

Parallel & Scientific Computing with Python

Contents

Python - The Basics

Parallel Python

- threading / multiprocessing: for multicore
- mpi4py: for cluster

Python – The Basics

What is Python?

- Developed by Guido van Rossum (1989)
- Modern, interpreted, object-oriented, full featured high level programming language
- Rapid development (Life is too short and long to do list ...)
- Portable (Unix/Linux, Mac OS X, Windows)
- Open source, intellectual property rights held by the Python Software Foundation
- Python versions: 2.x and 3.x : 3.x is not backwards compatible with 2.x

Why Python?

- Easy to learn and fun
- Intuitive and simple mind
- Powerful basic data structure (List, Dictionary, Set, ...)
- Powerful string process
- Productive (re-usability, Readability)
- Object oriented programming
- Large standard library - os, sys, string, math, random, ...
- Lots of third party libraries - Numpy, Scipy, Mpi4py, ...

Python Interpreter vs Scripting

```
% python
>>> str = 'Hello World !'
>>> print str
>>>                                     # Ctrl+D for Exit

% python hello.py
'Hello World !'
% chmod +x hello.py
% ./hello.py
'Hello World !'
```

Hello World !

```

program hello
  implicit none
  character(len=13) :: sent
  sent="Hello, World"
  print *,sent
end program hello

```

```

#!/usr/bin/python
sent="Hello, World"
print sent

```

```

#include<stdio.h>
#include<string.h>

int main(int argc, char* argv[]) {
  char sent[13];
  memset(sent,13,'\0');
  strcpy(sent,"Hello, World");
  printf("%s\n",sent);
  return 0;
}

```

```

subroutine out(line)
  character(len=13), intent(in) :: line
  print *,line
end subroutine

```

```

program hello
  implicit none
  character(len=13) :: sent
  sent="Hello, World"
  call out(sent)
end program hello

```

```

#!/usr/bin/python

```

```

def out(line) :
  print line

sent="Hello, World"
out(sent)

```

```

#include<stdio.h>
#include<string.h>

void out(char* line) {
  printf("%s\n",line);
}

```

```

int main(int argc, char* argv[]) {
  char sent[13];

  memset(sent,13,'\0');
  strcpy(sent,"Hello, World");
  out(sent);

  return 0;
}

```

Data types

- Integers
- Floats
- Complex numbers
- Basic operations: $+$, $-$, $*$, $/$ and $**$
- Strings are enclosed by `"`, `'`, or `'''`

```

>>> a = 2          # integer
>>> b = 3.0        # float
>>> print 'sum = ', a+b
>>> a, b = b, a     # tuple (exchange)
>>> x = 4.0 + 5.0j  # complex

```

```

>>> s1 = "this is string !"
>>> s2 = 'this is string !'
>>> s3 = "this isn't so simple !"
>>> s4 = 'is this "complex"'
>>> s5 = ''' STRING '''

```

– $+$ and $*$ operators

```

>>> "Strings can be " + "combined"
'Strings can be combined'
>>> "Repeat! " * 3
'Repeat! Repeat! Repeat!'

```

Data types

- Python is dynamically typed language – no type declarations for variables
- Variable does have a type

– incompatible types cannot be combined

```

>>> print "Starting example"
>>> x = 1.0
>>> for i in range(10):
    x += 1

```

```

>>> y = 4 * x
>>> print x, y
11.0 44.0

```

```
>>> s = 'Result'
>>> z=s+y          # Error
```

Dynamic typing

- No separate functions for different data types

```
>>> def add(x, y):
    result = x + y
    return result

>>> add(1, 2)
3
>>> add(1.0, 2.0)
3.0
```

- Works for any numeric type
- No duplicate code e.g. for real and complex numbers

Powerful data structures: List (mutable sequences)

- Python lists are dynamic arrays
- List items are indexed (index starts from 0)
- List item can be any Python object, items can be of different type
- New items can be added to any place in the list
- Items can be removed from any place of the list

```
>>> list = [2, 1, 'hello']
>>> dir(list)                #
>>> list.append(5)           # [2, 1, 'hello', 5]
>>> list.remove(1)           # [2, 'hello', 5]
>>> list.insert(1, 'paul')    # [2, 'paul', 'hello', 5]
>>> list.sort()              # [2, 5, 'hello', 'paul']
>>> del list[0]               # [5, 'hello', 'paul']
>>> del list[-1]              # [5, 'hello']
>>> list.append('world')      # [5, 'hello', 'world']
>>> list[0] = 1               # [1, 'hello', 'world']
>>> list[0:2] = [4, 5]        # [4, 5, 'world']

>>> a=range(4)
Exercise:
List a+b, list.index(2), list.pop(), list.reverse()
list.count(2), 2 in list, len(list), etc...
```

Sequences indexing and slicing

```
>>> a= [0, 1, 'a', 'b', 'c', 4, 5]
>>> a[4]                        # 'c'
>>> a[-1]                       # 5
>>> a[-2]                       # 4
>>> a[2:4]                      # ['a', 'b']
>>> a[2:-3]                     # ['a', 'b']
>>> a[-5:-3]                    # ['a', 'b']
>>>
>>> a=[1,2,3,4]
>>> b=a                        # references (b is alias of a)
>>> b[0] = -1
>>> print a, b                 # same value
>>> c=a[:]                     # values (c is another list)
```

```

>>> c[0] = 0
>>> print a, c
>>>
>>> x1 = [ 1, 2, 3 ]
>>> x2 = [ 4, 5, 6 ]
>>> y = [ x1, x2 ]                # [ [1,2,3], [4,5,6] ] list of
lists
>>> y[0][0]                       # 1
>>> y[1][0]                       # 4

```

Powerful data structures: Dictionary (Hash Table)

- Dictionaries are associative arrays
- Unordered list of key - value pairs
- Values are indexed by keys
- Keys can be strings or numbers
- Value can be any Python object

```

>>> d1 = {}                        # empty dictionary
>>> d2 = {'GLY':'G', 'LYS':'K'}
>>> d2['GLY']                      # indexing by key
>>> d2.has_key('LYS')             # membership test
>>> d2.keys()                     # keys list
>>> d2.values()                   # values list
>>> d2.items()                    # list of tuple (key, value)
>>> len(d2)                       # length

>>> d2['PRO'] = 'P'               # adding/changing
>>> del d2['LYS']                  # deleting

```

Iteration (List and String)

```

>>> for i in [0,1,2,3,4]:
>>>     print 'i=', i             # tab or spaces
>>> for i in range(5):
>>>     print 'i=%d' % i          # [0,1,2,3,4]
>>> List = [ 'a', 1, 'b', 'spam', [0,1,2,3] ]
>>> for i in List:
>>>     print 'element = %s' % i
>>> str = 'abcdef'                # string
>>> for c in str:
>>>     print 'Ascii value of %c is %3d' % (c,ord(c))

```

Iteration (Dictionary)

```

>>> Amino = {'A':1, 'R':2, 'N':3}
>>> for key in Amino:
>>>     print 'key = %s' % key     # print keys
>>> for key in Amino:
>>>     print '%s = %d' % (key, Amino[key]) # values
>>> Keys = Amino.keys()           # key list
>>> print keys

>>> for key, value in Amino.items():
>>>     print '%s = %d' % (key, value)

```

Control flows (Branchs)

```
>>> seq = range(0,10)
>>> print seq
>>> for s in seq:
    if s < 3:
        continue
    elif s == 8:
        break
    else:
        print s
```

Functions and arguments

```
>>> def func (x):
    x = x**2
    return x
>>> f(5)
25

>>> def f1 (x):    # local
    x = x+1
>>> x=1
>>> f1 (x)
>>> print x        # x=1
```

Files, I/O

```
>>> f = open('testfile')
>>> for line in f.readlines():
    print line[:-1]
>>> f.close()
>>>
>>> x = ['a','b','c','d','e','f']
>>> g = open('data','w')
>>> for i in range(len(x)):
    line = ' x[%d] = %s \n' % (i,x[i])
    g.write (line)
>>> g.close()
```

Class - Basic

```
>>> class Vector:
    def __init__(self,x,y,z):
        self.xyz = [x,y,z]
>>> a=Vector(1,0,0)           # instance a
>>> b=Vector(0,1,0)           # instance b
>>> c=Vector(0,0,1)           # instance c
>>> print a
>>> print b
>>> print c
>>> print a.xyz
>>> print a.xyz[0], a.xyz[1], a.xyz[2]
```

Class - methods, operator

```
#!/usr/bin/python
import math
class Vector:
    def __init__ (self,x,y,z):
        self.v = [x,y,z]
    def __str__ (self): return 'Vector (%f, %f, %f)' %
(self.v[0],self.v[1],self.v[2])
    def length (self): return math.sqrt(self.v[0]**2+self.v[1]**2+self.v[2]**2)
    def __add__ (self, other):
        return
Vector(self.v[0]+other.v[0],self.v[1]+other.v[1],self.v[2]+other.v[2])
def main ():
    a=Vector (1,0,0) ; b=Vector (0,1,0) ; c=Vector (0,0,1)
    print 'a=%s' % a
    print 'b=%s' % b
    print 'c=%s' % c
    print 'length of a = %f' % a.length()
    print 'length of b = %f' % b.length()
    print 'length of c = %f' % c.length()
    x = a + b + c
    print 'x=%s' % x
    print 'length of x = %f' % x.length()
if __name__ == '__main__':
    main ()
```

Modules - Standard

```
>>> import math                # math module
>>> math.sqrt(2)
1.41421
>>> dir(math)                  # show all methods, elements
>>> math.pi
>>> math.e

>>> import string              # string module
>>> dir(string)
>>> s='1 2 3 4 5'
>>> list = string.split(s) ; print list
>>> list = map(string.atoi, string.split(s))
>>> print list
>>> seq = 'ARNDCPPPPCCCYWCGATDEFCCC'
>>> string.find('PPP')

>>> import math, string, os, sys
>>> from math import sqrt, sin, cos, log, pi
>>> dir(math)
>>> dir(string)
>>> dir(os)
```

Extending python

- Numerical module (numpy) # similar to matlab
- Scientific module (scipy)
- Biopython module
- Your modules...

- SWIG module (C/C++, Fortran binding)
- MPI module
- Etc

F2PY example

```
subroutine fib(a,n)
  implicit none
  integer, intent (in) :: n
  real*8, intent (out) :: a(n)
  integer :: i
  do i=1, n
    if (i == 1) then
      a(i) = 0.0d0
    elseif (i == 2) then
      a(i) = 1.0d0
    else
      a(i) = a(i-1) + a(i-2)
    endif
  enddo
end subroutine fib
```

```
* Compile and run
% f2py2.7 -c -m fib fib.f90
% ls
% python test.py
```

```
#!/usr/bin/env python

import fib

print fib.__doc__
print fib.fib.__doc__

a = fib.fib(5)

print a
```

Links

1. Python Home: <http://www.python.org>
2. Python scientific lecture note: <http://scipy-lectures.github.com>
3. <https://github.com/thehackerwithin/PyTrieste/wiki/F2Py>

Parallel Programming with Python

"Decompose the complete task into independent subtasks and let the data flow between them"

"Distribute the subtasks over the processors to minimize the total execution time"

Multi-threading / processing: minimizing waiting time

- For multi-cores, multi-processors
- Not suitable for clusters

Message-passing (MPI): minimizing the communication time

- For cluster machines
- Suitable but less efficient for multi-cores, multi-processors

Python - Parallel Programming (multiprocessing)

Spawning Processes

- from multiprocessing import *
- Create and start a process:
 p = Process(target=myfunc, args = tuple) # create a process
 p.start() # start a process
- Get process info:
 current_process().pid # process id
 current_process().name # process name

Hello World ! (hello.py)

```
#!/usr/bin/env python

from multiprocessing import Process

def hello ():
    print 'Hello World'

if __name__ == '__main__':

    jobs = []
    for me in range(5):
        p = Process (target=hello)
        p.start()
        jobs.append(p)
```

*Note (Execution)

```
% python hello.py
% chmod +x hello.py
% ./hello.py
```

hello_me.py (argument passing)

```
#!/usr/bin/env python

from multiprocessing import Process

def hello (me):
    print 'Hello World, I am a process %4s' % me

if __name__ == '__main__':

    jobs = []
    for me in range(5):
        p = Process (target=hello, args=(me,))
        p.start()
        jobs.append(p)
```

hello_name.py

```
#!/usr/bin/env python

import os
from multiprocessing import Process, current_process
from time import sleep

def hello ():
    name = current_process().name
    pid = current_process().pid
    sleep(10)
    print 'Hello, I am a %s with pid= %s' % (name, pid)

if __name__ == '__main__':

    jobs = []
    for me in range(5):
        p = Process (target=hello)
        p.start()
        jobs.append(p)

    #for job in jobs:
    #    job.join()
    name = current_process().name
    pid = current_process().pid
    print 'Hello, I am a %s with pid= %s' % (name, pid)
```

Pool: Pool of worker

- Multiprocessing module has a Pool class that:
 - Distribute the work between workers
 - Collect the return value as a list
- It makes it easy to implement quick/simple concurrent program

sqrt_paralle.py

```
#!/usr/bin/env python
from multiprocessing import Process

def hello (me):
    print 'Hello World, I am a process %4s' % me

if __name__ == '__main__':
    jobs = []
    for me in range(5):
        p = Process (target=hello, args=(me,))
        p.start()
        jobs.append(p)
```

p = Pool (processes=N) : The number of processes
p.map (myfunc, args, chunksize)
- args is list of arguments to evaluate with myfunc
- myfunc can accept only one argument
- chunks can be specified by setting chunksize (positive integer)
* apply_async, map_async, imap, imap_unordered, close()

sqrt_paralle.py

```
#!/usr/bin/env python
import sys, time, random
import multiprocessing as mp
from multiprocessing import Pool

def monte_carlo_pi(n):
    count = 0
    for i in range(n):
        x=random.random(); y=random.random()
        if x*x + y*y <= 1:
            count += 1
    return count

if __name__ == '__main__':
    np = int(sys.argv[1]) # the number of processes
    #np = mp.cpu_count()
    print 'Spawn %4d Processes' % (np)
    # Nummber of points to use for the Pi estimation
    n = 90000000
    part = [n/np] * np
    pool = Pool(processes=np) # create worker pool
    # parallel map
    tic = time.time()
    count=pool.map(monte_carlo_pi, part)
    toc = time.time()
    print 'count= %s' % (count)
    print 'Estimated value of Pi = ', 4.0*sum(count)/n
    print 'Elapsed time = %.4f sec' % (toc-tic)
```

parallel_talign.py

```
#!/usr/bin/env python2.7
import os, sys, time
from multiprocessing import Pool

NPROCS = 8

def talign (item):
    pdbA, pdbB = item
    cmd = './talign.x ~/data/%s.pdb %s \
          | grep -a "TM-score="' % (pdbA, pdbB)
    line = os.popen(cmd).read().strip()
    return '%s %s' % (pdbA, line)

if __name__ == '__main__':

    inpdb = sys.argv[1]
    p = Pool (NPROCS)
    pdbA = []
    for line in file('list'):
        pdbA.append((line.strip(), inpdb))

    tic = time.time()
    out = p.map(talign, pdbA)
    toc = time.time()

    for line in out: print line
    print 'Walltime = %s sec' % (toc-tic)
```

Pipe & Queue: Inter Process Communication

- Pipe ()
 - For communication between two processes
 - A Pipe has two ends: Process A writes something into his end of the pipe and process B can read it from his
 - Pipes are bidirectional
- Queue ()
 - Multi-producer, multi-consumer FIFO (cf. Stack)
 - Multiple processes can put items into the Queue, others can get them

send.py

```
#!/usr/bin/env python
from multiprocessing import Process, Pipe, current_process

def worker (con):
    while True:
        data = con.recv()
        if data == 'END': break
        print 'Hello World, I am a %s: data= %s' % (current_process().name, data)

if __name__ == '__main__':
    con1, con2 = Pipe ()
    p = Process (target=worker, args=(con2,))
    p.start()
    for i in range(5):
        con1.send(i)
    con1.send ('END')
```

q.py

```
#!/usr/bin/env python

import os
from multiprocessing import Process, Queue

def f(q):
    while True:
        data=q.get()
        if data == 'END': break
        else: print data

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()

    q.put (1)
    q.put (2)
    q.put ('Test')
    q.put ([1,2,3,4])
    q.put ('END')
```

qiter.py

```
#!/usr/bin/env python

from multiprocessing import Process, Queue
from multiprocessing import current_process

NPROCS = 8

def f(q,i):
    name = current_process().name
    data = []
    for i in range(i):
        data.append(i)
    q.put((name,data))

if __name__ == '__main__':

    q = Queue ()

    jobs = []
    for i in range(NPROCS):
        p = Process (target=f, args=(q,i))
        p.start()
        jobs.append(p)

    #for job in jobs: job.join()

    while True:
        if q.empty(): break
        print q.get()
```

pi.py

```
#!/usr/bin/env python

import os, time, math
from multiprocessing import Process, Queue

NPROCS=8

def compute_pi(params, output):
    item = params.get()
    n, start, step = item
    h = 1.0 / float(n)
    s = 0.0
    for i in xrange(start, n, step):
        x = h * (i + 0.5)
        s += 4.0 / (1.0 + x**2)
    output.put(s*h)

if __name__ == '__main__':

    n = 10000000
    params = Queue()
    output = Queue()

    tic = time.time()
    for me in range(NPROCS):
        p = Process(target=compute_pi, args=(params, output))
        p.start()

    for me in range(NPROCS):
        item = (n, me, NPROCS)
        params.put(item)

    pi = 0.0
    count = 0
    while True:
        if count == NPROCS: break
        x = output.get()
        pi += x
        count += 1
    toc = time.time()

    err = abs(math.pi - pi)
    print 'pi is approximately %.16f, error is %.16f' % (pi, err)
    print 'elapsed time=%.8f' % (toc-tic)
```

talign.py & parallel_talign.py

talign.py

```
#!/usr/bin/env python

import os
from multiprocessing import Process

class process_talign (Process):
    def __init__(self, que, out):
        Process.__init__(self)
```

```

        self.que = que
        self.out = out
    def run(self):
        while True:
            item = self.que.get()
            if item == 'END': break
            else:
                pdbA, pdbB = item
                cmd = './talign.x ~/data/%s.pdb %s \
| grep -a "TM-score="' % (pdbA, pdbB)
                line = os.popen(cmd).read().strip()
                out = '%s %s' % (pdbA, line)
                self.out.put(out)
        self.out.put('END')

```

parallel_talign.py

```

#!/usr/bin/env python

import os, sys, time
from multiprocessing import Queue
from talign import *

NPROCS = 8
if __name__ == '__main__':

    inpdb = sys.argv[1]
    pdbs = []
    for line in file('list'):
        pdbs.append(line[:-1])
    que = Queue ()      # todo queue
    out = Queue ()      # output queue
    # spawn proceses
    pro = []
    for i in range(NPROCS):
        p = process_talign (que, out)
        p.start()
        pro.append(p)
    # queuing jobs
    for i in range(len(pdbs)):
        pdbA = pdbs[i]
        que.put ([pdbA, inpdb])
    for i in range(NPROCS):
        que.put('END')
    # output
    count = 0
    while True:
        if count == NPROCS: break
        line = out.get()
        if line == 'END':
            count +=1
            continue
        else: print line
    # joining proceses
    for p in pro:
        p.join()

```

Sharing state between processes

- Shared memory
 - Value: the return value is a synchronized wrapper for the object
 - Array: the return value is a synchronized wrapper for the array
- Server process
 - Manager: control a server process that holds python objects and allow other process to manipulate them.
 - It make sure the shared object get updated in all processes when anyone modifies it.

share.py

```
#!/usr/bin/env python

from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':

    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print '%s' % num.value
    print '%s' % arr[:]
```

manage.py

```
#!/usr/bin/env python

from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':

    manager = Manager ()

    d = manager.dict()
    l = manager.list(range(10))

    p = Process (target=f, args=(d, l))
    p.start()
    p.join()

    print d
    print l
```

Synchronization issues

- Deadlock:
 - Two processes are waiting for each other to finish

- Usually caused by locks or by blocking communication
 - Race condition:
 - Two or more processes modify a shared resource (variable, file, ...)
 - Result depends on which process comes first
 - Can be avoided using locks, but it often cause deadlocks.
-

Python - Parallel Programming (MPI)

Key concepts of MPI

- Used to create parallel **SPMD** programs on distributed-memory machines with explicit message passing
- A Parallel **MPI Program is launched as separate processes (tasks)**, each with their own address space.
 - Requires partitioning data across tasks.
- **Data is explicitly moved** from task to task
 - A task accesses the data of another task through a transaction called “message passing” in which a copy of the data (message) is transferred (passed) from one task to another.
- There are two classes of message passing (transfers)
 - **Point-to-Point messages** involve only two tasks
 - **Collective messages** involve a set of tasks
- Transfers use **synchronous or asynchronous protocols**

Mpi4py module as python extension

- MPI4Py provides an interface very similar to the MPI-2 standard C++ Interface
- Focus is in translating MPI syntax and semantics: If you know MPI, MPI4Py is “obvious”
- **You can communicate Python objects!!**
- What you lose in performance, you gain in shorter development time
- Communication of contiguous NumPy arrays at nearly C-speed

Mpi4py API

<http://mpi4py.scipy.org/docs/apiref/index.html>

Hello World ! (Parallel, no communication)

```
#!/usr/bin/env python

from mpi4py import MPI
import sys

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

print 'Hello world! I am process %d of %d.' % (rank, size)
```

*Note (Execution)

```
% mpiexec -np 2 python hello.py
% chmod +x hello.py
% mpiexec -np 2 hello.py
```

Point-to-Point

- Send a message from one process to another
- Message can contain any number of native or user defined types with an associated message tag
- MPI4Py (and MPI) handle the packing and unpacking for user defined data types
- Two types of communication: Blocking and non-Blocking

*Note:

Message = data + envelope

data = buffer, count, datatype
envelope = process identifier (dest/source), tag, communicator

Point-to-Point : Blocking communication

1. Python objects

```
comm.send (obj, dest = 1, tag = 0)  
obj = com.recv (source = 0, tag = 0)
```

2. Array data

```
comm.Send ([array, count, datatype], dest = 1, tag =0)  
comm.Recv ([array, count, datatype], source = 0, tag =0)
```

*python object send/recv

```
#!/usr/bin/env python  
  
from mpi4py import MPI  
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()  
  
if rank == 0:  
    data = {'a': 7, 'b': 3.14}  
    comm.send(data, dest=1, tag=11)  
    print "Message sent, data is: ", data  
elif rank == 1:  
    data = comm.recv(source=0, tag=11)  
    print "Message Received, data is: ", data
```

*python object send/recv with numpy (nearly C speed)

```
#!/usr/bin/env python  
from mpi4py import MPI  
import numpy  
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()  
  
# pass explicit MPI datatypes  
if rank == 0:  
    data = numpy.arange(1000, dtype='i')  
    comm.Send([data, MPI.INT], dest=1, tag=77)  
elif rank == 1:  
    data = numpy.empty(1000, dtype='i')  
    comm.Recv([data, MPI.INT], source=0, tag=77)  
  
# automatic MPI datatype discovery  
if rank == 0:  
    data = numpy.arange(100, dtype=numpy.float64)  
    comm.Send(data, dest=1, tag=13)  
elif rank == 1:  
    data = numpy.empty(100, dtype=numpy.float64)  
    comm.Recv(data, source=0, tag=13)
```

*Note the difference between upper/lower case!

- send/recv: general Python objects, slow
- Send/Recv: continuous arrays, fast Extra material

Point-to-Point : Non-Blocking communication

1. Python objects

```
req = comm.isend (obj, dest = 0, tag = 0)
req.Wait ()
```

2. Array data

```
req1 = comm.isend ([array, count, datatype], dest = 1, tag = 0)
req2 = comm.lrecv ([array, count, datatype], source = 0, tag = 0)
MPI.request.Waitall ([req1, req2])
```

*Ping Pong

```
#!/usr/bin/env python
from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.rank == 0:
    sendmsg = 123
    target = 1
else:
    sendmsg = 456
    target = 0

request = comm.isend(sendmsg, dest=target, tag=77)
recvmsg = comm.recv(source=target, tag=77)
request.Wait()
print 'rank = %s, recvmsg = %s' % (comm.rank, recvmsg)
```

*Ring (Exchange) (% mpiexec -np 4 ex.py)

```
#!/usr/bin/env python

from mpi4py import MPI
comm = MPI.COMM_WORLD

sendmsg = [comm.rank]*3

right = (comm.rank + 1) % comm.size
left = (comm.rank - 1) % comm.size

req1 = comm.isend (sendmsg, dest = right)
req2 = comm.isend (sendmsg, dest = left )
lmsg = comm.recv (source = left)
rmsg = comm.recv (source = right)

MPI.Request.Waitall ([req1, req2])

print 'rank = %s, lmsg = %s, rmsg = %s' % (comm.rank, lmsg, rmsg)
```

*Isend/Irecv (% mpiexec -np 2 sendran.py)

```
#!/usr/bin/env python

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

```

comm = MPI.COMM_WORLD
random.seed (123+comm.rank)
data = np.arange (100, dtype = 'f')
random.shuffle (data)

if comm.rank == 0:
    target = 1
else:
    target = 0

req1 = comm.Isend ([data, MPI.FLOAT], dest = target)
req2 = comm.Irecv ([data, MPI.FLOAT], source = target)

MPI.Request.Waitall ([req1, req2])

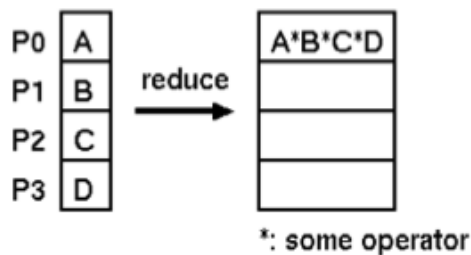
print 'rank = %s, data = %s' % (comm.rank, data)

```

Collective Communication

1. Reduce
2. Broadcast
3. Scatter/Gather
4. Synchronization

Reduce : compute pi



$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \frac{1}{n} \sum_{i=0}^{n-1} \frac{4}{1 + \left(\frac{i+0.5}{n}\right)^2}$$

* pi_parallel.py (% mpiexec -np 4 pi_parallel.py)

```

#!/usr/bin/env python

from mpi4py import MPI
import math

def compute_pi (n, start=0, step=1):
    h = 1.0 / n
    s = 0.0
    for i in xrange(start, n, step):
        x = h * (i + 0.5)
        s += 4.0 / (1.0 + x**2)
    return s*h

comm = MPI.COMM_WORLD

```

```

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

n      = 10000000

start = MPI.Wtime ()
mypi = compute_pi (n, rank, nproc)
pi    = comm.reduce (mypi, op=MPI.SUM, root = 0)
end    = MPI.Wtime ()
if rank == 0:
    err = abs (pi - math.pi)
    print 'pi is approximately %.16f, error is %.16f' % (pi, err)
    print 'elapsed time = %.8f' % (end - start)

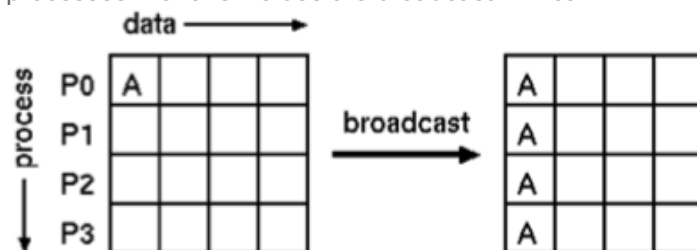
```

*Note: Reduce Operator (see API)

Name	Meaning
MPI.MAX	maximum
MPI.MIN	minimum
MPI.SUM	sum
MPI.PROD	product
MPI.LAND	logical and
MPI.BAND	bit-wise and
MPI.LOR	logical or
MPI.BOR	bit-wise or
MPI.LXOR	logical xor
MPI.BXOR	bit-wise xor
MPI.MAXLOC	max value and location
MPI.MINLOC	min value and location

Broadcast

Point to point doesn't always have to be one process to another, it can be one process to a number of processes. For this we use the broadcast API call.



* bcast.py (% mpiexec -np 4 bcast.py)

```

#!/usr/bin/env python

from mpi4py import MPI
comm = MPI.COMM_WORLD

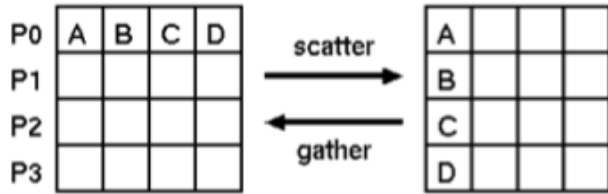
if comm.rank == 0:
    sendmsg = (1, "abc", [1.0, 2+3j], {'A':'ALA', 'R':'ARG'})
else:
    sendmsg = None

```

```
recvmsg = comm.bcast (sendmsg, root = 0)

print 'rank=%d, recvmsg = %s' % (comm.rank, recvmsg)
```

Scatter / Gather



* scatter.py (% mpiexec -np 10 scatter.py)

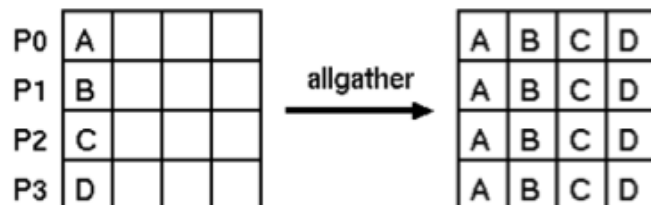
```
#!/usr/bin/env python

from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.rank == 0:
    sendmsg = [i**2 for i in range(comm.size)]
else:
    sendmsg = None

recvmsg = comm.scatter (sendmsg, root = 0)
print 'rank = %d, recvmsg = %s' % (comm.rank, recvmsg)
```

Allgather



* gather.py (% mpiexec -np 4 gather.py)

```
#!/usr/bin/env python

from mpi4py import MPI
comm = MPI.COMM_WORLD

sendmsg = comm.rank**2

recvmsg1 = comm.gather (sendmsg, root=0)
recvmsg2 = comm.allgather (sendmsg)
print 'rank=%d, recvmsg1 = %s' % (comm.rank, recvmsg1)
print 'rank=%d, recvmsg2 = %s' % (comm.rank, recvmsg2)
```

Synchronization

comm.Barrier ()

Pi calculation (Monte Carlo)

```
#!/usr/bin/env python

from mpi4py import MPI
import math
import random

def compute_pi (nsamples):
    random.seed (rank)
    count = 0
    for i in xrange(nsamples):
        # randomly choose a point in the box
        x = random.random()
        y = random.random()
        if x*x + y*y < 1.0:
            count += 1
    mypi = float(count)/nsamples
    pi = (4.0 / nproc) * comm.allreduce (mypi, op=MPI.SUM)
    return pi

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

if __name__ == '__main__':
    n = 100000
    start = MPI.Wtime ()
    pi = compute_pi (n)
    end = MPI.Wtime ()

    if rank == 0:
        err = abs (pi - math.pi)
        print 'pi is approximately %.16f, error is %.16f' % (pi, err)
        print 'elapsed time = %.8f' % (end - start)
```

Integration (Trapezoidal rule)

$$area \approx \sum_{i=0}^n \frac{[f(a) + f(b)]}{2} \cdot \Delta x = \left[\frac{f(a) + f(b)}{2} + \sum_{i=0}^n f(a + i\Delta x) + f(a + (i + 1)\Delta x) \right] \cdot \Delta x$$

where $\Delta x = (b - a)/n$

* integral.py

```
#!/usr/bin/env python
#example to run: % mpiexec -n 4 python integral.py 0.0 1.0 10000
import sys,numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def f(x):
    return x*x
```

```

def F(a,b):
    return (1./3)*(b**3 - a**3)

def integrateRange(a, b, n):
    integral = -(f(a) + f(b))/2.0
    # n+1 endpoints, but n trapazoids
    for x in numpy.linspace(a,b,n+1):
        integral = integral + f(x)
    integral = integral* (b-a)/n
    return integral

if __name__ == '__main__':
    #takes in command-line arguments [a,b,n]
    a = float(sys.argv[1])
    b = float(sys.argv[2])
    n = int(sys.argv[3])
    #h is the step size. n is the total number of trapezoids
    h = (b-a)/n
    #local_n is the number of trapezoids each process will calculate
    #note that size must divide n
    local_n = n/size

    #we calculate the interval that each process handles
    #local_a is the starting point and local_b is the endpoint
    local_a = a + rank*local_n*h
    local_b = local_a + local_n*h

    #initializing variables. mpi4py requires that we pass numpy objects.
    integral = numpy.zeros(1)
    total = numpy.zeros(1)

    # perform local computation. Each process integrates its own interval
    start = MPI.Wtime ()
    integral[0] = integrateRange(local_a, local_b, local_n)
    end = MPI.Wtime ()

    # communication
    # root node receives results with a collective "reduce"
    comm.Reduce(integral, total, op=MPI.SUM, root=0)

    # root process prints results
    if comm.rank == 0:
        print "With n =", n, "trapezoids, our estimate of the integral from"\
        , a, "to", b, "is", total
        print "error = ", F(a,b)-total
        print "Wall time = ", end-start

```

Links

1. Mpi4py Home: <http://mpi4py.scipy.org>
2. Another Mpi4py tutorial: <http://jeremybejarano.zzl.org/MPIwithPython/index.html>