Jussi Enkovaara
Martti Louhivuori

Python in High-Performance Computing

CSC – IT Center for Science Ltd, Finland

```
import sys, os
try:
    from Bio.PDB import PDBParser
    __biopython_installed__ = True
except ImportError:
    __biopython_installed__ = False

__default_bfactor__ = 0.0     # default B-factor
__default_occupancy__ = 1.0   # default occupancy level
__default_segid__ = ''        # empty segment ID

class EOF(Exception):
    def __init__(self): pass

class FileCrawler:
    """
    Crawl through a file reading back and forth without loading
    anything to memory.
    """
    def __init__(self, filename):
        try:
            self.__fp__ = open(filename)
        except IOError:
            raise ValueError, "Couldn't open file '%s' for reading." % filename
        self.tell = self.__fp__.tell
        self.seek = self.__fp__.seek
    def prevline(self):
        try:
            self.prev()
```

Agenda

Wednesday

9:00-9:15	Python and HPC
9:15-10:00	NumPy – fast array interface to Python
10:00-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:00	NumPy tools
11:00-12:00	Exercises
12:00-13:00	Lunch break
13:00-13:45	Advanced indexing, Vectorized operations & broadcasting, numexpr
13:45-14:30	Exercises
14:30-14:45	Coffee Break
14:45-15:30	Performance analysis
15:30-16:30	Exercises

Thursday

9.00-9.45	Optimising Python with Cython
9:45-10:30	Cython cont.
10.30-10.45	Coffee break
10:45-12:00	Exercises
12.00-13.00	Lunch break
13.00-13:45	Interfacing external libraries
13:45-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.30	Multiprocessing
15:30-16:30	Exercises

Friday

9.00-9.45	MPI introduction
9:45-10:30	Point-to-point communication
10.30-10.45	Coffee break
10:45-12:00	Exercises
12.00-13.00	Lunch break
13.00-13:45	Non-blocking communication and communicators
13:45-14:30	Exercises
14.30-14.15	Coffee break
14.15-15.30	Collective communications
15:30-16:30	Exercises
16:30-16:45	Summary of Python HPC strategies

PYTHON AND HIGH-PERFORMANCE COMPUTING

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 and 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, 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 if 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

Agenda

Monday

9:00-9:15	Python and HPC
9:15-10:00	NumPy – fast array interface to Python
10:00-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:00	NumPy tools
11:00-12:00	Exercises
12:00-13:00	Lunch break
13:00-13:45	Advanced indexing, Vectorized operations & broadcasting, numexpr
13:45-14:30	Exercises
14:30-14:45	Coffee Break
14:45-15:30	Performance analysis
15:30-16:30	Exercises

Tuesday

9.00-9.45	Optimising Python with Cython
9:45-10:30	Cython cont.
10.30-10.45	Coffee break
10:45-12:00	Exercises
12.00-13.00	Lunch break
13.00-13:45	Interfacing external libraries
13:45-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.30	Multiprocessing
15:30-16:30	Exercises

Wednesday

9.00-9.45	MPI introduction
9:45-10:30	Point-to-point communication
10.30-10.45	Coffee break
10:45-12:00	Exercises
12.00-13.00	Lunch break
13.00-13:45	Non-blocking communication and communicators
13:45-14:30	Exercises
14.30-14.15	Coffee break
14.15-15.30	Collective communications
15:30-16:30	Exercises
16:30-16:45	Summary of Python HPC strategies

NUMPY BASICS

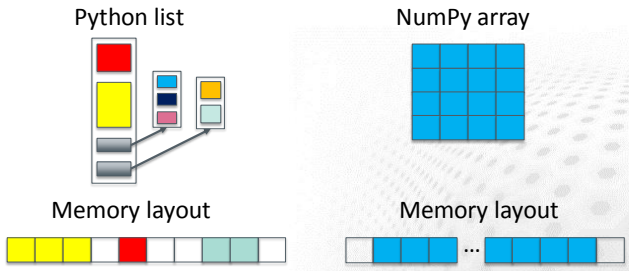
Numpy – fast array interface

- 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
  - some linear algebra, random numbers

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



Creating numpy arrays

- From a list:

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>>
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.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
```

Creating numpy arrays

- More ways for creating arrays:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5 , -2.25,  0.  ,  2.25,  4.5 ])
>>>
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>>
>>> d = np.ones((2, 4))
>>> d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Non-numeric data

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

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

- Character arrays can, however, be sometimes useful

```
>>> dna = 'AAAGTCTGAC'
>>> a = np.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')
```

Indexing and slicing arrays

- Simple indexing:

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

- Slicing:

```
>>> a = np.arange(5)
>>> a[2:]
array([2, 3, 4])
>>> a[:-1]
array([0, 1, 2, 3])
>>> a[1:3] = -1
>>> a
array([0, -1, -1, 3, 4])
```

Indexing and slicing arrays

- Slicing is possible over all dimensions:

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>>
>>> a = np.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

example.py

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

- split : split array to N pieces

```
>>> np.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 = np.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
```

NUMPY TOOLS

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(np.arange(4), 10)
array([0, 1, 1, 2, 1, 1, 2, 0, 2, 3])
```

Polynomials

- Polynomial is defined by array of coefficients p
- p(x, N) = p[0] x^{N-1} + p[1] x^{N-2} + ... + p[N-1]
- Least square fitting: numpy.polyfit
- Evaluating polynomials: numpy.polyval
- Roots of polynomial: numpy.roots
- ...

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

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

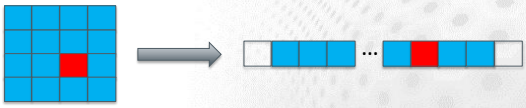
Linear algebra

- Normally, NumPy utilises high performance libraries in linear algebra operations
- Exampe: matrix multiplication
  C = A * B
  matrix dimension 200
  - pure python: 5.30 s
  - naive C: 0.09 s
  - numpy.dot: 0.01 s

NUMPY ADVANCED TOPICS

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

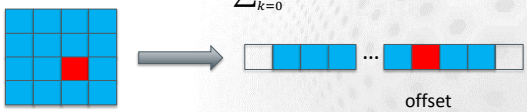


NumPy indexing

- 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, ... n_{N-1}) corresponds to offset from the beginning of 1-dimensional block

$$offset = \sum_{k=0}^{N-1} s_k n_k$$

s_k stride in dimension k



(n_0, n_1)

ndarray attributes

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

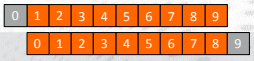
- Boolean "mask" arrays

```
>>> 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



example.py

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

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

  - for loop is ~80 times slower!

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

example.py

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

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

Temporary arrays

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

example.py

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

# two temporary arrays will be created
c = 2.0 * a - 4.5 * b
      temp1   temp2

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

example.py

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

Numexpr

- Evaluation of complex expressions with one operation at a time can lead also into suboptimal performance
  - Effectively, one carries out multiple *for* loops in the NumPy C-code
- Numexpr package provides fast evaluation of array expressions

example.py

```
import numexpr as ne
x = np.random.random((1000000, 1))
y = np.random.random((1000000, 1))
poly = ne.evaluate("((.25*x + .75)*x - 1.5)*x - 2")
```

Numexpr

- By default, numexpr tries to use multiple threads
- Number of threads can be queried and set with ne.set_num_threads(nthreads)
- Supported operators and functions
  - +,-,*,/,**, sin, cos, tan, exp, log, sqrt
- Speedups in comparison to NumPy are typically between 0.95 and 4
- Works best on arrays that do not fit in CPU cache

Summary

- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Tools for linear algebra and random numbers
- Arrays can be broadcasted into same shapes
- Expression evaluation can lead into temporary arrays

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
- 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.time()**, **time.clock()**,
  - In Python 3: **time.perf_counter()**, **time.process_time()**

timing.py

```
import time

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

```
$ python -m timeit -s "from mymodule import func" "func()"
10 loops, best of 3: 433 msec per loop
```

- %timeit magic in IPython

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

profile.py

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

- Profiling whole program from command line

```
$ 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

CYTHON

Cython

- Optimising static compiler for Python
- Extended Cython programming language
- Tune readable Python code into plain C performance by adding static type declarations
- Easy interfacing to external C libraries

Python overheads

- Interpreting
- ”Boxing ”- everything is an object
- Function call overhead
- Global interpreter lock – no threading benefits (CPython)

Interpreting

- Cython command generates a C /C++ source file from a Cython source file
- C/C++ source is then compiled into an extension module
- Interpreting overhead is normally not drastic

setup.py

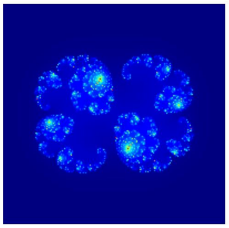
```
from distutils.core import setup
from Cython.Build import cythonize

# Normally, one compiles cython extended code with .pyx ending
setup(ext_modules=cythonize(“mandel_cyt.py”), )
```

```
$ python setup.py build_ext --inplace
```

```
In [1]: import mandel_cyt
```

Case study: Mandelbrot fractal

- Pure Python: 2.71 s
- Compiled with Cython: 2.61 s

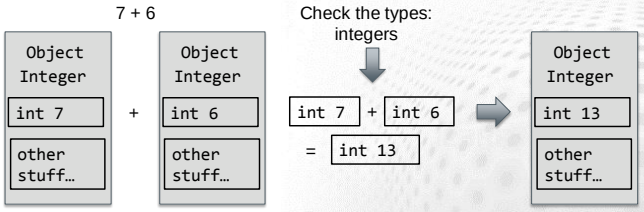


mandel.py

```
def kernel(zr, zi, cr, ci, lim, cutoff):

    count = 0
    while ((zr*zr + zi*zi) < (lim*lim)) \
              and count < cutoff:
        zr = zr * zr - zi * zi + cr
        zi = zr * zr - zi * zi + cr
        count += 1
    return count
```

”Boxing”

- In Python, everything is an object



Static type declarations

- Cython extended code should have **.pyx** ending
  - Cannot be run with normal Python
- Types are declared with **cdef** keyword
  - In function signatures only type is given

example.py

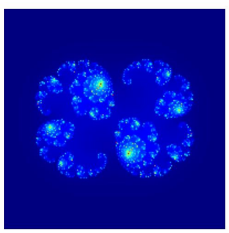
```
def integrate(f, a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

example.pyx

```
def integrate(f, double a,
              double b, int N):
    cdef double s = 0
    cdef int i
    cdef double dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Static type declarations

- Pure Python: 2.71 s
- Type declarations in kernel: 20.2 ms

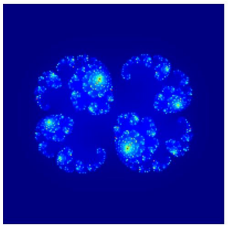


Function call overhead

- Function calls in Python can involve lots of checking and ”boxing”
- Overhead can be reduced by declaring functions to be C-functions
  - **cdef** keyword: functions can be called only from Cython
  - **cpdef** keyword: generate also Python wrapper (can have additional overhead in some cases)

Using C functions

- Static type declarations: 20.2 ms
- Kernel as C function: 12.5 ms



mandel.py

```
cdef int kernel(double zr, double zi, …):

    cdef int count = 0
    while ((zr*zr + zi*zi) < (lim*lim)) \
              and count < cutoff:
        zr = zr * zr - zi * zi + cr
        zi = zr * zr - zi * zi + cr
        count += 1
    return count
```

NumPy arrays with Cython

- Cython supports fast indexing for NumPy arrays
- Type and dimensions of array have to be declared

numpy_example.py

```
import numpy as np   # Normal NumPy import
cimport numpy as cnp # Import for NumPY C-API

def func():  # declarations can be made only in function scope
    cdef cnp.ndarray[cnp.int_t, ndim=2] data
    data = np.empty((N, N), dtype=int)

    …
    for i in range(N):
        for j in range(N):
            data[i,j] = …       # double loop is done in nearly C speed
```

Compiler directives

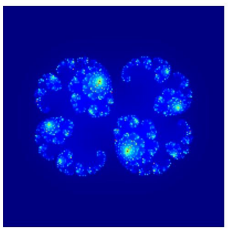
- Compiler directives can be used for turning of certain Python features for additional performance
  - boundscheck (False) : assume no IndexErrors
  - wraparound (False): no negative indexing
  - …

numpy_example.py

```
import numpy as np   # Normal NumPy import
cimport numpy as cnp # Import for NumPY C-API
import cython

@cython.boundscheck(False)
def func():  # declarations can be made only in function scope
    cdef cnp.ndarray[cnp.int_t, ndim=2] data
    data = np.empty((N, N), dtype=int)
```

Final performance

- Pure Python: 2.7 s
- Static type declarations: 20.2 ms
- Kernel as C function: 12.5 ms
- Fast indexing and directives: 2.4 ms

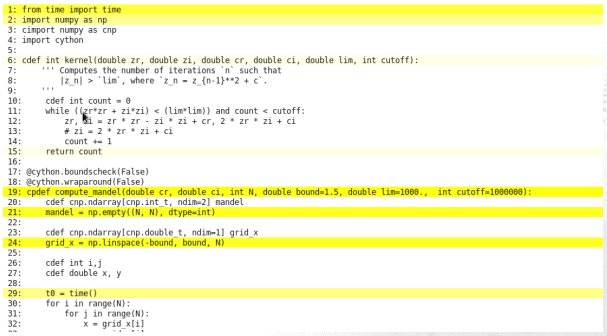


Where to add types?

- Typing everything reduces readibility and can even slow down the performance
- Profiling should be first step when optimising
- Cython is able to provide annotated HTML-report
- Lines are colored according to the level of “typedness”
  - white lines translate to pure C
  - lines that require the Python C-API are yellow (darker as they translate to more C-API interaction)

```
$cython -a cython_module.pyx
$firefox cython_module.html
```

HTML-report



Profiling Cython code

- By default, Cython code does not show up in profile produced by cProfile
- Profiling can be enabled for entire source file or on per function basis

profiling.py

```
# cython: profile=True

import cython
…
@cython.profile(False)
cdef func():
    …
```

profiling.py

```
# cython: profile=False

import cython
…
@cython.profile(True)
cdef func():
    …
```

Summary

- Cython is optimising static compiler for Python
- Possible to add type declarations with Cython language
- Fast indexing for NumPy arrays
- At best cases, huge speed ups can be obtained
  - Some compromise for Python flexibility

Further functionality in Cython

- Using C structs and C++ classes in Cython
- Exceptions handling
- Parallelisation (threading) with Cython
- …

INTERFACING EXTERNAL LIBRARIES

Increasing performance with compiled code

- There are Python interfaces for many high performance libraries
- However, sometimes one might want to utilize a library without Python interface
  - Existing libraries
  - Own code written in C or Fortran
- Python C-API provides the most comprehensive way to extend Python
- **Cffi**, cython, and f2py can provide easier approaches

cffi

- C Foreign Function Interface for Python
- Interact with almost any C code
- C-like declarations within Python
  - Can often be copy-pasted from headers / documentation
- ABI and API modes
  - ABI does not require compilation
  - API can be more robust
  - Only ABI discussed here
- Some understanding of C required

cffi example 1

cffi_example.py

```
from cffi import FFI

ffi = FFI()
# Use sqrt from C standard math library
lib = ffi.dlopen("libm.so")
ffi.cdef("""float sqrtf(float x);""")

# Python takes care of proper datatype conversion
a = lib.sqrtf(4)
print(a)
```

cffi example 2

cffi_example.py

```
from cffi import FFI
import numpy as np

ffi = FFI()
lib = ffi.dlopen("./myclib.so") # Use functions from users own library
ffi.cdef("""void add(double *x, double *y, int n);""")
ffi.cdef("""void subtract(double *x, double *y, int n);""")

a = np.random.random((1000000,1))
b = np.zeros_like(a)

# "Pointer" objects need to be passed to library
aptr = ffi.cast("double *", ffi.from_buffer(a))
bptr = ffi.cast("double *", ffi.from_buffer(b))

lib.add(bptr, aptr, len(a))
lib.subtract(bptr, aptr, len(a))
```

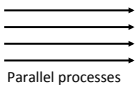
Summary

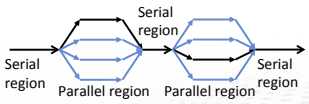
- External libraries can be interfaced in various ways
- cffi provides easy interfacing to C libraries
  - System libraries and user libraries
  - Python can take care of some datatype conversions

MULTIPROCESSING

– PROCESS BASED ”THREADING”

Processes and threads





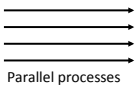
Process

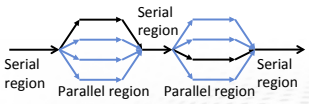
- Independent execution units
- Have their own state information and *own memory* address space

Thread

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

Processes and threads





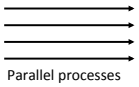
Process

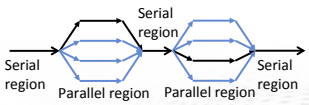
- Long-lived: created when parallel program started, killed when program is finished
- Explicit communication between processes

Thread

- Short-lived: created when entering a parallel region, destroyed (joined) when region ends
- Communication through shared memory

Processes and threads





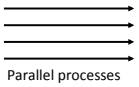
Process

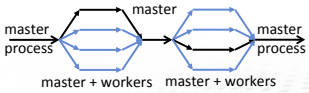
- MPI
  - good performance
  - scales from a laptop to a supercomputer

Thread

- OpenMP
  - C / Fortran, not Python
- threading module
  - only for I/O bound tasks (maybe)
  - Global Interpreter Lock (GIL) limits usability

Processes and threads





Process

- MPI
  - good performance
  - scales from a laptop to a supercomputer

~~Thread~~ Process

- multiprocessing module
  - relies on OS for forking worker processes that mimic threads
  - limited communication between the parallel processes

Multiprocessing

- Underlying OS used to spawn new independent subprocesses
  - processes are independent and execute code in an asynchronous manner
    - no guarantee on the order of execution
- Communication possible only through dedicated, shared communication channels
  - Queues, Pipes
  - must be created before a new process is forked

Spawn a process

spawn.py

```
from multiprocessing import Process
import os

def hello(name):
    print 'Hello', name
    print 'My PID is', os.getpid()
    print "My parent's PID is", os.getppid()

# Create a new process
p = Process(target=hello, args=('Alice', ))

# Start the process
p.start()
print 'Spawned a new process from PID', os.getpid()

# End the process
p.join()
```

Communication

- Sharing data
  - shared memory, data manager
- Pipes
  - direct communication between two processes
- Queues
  - work sharing among a group of processes
- Pool of workers
  - offloading tasks to a group of worker processes

Queues

- FIFO (*first-in-first-out*) task queues that can be used to distribute work among processes
- Shared among all processes
  - all processes can add and retrieve data from the queue
- Automatically takes care of locking, so can be used safely with minimal hassle

Queues

task-queue.py

```
from multiprocessing import Process, Queue

def f(q):
    while True:
        x = q.get()
        if x is None:
            break
        print(x**2)

q = Queue()
for i in range(100):
    q.put(i)

# task queue: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 99]

for i in range(3):
    q.put(None)
    p = Process(target=f, args=(q, ))
    p.start()
```

Queues

task-queue.py

```
from multiprocessing import Process, Queue

def f(q):
    while True:
        x = q.get()
        if x is None:  # if sentinel, stop execution
            break
        print(x**2)

q = Queue()
for i in range(100):
    q.put(i)

# task queue: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 99]

for i in range(3):
    q.put(None)  # add sentinels to the queue to signal STOP
    p = Process(target=f, args=(q, ))
    p.start()
```

Pool of workers

- Group of processes that carry out tasks assigned to them

1. Master process submits tasks to the pool
2. Pool of worker processes perform the tasks
3. Master process retrieves the results from the pool

- Blocking and non-blocking (= asynchronous) calls available

Pool of workers

pool.py

```
from multiprocessing import Pool
import time

def f(x):
    return x**2

pool = Pool(8)

# Blocking execution (with a single process)
result = pool.apply(f, (4,))
print(result)

# Non-blocking execution "in the background"
result = pool.apply_async(f, (12,))
while not result.ready():
    time.sleep(1)
print(result.get())

# an alternative to "sleeping" is to use e.g. result.get(timeout=1)
```

Pool of workers

pool-map.py

```
from multiprocessing import Pool
import time

def f(x):
    return x**2

pool = Pool(8)

# calculate x**2 in parallel for x in 0..9
result = pool.map(f, range(10))
print(result)

# non-blocking alternative
result = pool.map_async(f, range(10))
while not result.ready():
    time.sleep(1)
print(result.get())
```

Multiprocessing summary

- Parallelism achieved by launching new OS processes
- Only limited communication possible
  - work sharing: queues / pool of workers
- Non-blocking execution available
  - do something else while waiting for results

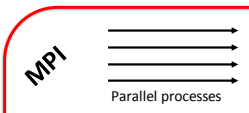
- Further information:
  https://docs.python.org/2/library/multiprocessing.html

MESSAGE PASSING INTERFACE

Message passing interface

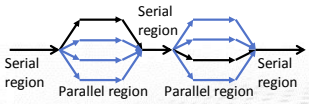
- MPI is an application programming interface (API) for communication between separate processes
- The most widely used approach for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from PC's to supercomputers
- MPI 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
  - mpirun, mpiexec, aprun, srun,...
  - aprun on sisu.csc.fi, srun on taito.csc.fi

MPI rank

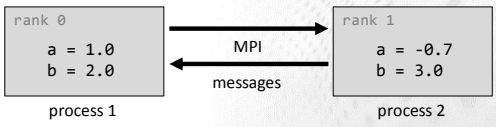
- Rank: ID number given to a process
  - it is possible to query for rank
  - processes can perform different tasks based on their rank

example.py

```
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

hello.py

```
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

hello.py

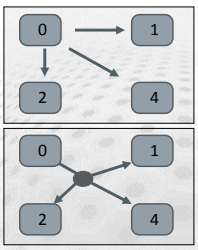
```
from mpi4py import MPI

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))
```

```
$ mpirun -np 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
```

POINT-TO-POINT COMMUNICATION

MPI communication

- MPI processes are independent, they 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

Sending and receiving data

- Sending and receiving a dictionary

send.py

```
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
  - 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*

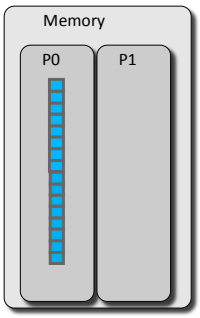
Typical point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data

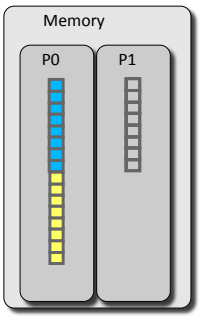
- Incorrect ordering of sends and receives may result in a deadlock

Case study: parallel sum

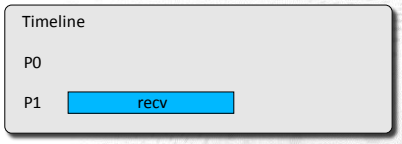


- Array originally on process #0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segments
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

Case study: parallel sum

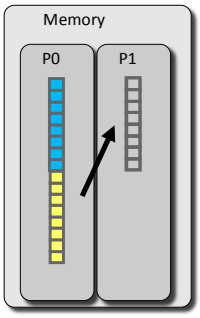


Step 1.1: Receive operation in scatter

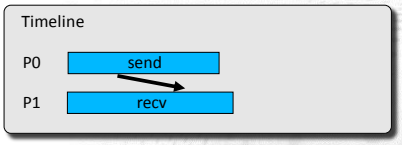


P1 posts a receive to receive half of the array from P0

Case study: parallel sum

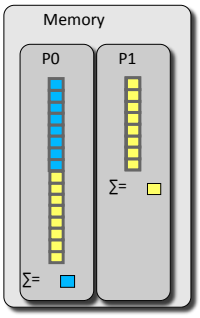


Step 1.2: Send operation in scatter

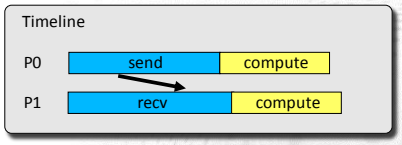


P0 posts a send to send the lower part of the array to P1

Case study: parallel sum

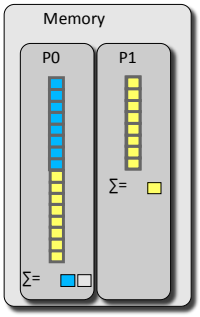


Step 2: Compute the sum in parallel

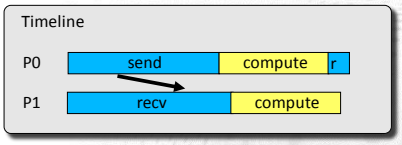


P0 & P1 computes their parallel sums and store them locally

Case study: parallel sum

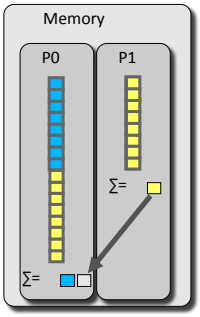


Step 3.1: Receive operation in reduction

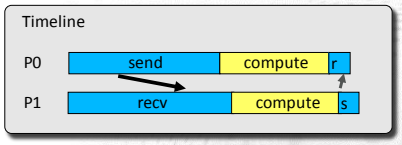


P0 posts a receive to receive partial sum

Case study: parallel sum

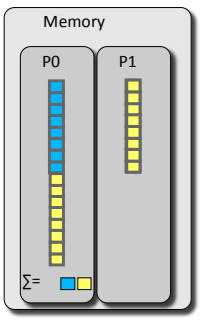


Step 3.2: send operation in reduction

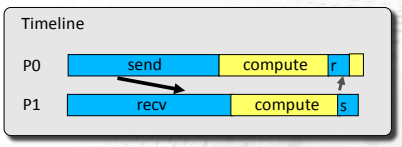


P1 posts a send with partial sum

Case study: parallel sum



Step 4: compute final answer



P0 sums 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

send-array.py

```
from mpi4py import MPI
import numpy

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

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

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

Combined send and receive

sendrecv.py

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

if rank == 0:
    comm.Sendrecv(data, dest=1, recvbuf=buffer, source=1)
elif rank == 1:
    comm.Sendrecv(data, dest=0, recvbuf=buffer, source=0)
```

- 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*

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

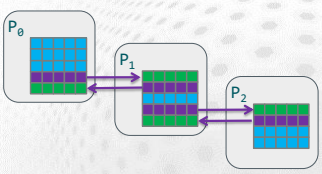
- Non-blocking sends and receives
  - `isend` & `irecv`
  - returns immediately and sends/receives in background
  - return value is a `Request` object
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations

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`

Typical usage pattern

```
request = comm.Irecv(ghost_data)
request2 = comm.Isend(border_data)
compute(ghost_independent_data)
request.wait()
compute(border_data)
```



Non-blocking send/receive

- Interleaving communication and computation

isend.py

```
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

waitall.py

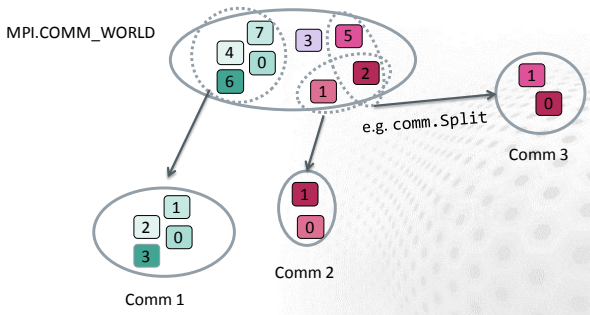
```
from mpi4py.MPI import Request
data = numpy.arange(10, dtype=float) * (rank + 1)  # send buffer
buffer = numpy.empty(10, float)                    # receive buffer
if rank == 0:
    req = [comm.Isend(data, dest=1))]
    req.append(comm.Irecv(buffer, source=1))
    Request.waitall(req)
```

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



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

split.py

```
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:

p2p.py

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

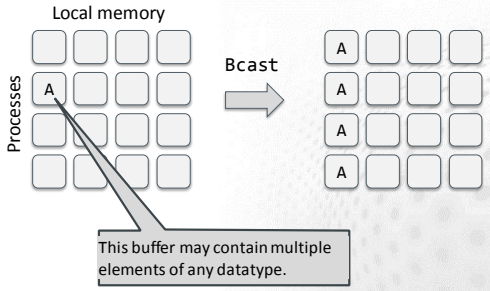
collective.py

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

Communicating a Numpy array of 1M elements from task 0 to all other tasks

Broadcasting

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



Broadcasting

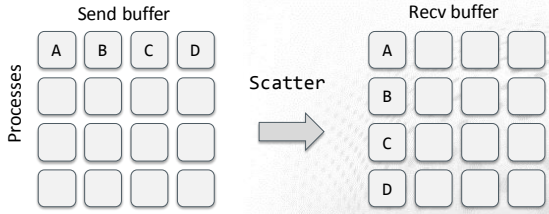
- Broadcast sends same data to all processes

bcast.py

```
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)
```

Scattering

- Send equal amount of data from one process to others



- Segments A, B, … may contain multiple elements

Scattering

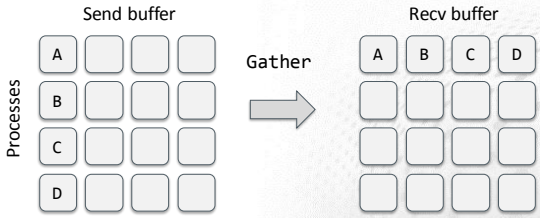
- Scatter distributes data to processes

scatter.py

```
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
buffer = empty(size, float)  # prepare a receive buffer
new_data = comm.scatter(py_data, root=0)      # returns the value
comm.Scatter(data, buffer, root=0)  # in-place modification
```

Gathering

- Collect data from all the process to one process



- Segments A, B, … may contain multiple elements

Gathering

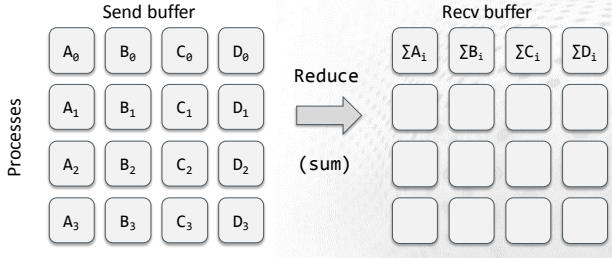
- Gather pulls data from all processes

gather.py

```
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 operation

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



Reduce operation

- Reduce gathers data and applies an operation on it

reduce.py

```
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
```

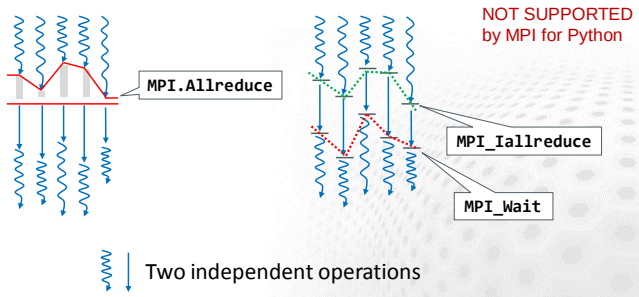
Other common collective operations

- 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 to/from each other

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

Non-blocking collectives



Common mistakes with collectives

- Using a collective operation within one branch of an if-test of the rank

```
if rank == 0: comm.bcast(...)
```

  - all processes in a communicator must call a collective routine!
- Assuming that all processes making a collective call would complete at the same time
- Using the input buffer as the output buffer

```
comm.Scatter(a, a, MPI.SUM)
```

Summary

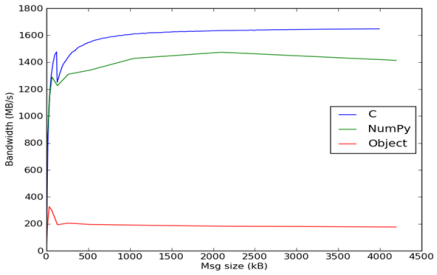
- Collective communications involve all the processes within a communicator
  - all processes must call them
- Collective operations make code more transparent and compact
- Collective routines allow optimizations by MPI library
- MPI-3 contains also non-blocking collectives, but these are currently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

On-line resources

- Documentation for mpi4py is quite limited
  - short on-line manual and API reference available at http://pythonhosted.org/mpi4py/
- 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

mpi4py performance

- Ping-pong test



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 from C/Fortran