MPI (Message Passing Interface) & mpi4py

Eero Vainikko

eero.vainikko@ut.ee
6 basic MPI calls

MPI_Init: initialise MPI
MPI_Comm_Size: how many PE?
MPI_Comm_Rank: identify the PE
MPI_Send
MPI_Receive
MPI_Finalise: close MPI

Send, Ssend, Bsend, Rsend - blocking calls
Issend, Issend, Ibsend, Irsend - nonblocking calls

Full range of MPI calls:
http://www.mpich.org/static/docs/latest/

EXAMPLE (fortran90):
http://www.ut.ee/~eero/SC/konspekt/Naited/greetings.f90.html
Example (C):

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD  # Defines the default communicator
num_procs = comm.Get_size()  # Stores the number of processes in size.
rank = comm.Get_rank()  # Stores the rank (pid) of the current process
stat = MPI.Status()

msg = "Hello world, say process %s !", % rank

if rank == 0:
    # Master work
    print(msg
    for i in range(num_procs - 1):
        msg = comm.recv(source=i+1, tag=MPI.ANY_TAG, status=stat)
        print(msg

elif rank == 1:
    # Worker work
    comm.send(msg, dest = 0)
```
Non-Blocking Send and Receive, Avoiding Deadlocks

- Non-Blocking communications allows the separation between the initiation of the communication and the completion
- **Advantages:** between the initiation and completion the program could do other useful computation (latency hiding)
- **Disadvantages:** the programmer has to insert code to check for completion
- Sending objects (pickling underneath)
  - Blocking commands: `send`, `recv`
  - Non-blocking commands: `isend`, `irecv`
    └ Return request object to be able to check the message status
- Sending contiguous memory contents:
  - Blocking commands: `Send`, `Recv`
  - Non-blocking commands: `isend`, `irecv`
    └ Return request object to be able to check the message status

```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) # blocking
elif rank == 1:
    data = comm.recv(source=0, tag=11) # blocking
```

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11) # non-blocking
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11) # non-blocking
    data = req.wait()
```
Unidirectional communication between processors

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

size = MPI.COMM_WORLD.size
rank = MPI.COMM_WORLD.rank
comm = MPI.COMM_WORLD
len = 100

# 1. Blocking send and blocking receive
data = np.arange(len, dtype=float)  # (or similar)
tag = 99

if rank==0:
    print "[0] Sending: ", data
    comm.Send([data, MPI.FLOAT], 1, tag)
elif rank == 1:
    print "[1] Receiving..."
    comm.Recv([data, MPI.FLOAT], 0, tag)
    print "[1] Data: ", data

# 2. Non-blocking send and blocking receive
request = comm.Isend([data, MPI.FLOAT], 1, tag)
... # calculate or do something useful...
request.Wait()

# 3. Blocking send and non-blocking receive
request = comm.Irecv([data, MPI.FLOAT], 0, tag)
... # calculate or do something useful...
request.Wait()
print "[1] Data: ", data
```
# 4. Non-blocking send and non-blocking receive

```python
if rank==0:
    print "[0] Sending: ", data
    request = comm.Isend([data, MPI.FLOAT], 1, tag)
    ... # calculate or do something useful...
    request.Wait()

elif rank == 1:
    print "[1] Receiving..."
    request=comm.Irecv([data, MPI.FLOAT], 0, tag)
    ... # calculate or do something useful...
    request.Wait()

print "[1] Data: ", data
```

Unidirectional communication between processors

Wildcards:

- ANY_SOURCE
- ANY_TAG

Can be used

In * recv
Possibilities for checking received message’s details

```python
# probe.py
from mpi4py import MPI
import numpy

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

if myid == 0:
    data = myid * numpy.ones(5, dtype=numpy.float64)
    comm.Send([data, 3, MPI.DOUBLE], dest=1, tag=1)
if myid == 1:
    info = MPI.Status()
    comm.Probe(MPI.ANY_SOURCE, MPI.ANY_TAG, info)
    count = info.Get_elements(MPI.DOUBLE)
    data = numpy.empty(count, dtype=numpy.float64)
    comm.Recv(data, MPI.ANY_SOURCE, MPI.ANY_TAG, info)
print 'on', myid, 'data:', data
```

```python
# status.py
from mpi4py import MPI
import numpy

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

data = myid * numpy.ones(5, dtype=numpy.float64)

if myid == 0:
    comm.Send([data, 3, MPI.DOUBLE], dest=1, tag=1)
if myid == 1:
    info = MPI.Status()
    comm.Recv(data, MPI.ANY_SOURCE, MPI.ANY_TAG, info)
    source = info.Get_source()
    tag = info.Get_tag()
    count = info.Get_elements(MPI.DOUBLE)
    size = info.Get_count()
    print 'on', myid, 'source, tag, count, size is', source, tag, count, size
```

Mutual communication and avoiding deadlocks

Non-blocking operations can be used also for avoiding deadlocks

Deadlock is a situation where processes wait after each other without any of them able to do anything useful.

Deadlocks can occur:

- caused of false order of send and receive
- caused by system send-buffer fill-up

In case of mutual communication there are 3 possibilities:

1. Both processes start with send followed by receive
2. Both processes start with receive followed by send
3. One process starts with send followed by receive, another vica versa

Depending on blocking there are different possibilities:
# Mutual communication and avoiding deadlocks

## 1. Send followed by receive (vers. 1)

```python
if rank == 0:
    comm.Send([sendbuf, MPI.FLOAT], 1, tag)
    comm.Recv([recvbuf, MPI.FLOAT], 1, tag)
elif rank == 1:
    comm.Send([sendbuf, MPI.FLOAT], 0, tag)
    comm.Recv([recvbuf, MPI.FLOAT], 0, tag)
```

Is this OK?
- OK with small messages only (if sendbuf is smaller than system message send-buffer)

But what about large messages?
- Large messages produce Deadlock!

## 1.1 Send followed by receive (vers. 2)

```python
if rank == 0:
    request = comm.Isend([sendbuf, MPI.FLOAT], 1, tag)
    comm.Recv([recvbuf, MPI.FLOAT], 1, tag)
    request.Wait()
elif rank == 1:
    request = comm.Isend([sendbuf, MPI.FLOAT], 0, tag)
    comm.Recv([recvbuf, MPI.FLOAT], 0, tag)
    request.Wait()
```

Is this deadlock-free?
- It is!

Why `Wait()` cannot follow right after `Isend(...)`?
Mutual communication and avoiding deadlocks

# 2. Receive followed by send (version 1)

```python
if rank == 0:
    comm.Recv([recvbuf, MPI.FLOAT], 1, tag)
    comm.Send([sendbuf, MPI.FLOAT], 1, tag)
elif rank == 1:
    comm.Recv([recvbuf, MPI.FLOAT], 0, tag)
    comm.Send([sendbuf, MPI.FLOAT], 0, tag)
```

... Is this OK?

- No it is not!
  - Produces deadlock in any message buffer size

# 2. Receive followed by send (version 2)

```python
if rank == 0:
    request=comm.Irecv([recvbuf, MPI.FLOAT], 1, tag)
    comm.Send([sendbuf, MPI.FLOAT], 1, tag)
    request.Wait()
elif rank == 1:
    request=comm.Irecv([recvbuf, MPI.FLOAT], 0, tag)
    comm.Send([sendbuf, MPI.FLOAT], 0, tag)
    request.Wait()
```

... deadlock-free?

- Yes, no deadlock!
Mutual communication and avoiding deadlocks

# 3. One starts with Send, the other one with receive

```python
if rank==0:
    comm.Send([sendbuf, MPI.FLOAT], 1, tag)
    comm.Recv([recvbuf, MPI.FLOAT], 1, tag)
else:
    comm.Recv([recvbuf, MPI.FLOAT], 0, tag)
    comm.Send([sendbuf, MPI.FLOAT], 0, tag)
```

… Could we use non-blocking commands instead?

- (Non-blocking commands can be used in whichever call here as well)

# Generally, the following communication pattern is advised:

```python
if rank==0:
    req1=comm.Isend([sendbuf, MPI.FLOAT], 1, tag)
    req2=recomm.Irecv([recvbuf, MPI.FLOAT], 1, tag)
else:
    req1=comm.Isend([sendbuf, MPI.FLOAT], 0, tag)
    req2=comm.Irecv([recvbuf, MPI.FLOAT], 0, tag)
req1.Wait()
req2.Wait()
```

Alternatively, use `comm.Sendrecv`

Docstring:

```
Comm.Sendrecv(self, sendbuf, int dest, int sendtag=0, recvbuf=None, int source=ANY_SOURCE, int recvtag=ANY_TAG, Status status=None)
```

Send and receive a message

.. note:: This function is guaranteed not to deadlock in situations where pairs of blocking sends and receives may deadlock.

.. caution:: A common mistake when using this function is to mismatch the tags with the source and destination ranks, which can result in deadlock.

Type: `builtin_function_or_method`