MPI for Python

http://mpi4py.scipy.org

Lisandro Dalcin
dalcinl@gmail.com

Centro Internacional de Métodos Computacionales en Ingeniería
Consejo Nacional de Investigaciones Científicas y Técnicas
Santa Fe, Argentina

January, 2011
Python for parallel scientific computing
PASI, Valparaíso, Chile

Outline

What is **mpi4py**?

- Full-featured Python bindings for **MPI**.
- API based on the standard MPI-2 C++ bindings.
- Almost all MPI calls are supported.
  - targeted to MPI-2 implementations.
  - also works with MPI-1 implementations.

Implementation

Implemented with **Cython**

- Code base far easier to write, maintain, and extend.
- Faster than other solutions (mixed Python and C codes).
- A *pythonic* API that runs at C speed !

Features – MPI-1

- Process groups and communication domains.
  - intracommunicators
  - intercommunicators
- Point to point communication.
  - blocking (send/recv)
  - nonblocking (isend/irecv + test/wait)
- Collective operations.
  - Synchronization (barrier)
  - Communication (broadcast, scatter/gather)
  - Global reductions (reduce, scan)

Features – MPI-2

- Dynamic process management (spawn, connect/accept).
- Parallel I/O (read/write).
- One sided operations, a.k.a. RMA (put/get/accumulate).
- Extended collective operations.

Features – Python

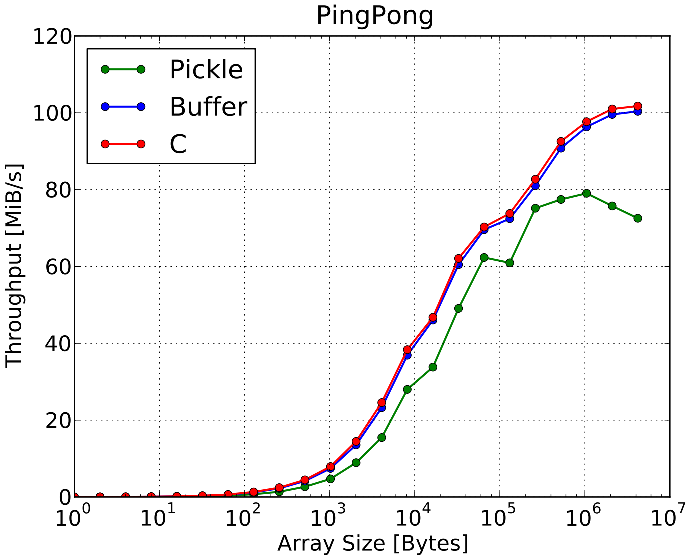
- Communication of Python objects.
  - high level and very convenient, based in `pickle` serialization
  - can be slow for large data (CPU and memory consumming)

```
<object> ⟶ pickle.dump() ⟶ send()
                              ↓
<object> ⟵ pickle.load() ⟵ recv()
```

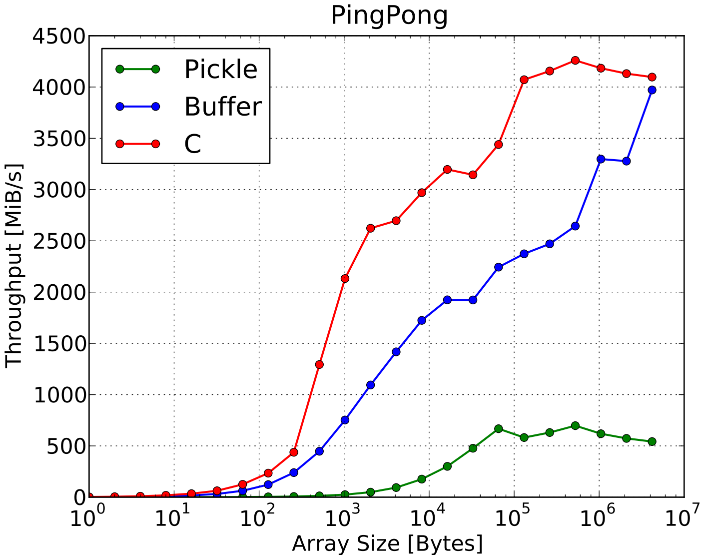
- Communication of array data (e.g. **NumPy** arrays).
  - lower level, slightly more verbose
  - very fast, almost C speed (for messages above 5-10 KB)

```
message = [<object>, (count, displ), datatype]
```

Point to Point Throughput – Gigabit Ethernet



Point to Point Throughput – Shared Memory



Features – IPython

Integration with **IPython** enables MPI to be used *interactively*.

- Start engines with MPI enabled

```
$ ipcluster mpiexec -n 16 --mpi=mpi4py
```

- Connect to the engines

```
$ ipython
In [1]: from IPython.kernel import client
In [2]: mec = client.MultiEngineClient()
In [3]: mec.activate()
```

- Execute commands using %px

```
In [4]: %px from mpi4py import MPI
In [5]: %px print(MPI.Get_processor_name())
```

Features – Interoperability

Good support for wrapping other MPI-based codes.

- You can use **Cython** (`cimport` statement).
- You can use **SWIG** (*typemaps* provided).
- You can use **F2Py** (`py2f()`/`f2py()` methods).
- You can use **Boost::Python** or **hand-written C** extensions.

mpi4py will allow you to use virtually any MPI based C/C++/Fortran code from Python.

Hello World!

```
from mpi4py import MPI

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

print ("Hello, World! "
       "I am process %d of %d on %s" %
       (rank, size, name))
```

Hello World! – Wrapping with **SWIG**

C source

```c
/* file: helloworld.c */
void sayhello(MPI_Comm comm)
{
  int size, rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  printf("Hello, World! "
         "I am process "
         "%d of %d.\n",
         rank, size);
}
```

SWIG interface file

```
// file: helloworld.i
%module helloworld
%{
#include <mpi.h>
#include "helloworld.c"
}%

%include mpi4py/mpi4py.i
%mpi4py_typemap(Comm, MPI_Comm);

void sayhello(MPI_Comm comm);
```

At the Python prompt . . .

```
>>> from mpi4py import MPI
>>> import helloworld
>>> helloworld.sayhello(MPI.COMM_WORLD)
Hello, World! I am process 0 of 1.
```

Hello World! – Wrapping with **Boost.Python**

```
// file: helloworld.cxx
#include <boost/python.hpp>
#include <mpi4py/mpi4py.h>
using namespace boost::python;

#include "helloworld.c"
static void wrap_sayhello(object py_comm) {
  PyObject* py_obj = py_comm.ptr();
  MPI_Comm *comm_p = PyMPIComm_Get(py_obj);
  if (comm_p == NULL) throw_error_already_set();
  sayhello(*comm_p);
}

BOOST_PYTHON_MODULE(helloworld) {
  if (import_mpi4py() < 0) return;
  def("sayhello", wrap_sayhello);
}
```

Hello World! – Wrapping with **F2Py**

Fortran 90 source

```fortran
! file: helloworld.f90
subroutine sayhello(comm)
  use mpi
  implicit none
  integer :: comm, rank, size, ierr
  call MPI_Comm_size(comm, size, ierr)
  call MPI_Comm_rank(comm, rank, ierr)
  print *, 'Hello, World! I am process ',rank,' of ',size,'.'
end subroutine sayhello
```

At the Python prompt . . .

```
>>> from mpi4py import MPI
>>> import helloworld
>>> fcomm = MPI.COMM_WORLD.py2f()
>>> helloworld.sayhello(fcomm)
Hello, World! I am process 0 of 1.
```

Communicators

communicator = process group + communication context

- Predefined instances
  - `COMM_WORLD`
  - `COMM_SELF`
  - `COMM_NULL`
- Accessors
  - `rank = comm.Get_rank() # or comm.rank`
  - `size = comm.Get_size() # or comm.size`
  - `group = comm.Get_group()`
- Constructors
  - `newcomm = comm.Dup()`
  - `newcomm = comm.Create(group)`
  - `newcomm = comm.Split(color, key)`

Communicators – Create()

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
group = comm.Get_group()

newgroup = group.Excl([0])
newcomm = comm.Create(newgroup)

if comm.rank == 0:
    assert newcomm == MPI.COMM_NULL
else:
    assert newcomm.size == comm.size - 1
    assert newcomm.rank == comm.rank - 1

group.Free(); newgroup.Free()
if newcomm: newcomm.Free()
```

Communicators – Split()

```python
from mpi4py import MPI

world_rank = MPI.COMM_WORLD.Get_rank()
world_size = MPI.COMM_WORLD.Get_size()

if world_rank < world_size//2:
    color = 55
    key = -world_rank
else:
    color = 77
    key = +world_rank

newcomm = MPI.COMM_WORLD.Split(color, key)
# ...
newcomm.Free()
```

Exercise #1

a) Create a new process group containing the processes in the group of `COMM_WORLD` with **even** rank. Use the new group to create a new communicator.
   Tip: use `Group.Incl()` or `Group.Range_incl()`

b) Use `Comm.Split()` to split `COMM_WORLD` in two halves.
   - The first half contains the processes with **even** rank in `COMM_WORLD`. The process rank ordering in the new communication is **ascending**.
   - The second half contains the processes with **odd** rank in `COMM_WORLD`. The process rank ordering in the new communication is **descending**.

- Blocking communication
  - Python objects

    ```
    comm.send(obj, dest=0, tag=0)
    obj = comm.recv(None, src=0, tag=0)
    ```

  - Array data

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

- Nonblocking communication
  - Python objects

    ```
    request = comm.isend(object, dest=0, tag=0)}
    request.Wait()
    ```

  - Array data

    ```
    req1 = comm.Isend([array, count, datatype], dest=0, tag=0)
    req2 = comm.Irecv([array, count, datatype], src=0, tag=0)
    MPI.Request.Waitall([req1, req2])}
    ```

PingPong

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
assert comm.size == 2

if comm.rank == 0:
    sendmsg = 777
    comm.send(sendmsg, dest=1, tag=55)
    recvmsg = comm.recv(source=1, tag=77)
else:
    recvmsg = comm.recv(source=0, tag=55)
    sendmsg = "abc"
    comm.send(sendmsg, dest=0, tag=77)
```

PingPing

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
assert comm.size == 2

if comm.rank == 0:
    sendmsg = 777
    target = 1
else:
    sendmsg = "abc"
    target = 0

request = comm.isend(sendmsg, dest=target, tag=77)
recvmsg = comm.recv(source=target, tag=77)
request.Wait()
```

Exchange

```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])
assert lmsg == [left] * 3
assert rmsg == [right] * 3
```

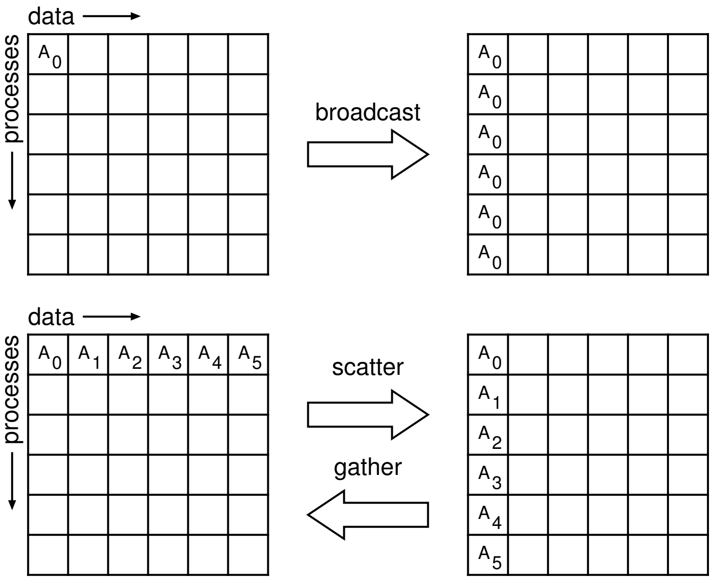
PingPing with NumPy arrays

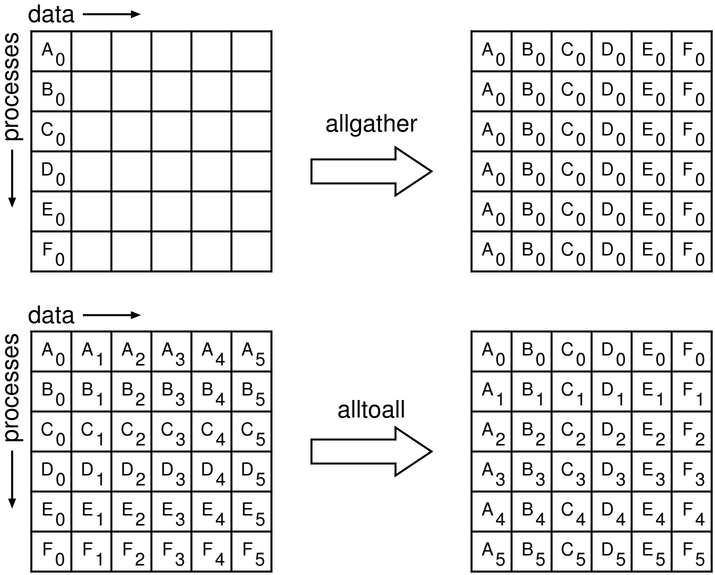
```
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
assert comm.size == 2

if comm.rank == 0:
    array1 = numpy.arange(10000, dtype='f')
    array2 = numpy.empty(10000, dtype='f')
    target = 1
else:
    array1 = numpy.ones(10000, dtype='f')
    array2 = numpy.empty(10000, dtype='f')
    target = 0

request = comm.Isend([array1, MPI.FLOAT], dest=target)
comm.Recv([array2, MPI.FLOAT], source=target)
request.Wait()
```

Exercise #2

a) Modify *PingPong* example to communicate NumPy arrays.
   Tip: use `Comm.Send()` and `Comm.Recv()`

b) Modify *Exchange* example to communicate NumPy arrays.
   Use nonblocking communication for both sending and receiving.
   Tip: use `Comm.Isend()` and `Comm.Irecv()`


data
processes
A_0
broadcast
A_0
A_0
A_0
A_0
A_0
A_0
data
processes
A_0 A_1 A_2 A_3 A_4 A_5
scatter
gather
A_0
A_1
A_2
A_3
A_4
A_5


data
processes
A_0
B_0
C_0
D_0
E_0
F_0
allgather
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
data
processes
A_0 A_1 A_2 A_3 A_4 A_5
B_0 B_1 B_2 B_3 B_4 B_5
C_0 C_1 C_2 C_3 C_4 C_5
D_0 D_1 D_2 D_3 D_4 D_5
E_0 E_1 E_2 E_3 E_4 E_5
F_0 F_1 F_2 F_3 F_4 F_5
alltoall
A_0 B_0 C_0 D_0 E_0 F_0
A_1 B_1 C_1 D_1 E_1 F_1
A_2 B_2 C_2 D_2 E_2 F_2
A_3 B_3 C_3 D_3 E_3 F_3
A_4 B_4 C_4 D_4 E_4 F_4
A_5 B_5 C_5 D_5 E_5 F_5

Broadcast

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.rank == 0:
    sendmsg = (7, "abc", [1.0,2+3j], {3:4})
else:
    sendmsg = None

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

Scatter

```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)
```

Gather & Gather to All

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD

sendmsg = comm.rank**2

recvmsg1 = comm.gather(sendmsg, root=0)

recvmsg2 = comm.allgather(sendmsg)
```

Reduce & Reduce to All

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD

sendmsg = comm.rank

recvmsg1 = comm.reduce(sendmsg, op=MPI.SUM, root=0)

recvmsg2 = comm.allreduce(sendmsg)
```

Exercise #3

a) Modify *Broadcast*, *Scatter*, and *Gather* example to communicate NumPy arrays.

b) Write a routine implementing parallel *matrix–vector* product `y = matvec(comm,A,x)`.

   - the global matrix is dense and square.
   - matrix rows and vector entries have matching block distribution.
   - all processes own the same number of matrix rows.

   Tip: use `Comm.Allgather()` and `numpy.dot()`

Compute Pi

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

Compute Pi – sequential

```
import math

def compute_pi(n):
    h = 1.0 / n
    s = 0.0
    for i in range(n):
        x = h * (i + 0.5)
        s += 4.0 / (1.0 + x**2)
    return s * h

n = 10
pi = compute_pi(n)
error = abs(pi - math.pi)

print ("pi is approximately %.16f, "
       "error is %.16f" % (pi, error))
```

Compute Pi – parallel [1]

```
from mpi4py import MPI
import math

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

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
myrank = comm.Get_rank()
```

Compute Pi – parallel [2]

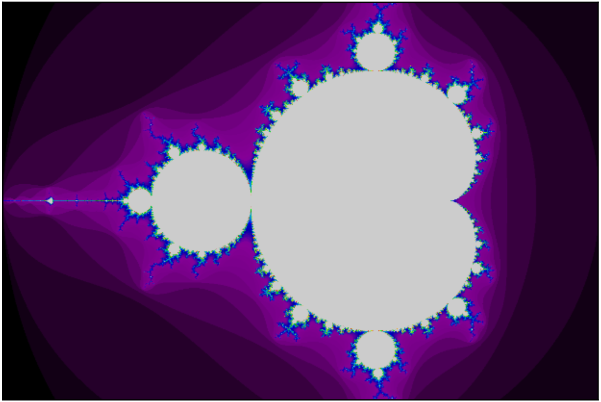
```
if myrank == 0:
    n = 10
else:
    n = None

n = comm.bcast(n, root=0)

mypi = compute_pi(n, myrank, nprocs)

pi = comm.reduce(mypi, op=MPI.SUM, root=0)

if myrank == 0:
    error = abs(pi - math.pi)
    print ("pi is approximately %.16f, "
           "error is %.16f" % (pi, error))
```

Exercise #4

Modify *Compute Pi* example to employ NumPy.

Tip: you can convert a Python `int`/`float` object to a NumPy *scalar* with `x = numpy.array(x)`.

Mandelbrot Set



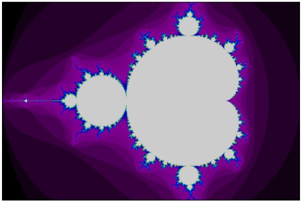
Mandelbrot Set – sequential [1]

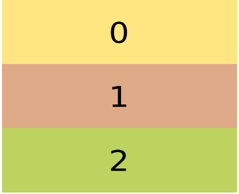
```
def mandelbrot(x, y, maxit):
    c = x + y*1j
    z = 0 + 0j
    it = 0
    while abs(z) < 2 and it < maxit:
        z = z**2 + c
        it += 1
    return it

x1, x2 = -2.0, 1.0
y1, y2 = -1.0, 1.0
w,  h  = 150, 100
maxit  = 127
```

Mandelbrot Set – sequential [2]

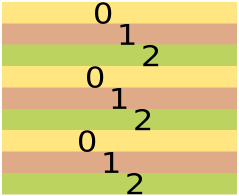
```
import numpy
C = numpy.zeros([h, w], dtype='i')
dx = (x2 - x1) / w
dy = (y2 - y1) / h
for i in range(h):
    y = y1 + i * dy
    for j in range(w):
        x = x1 + j * dx
        C[i, j] = mandelbrot(x, y, maxit)

from matplotlib import pyplot
pyplot.imshow(C, aspect='equal')
pyplot.spectral()
pyplot.show()
```

Mandelbrot Set – partitioning





Block distribution



Cyclic distribution

Mandelbrot Set – parallel, block [1]

```
def mandelbrot(x, y, maxit):
    c = x + y*1j
    z = 0 + 0j
    it = 0
    while abs(z) < 2 and it < maxit:
        z = z**2 + c
        it += 1
    return it

x1, x2 = -2.0, 1.0
y1, y2 = -1.0, 1.0
w,  h  = 150, 100
maxit  = 127
```

Mandelbrot Set – parallel, block [2]

```python
from mpi4py import MPI
import numpy

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

# number of rows to compute here
N = h // size + (h % size > rank)

# first row to compute here
start = comm.scan(N)-N

# array to store local result
Cl = numpy.zeros([N, w], dtype='i')
```

Mandelbrot Set – parallel, block [3]

```
# compute owned rows

dx = (x2 - x1) / w
dy = (y2 - y1) / h
for i in range(N):
    y = y1 + (i + start) * dy
    for j in range(w):
        x = x1 + j * dx
        Cl[i, j] = mandelbrot(x, y, maxit)
```

Mandelbrot Set – parallel, block [4]

```
# gather results at root (process 0)

counts = comm.gather(N, root=0)
C = None
if rank == 0:
    C = numpy.zeros([h, w], dtype='i')

rowtype = MPI.INT.Create_contiguous(w)
rowtype.Commit()

comm.Gatherv(sendbuf=[Cl, MPI.INT],
             recvbuf=[C, (counts, None), rowtype],
             root=0)

rowtype.Free()
```

Mandelbrot Set – parallel, block [5]

```
if comm.rank == 0:
    from matplotlib import pyplot
    pyplot.imshow(C, aspect='equal')
    pyplot.spectral()
    pyplot.show()
```

Exercise #5

Measure the wall clock time T_i of local computations at each process for the *Mandelbrot Set* example with **block** and **cyclic** row distributions.
What row distribution is better regarding load balancing?

Tip: use `Wtime()` to measure wall time, compute the ratio $T_{\max}/T_{\min}$ to compare load balancing.

Dynamic Process Management

- Useful in assembling complex distributed applications. Can couple **independent parallel codes** written in **different languages**.
- Create new processes from a running program.
  – `Comm.Spawn()` and `Comm.Get_parent()`
- Connect two running applications together.
  – `Comm.Connect()` and `Comm.Accept()`

Dynamic Process Management – Spawning

Spawning new processes is a *collective operation* that creates an **intercommunicator**.

- Local group is group of spawning processes (parent).
- Remote group is group of new processes (child).
- `Comm.Spawn()` lets parent processes spawn the child processes. It returns a new intercommunicator.
- `Comm.Get_parent()` lets child processes find intercommunicator to the parent group. Child processes have own `COMM_WORLD`.
- `Comm.Disconnect()` ends the parent–child connection. After that, both groups can continue running.

Dynamic Process Management – Compute Pi (parent)

```
from mpi4py import MPI
import sys, numpy

comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['compute_pi-child.py'],
                           maxprocs=5)

N = numpy.array(10, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE],
            op=MPI.SUM, root=MPI.ROOT)
comm.Disconnect()

error = abs(PI - numpy.pi)
print ("pi is approximately %.16f, "
       "error is %.16f" % (PI, error))
```

Dynamic Process Management – Compute Pi (child)

```python
from mpi4py import MPI
import numpy

comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)
h = 1.0 / N; s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None,
            op=MPI.SUM, root=0)

comm.Disconnect()
```

Exercise #5

a) Implement the *Compute Pi* **child** code in **C** or **C++** . Adjust the parent code in Python to spawn the new implementation.

b) Compute and plot the *Mandelbrot Set* using spawning with parent/child codes implemented in Python.
Tip: Reuse the provided parent code in Python and translate the child code in Fortran 90 to Python.

Do not hesitate to ask for help ...

- Mailing List: `mpi4py@googlegroups.com`
- Mail&Chat: `dalcinl@gmail.com`

Thanks!