MPI By Example

Mike Nolta

2019 May 10

What is MPI?

- "MPI" stands for Message Passing Interface.

- It's a library for passing messages & data between isolated, distributed, & parallel processes.

  - *isolated*: they don't share memory
  - *distributed*: can be running on multiple computers
  - *parallel*: execute simultaneously

- Usually used by C/C++/Fortran codes, but we'll use Python for this lecture (specifically the `mpi4py` module).

Running an MPI job

Use `mpirun` to run a job:

```
$ mpirun -np 4 hostname
nia-login07
nia-login07
nia-login07
nia-login07
```

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

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

print("proc", rank, "of", size)
```

First example (0)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

print("proc", rank, "of", size)
```

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

First example (1)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

print("proc", rank, "of", 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)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

print("proc", rank, "of", 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 python3

from mpi4py import MPI

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

print("proc", rank, "of", size)
```

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

First example (4)

```
#!/usr/bin/env python3

from mpi4py import MPI

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

print("proc", rank, "of", size)
```

Output:

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

First example (5)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

print("proc", rank, "of", size)
```

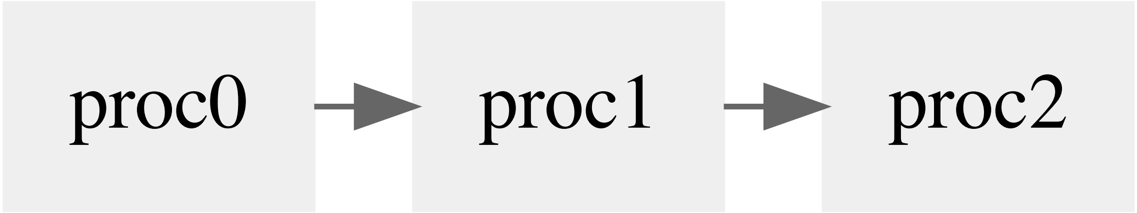
Output:

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

Note the lines can be out of order.

Second example

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



Second example (0)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

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

Second example (1)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

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

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

Second example (2)

```python
#!/usr/bin/env python3

from mpi4py import MPI

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

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

Receive message from process `rank-1`.

Second example (3)

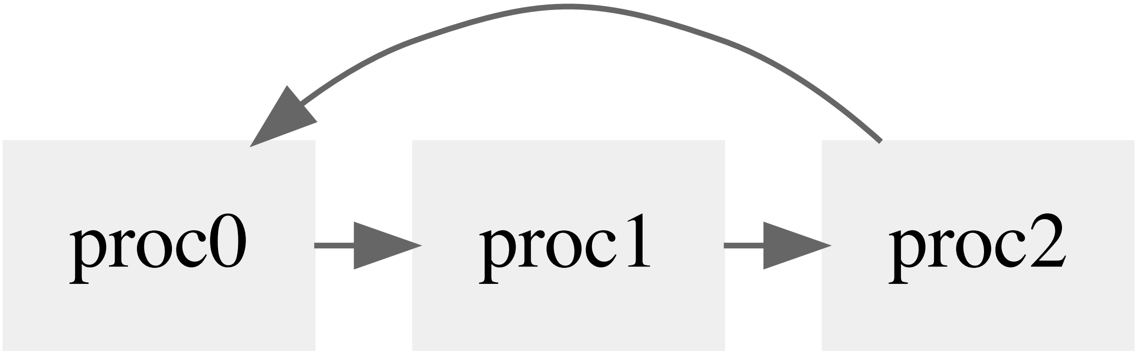
```
#!/usr/bin/env python3

from mpi4py import MPI

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

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

```python
#!/usr/bin/env python3

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", rank, "got", x)
```

Third example (0)

```
#!/usr/bin/env python3

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", rank, "got", x)
```

Now all processes send and receive messages.

Third example (1)

```python
#!/usr/bin/env python3

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", rank, "got", 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 previous example, proc2 didn't send a message and thus was ready to receive proc1's message.
- But in this example, all processes are waiting for the messages to be received.

Fourth example

```
#!/usr/bin/env python3

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", rank, "got", x)
```

Fourth example

```python
#!/usr/bin/env python3

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", rank, "got", 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 python3

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", rank, "got", x)
```

Output:

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

Example Five

```python
#!/usr/bin/env python3

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", rank, "got", 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

- `ssend` blocks until message received (synchronous).
- `isend` doesn't block until explicitly waited for (asynchronous).
- Each send must be matched with one and only one receive.

Collective operations

For convenience and efficiency, MPI has routines for some common communication patterns:

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

Crummy distributed memory diagrams

```
      memory->
    p +-----------+         +-----------+
    r |   |   |   |         |   |   |   |
    o +-----------+   op    +-----------+
    c |   |   |   |  ===>   |   |   |   |
    | +-----------+         +-----------+
    v |   |   |   |         |   |   |   |
      +-----------+         +-----------+
          before                after
```

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

The first block is the state of the memory before, and the second after the operation.

Broadcast

```
+-----------+         +-----------+
| A |   |   |         | A |   |   |
+-----------+         +-----------+
|   |   |   |  ===>   | A |   |   |
+-----------+         +-----------+
|   |   |   |         | A |   |   |
+-----------+         +-----------+
```

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

Scatter

```
+-----------+          +-----------+
| A | B | C |          | A |   |   |
+-----------+          +-----------+
|   |   |   |   ===>   | B |   |   |
+-----------+          +-----------+
|   |   |   |          | C |   |   |
+-----------+          +-----------+
```

Example: distribute the rows of a matrix:

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

Gather

```
+-----------+          +-----------+
| A |   |   |          | A | B | C |
+-----------+          +-----------+
| B |   |   |   ===>   |   |   |   |
+-----------+          +-----------+
| C |   |   |          |   |   |   |
+-----------+          +-----------+
```

Example: inverse of previous example:

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

All Gather

```
+-----------+          +-----------+
| A |   |   |          | A | B | C |
+-----------+          +-----------+
| B |   |   |   ===>   | A | B | C |
+-----------+          +-----------+
| C |   |   |          | A | B | C |
+-----------+          +-----------+
```

All To All

```
+-----------+         +-----------+
| A | B | C |         | A | D | G |
+-----------+         +-----------+
| D | E | F |   ===>  | B | E | H |
+-----------+         +-----------+
| G | H | I |         | C | F | I |
+-----------+         +-----------+
```