HPC Python Tutorial: Introduction to MPI4Py

4/23/2012

Instructor:
Yaakoub El Khamra, Research Associate, TACC
yaakoub@tacc.utexas.edu

What is MPI

- Message Passing Interface
- Most useful on distributed memory machines
- Many implementations, interfaces in C/C++/Fortran
- Why python?
  - Great for prototyping
  - Small to medium codes
- Can I use it for production?
  - Yes, if the communication is not very frequent and performance is not the primary concern

Message Passing Paradigm

- 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
- Access to subsets of complex data structures is simplified
  - A data subset is described as a single Data Type entity
- Transfers use synchronous or asynchronous protocols
- Messaging can be arranged into efficient topologies

Key Concepts-- Summary

- Used to create parallel SPMD programs on distributed-memory machines with explicit message passing
- Routines available for
  - Point-to-Point Communication
  - Collective Communication
    - 1-to-many
    - many-to-1
    - many-to-many
  - Data Types
  - Synchronization (barriers, non-blocking MP)
  - Parallel IO
  - Topologies

Advantages of Message Passing

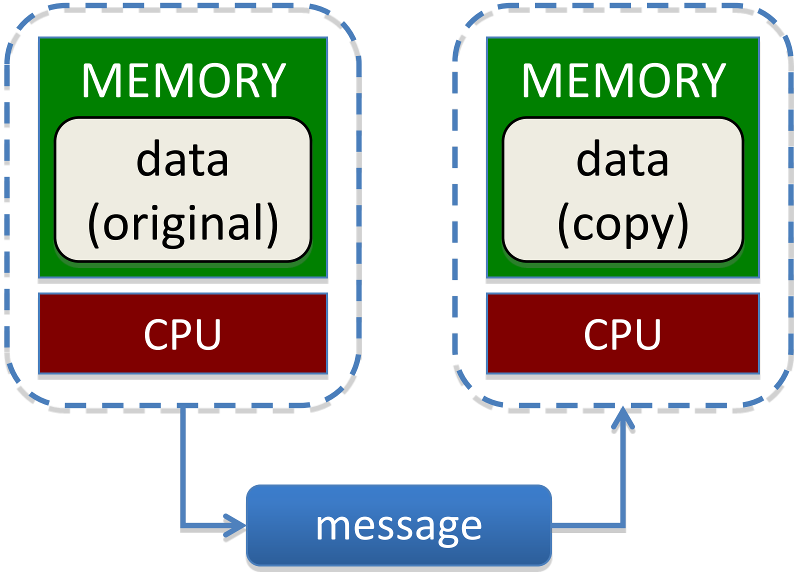
- Universality
  - Message passing model works on separate processors connected by any network (and even on shared memory systems)
  - matches the hardware of most of today's parallel supercomputers as well as ad hoc networks of computers
- Performance/Scalability
  - Scalability is the most compelling reason why message passing will remain a permanent component of HPC (High Performance Computing)
  - As modern systems increase core counts, management of the memory hierarchy (including distributed memory) is the key to extracting the highest performance
  - Each message passing process only directly uses its local data, avoiding complexities of process-shared data, and allowing compilers and cache management hardware to function without contention.

Communicators

- Communicators
  - MPI uses a communicator objects (and groups) to identify a set of processes which communicate only within their set.
  - MPI_COMM_WORLD is defined in the MPI include file as all processes (ranks) of your job
  - Required parameter for most MPI calls
  - You can create subsets of MPI_COMM_WORLD
- Rank
  - Unique *process ID* within a communicator
  - Assigned by the system when the process initializes (for MPI_COMM_WORLD)
  - Processors within a communicator are assigned numbers 0 to n-1 (C/F90)
  - Used to specify sources and destinations of messages, process specific indexing and operations.

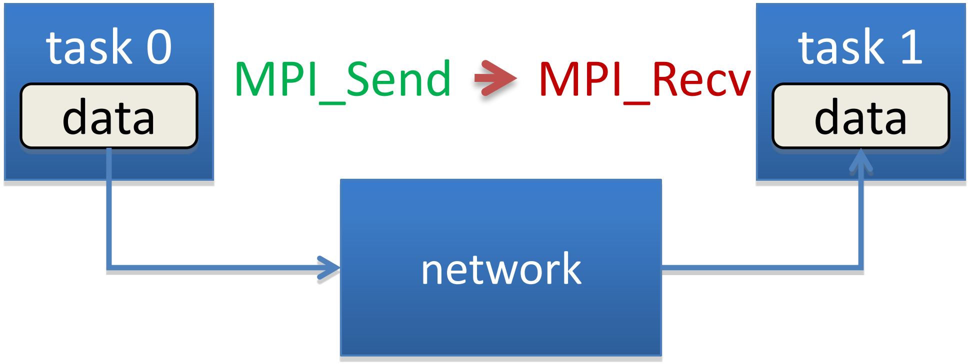
Parallel Code

- The programmer is responsible for determining all parallelism.
  - Data Partitioning
  - Deriving Parallel Algorithms
  - Moving Data between Processes
- Tasks (independent processes executing anywhere) send and receive "messages" to exchange data.
- Data transfer requires cooperative operation to be performed by each process (point to point communications).
- Message Passing Interface (MPI) was released in 1994. (MPI-2 in 1996) Now the MPI is the de facto standard for message passing.
- http://www-unix.mcs.anl.gov/mpi/



Point-to-Point Communication

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another receives



task 0

data

MPI_Send → MPI_Recv

task 1

data

network

Basic Communications in MPI

- Standard MPI_Send/MPI_Recv routines
  - Used for basic messaging

Modes of Operation

- Blocking
  - Call does not return until the data area is safe to use
- Non-blocking
  - Initiates send or receive operation, returns immediately
  - Can check or wait for completion of the operation
  - Data area is not safe to used until completion.
- Synchronous and Buffered (later)

What is available…

- Pypar
  - Its interface is rather minimal. There is no support for communicators or process topologies.
  - It does not require the Python interpreter to be modified or recompiled, but does not permit interactive parallel runs.
  - General (picklable) Python objects of any type can be communicated. There is good support for numeric arrays, practically full MPI bandwidth can be achieved.
- pyMPI
  - It rebuilds the Python interpreter providing a built-in module for message passing. It does permit interactive parallel runs, which are useful for learning and debugging.
  - It provides an interface suitable for basic parallel programing. There is not full support for defining new communicators or process topologies.
  - General (picklable) Python objects can be messaged between processors. There is not support for numeric arrays.
- Scientific Python
  - It provides a collection of Python modules that are useful for scientific computing.
  - There is an interface to MPI and BSP (Bulk Synchronous Parallel programming).
  - The interface is simple but incomplete and does not resemble the MPI specification. There is support for numeric arrays.

MPI4Py

- 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

Functionality

- There are hundreds of functions in the MPI standard, not all of them are necessarily available in MPI4Py, most commonly used are
- No need to call MPI_Init() or MPI_Finalize()
  - MPI_Init() is called when you import the module
  - MPI_Finalize() is called before the Python process ends
- To launch:

```
mpirun –np <number of process> -machinefile <hostlist>
python <my MPI4Py python script>
```

HelloWorld.py

```
# helloworld.py

from mpi4py import MPI
import sys

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

print(“Helloworld! I am process \
 %d of %d on %s.\n” % (rank, size, name))
```

```
Output:
Helloworld! I am process      0 of 4 on Sovereign.
Helloworld! I am process      1 of 4 on Sovereign.
Helloworld! I am process      2 of 4 on Sovereign.
Helloworld! I am process      3 of 4 on Sovereign.
```

Communicators

- COMM_WORLD is available (MPI.COMM_WORLD)
- To get size: MPI.COMM_WORLD.Get_size() or MPI.COMM_WORLD.size
- To get rank: MPI.COMM_WORLD.Get_rank() or MPI.COMM_WORLD.rank
- To get group (MPI Group): MPI.COMM_WORLD.Get_group() . This returns a Group object
  - Group objects can be used with Union(), Intersect(), Difference() to create new groups and new communicators using Create()

More On Communicators

- To duplicate a communicator: Clone() or Dup()
- To split a communicator based on a color and key: Split()
- Virtual topologies are supported!
  - Cartcomm, Graphcomm, Distgraphcomm fully supported
  - Use: Create_cart(), Create_graph()

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

Point-To-Point (cont)

- Blocking: the function return when the buffer is safe to be used
- Send(), Recv(), Sendrecv() can communicate generic Python objects

```
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
```

```
Output:
Message sent, data  is:  {'a': 7, 'b': 3.14}
Message Received, data is:  {'a': 7, 'b': 3.14}
```

Point-To-Point with Numpy

```
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)
```

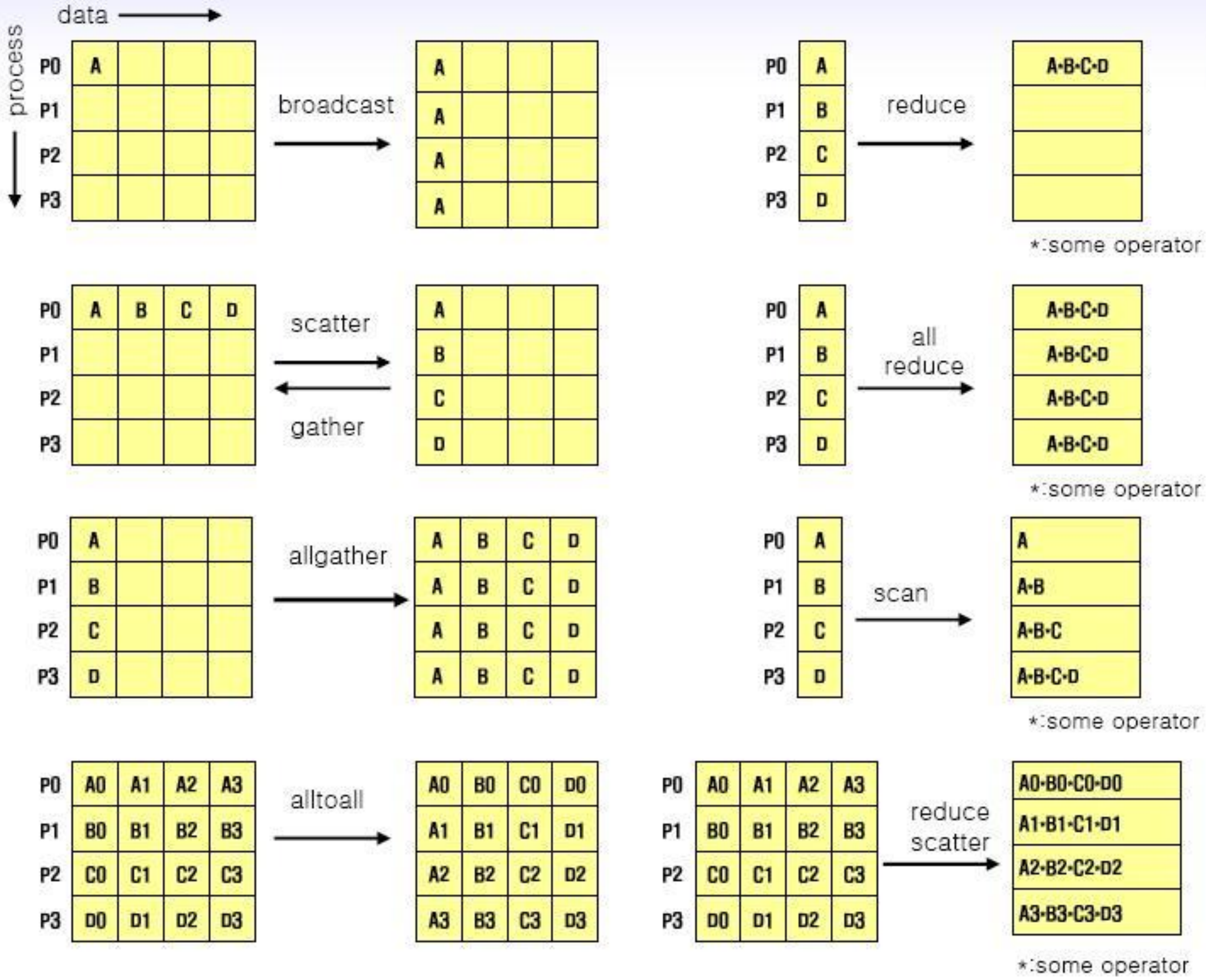
Point-To-Point (cont)

- You can use nonblocking communication to overlap communication with computation
- These functions: Isend() and Irecv() return immediately: the buffers are NOT SAFE for reuse
- You have to Test() or Wait() for the communication to finish
- Optionally you can Cancel() the communication
- Test(), Wait(), Cancel() Operate on the Request object used in the nonblocking function

Collective Communications

- Collective Communications allow multiple processes within the same communicator to exchange messages and possibly perform operations
- Collective Communications are always blocking, there are no tags (organized by calling order)
- Functions perform typical operations such as Broadcast, Scatter, Gather, Reduction and so on

Collective Communication: Summary



broadcast

P0	A			
P1				
P2				
P3				

A			
A			
A			
A			

reduce

P0	A
P1	B
P2	C
P3	D

A*B*C*D

*:some operator

scatter / **gather**

P0	A	B	C	D
P1				
P2				
P3				

A			
B			
C			
D			

all reduce

P0	A
P1	B
P2	C
P3	D

A*B*C*D
A*B*C*D
A*B*C*D
A*B*C*D

*:some operator

allgather

P0	A			
P1	B			
P2	C			
P3	D			

A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

scan

P0	A
P1	B
P2	C
P3	D

A
A*B
A*B*C
A*B*C*D

*:some operator

alltoall

P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

reduce scatter

P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

A0*B0*C0*D0
A1*B1*C1*D1
A2*B2*C2*D2
A3*B3*C3*D3

*:some operator

Collective Communications (cont)

- Bcast(), Scatter(), Gather(), Allgather(), Alltoall() can communicate generic Python objects
- Scatterv(), Gatherv(), Allgatherv() and Alltoallv() can only communicate explicit memory buffers
- No Alltoallw() and no Reduce_scatter()

Bcast() Example

```python
from mpi4py import MPI

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

if rank == 0:
   data = {'key1' : [7, 2.72, 2+3j],
          'key2' : ( 'abc', 'xyz')}
else:
   data = None
data = comm.bcast(data, root=0)
print "bcast finished and data \
 on rank %d is: "%comm.rank, data
```

Output:

```
bcast finished and data on rank 0 is:  {'key2':
('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
bcast finished and data on rank 2 is:  {'key2':
('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
bcast finished and data on rank 3 is:  {'key2':
('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
bcast finished and data on rank 1 is:  {'key2':
('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
```

Scatter() example

```
from mpi4py import MPI

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

if rank == 0:
   data = [(i+1)**2 for i in range(size)]
else:
   data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
print "data on rank %d is: "%comm.rank, data
```

```
Output:
data on rank 0 is:  1
data on rank 1 is:  4
data on rank 2 is:  9
data on rank 3 is:  16
```

Gather() & Barrier()

```python
from mpi4py import MPI

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

data = (rank+1)**2
print "before gather, data on \
  rank %d is: "%rank, data

comm.Barrier()
data = comm.gather(data, root=0)
if rank == 0:
  for i in range(size):
     assert data[i] == (i+1)**2
else:
  assert data is None
print "data on rank: %d is: "%rank, data
```

```
Output:
before gather, data on rank 3 is:  16
before gather, data on rank 0 is:  1
before gather, data on rank 1 is:  4
before gather, data on rank 2 is:  9
data on rank: 1 is:  None
data on rank: 3 is:  None
data on rank: 2 is:  None
data on rank: 0 is:  [1, 4, 9, 16]
```

Advanced Capabilities

- MPI4Py supports dynamic processes through spawning: Spawning(), Connect() and Disconnect()
- MPI4PY supports one sided communication Put(), Get(), Accumulate()
- MPI4Py supports MPI-IO: Open(), Close(), Get_view() and Set_view()

Spawn() and Disconnect()

Pi.py

```
from mpi4py import MPI
import numpy
import sys

print "Spawning MPI processes"
comm =
MPI.COMM_SELF.Spawn(sys.executable,
                    args=[‘Cpi.py'],
                    maxprocs=8)

N = numpy.array(100, '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)

print "Calculated value of PI is: %f16"
%PI
```

Cpi.py

```
#!/usr/bin/env 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)
print "Disconnecting from rank %d"%rank
comm.Barrier()

comm.Disconnect()
```

Output

```
Disconnecting from rank 5
Disconnecting from rank 1
Disconnecting from rank 7
Disconnecting from rank 3
Disconnecting from rank 2
Disconnecting from rank 6
Disconnecting from rank 4
Calculated value of PI is: 3.14160116
Disconnecting from rank 0
```