## MPI for Python

http://mpi4py.scipy.org

Lisandro Dalcin dalcinl@gmail.com

Extreme Computing Research Center King Abdullah University of Science and Technology Thuwal, Saudi Arabia

March 2, 2017

### Outline

Overview

Hello World!

Point to Point Communication

Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

#### Overview

Hello World

Point to Point Communication

Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

# 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-3 implementations.
  - ▶ also works with MPI-1/2 implementations.

### Implementation

#### Implemented with Cython

- Code base far easier to write, maintain, and extend.
- ▶ Faster than other solutions (mixed Python and C codes).
- Pythonic APIs that runs at C speed!

# Implementation - Cython [1]

```
cdef import from "mpi.h":
    ctypedef void* MPI_Comm
    MPI_Comm MPI_COMM_NULL
    MPI_Comm MPI_COMM_SELF
    MPI_Comm MPI_COMM_WORLD
    int MPI_Comm_size(MPI_Comm,int*)
    int MPI_Comm_rank(MPI_Comm,int*)

cdef inline int CHKERR(int ierr) except -1:
    if ierr != 0:
        raise RuntimeError("MPI error code %d" % ierr)
    return 0
```

# Implementation - Cython [2]

```
cdef class Comm:
    cdef MPI_Comm ob_mpi
    . . .
    def Get_size(self):
        cdef int size
        CHKERR( MPI_Comm_size(self.ob_mpi, &size) )
        return size
    def Get rank(self):
        cdef int rank
        CHKERR( MPI_Comm_rank(self.ob_mpi, &rank) )
        return rank
cdef inline Comm NewComm(MPI_Comm comm_c):
    cdef Comm comm_py = Comm()
    comm_py.ob_mpi = comm_c
    return comm_py
COMM_NULL = NewComm(MPI_COMM_NULL)
COMM_SELF = NewComm(MPI_COMM_SELF)
COMM WORLD = NewComm(MPI COMM WORLD)
```

### 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 – Python

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

```
    send(object)

    object → pickle.dump() → MPI_Send()

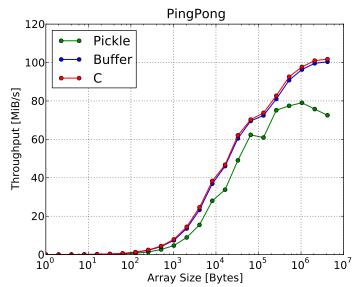
    object = recv()
```

```
object ← pickle.load() ← MPI_Recv()
```

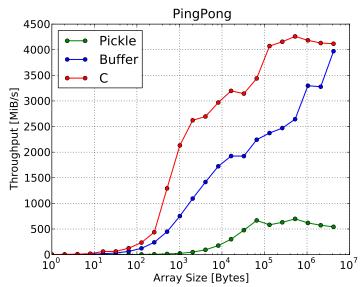
- Communication of array data (e.g. NumPy arrays).
  - lower level, slightly more verbose
  - very fast, almost C speed (for messages above 5-10 KB)

```
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 start -n 16 --engines=MPI
- Connect to the engines
  - \$ ipython
  - In [1]: from ipyparallel import Client
  - In [2]: rc = Client()
- Execute commands using %px
  - In [4]: %px from mpi4py import MPI
  - In [5]: %px print(MPI.Get\_processor\_name())

#### Overview

#### Hello World!

Point to Point Communication

Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

### Hello World! - C

```
#include <mpi.h>
   #include <stdio.h>
3
   int main(int argc, char *argv[]) {
     int rank, size, len;
5
     char name[MPI_MAX_PROCESSOR_NAME];
6
7
     MPI_Init(&argc, &argv);
8
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9
     MPI_Comm_size(MPI_COMM_WORLD, &size);
10
     MPI_Get_processor_name(name, &len);
11
     printf("Hello, World! "
12
             "I am process %d of %d on %s\n",
13
             rank, size, name);
14
     MPI_Finalize();
15
     return 0;
16
17
```

### Hello World! - Fortran 90

```
program main
2
     use mpi
     integer :: ierr, rank, size, len
3
     character (len=MPI_MAX_PROCESSOR_NAME) :: name
4
5
     call MPI_Init(ierr)
6
     call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
     call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
8
     call MPI_Get_processor_name(name, len, ierr)
9
     print*, 'Hello, World!', &
10
           'I am process', rank, ' of', size, &
11
           ' on ', name(1:len)
12
     call MPI_Finalize(ierr)
13
   end program main
14
```

### Hello World! - C++

```
1 #include <mpi.h>
   #include <iostream>
3
   int main(int argc, char *argv[]) {
     MPI::Init();
5
     int rank = MPI::COMM_WORLD.Get_rank();
6
     int size = MPI::COMM_WORLD.Get_size();
7
     int len; char name[MPI_MAX_PROCESSOR_NAME];
8
     MPI::Get_processor_name(name, len);
9
     std::cout << "Hello, World! " <<
10
       "I am process " << rank << " of " << size <<
11
       " on " << name << std::endl:
12
     MPI::Finalize();
13
     return 0;
14
   }
15
```

# Hello World! - Python

```
from mpi4py import MPI

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

ril am process {} of {} on {}"
format(rank, size, name))
```

\$ mpiexec -n 4 python helloworld.py
Hello, World! I am process 0 of 4 on localhost
Hello, World! I am process 1 of 4 on localhost
Hello, World! I am process 2 of 4 on localhost
Hello, World! I am process 3 of 4 on localhost

### Exercise #0

- a) Compile and run the C/Fortran/C++ version of Hello, World!.
- b) Run the Python version of Hello, World!.

#### Overview

Hello World!

#### Point to Point Communication

Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

- 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
4
   if comm.rank == 0:
       sendmsg = 777
6
       comm.send(sendmsg, dest=1, tag=55)
       recvmsg = comm.recv(source=1, tag=77)
8
   else:
       recvmsg = comm.recv(source=0, tag=55)
10
       sendmsg = "abc"
11
       comm.send(sendmsg, dest=0, tag=77)
12
```

# **PingPing**

```
from mpi4py import MPI
   comm = MPI.COMM_WORLD
   assert comm.size == 2
4
   if comm.rank == 0:
       sendmsg = 777
6
       target = 1
  else:
       sendmsg = "abc"
       target = 0
10
11
   request = comm.isend(sendmsg, dest=target, tag=77)
12
   recvmsg = comm.recv(source=target, tag=77)
13
   request.wait()
14
```

## Exchange

```
from mpi4py import MPI
   comm = MPI.COMM_WORLD
3
   sendmsg = [comm.rank]*3
   right = (comm.rank + 1) % comm.size
   left = (comm.rank - 1) % comm.size
7
   req1 = comm.isend(sendmsg, dest=right)
8
   req2 = comm.isend(sendmsg, dest=left)
   lmsg = comm.recv(source=left)
10
   rmsg = comm.recv(source=right)
11
12
   MPI.Request.waitall([req1, req2])
13
   assert lmsg == [left] * 3
14
   assert rmsg == [right] * 3
15
```

# PingPing with NumPy arrays

```
from mpi4py import MPI
   import numpy
2
   comm = MPI.COMM_WORLD
3
  assert comm.size == 2
4
5
   if comm.rank == 0:
       array1 = numpy.arange(10000, dtype='f')
       array2 = numpy.empty(10000, dtype='f')
8
       target = 1
9
   else:
10
       array1 = numpy.ones(10000, dtype='f')
11
       array2 = numpy.empty(10000, dtype='f')
12
       target = 0
13
14
   request = comm.Isend([array1, MPI.FLOAT], dest=target)
15
   comm.Recv([array2, MPI.FLOAT], source=target)
16
   request.Wait()
17
                                         4□ > 4□ > 4 = > 4 = > = 900
```

### Exercise #2

- a) Modify the *PingPong* example to communicate NumPy arrays.Tip: use Comm.Send() and Comm.Recv()
- b) Modify the *Exchange* example to communicate NumPy arrays.

  Use nonblocking communication for both sending and receiving. **Tip**: use Comm.Isend() and Comm.Irecv()

Overview

Hello World

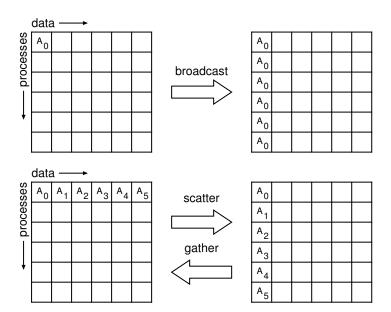
Point to Point Communication

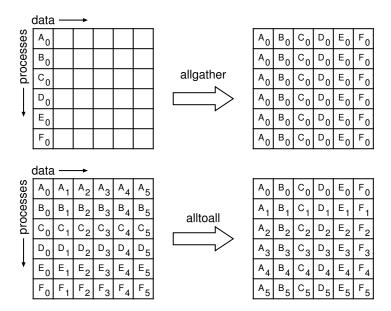
Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management





#### **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

```
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

```
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

```
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 the *Broadcast*, *Scatter*, and *Gather* examples 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()

Overview

Hello World

Point to Point Communication

Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

## 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
2
   def compute_pi(n):
3
       h = 1.0 / n
4
    s = 0.0
5
for i in range(n):
           x = h * (i + 0.5)
7
           s += 4.0 / (1.0 + x**2)
8
       return s * h
9
10
   n = 10
11
   pi = compute_pi(n)
12
   error = abs(pi - math.pi)
13
14
   print ("pi is approximately %.16f, "
15
           "error is %.16f" % (pi, error))
16
```

## Compute Pi – parallel [1]

```
from mpi4py import MPI
   import math
2
3
   def compute_pi(n, start=0, step=1):
       h = 1.0 / n
5
       s = 0.0
6
       for i in range(start, n, step):
7
            x = h * (i + 0.5)
8
            s += 4.0 / (1.0 + x**2)
9
       return s * h
10
11
   comm = MPI.COMM_WORLD
12
   nprocs = comm.Get_size()
13
   myrank = comm.Get_rank()
14
```

## Compute Pi – parallel [2]

```
if myrank == 0:
       n = 10
2
3 else:
   n = None
4
5
   n = comm.bcast(n, root=0)
7
   mypi = compute_pi(n, myrank, nprocs)
8
9
   pi = comm.reduce(mypi, op=MPI.SUM, root=0)
10
11
   if myrank == 0:
12
       error = abs(pi - math.pi)
13
       print ("pi is approximately %.16f, "
14
               "error is %.16f" % (pi, error))
15
```

#### 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).

#### Overview

Hello World

Point to Point Communication

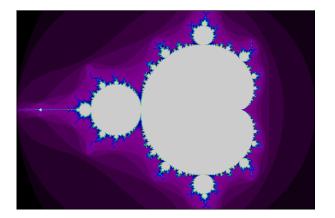
Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

## Mandelbrot Set



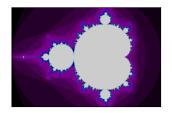
## Mandelbrot Set – sequential [1]

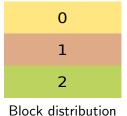
```
def mandelbrot(x, y, maxit):
      c = x + y*1j
2
      z = 0 + 0j
3
      it = 0
4
while abs(z) < 2 and it < maxit:
       z = z**2 + c
6
        it += 1
7
      return it
8
9
   x1, x2 = -2.0, 1.0
10
   y1, y2 = -1.0, 1.0
11
w, h = 150, 100
   maxit = 127
13
```

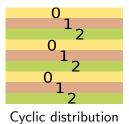
## Mandelbrot Set – sequential [2]

```
import numpy
  C = numpy.zeros([h, w], dtype='i')
_3 dx = (x2 - x1) / w
_{4} dy = (y2 - y1) / h
5 for i in range(h):
       y = y1 + i * dy
6
7
       for j in range(w):
           x = x1 + j * dx
8
           C[i, j] = mandelbrot(x, y, maxit)
9
10
   from matplotlib import pyplot
11
   pyplot.imshow(C, aspect='equal')
12
   pyplot.spectral()
13
14 pyplot.show()
```

#### Mandelbrot Set – partitioning







## Mandelbrot Set – parallel, block [1]

```
def mandelbrot(x, y, maxit):
      c = x + y*1j
2
      z = 0 + 0j
3
      it = 0
4
5 while abs(z) < 2 and it < maxit:</pre>
    z = z**2 + c
6
         it += 1
7
       return it
8
9
   x1, x2 = -2.0, 1.0
10
   y1, y2 = -1.0, 1.0
11
w, h = 150, 100
   maxit = 127
13
```

#### Mandelbrot Set – parallel, block [2]

```
from mpi4py import MPI
   import numpy
3
   comm = MPI.COMM WORLD
   size = comm.Get_size()
   rank = comm.Get_rank()
6
7
   # number of rows to compute here
   N = h // size + (h \% size > rank)
10
   # first row to compute here
11
   start = comm.scan(N)-N
12
13
   # array to store local result
14
   Cl = numpy.zeros([N, w], dtype='i')
15
```

## Mandelbrot Set – parallel, block [3]

```
1  # compute owned rows
2
3  dx = (x2 - x1) / w
4  dy = (y2 - y1) / h
5  for i in range(N):
6     y = y1 + (i + start) * dy
7     for j in range(w):
8          x = x1 + j * dx
9     Cl[i, j] = mandelbrot(x, y, maxit)
```

#### Mandelbrot Set – parallel, block [4]

```
# gather results at root (process 0)
2
   counts = comm.gather(N, root=0)
3
   C = None
   if rank == 0:
       C = numpy.zeros([h, w], dtype='i')
6
7
   rowtype = MPI.INT.Create_contiguous(w)
8
   rowtype.Commit()
9
10
   comm.Gatherv(sendbuf=[Cl, MPI.INT],
11
                 recvbuf=[C, (counts, None), rowtype],
12
                 root=0)
13
14
   rowtype.Free()
15
```

#### Mandelbrot Set – parallel, block [5]

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

#### 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 is the best row distribution regarding load balancing?

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

Overview

Hello World

Point to Point Communication

Collective Operations

Compute Pi

Mandelbrot Set

Dynamic Process Management

#### 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 child processes. It returns a new intercommunicator.
- Comm.Get\_parent() lets child processes get the intercommunicator to the parent group. Child processes have their own COMM\_WORLD.
- ► Comm.Disconnect() ends the parent—child connection. Both groups can continue execution.

#### Dynamic Process Management - Compute Pi (parent)

```
from mpi4py import MPI
1
   import sys, numpy
2
3
   comm = MPI.COMM_SELF.Spawn(sys.executable,
                                args=['compute_pi-child.py'],
5
                                maxprocs=5)
6
7
   N = numpy.array(10, 'i')
   comm.Bcast([N, MPI.INT], root=MPI.ROOT)
   PI = numpy.array(0.0, 'd')
10
   comm.Reduce(None, [PI, MPI.DOUBLE],
11
                op=MPI.SUM, root=MPI.ROOT)
12
   comm.Disconnect()
13
14
   error = abs(PI - numpy.pi)
15
   print ("pi is approximately %.16f, "
16
           "error is %.16f" % (PI, error))
17
```

#### Dynamic Process Management – Compute Pi (child)

```
1
   from mpi4py import MPI
   import numpy
3
   comm = MPI.Comm.Get_parent()
   size = comm.Get_size()
   rank = comm.Get_rank()
6
7
   N = numpy.array(0, dtype='i')
   comm.Bcast([N, MPI.INT], root=0)
   h = 1.0 / N; s = 0.0
10
   for i in range(rank, N, size):
11
        x = h * (i + 0.5)
12
        s += 4.0 / (1.0 + x**2)
13
   PI = numpy.array(s * h, dtype='d')
14
   comm.Reduce([PI, MPI.DOUBLE], None,
15
                op=MPI.SUM, root=0)
16
17
   comm.Disconnect()
18
```

#### Exercise #6

- 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!