Parallel Computing in Python using mpi4py

Stephen Weston

Yale Center for Research Computing
Yale University

June 2017

Parallel computing modules

There are many Python modules available that support parallel computing. See `http://wiki.python.org/moin/ParallelProcessing` for a list, but a number of the projects appear to be dead.

- mpi4py
- multiprocessing
- jug
- Celery
- dispy
- Parallel Python

Notes:

- **multiprocessing** included in the Python distribution since version 2.6
- **Celery** uses different transports/message brokers including RabbitMQ, Redis, Beanstalk
- **IPython** includes parallel computing support
- **Cython** supports use of OpenMP

Multithreading support

Python has supported multithreaded programming since version 1.5.2. However, the C implementation of the Python interpreter (CPython) uses a *Global Interpreter Lock* (GIL) to synchronize the execution of threads. There is a lot of confusion about the GIL, but essentially it prevents you from using multiple threads for parallel computing. Instead, you need to use multiple Python interpreters executing in separate processes.

- For parallel computing, don't use multiple threads: use multiple processes
- The **multiprocessing** module provides an API very similar to the **threading** module that supports parallel computing
- There is no GIL in Jython or IronPython
- **Cython** supports multitheaded programming with the GIL disabled

What is MPI?

- Stands for "Message Passing Interface"
- Standard for message passing library for parallel programs
- MPI-1 standard released in 1994
- Most recent standard is MPI-3.1 (not all implementations support it)
- Enables parallel computing on distributed systems (clusters)
- Influenced by previous systems such as PVM
- Implementations include:
  - Open MPI
  - MPICH
  - Intel MPI Library

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickleable Python objects
- Optimized communicaton of NumPy arrays
- API docs: `http://pythonhosted.org/mpi4py/apiref/index.html`

Installing mpi4py

Easy to install with Anaconda:

```
$ conda create -n mpi mpi4py numpy scipy
```

Already installed on Omega and Grace clusters:

```
$ module load Langs/Python
$ module load Libs/MPI4PY
$ module load Libs/NUMPY
$ module load Libs/SCIPY
```

Minimal mpi4py example

In this mpi4py example every worker displays its rank and the world size:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("%d of %d" % (comm.Get_rank(), comm.Get_size()))
```

Use **mpirun** and **python** to execute this script:

```
$ mpirun -n 4 python script.py
```

Notes:

- MPI_Init is called when mpi4py is imported
- MPI_Finalize is called when the script exits

Running MPI programs with mpirun

MPI distributions normally come with an implementation-specific execution utility.

- Executes program multiple times (SPMD parallel programming)
- Supports multiple nodes
- Integrates with batch queueing systems
- Some implementations use “mpiexec”

Examples:

```
$ mpirun -n 4 python script.py   # on a laptop
$ mpirun --host n01,n02,n03,n04 python script.py
$ mpirun --hostfile hosts.txt python script.py
$ mpirun python script.py        # with batch queueing system
```

Point to point communcations

"send" and "recv" are the most basic communication operations. They're also a bit tricky since they can cause your program to hang.

- comm.send(obj, dest, tag=0)
- comm.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG, status=None)
- "tag" can be used as a filter
- "dest" must be a rank in communicator
- "source" can be a rank or MPI.ANY_SOURCE (wild card)
- "status" used to retrieve information about recv'd message
- These are blocking operations

Example of send and recv

```
from mpi4py import MPI

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

if rank == 0:
    msg = 'Hello, world'
    comm.send(msg, dest=1)
elif rank == 1:
    s = comm.recv()
    print "rank %d: %s" % (rank, s)
```

Ring example

send and recv are blocking operations, so be careful, especially with large objects!

```
s = range(1000000)
src = rank - 1 if rank != 0 else size - 1
dst = rank + 1 if rank != size - 1 else 0

comm.send(s, dest=dst) # This will probably hang
m = comm.recv(source=src)
```

The chain of send's can be broken using:

```
if rank % 2 == 0:
    comm.send(s, dest=dst)
    m = comm.recv(source=src)
else:
    m = comm.recv(source=src)
    comm.send(s, dest=dst)
```

Collective operations

- High level operations
- Support 1-to-many, many-to-1, many-to-many operations
- Must be executed by all processes in specified communicator at the same time
- Convenient and efficient
- Tags not needed
- “root” argument used for 1-to-many and many-to-1 operations

Communicators

- Objects that provide the appropriate scope for all communication operations
- intra-communicators for operations within a group of processes
- inter-communicators for operations between two groups of processes
- MPI.COMM_WORLD is most commonly used communicator

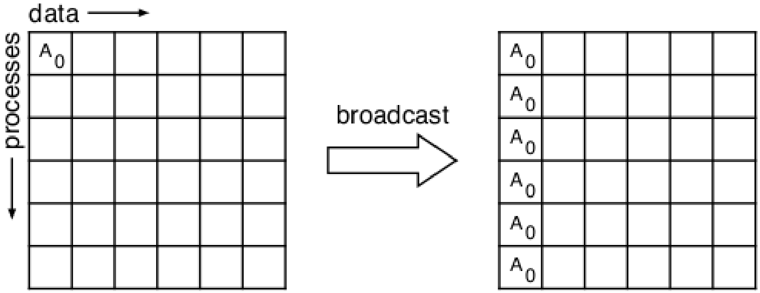
Collectives: Barrier

comm.barrier()

- Synchronization operation
- Every process in communicator group must execute before any can leave
- Try to avoid this if possible

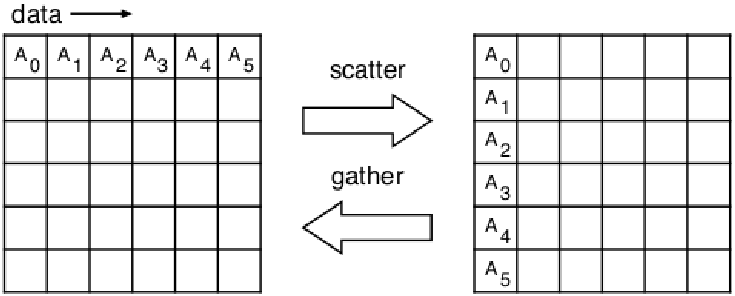
Collectives: Broadcast

comm.bcast(obj, root=0)



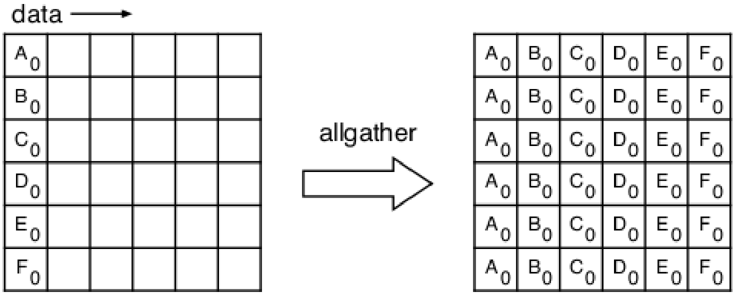
Collectives: Scatter and Gather

comm.scatter(sendobj, root=0) - where sendobj is iterable
comm.gather(sendobj, root=0)



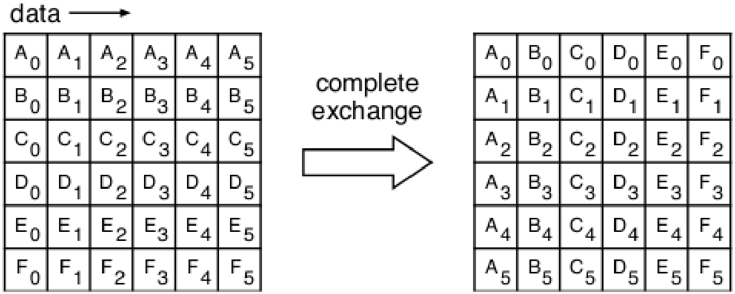
Collectives: All Gather

comm.allgather(sendobj) - where sendobj is iterable



Collectives: All to All

comm.alltoall(sendobj) - where sendobj is iterable


data
complete exchange
A_0 A_1 A_2 A_3 A_4 A_5
B_0 B_1 B_2 B_3 B_4 B_5
C_0 C_1 C_2 C_3 C_4 C_5
D_0 D_1 D_2 D_3 D_4 D_5
E_0 E_1 E_2 E_3 E_4 E_5
F_0 F_1 F_2 F_3 F_4 F_5
A_0 B_0 C_0 D_0 E_0 F_0
A_1 B_1 C_1 D_1 E_1 F_1
A_2 B_2 C_2 D_2 E_2 F_2
A_3 B_3 C_3 D_3 E_3 F_3
A_4 B_4 C_4 D_4 E_4 F_4
A_5 B_5 C_5 D_5 E_5 F_5

Collectives: Reduction operations

```
comm.reduce(sendobj, op=MPI.SUM, root=0)
comm.allreduce(sendobj, op=MPI.SUM)
```

- **reduce** is similar to **gather** but result is “reduced”
- **allreduce** is likewise similar to **allgather**
- MPI reduction operations include:
  - MPI.MAX
  - MPI.MIN
  - MPI.SUM
  - MPI.PROD
  - MPI.LAND
  - MPI.LOR
  - MPI.BAND
  - MPI.BOR
  - MPI.MAXLOC
  - MPI.MINLOC

Sending pickleable Python objects

Generic Python objects can be sent between processes using the “lowercase” communication methods if they can be pickled.

```
import numpy as np
from mpi4py import MPI

def rbind(comm, x):
    return np.vstack(comm.allgather(x))

comm = MPI.COMM_WORLD
x = np.arange(4, dtype=np.int) * comm.Get_rank()
a = rbind(comm, x)
```

Sending buffer-provider objects

Buffer-provider objects can be sent between processes using the “uppercase” communication methods which can be significantly faster.

```
import numpy as np
from mpi4py import MPI

def rbind2(comm, x):
    size = comm.Get_size()
    m = np.zeros((size, len(x)), dtype=np.int)
    comm.Allgather([x, MPI.INT], [m, MPI.INT])
    return m

comm = MPI.COMM_WORLD
x = np.arange(4, dtype=np.int) * comm.Get_rank()
a = rbind2(comm, x)
```

Parallel map

The “map” function can be parallelized

```
x = range(20)
r = map(sqrt, x)
```

The trick is to split “x” into chunks, compute on your chunk, and then combine everybody’s results:

```
m = int(math.ceil(float(len(x)) / size))
x_chunk = x[rank*m:(rank+1)*m]
r_chunk = map(sqrt, x_chunk)
r = comm.allreduce(r_chunk)
```

K-Means Algorithm

```
repeat nstart times
    Randomly select K points from the data set as initial centroids
    do
        Form K clusters by assigning each point to closet centroid
        Recompute the centroid of each cluster
    until centroids do not change
    Compute the quality of the clustering
    if this is the best set of centroids found so far
        Save this set of centroids
    end
end
```

Sequential K-Means using SciPy

```
import numpy as np
from scipy.cluster.vq import kmeans, whiten

obs = whiten(np.genfromtxt('data.csv', dtype=float, delimiter=','))
K = 10
nstart = 100
np.random.seed(0)  # for testing purposes
centroids, distortion = kmeans(obs, K, nstart)
print('Best distortion for %d tries: %f' % (nstart, distortion))
```

K-Means example

```
import numpy as np
from scipy.cluster.vq import kmeans, whiten
from operator import itemgetter
from math import ceil
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank(); size = comm.Get_size()
np.random.seed(seed=rank)  # XXX should use parallel RNG
obs = whiten(np.genfromtxt('data.csv', dtype=float, delimiter=','))
K = 10; nstart = 100
n = int(ceil(float(nstart) / size))
centroids, distortion = kmeans(obs, K, n)
results = comm.gather((centroids, distortion), root=0)
if rank == 0:
    results.sort(key=itemgetter(1))
    result = results[0]
    print('Best distortion for %d tries: %f' % (nstart, result[1]))
```

K-Means example: alternate ending

Instead of sending all of the results to rank 0, we can perform an “allreduce” on the distortion values so that all of the workers know which worker has the best result. Then the winning worker can broadcast its centroids to everyone else.

```
centroids, distortion = kmeans(obs, K, n)
distortion, i = comm.allreduce(distortion, op=MPI.MINLOC)
comm.Bcast([centroids, MPI.FLOAT], root=i)
```