MPI By Example

Mike Nolta

6 May 2016

What is MPI?

- "MPI" stands for Message Passing Interface.
- It is a portable API for passing messages (i.e., data) between isolated, distributed, & parallel processes.
    - *isolated*: don't share memory
    - *distributed*: running on multiple computers
    - *parallel*: execute simultaneously
- Usually used by C/C++/Fortran codes, but we'll use Python for this class.

Getting setup

Login to SciNet:

```
$ ssh login.scinet.utoronto.ca
```

Login to a GPC devel node:

```
$ ssh gpc
```

Load the following modules:

```
$ module load intel/15.0.2 intelmpi/5.0.3.048 python/2.7.8
```

Running an MPI job

Use `mpirun` to run a job:

```
$ mpirun -np 4 hostname
gpc-f102n001-ib0
gpc-f102n001-ib0
gpc-f102n001-ib0
gpc-f102n001-ib0
```

The `-np 4` option tells `mpirun` to start 4 processes, each of which runs the `hostname` program.

```
$ mpirun [mpirun args] program [program args]
```

First example

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

First example (0)

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

Behind the scenes, this initializes the MPI library, and hooks all the processes together.

First example (1)

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

`comm` is a *communicator*, which is a group of processes.

`COMM_WORLD` is the set of all processes in this job.

First example (2)

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

The *size* of a communicator is the number of processes in the group.

Because we're using the WORLD communicator, this equals the number started by mpirun (i.e., the `-np` option).

First example (3)

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

The *rank* is the process index, running from `0` to `size-1`.

First example (4)

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

Output:

```
$ mpirun -np 3 python 1.py
proc 0/3
proc 2/3
proc 1/3
```

First example (5)

```
#!/usr/bin/env python

from mpi4py import MPI

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

print "proc %d/%d" % (rank, size)
```

Output:

```
$ mpirun -np 3 python 1.py
proc 0/3
proc 2/3
proc 1/3
```

Note the lines are out of order.

Second example

Let's send a message from each process to the next process:

Second example (0)

```
#!/usr/bin/env python

from mpi4py import MPI

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

if rank + 1 < size:
    comm.ssend(rank, dest=rank+1)
if rank - 1 >= 0:
    x = comm.recv(source=rank-1)
    print "proc %d got %d" % (rank, x)
```

Second example (1)

```
#!/usr/bin/env python

from mpi4py import MPI

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

if rank + 1 < size:
    comm.ssend(rank, dest=rank+1)
if rank - 1 >= 0:
    x = comm.recv(source=rank-1)
    print "proc %d got %d" % (rank, x)
```

Send `rank` to the process with `rank+1`.

Second example (2)

```
#!/usr/bin/env python

from mpi4py import MPI

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

if rank + 1 < size:
    comm.ssend(rank, dest=rank+1)
if rank - 1 >= 0:
    x = comm.recv(source=rank-1)
    print "proc %d got %d" % (rank, x)
```

Receive message from process `rank-1`.

Second example (3)

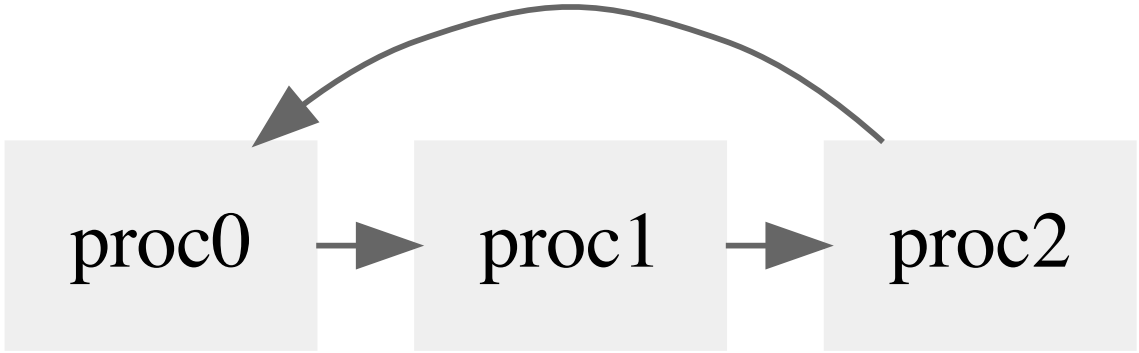
```
#!/usr/bin/env python

from mpi4py import MPI

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

if rank + 1 < size:
    comm.ssend(rank, dest=rank+1)
if rank - 1 >= 0:
    x = comm.recv(source=rank-1)
    print "proc %d got %d" % (rank, x)
```

Output:

```
$ mpirun -np 3 python 2.py
proc 1 got 0
proc 2 got 1
```

Third example

Send messages in a ring:



Third example

```
#!/usr/bin/env python

from mpi4py import MPI

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

comm.ssend(rank, dest=(rank+1)%size)
x = comm.recv(source=(rank-1)%size)
print "proc %d got %d" % (rank, x)
```

Third example (0)

```
#!/usr/bin/env python

from mpi4py import MPI

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

comm.ssend(rank, dest=(rank+1)%size)
x = comm.recv(source=(rank-1)%size)
print "proc %d got %d" % (rank, x)
```

Now all processes send and receive messages.

Third example (1)

```
#!/usr/bin/env python

from mpi4py import MPI

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

comm.ssend(rank, dest=(rank+1)%size)
x = comm.recv(source=(rank-1)%size)
print "proc %d got %d" % (rank, x)
```

Output:

```
$ mpirun -np 3 python 3.py
... waiting ...
```

DEADLOCK 

Why the deadlock?

- The `ssend` command waits for the message to be received before returning.
- In the second example, proc2 didn't send a message and thus was ready to receive proc1's message.
- But in the this example, all processes are waiting for the messages to be received.

One way to fix it -- swap send/recv order on alternate processes:

```
if rank % 2 == 0:
    comm.ssend(rank, dest=(rank+1)%size)
    x = comm.recv(source=(rank-1)%size)
else:
    x = comm.recv(source=(rank-1)%size)
    comm.ssend(rank, dest=(rank+1)%size)
```

Fourth example

```
#!/usr/bin/env python

from mpi4py import MPI

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

s = comm.isend(rank, dest=(rank+1)%size)
x = comm.recv(source=(rank-1)%size)
s.Wait()
print "proc %d got %d" % (rank, x)
```

Fourth example

```
#!/usr/bin/env python

from mpi4py import MPI

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

s = comm.isend(rank, dest=(rank+1)%size)
x = comm.recv(source=(rank-1)%size)
s.Wait()
print "proc %d got %d" % (rank, x)
```

`isend` is *non-blocking* -- doesn't wait for send to be received before returning. It immediately returns a `Request` object.

The `Wait` command waits for the send to complete.

There's also a `Test` command to check whether the send has completed.

Fourth example (1)

```
#!/usr/bin/env python

from mpi4py import MPI

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

s = comm.isend(rank, dest=(rank+1)%size)
x = comm.recv(source=(rank-1)%size)
s.Wait()
print "proc %d got %d" % (rank, x)
```

Output:

```
proc 0 got 2
proc 1 got 0
proc 2 got 1
```

Example Five (numpy arrays)

```
#!/usr/bin/env python

from mpi4py import MPI
import numpy

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

sbuf = numpy.arange(rank, rank+2, dtype='i')
rbuf = numpy.empty(2, dtype='i')

s = comm.Isend(sbuf, dest=(rank+1)%size)
comm.Recv(rbuf, source=(rank-1)%size)
s.Wait()
print "proc %d got %s" % (rank, repr(rbuf))
```

Output:

```
$ mpirun -np 3 python 5.py
proc 0 got array([2, 3], dtype=int32)
proc 1 got array([0, 1], dtype=int32)
proc 2 got array([1, 2], dtype=int32)
```

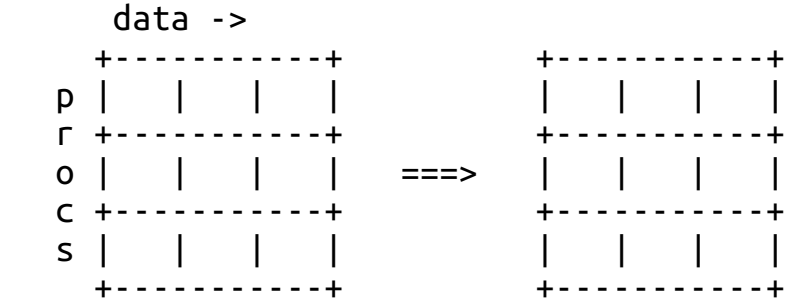
Point-to-point summary

- At its heart, MPI is a point-to-point, two-sided communication protocol.
- Every send *must* be matched with a receive.
- `ssend` blocks until message received.
- `send` may block until received.
- `isend` doesn't block until explicitly waited for.

Collective operations

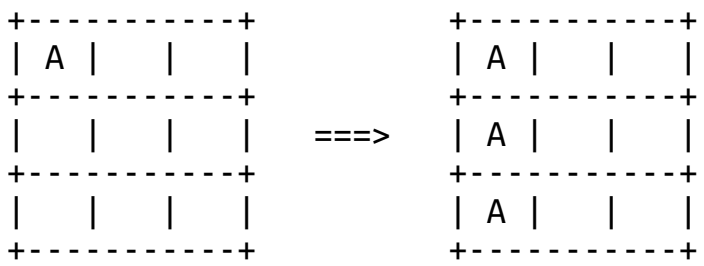
- *broadcast*
- *scatter*
- *gather*
- *alltoall*
- *allgather*

Crummy distributed memory diagrams



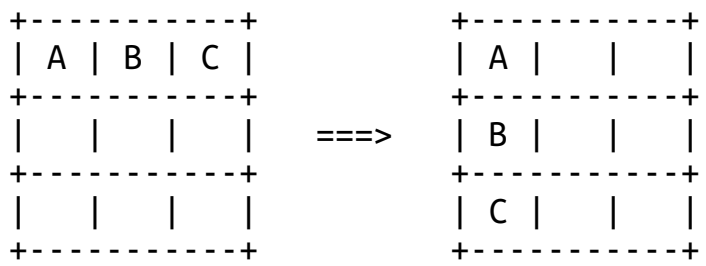
Each row represents a single process, and each cell is a block of memory.

Broadcast



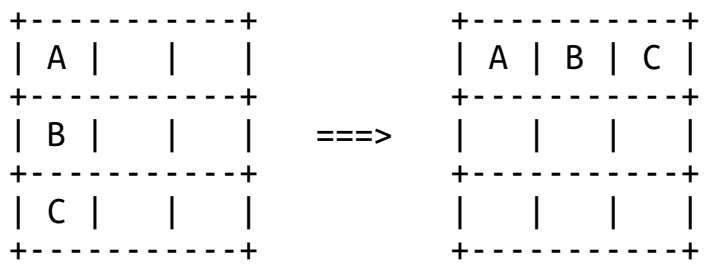

```
data = np.empty(2)
if rank == 0:
    data[:] = range(2)
comm.Bcast(data, root=0)
```

Scatter



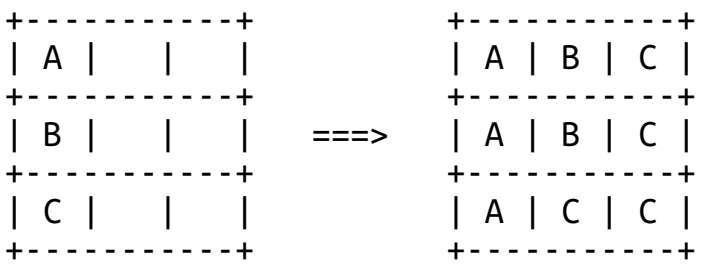

```
if rank == 0:
    sbuf = numpy.empty([size, 2])
    sbuf.T[:,:] = range(size)
else:
    sbuf = None
rbuf = numpy.empty(2)
comm.Scatter(sbuf, rbuf, root=0)
```

Gather




```
sbuf = numpy.empty(2)
sbuf[:] = rank
if rank == 0:
    rbuf = numpy.empty([size, 2])
else:
    rbuf = None
comm.Gather(sbuf, rbuf, root=0)
```

AllGather



AllToAll