4 processes
I am rank 0 in group of 4 processes
I am rank 3 in group of 4 processes
I am rank 1 in group of 4 processes
```

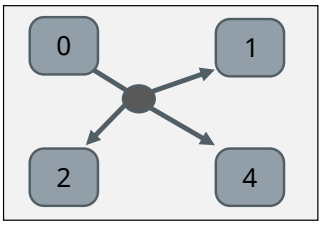
```
from mpi4py import MPI

comm = MPI.COMM_WORLD # communicator object containing all processes

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

print("I am rank %d in group of %d processes" % (rank, size))
```


Point-to-Point Communication

MPI communication

- Data is local to the MPI processes
  - They need to *communicate* to coordinate work
- Point-to-point communication
  - Messages are sent between two processes
- Collective communication
  - Involving a number of processes at the same time





MPI point-to-point operations

- One process *sends* a message to another process that *receives* it
- Sends and receives in a program should match - one receive per send
- Each message contains
  - The actual *data* that is to be sent
  - The *datatype* of each element of data
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process
- With **mpi4py** it is often enough to specify only *data* and *source* and *destination*

Sending and receiving data

- Sending and receiving a dictionary

```
from mpi4py import MPI

comm = MPI.COMM_WORLD # communicator object containing all processes
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source=0)
```


Sending and receiving data

- Arbitrary Python objects can be communicated with the send and receive methods of a communicator

`.send(data, dest)`

data

Python object to send

dest

destination rank

`.recv(source)`

source

source rank

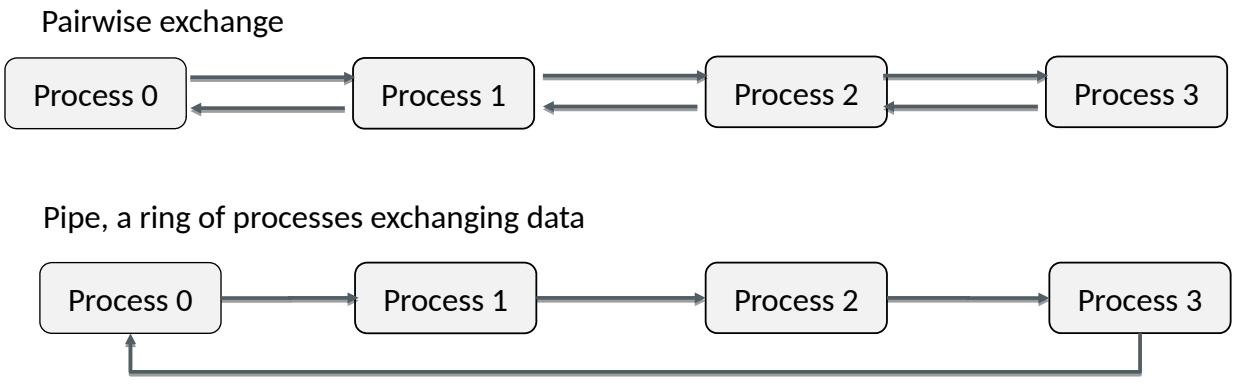
note: data is provided as return value

- Destination and source ranks have to match!

Blocking routines & deadlocks

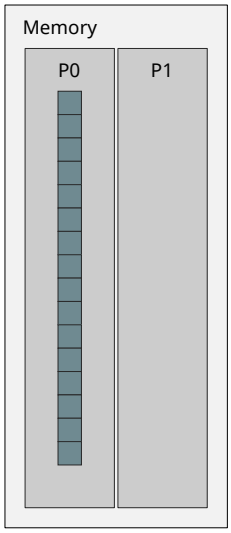
- `send()` and `recv()` are *blocking* routines
  - the functions exit only once it is safe to use the data (memory) involved in the communication
- Completion depends on other processes => risk for *deadlocks*
  - for example, if all processes call `recv()` there is no-one left to call a corresponding `send()` and the program is *stuck forever*

Typical point-to-point communication patterns



- Incorrect ordering of sends and receives may result in a deadlock

Case study: parallel sum



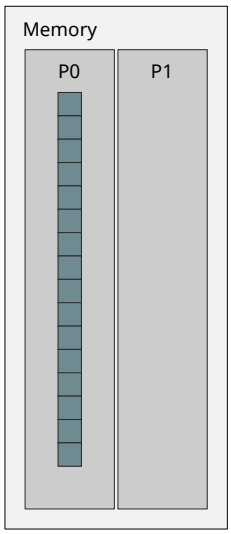
Initial state

An array A containing floating point numbers read from a a file by the first MPI task (rank 0).

Goal

Calculate the total sum of all elements in array A in parallel.

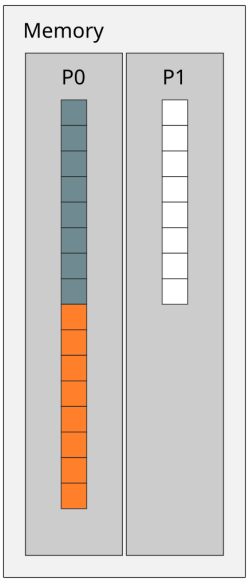
Case study: parallel sum



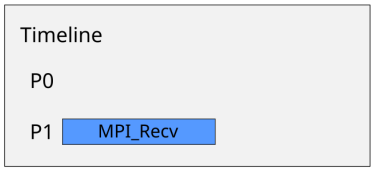
Parallel algorithm

1. Scatter the data
   1.1. receive operation for scatter
   1.2. send operation for scatter
2. Compute partial sums in parallel
3. Gather the partial sums
   3.1. receive operation for gather
   3.2. send operation for gather
4. Compute the total sum

Step 1.1: Receive operation for scatter

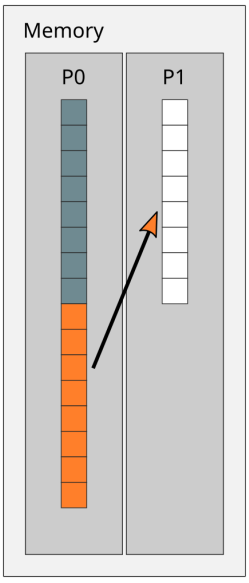


Step 1: Scatter the data
1.1: receive operation for scatter

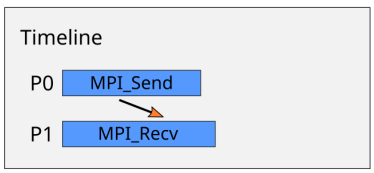


P1 posts a receive to receive half of the array from P0

Step 1.2: Send operation for scatter

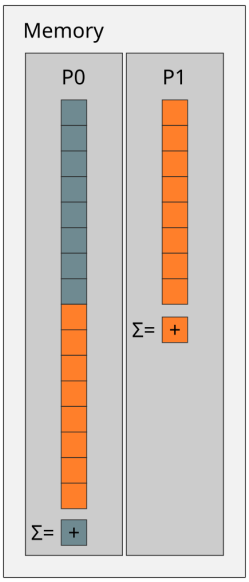


Step 1: Scatter the data
1.1: receive operation for scatter
1.2: send operation for scatter

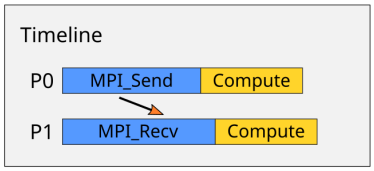


P0 posts a send to send the lower part of the array to P1

Step 2: Compute partial sums in parallel

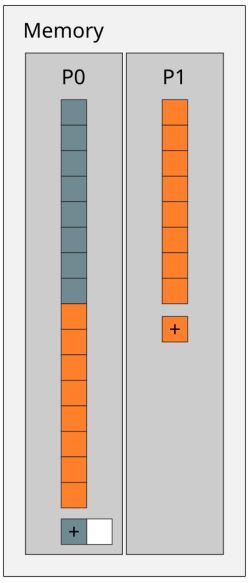


Step 1: Scatter the data
1.1: receive operation for scatter
1.2: send operation for scatter
Step 2: Compute partial sums in parallel

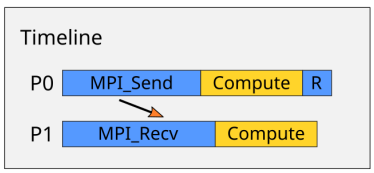


P0 and P1 compute their partial sums in parallel and store them locally

Step 3.1: Receive operation for gather

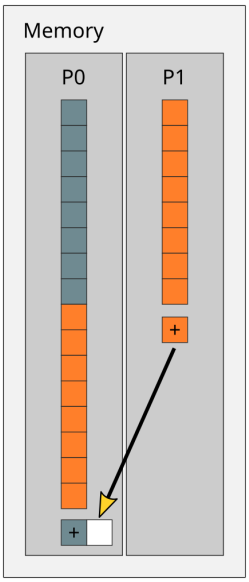


Step 1: Scatter the data
1.1: receive operation for scatter
1.2: send operation for scatter
Step 2: Compute partial sums in parallel
Step 3: Gather the partial sums
3.1: receive operation for gather

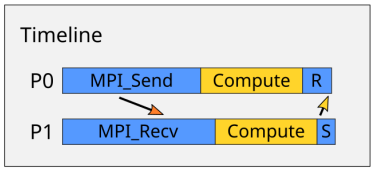


P0 posts a receive for the partial sum of P1

Step 3.2: Send operation for gather

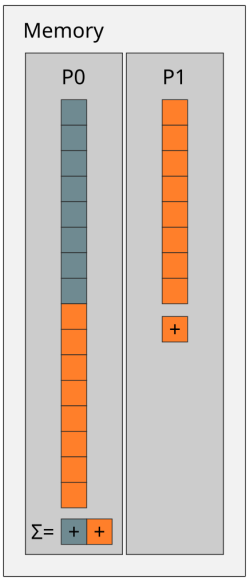


Step 1: Scatter the data
1.1: receive operation for scatter
1.2: send operation for scatter
Step 2: Compute partial sums in parallel
Step 3: Gather the partial sums
3.1: receive operation for gather
3.2: send operation for gather

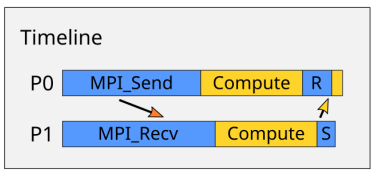


P1 posts a send for the partial sum

Step 4: Compute the total sum



Step 1: Scatter the data
1.1: receive operation for scatter
1.2: send operation for scatter
Step 2: Compute partial sums in parallel
Step 3: Gather the partial sums
3.1: receive operation for gather
3.2: send operation for gather
Step 4: Compute the total sum



P0 sums up the partial sums

Communicating NumPy arrays

- Arbitrary Python objects are converted to byte streams (pickled) when sending and back to Python objects (unpickled) when receiving
  - these conversions may be a serious overhead to communication
- Contiguous memory buffers (such as NumPy arrays) can be communicated with very little overhead using upper case methods:
  - `Send(data, dest)`
  - `Recv(data, source)`
  - note the difference in receiving: the data array has to exist at the time of call

Send/receive a NumPy array

- Note the difference between upper/lower case!
  - send/recv: general Python objects, slow
  - Send/Recv: continuous arrays, fast

```
from mpi4py import MPI
import numpy

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

data = numpy.empty(100, dtype=float)
if rank == 0:
    data[:] = numpy.arange(100, dtype=float)
    comm.Send(data, dest=1)
elif rank == 1:
    comm.Recv(data, source=0)
```

Combined send and receive

- Send one message and receive another with a single command
  - reduces risk for deadlocks
- Destination and source ranks can be same or different
  - `MPI.PROC_NULL` can be used for *no destination/source*

```
data = numpy.arange(10, dtype=float) * (rank + 1)
buffer = numpy.empty(data.shape, dtype=data.dtype)

if rank == 0:
    dest, source = 1, 1
elif rank == 1:
    dest, source = 0, 0

comm.Sendrecv(data, dest=dest, recvbuf=buffer, source=source)
```


MPI datatypes

- MPI has a number of predefined datatypes to represent data
  - e.g. `MPI.INT` for integer and `MPI.DOUBLE` for float
- No need to specify the datatype for Python objects or Numpy arrays
  - objects are serialised as byte streams
  - automatic detection for NumPy arrays
- If needed, one can also define custom datatypes
  - for example to use non-contiguous data buffers

Summary

- Point-to-point communication = messages are sent between two MPI processes
- Point-to-point operations enable any parallel communication pattern (in principle)
- Arbitrary Python objects (that can be pickled!)
  - `send` / `recv`
  - `sendrecv`
- Memory buffers such as Numpy arrays
  - `Send` / `Recv`
  - `Sendrecv`

Non-blocking Communication

Non-blocking communication

- Non-blocking sends and receives
  - `isend` & `irecv`
  - returns immediately and sends/receives in background
  - return value is a Request object
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations

Non-blocking communication

- Have to finalize send/receive operations
  - `wait()`
    - Waits for the communication started with `isend` or `irecv` to finish (blocking)
  - `test()`
    - Tests if the communication has finished (non-blocking)
- You can mix non-blocking and blocking p2p routines
  - e.g., receive `isend` with `recv`

Example: non-blocking send/receive

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

if rank == 0:
    data = arange(size, dtype=float) * (rank + 1)
    req = comm.Isend(data, dest=1)    # start a send
    calculate_something(rank)         # .. do something else ..
    req.wait()                        # wait for send to finish
    # safe to read/write data again

elif rank == 1:
    data = empty(size, float)
    req = comm.Irecv(data, source=0)  # post a receive
    calculate_something(rank)         # .. do something else ..
    req.wait()                        # wait for receive to finish
    # data is now ready for use
```


Multiple non-blocking operations

- Methods `waitall()` and `waitany()` may come handy when dealing with multiple non-blocking operations (available in the `MPI.Request` class)
  - `Request.waitall(requests)`
    - wait for all initiated requests to complete
  - `Request.waitany(requests)`
    - wait for any initiated request to complete
- For example, assuming `requests` is a list of request objects, one can wait for all of them to be finished with:

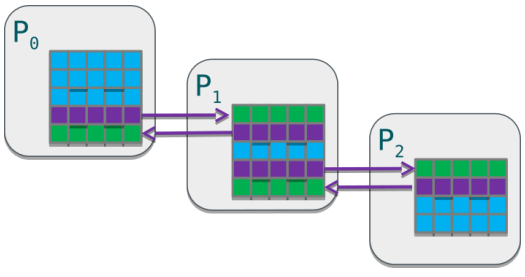
```
MPI.Request.waitall(requests)
```

Example: non-blocking message chain

```
from mpi4py import MPI
import numpy

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

data = numpy.arange(10, dtype=float) * (rank + 1)  # send buffer
buffer = numpy.zeros(10, dtype=float)              # receive buffer

tgt = rank + 1
src = rank - 1
if rank == 0:
    src = MPI.PROC_NULL
if rank == size - 1:
    tgt = MPI.PROC_NULL

req = []
req.append(comm.Isend(data, dest=tgt))
req.append(comm.Irecv(buffer, source=src))

MPI.Request.waitall(req)
```


Overlapping computation and communication

```
request_in = comm.Irecv(ghost_data)
request_out = comm.Isend(border_data)

compute(ghost_independent_data)
request_in.wait()

compute(border_data)
request_out.wait()
```



Summary

- Non-blocking communication is usually the smart way to do point-to-point communication in MPI
- Non-blocking communication realization
  - `isend` / `Isend`
  - `irecv` / `Irecv`
  - `request.wait()`

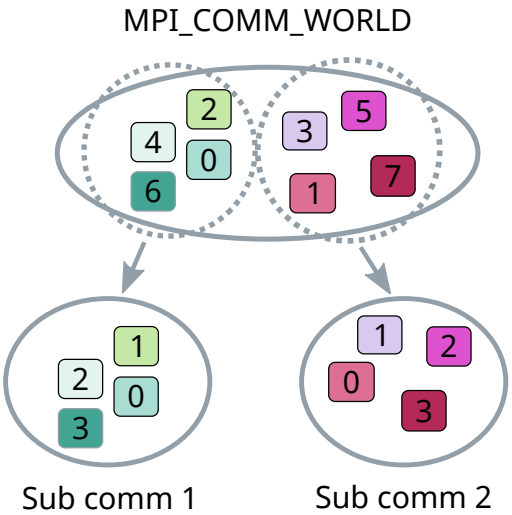
Communicators

Communicators

- The communicator determines the "communication universe"
  - The source and destination of a message is identified by process rank *within* the communicator
- So far: `MPI.COMM_WORLD`
- Processes can be divided into subcommunicators
  - Task level parallelism with process groups performing separate tasks
  - Collective communication within a group of processes
  - Parallel I/O

Communicators

- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - Unique rank in each communicator



User-defined communicators

- By default a single, universal communicator exists to which all processes belong (`MPI.COMM_WORLD`)
- One can create new communicators, e.g. by splitting this into sub-groups

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

color = rank % 4

local_comm = comm.Split(color)
local_rank = local_comm.Get_rank()

print("Global rank: %d Local rank: %d" % (rank, local_rank))
```


Collective Communication

Collective communication

- Collective communication transmits data among all processes in a process group (communicator)
  - these routines must be called by all the processes in the group
  - amount of sent and received data must match
- Collective communication includes
  - data movement
  - collective computation
  - synchronization
- Example
  - `comm.barrier()` makes every task hold until all tasks in the communicator `comm` have called it

Collective communication

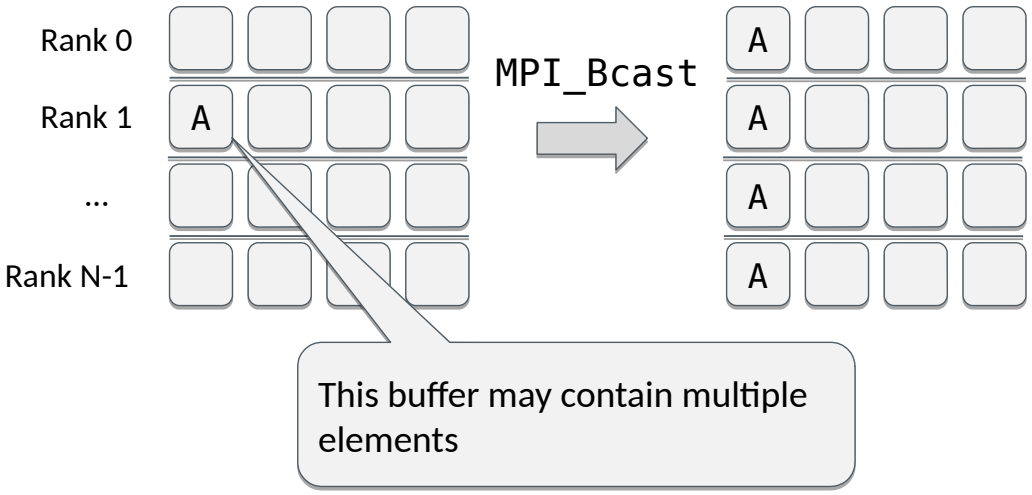
- Collective communication typically outperforms point-to-point communication
- Code becomes more compact (and efficient!) and easier to maintain:
  - For example, communicating a Numpy array of 1M elements from task 0 to all other tasks:

```
if rank == 0:
    for i in range(1, size):
    comm.Send(data, i)
else:
    comm.Recv(data, 0)
```

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

Broadcast

- Send the same data from one process to all the other



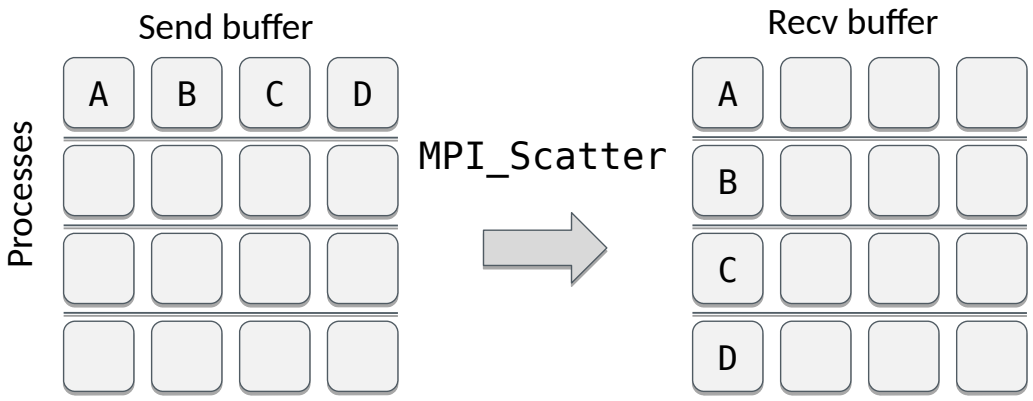
Broadcast

- Broadcast sends same data to all processes

```
from mpi4py import MPI
import numpy

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

if rank == 0:
    py_data = {'key1' : 0.0, 'key2' : 11}  # Python object
    data = np.arange(8) / 10.              # NumPy array
else:
    py_data = None
    data = np.zeros(8)

new_data = comm.bcast(py_data, root=0)

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

Scatter

- Send equal amount of data from one process to others
- Segments A, B, ... may contain multiple elements



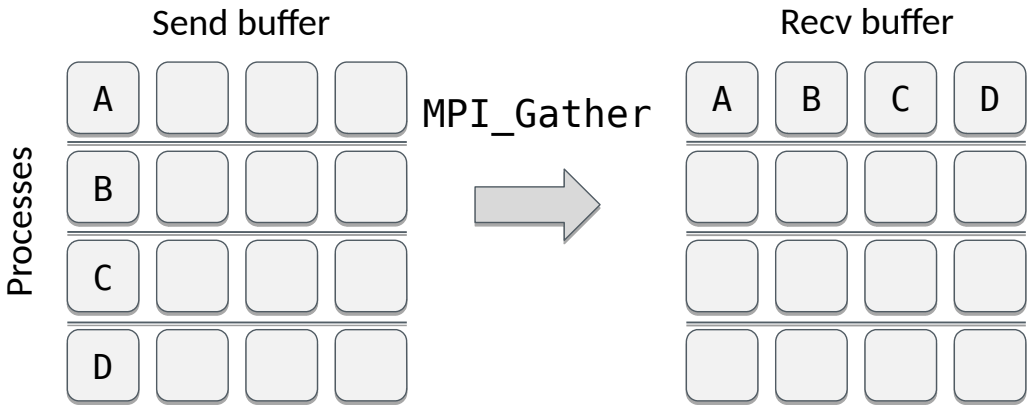
Scatter

- Scatter distributes data to processes

```
from mpi4py import MPI
from numpy import arange, empty

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
if rank == 0:
    py_data = range(size)
    data = arange(size**2, dtype=float)
else:
    py_data = None
    data = None

new_data = comm.scatter(py_data, root=0)  # returns the value

buffer = empty(size, float)         # prepare a receive buffer
comm.Scatter(data, buffer, root=0)  # in-place modification
```

Gather

- Collect data from all the process to one process
- Segments A, B, ... may contain multiple elements



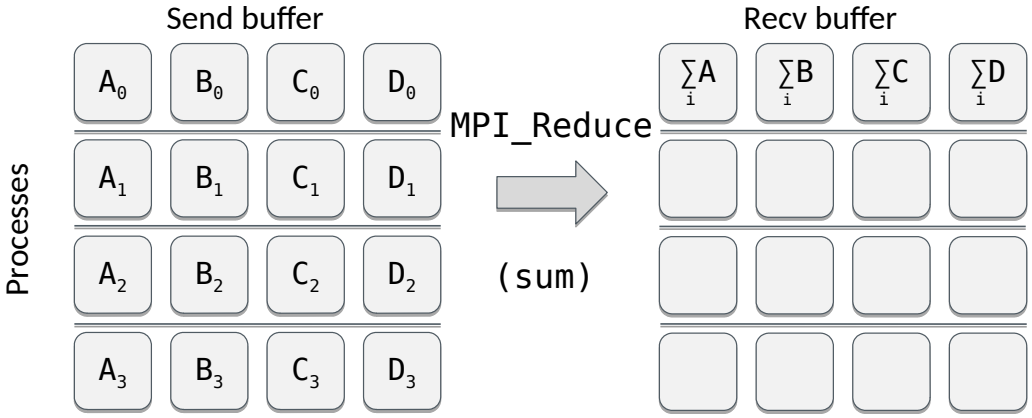
Gather

- Gather pulls data from all processes

```
from mpi4py import MPI
from numpy import arange, zeros

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

data = arange(10, dtype=float) * (rank + 1)
buffer = zeros(size * 10, float)

n = comm.gather(rank, root=0)     # returns the value
comm.Gather(data, buffer, root=0) # in-place modification
```

Reduce

- Applies an operation over set of processes and places result in single process



Reduce

- Reduce gathers data and applies an operation on it

```
from mpi4py import MPI
from numpy import arange, empty

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

data = arange(10 * size, dtype=float) * (rank + 1)
buffer = zeros(size * 10, float)

n = comm.reduce(rank, op=MPI.SUM, root=0)     # returns the value
comm.Reduce(data, buffer, op=MPI.SUM, root=0) # in-place modification
```

Other common collective operations

`Scatterv`
each process receives different amount of data

`Gatherv`
each process sends different amount of data

`Allreduce`
all processes receive the results of reduction

`Alltoall`
each process sends and receives to/from each other

`Alltoallv`
each process sends and receives different amount of data

Non-blocking collectives

- New in MPI 3: no support in mpi4py
- Non-blocking collectives enable the overlapping of communication and computation together with the benefits of collective communication
- Restrictions
  - have to be called in same order by all ranks in a communicator
  - mixing of blocking and non-blocking collectives is not allowed

Common mistakes with collectives

1. Using a collective operation within one branch of an if-else test based on the rank of the process
   - for example: `if rank == 0: comm.bcast(...)`
   - all processes in a communicator must call a collective routine!
2. Assuming that all processes making a collective call would complete at the same time.
3. Using the input buffer also as an output buffer:
   - for example: `comm.Scatter(a, a, MPI.SUM)`
   - always use different memory locations (arrays) for input and output!

Summary

- Collective communications involve all the processes within a communicator
  - all processes must call them
- Collective operations make code more transparent and compact
- Collective routines allow optimizations by MPI library
- MPI-3 contains also non-blocking collectives, but these are currently not supported by MPI for Python

On-line resources

- Documentation for mpi4py is quite limited
  - short on-line manual available at https://mpi4py.readthedocs.io/
- Some good references:
  - "A Python Introduction to Parallel Programming with MPI" *by Jeremy Bejarano* http://materials.jeremybejarano.com/MPIwithPython/
  - "mpi4py examples" *by Jörg Bornschein* https://github.com/jbornschein/mpi4py-examples

Summary

- mpi4py provides Python interface to MPI
- MPI calls via communicator object
- Possible to communicate arbitrary Python objects
- NumPy arrays can be communicated with nearly same speed as in C/Fortran