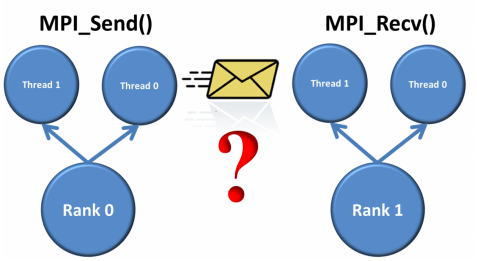
Parallel programming: MPI and a basic introduction to MPI4py

Cecilia Jarne

cecilia.jarne@unq.edu.ar



Message Passing Interface (MPI)

The MPI interface:
Is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances)

Message Passing Interface (MPI)

MPI programs always work with processes:
Programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

Message Passing Interface (MPI)

The following concepts help in understanding and providing context for all of the abilities and help the programmer to decide what functionality to use in their application programs.

- Communicator.
- Point-to-point basics.
- Collective basics.
- Derived datatypes.

Message Passing Interface (MPI)

MPI's prime goals are:

- To provide source-code portability.
- To allow efficient implementation.

It also offers:

- A great deal of functionality.
- Support for heterogeneous parallel architectures.

Message Passing Interface (MPI)

Header files

C:

```
#include <mpi.h>
```

Fortran:

```
include 'mpif.h'
```

Python:

```
from mpi4py import MPI
```

Initialising MPI:

- MPI controls its own internal data structures. An example with C:

```
int MPI_Init(int *argc, char ***argv)
```

- MPI releases 'handles' to allow programmers to refer to these.

Parallel model(MPI)

Communicator *MPI_COMM_WORLD*

In python:

```
comm                    = MPI.COMM_WORLD
```

MPI: Size

How many processes are contained within a communicator?

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

MPI: Rank

How do you identify different processes in a communicator?

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

Indicates the rank of the process that calls it in the range from size-1.

MPI: Exiting MPI

Must be the last MPI procedure called.

```
MPI.Finalize()
```

MPI: Messages

A message contains a number of elements of some particular datatype.
MPI datatypes:

- Basic types.
- Derived types.
- Derived types can be built up from basic types.
- C types are different from Fortran types.
- Python MPI no need to specify

MPI Basic Datatypes

For C:

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

For FORTRAN:

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI: Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator

MPI: Sending a message

Python:

```
comm.send(data, dest=comm)
```

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Fortran:

```

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG,COMM,
      IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG
INTEGER COMM, IERROR
```

MPI: Receiving a message

Python:

```
comm.recv(source=rank_i)
```

C:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *
      status)
```

Fortran:

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE),IERROR
```

MPI: Synchronous Blocking Message-Passing

- Processes synchronise.
- Sender process specifies the synchronous mode.
- Blocking both processes wait until the transaction has completed

MPI: For a communication to succeed...

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.

MPI: Message Order Preservation

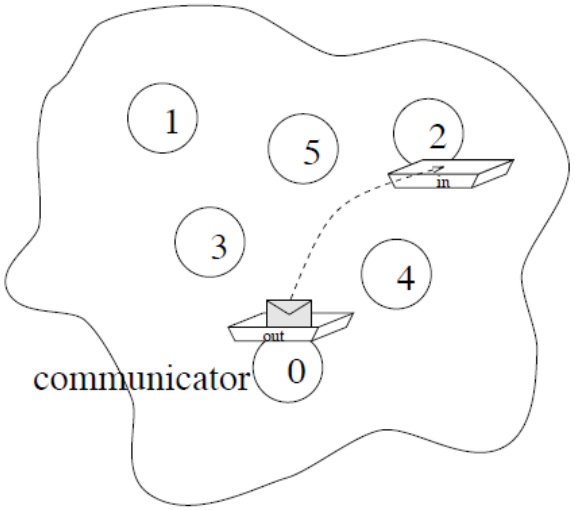
- Messages do not overtake each other.
- This is true even for non-synchronous sends.

MPI: Non-Blocking Communications

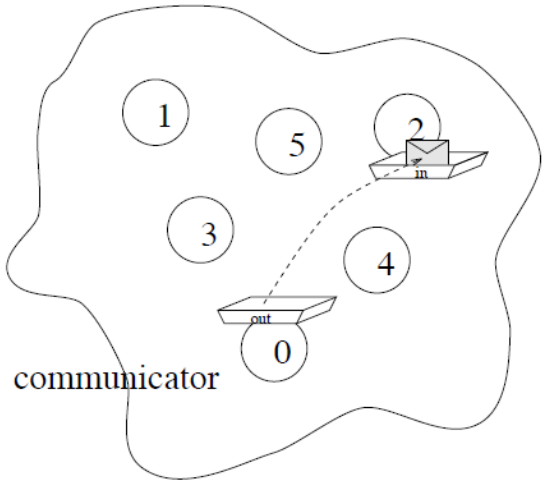
Separate communication into three phases:

- Initiate non-blocking communication.
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete.

MPI: Non-Blocking Send

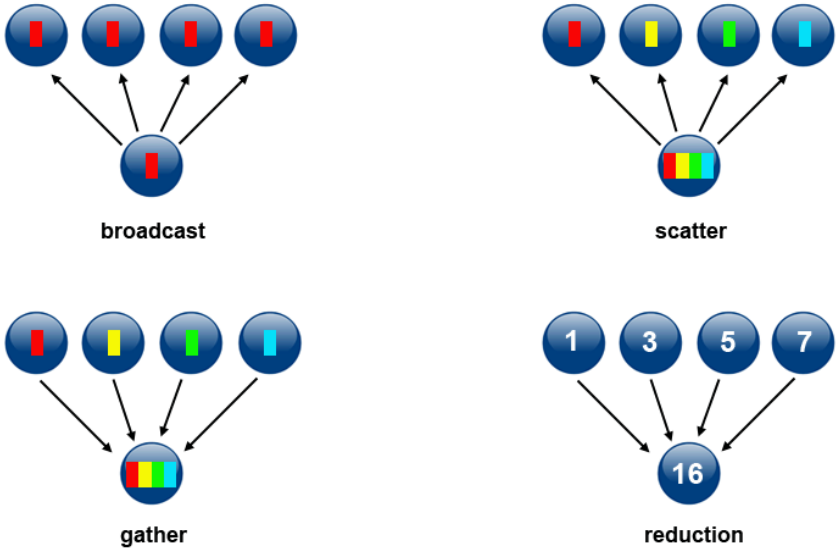


MPI: Non-Blocking Receive



MPI: Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.



MPI: Characteristics of Collective Comms

- Collective action over a communicator.
- All processes must communicate.
- Synchronization may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size.

MPI: Barrier Synchronisation

Python:

```
MPI_Barrier(MPI_Comm)
```

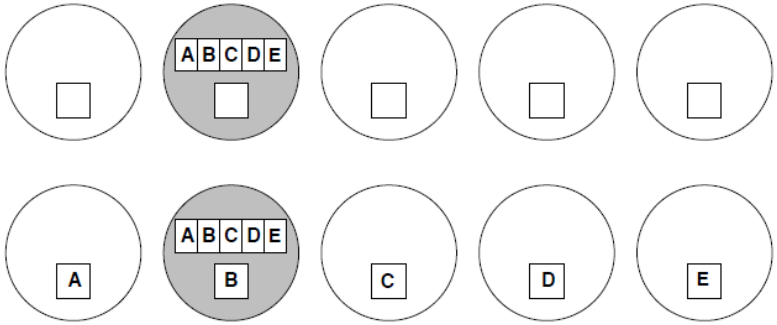
C:

```
int MPI_Barrier(MPI_Comm comm)
```

Fortran:

```
MPI_BARRIER (COMM, IERROR)
INTEGER COMM, IERROR
```

MPI:Scatter



MPI:Scatter

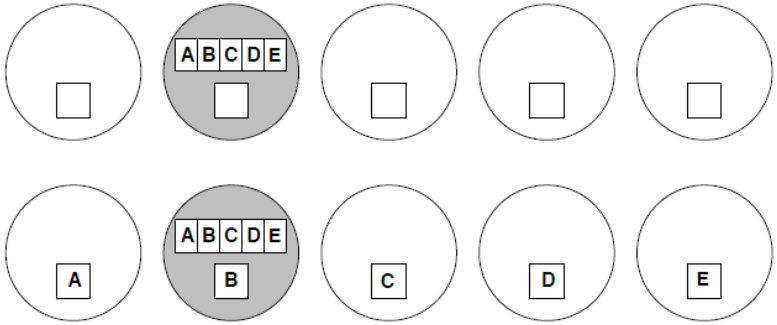
C:

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

Fortran:

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,RECVBUF, RECVCOUNT, RECVTYPE,
    ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

MPI:Gather



MPI:Scatter

C:

```
int MPI_Gather(void *sendbuf, int sendcount,MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype recvtype,int root, MPI_Comm
    comm)
```

Fortran:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,RECVBUF, RECVCOUNT, RECVTYPE,
    ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

MPI:Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  global sum or product
  global maximum or minimum
  global user-defined operation

MPI: Some Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

MPI: MPI_REDUCE

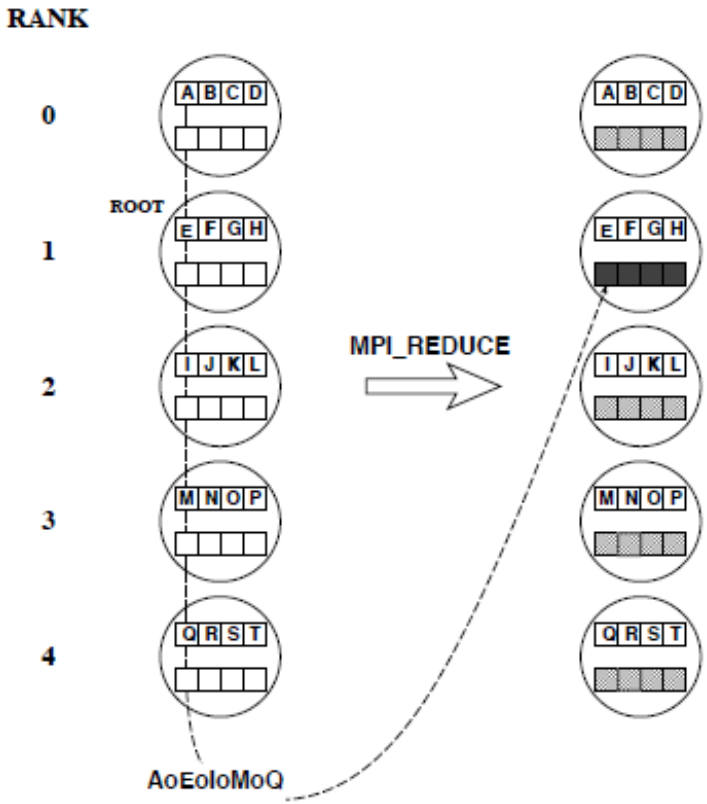
C:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op,int root, MPI_Comm comm)
```

Fortran:

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT,DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

MPI: MPI_REDUCE



What is mpi4py?

- Full-featured Python bindings for MPI.
- API based on the standard MPI-2 C++ bindings.
- Almost all MPI calls are supported.

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

- 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)

WTPC workshop en técnicas de programación científica

Features

- 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

Communication of Python objects.

- High level and very convenient, based in pickle serialization
- Can be slow for large data (CPU and memory consuming)

Communication of array data (e.g. NumPy arrays).

- lower level, slightly more verbose
- very fast, almost C speed (for messages above 5-10 KB)

Features: Intgration with 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())
```

Let's see an example

```
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))
```

How do we run it?

```
mpiexec -n N python hello.py
```

Communicators

communicator = process group + communication context

Prededed 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 Splite()

```
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()
```

Blocking and Nonblocking communication

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])}
```

Example II: Ping Pong:

A Ping pong task between ranks:

```
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()
```

Example III: Exchange

An exchange example:

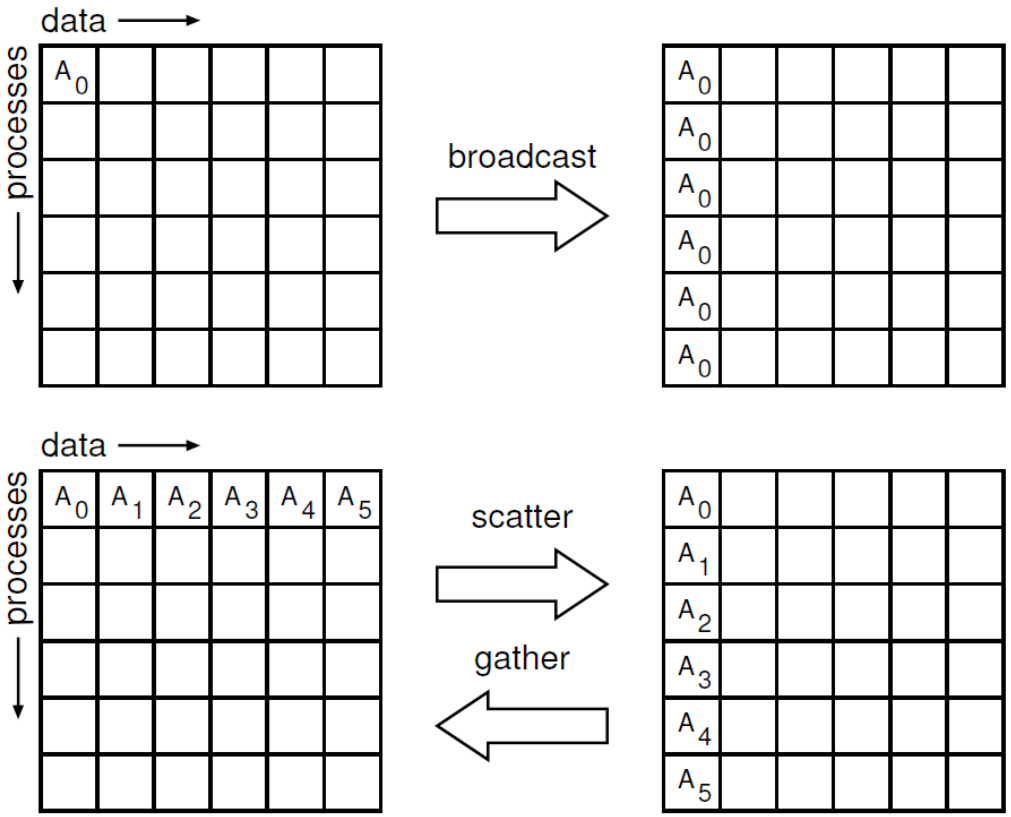
```
from mpi4py import MPI
comm = MPI.COMM_WORLD

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
```

Example IV:

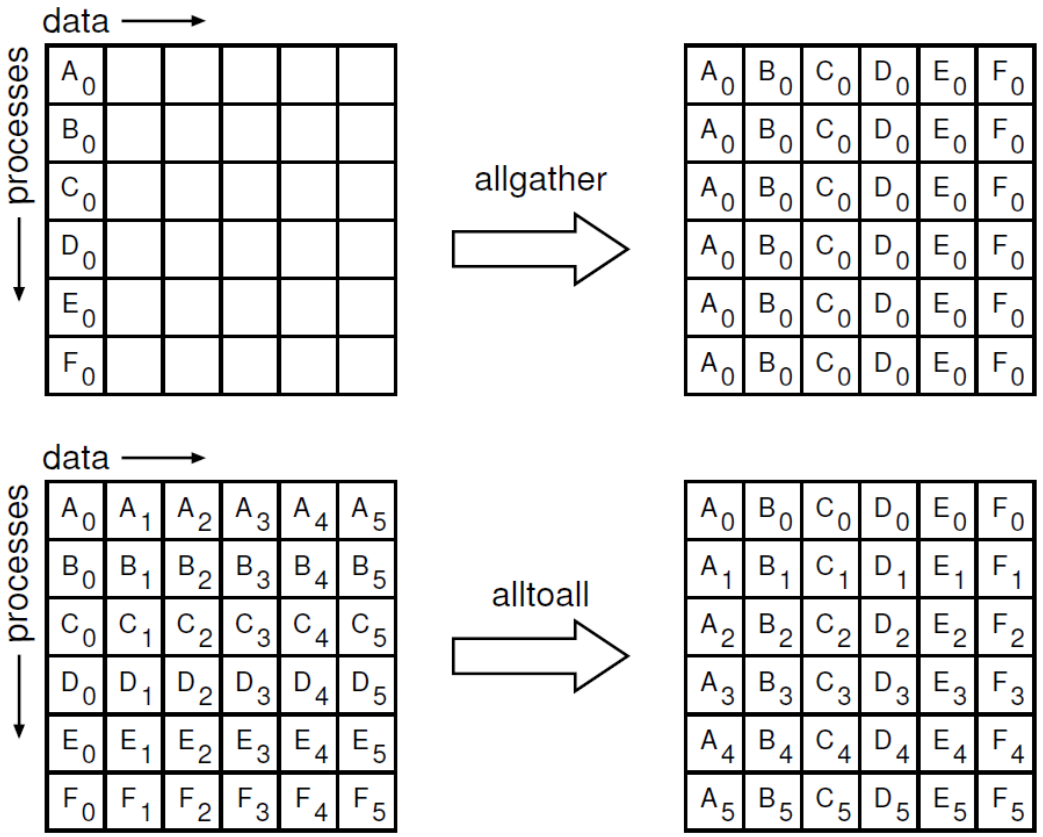
PingPing with NumPy arrays

```
1  from mpi4py import MPI
2  import numpy
3
4  comm = MPI.COMM_WORLD
5  assert comm.size == 2
6
7  if comm.rank == 0:
8      array1 = numpy.arange(10000, dtype='f')
9      array2 = numpy.empty(10000, dtype='f')
10     target = 1
11 else:
12     array1 = numpy.ones(10000, dtype='f')
13     array2 = numpy.empty(10000, dtype='f')
14     target = 0
15
16 request = comm.Isend([array1, MPI.FLOAT], dest=
       target)
17 comm.Recv([array2, MPI.FLOAT], source=target)
18 request.Wait()
```

Collective Operations



Collective Operations



Broadcast

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3
4 if comm.rank == 0:
5     sendmsg = (7, "abc", [1.0,2+3j], {3:4})
6 else:
7     sendmsg = None
8
9 recvmsg = comm.bcast(sendmsg, root=0)
```

Scatter

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3
4 if comm.rank == 0:
5     sendmsg = [i**2 for i in range(comm.size)]
6 else:
7     sendmsg = None
8
9 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)
```

Example V: 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

Compute Pi Sequential

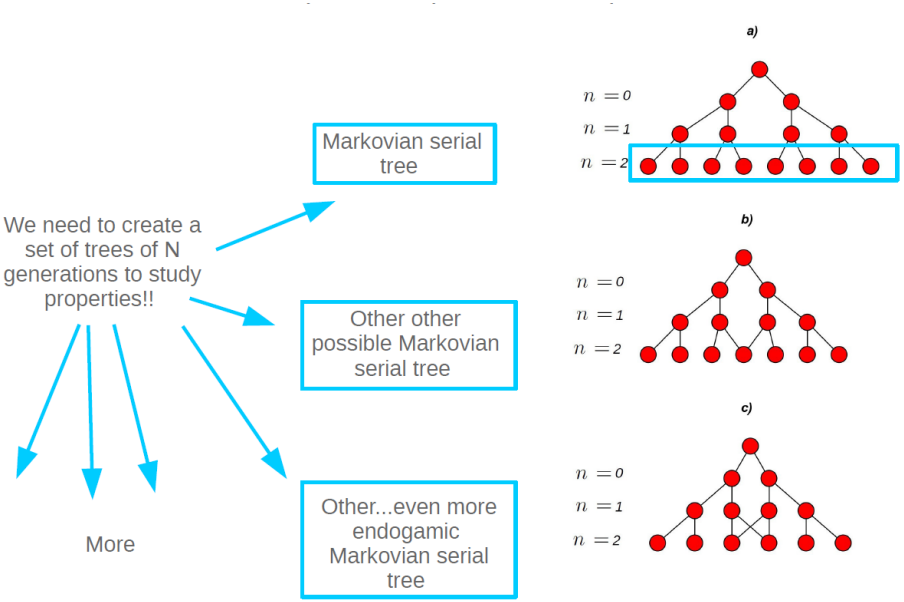
```
import math

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

```
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()
```

One example of code optimization: Tree of ancestors

The problem:



Some steps:

- I installed OpenBLAS library and compile numpy with OpenBLAS integration (not as easy as I expected).
- Then I used MPI for python (I installed the package MPI4py)
- Now the hardest part! I adapted my serial code to used MPI4py.

One example of code optimization

```
1
2 import numpy as np
3 from get_tree import *
4 import scipy
5 import time
6 from itertools import permutations
7 from random import sample
8 from mpi4py import MPI
9
10 start_time = time.time()
11
12 Nlines    = 200
13 color_lvl = 8
14 rgb = np.array(list(permutations(range(0,256,
       color_lvl),3)))/255.0
15 colors = sample(rgb,Nlines)
```

```
1
2 N                = 20   # track x gen back
3 generation_set   = []
4 ances_solo       = []
5 ances_par_impar  = []
6 ances_mariano    = []
7 trees_set        = []
8
9 generation = np.arange(0,N,1)
10 trees      = 50 #number of trees per core
11 generation.tolist()
12 bins       = np.arange(0, N, 1)
13 print 'bins', bins
14 width      = 1.0
```

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

if rank == 0:
  #print 'rank',rank
  ances_solo_0     = []
  ances_par_0      = []
  ances_mariano_0  = []
  trees_set_0      = []
  generation_set_0 = []

  for kk in np.arange(0,trees,1):
    j = get_tree(N,kk)
    generation_set_0.append(j[0])
    ances_solo_0.append(j[1])
    ances_par_impar.append(j[2])
    ances_mariano_0.append(j[3])
    trees_set_0.append(np.c_[j[1],j[0]])
    print"g:\n", j[0],"an:\n",j[1]
```

```
  for j, pepe in enumerate(generation_set_0):
    print 'j',j
    if len(generation_set_0[j])<N:
      agrego= N-len(generation_set_0[j])
      for i in np.arange(agrego):
        generation_set_0[j].append(len(
      generation_set_0[j])+i)
        ances_solo_0[j].append(0)
        ances_par_impar[j].append(0)
        ances_mariano_0[j].append(0)
    else:
        print 'not add'
  ances_solo.extend(ances_solo_0)
  for rank_i in np.arange(1,size,1):
      pepe=comm.recv(source=rank_i)
      ances_solo.extend(pepe)
      print 'recived from:1'
```

```
if rank != 0: #no for needed in each core!
   ances_solo_1     = []
   ances_par_1      = []
   ances_mariano_1  = []
   trees_set_1      = []
   generation_set_1 = []
   print 'rank',rank
   for kk in np.arange(rank*trees,(rank+1)*trees,1):
     j = get_tree(N,kk)
     generation_set_1.append(j[0])
     ances_solo_1.append(j[1])
     ances_par_impar.append(j[2])
     ances_mariano_1.append(j[3])
     trees_set_1.append( np.c_[j[1],j[0]])
     print "gen:\n", j[0],"an:\n", j[1]
    comm.send(ances_solo_1, dest=0)
```

```

 for j, pepe in enumerate(generation_set_1):
  if len(generation_set_1[j])<N:
   agrego= N-len(generation_set_1[j])
   for i in np.arange(agrego):
    generation_set_1[j].append(len(generation_set_1[
      j])+i)
    ances_solo_1[j].append(0)
    ances_par_impar[j].append(0)
    ances_mariano_1[j].append(0)
  else:
   print 'no agrego'
MPI.Finalize()
tam =len(ances_solo)
print'size:',tam,'from rank:',rank
```

```python
if rank == 0:
 media      = np.average(ances_solo, axis=0)
 uncertanty = np.std(ances_solo, axis=0)*float(1)/
      float(trees)
 print 'mean:',media,'uncer: ',uncertanty
 valor_pedido_mariano = []
 valor_pedido_mariano.append(2)
 valor_pedido_mariano.append(4)
 f_out = open('ances.txt', 'w')
 xxx   = np.c_[bins,media,uncertanty]
 np.savetxt(f_out,xxx,fmt='%f  %f  %f',delimiter='\t
      ',header="f  #gen  #ances #error")
 print'gen: ',bins, 'anc: ',media
 print("- %s seconds -" % (time.time() - start_time
      ))
```

```
mpiexec -n N python
    set_of_tree_par_v4.0.py
```

Where N is core number:

Standar result:

Set size: 100

20.6414310932 seconds

Parallel result: (with 2 cores)

Set size: 100

— 13.1909019947 seconds —

Remarks on Message Passing Interface (MPI)

- Standardized and portable message-passing standard.
- designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures.
- The standard defines the syntax and semantics of a core of library routines.
- Useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.
- There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain MPI4PY for exmple.

WTPC workshop en técnicas de programación científica