Lab Course: distributed data analytics

01. Message Passing Interface (MPI) for Python

Mohsan Jameel

Information Systems and Machine Learning Lab (ISMLL)
University of Hildesheim, Germany

International Master's Program in Data Analytics
Summer Semester 2018

Outline

1. MPI Basics
2. Point-to-point communication
3. Collective communication

Outline

1. MPI Basics

2. Point-to-point communication

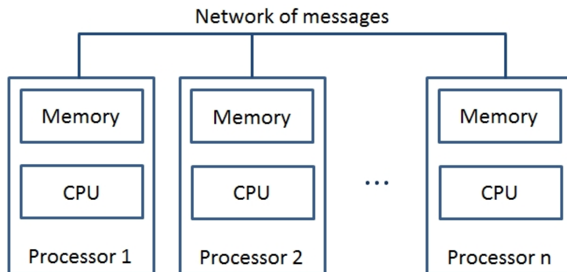
3. Collective communication

The MPI Standard

- The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function on a wide variety of parallel computers.
- Authored by a consortium of academics and industry.
  - MPI 1.0 standard (1994)
  - MPI 2.0 standard (1998)
  - MPI 3.0 standard (2012; 852 pages, 430 functions)
  - MPI 3.1 standard (2015; 868 pages)
  - Currently discussions for MPI 4.0
- MPI docs `http://mpi-forum.org/docs/`
- Basic concepts:
  - Processes run in parallel.
  - Processes synchronize and exchange data by passing messages from one to another.

Message-passing Paradigm

- A parallel program is decomposed into processes, called ranks.
- Each rank holds a portion of the program's data into its private memory.
- Communication among ranks is made explicit through messages.
- All process are launched simultaneously.



MPI for Python

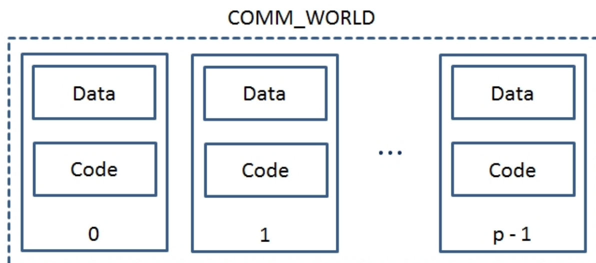
- `mpi4py` is the MPI for Python.
- `mpi4py` provides bindings of the MPI standard for the Python programming language, allowing any Python program to exploit multiple processors.
- `mpi4py` package can be found here: `http://mpi4py.readthedocs.io/en/stable/`.
- This package is constructed on top of the MPI-1/2/3 specifications and provides an object oriented interface which resembles the MPI-2 C++ bindings.
- One can follow the package installation at `http://mpi4py.readthedocs.io/en/stable/install.html`.

MPI for Python - continued

- For anyone who are using Windows, you might set up your environment as follows:
  - Install Anaconda, Python 3.5 `https://www.continuum.io/downloads`
  - Install pip package `https://anaconda.org/anaconda/pip`
  - Install Microsoft MPI `https://www.microsoft.com/en-us/download/details.aspx?id=54607`
    - You need to run both files `msmpisdk.msi` and `MSMpiSetup.exe`
    - Add `$PATH$` in the Environment Variables, e.g. `C:\Program Files (x86)\Microsoft SDKs\MPI`
  - Install `mpi4py` package by `conda install mpi4py`
  - Run a python program by command `mpiexec -n N python your_file.py`
    - N is the number of copies in parallel.

MPI Ranks

- In MPI, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers called ranks.
- Ranks have private memory.
- If we have a number p of processes that runs a program, the processes will have a rank that goes from 0 to $p - 1$.
- Groups of ranks among which a rank can communicate.
- `COMM_WORLD` is a communicator including all ranks in the system.



MPI Ranks- continued

Hello world script helloWorld.py:

```
from mpi4py import MPI
# MPI.Init() not required in python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
name = MPI.Get_processor_name()
print("COMM_WORLD size is {} running on the machine {}".format(size,name))
print ("hello world from process {}".format(rank))
# MPI.Finalize() not required in python
```

- run the command: `mpiexec -n 4 python helloWorld.py`

Outline

1. MPI Basics

2. Point-to-point communication

3. Collective communication

Point-to-point communication

- Point-to-point communication is a mechanism that enables data transmission between two processes: a process `receiver`, and a process `sender`.

- The `mpi4py` module enables point-to-point communication via two functions:

    - `Comm.Send(data, process_destination)`: this sends data to the destination process identified my its rank in the communicator group.

    - `Comm.Recv(process_source)`: this receives data from the source process identified by its rank in the communicator group.

Point-to-point communication - continued

- It is a two-step process:
  - Step 1: sending some data from one task (`sender`)
    - The sending task must specify the data to be sent and its destination.
  - Step 2: receiving these data by another task (`receiver`)
    - The receiving task must specify the source of the message to be received.
- There are two management methods of sending and receiving messages:
  - The buffered mode: the flow control returns to the program as soon as the data to be sent has been copied to a buffer.
  - The synchronous mode: the function gets terminated when the corresponding receive function begins receiving the message.

Point-to-point communication - continued

```
1  from mpi4py import MPI
2
3  comm=MPI.COMM_WORLD
4  rank = comm.rank
5  print("my rank is : " , rank)
6
7  if rank==8:
8      data = 1000
9      destination_process = 7
10     comm.send(data,dest=destination_process)
11     print ("sending data {} to process {}".format(data,destination_process))
12
13 if rank==1:
14     destination_process = 4
15     data = "hello"
16     comm.send(data,dest=destination_process)
17     print ("sending data {} to process {}".format(data, destination_process))
18
19 if rank==4:
20     data=comm.recv(source=1)
21     print ("data received is {}".format(data))
22
23 if rank==7:
24     data=comm.recv(source=8)
25     print ("data1 received is {}".format(data))
26
```

- command: `mpiexec -n 9 python p2p_com.py`

Avoiding deadlock

- Review deadlock from previous lecture.
- How to solve the problem of deadlock in the following code:
- run command: `mpiexec -n 9 python deadLock.py`

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==1:
  data= "a"
  destination_process = 5
  source_process = 5
  data_received=comm.recv(source=source_process)
  comm.send(data,dest=destination_process)
  print ("sending data {} to process {}".format(data,destination_process))
  print ("data received is {}".format(data_received))

if rank==5:
  data= "b"
  destination_process = 1
  source_process = 1
  data_received=comm.recv(source=source_process)
  comm.send(data,dest=destination_process)
  print ("sending data {} to process {}".format(data,destination_process))
  print ("data received is {}".format(data_received))

```

Outline

1. MPI Basics

2. Point-to-point communication

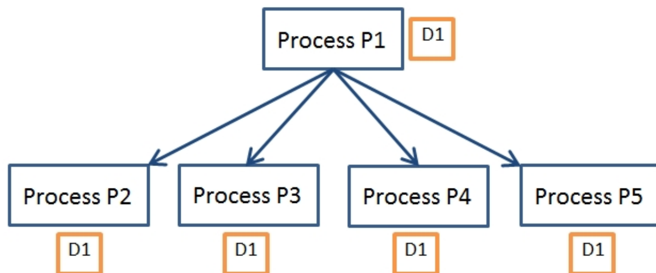
3. Collective communication

Collective communication

- In parallel programming, ones often find themselves in the situation where ones have to share between multiple processes the value of a variable at certain operations.
- Collective communication allows simultaneous data transmission between multiple processes in a group.
- A communication method that involves all the processes belonging to a communicator is called a collective communication.
  - broadcast: bcast / Bcast
  - scatter / Scatter
  - gather / Gather / allgather / Allgather
  - alltoall / Alltoall
  - scan / Scan
  - reduce / Reduce / allreduce / Allreduce

broadcast: bcast

- We call the collective communication broadcast wherein a single process sends the same data to any other processes that belong to the same `comm` communicator.
- Each process must call it by the same values of `root` and `comm`.



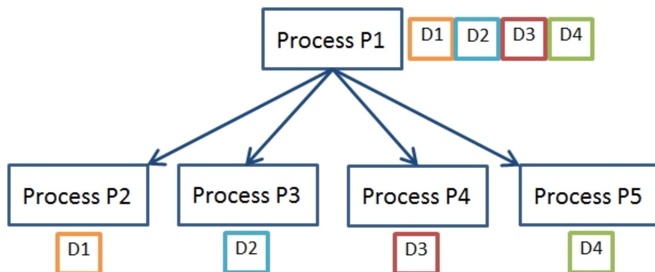
broadcast: bcast - continued

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 5:
    variable_to_share = {"key1":"value1",
                    "key2":[1,2,3.4],
                    "key3":("a","b","c")}
else:
    variable_to_share = None

variable_to_share = comm.bcast(variable_to_share, root=5)
print("process = {}, variable shared = {} ".format(rank,variable_to_share))
```

- Run command: `mpiexec -n 9 python broadcast.py`
- In the example, we have a root process of rank equal to 5 that shares its own data, `variable_to_share`, with the other processes defined in the communicator group `comm`.
- Try to modify the code at `rank == 5` and `root = 5`.

scatter

- The `scatter` function is very similar to a `broadcast` but has one major difference that it sends the chunks of data in an array to different processes.
- The `scatter` function takes elements of a array and distributes them to the processes according to their rank:
  - the first element will be sent to the first process, the second element to the second process, and so on.



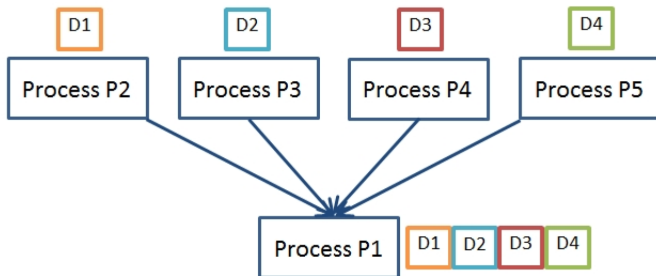
scatter - continued

```
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  size = comm.Get_size()
4  rank = comm.Get_rank()
5
6  if rank == 5:
7      variable_to_share = [(i+1)**2 for i in range(size)]
8  else:
9      variable_to_share = None
10
11 recv = comm.scatter(variable_to_share, root=5)
12 print("process = {}, variable shared = {} ".format(rank,recv))
13
```

- Run command: `mpiexec -n 9 python scatter.py`
- One restriction to `scatter` is that one can scatter as many elements as the processors specified in the execution statement.

gather

- The `gather` function performs the inverse of `scatter`. All processes send data to a root process that collects the data received.



gather - continued

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

data = (rank+1)**2
root = 5
data = comm.gather(data, root=root)

if rank == root:
  print ("process {} is receiving data to other processes".format(rank))
  for i in range(0,comm.Get_size()):
    if i != root:
      data[i] = (i+1)**2
      value = data[i]
      print("process {} receiving {} from process {}".format(rank , value , i))

```

- run command: `mpiexec -n 9 python gather.py`

allgather

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3 size = comm.Get_size()
4 rank = comm.Get_rank()
5
6 data = (rank + 1)**2
7
8 recv = comm.allgather(data)
9 print("rank {} received {}".format(rank,recv))
```

- run command: `mpiexec -n 9 python allgather.py`

alltoall

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = [(i + rank)**2 for i in range(size)]

recv = comm.alltoall(data)
print("rank {} received {}".format(rank,recv))
```

- run command: `mpiexec -n 5 python alltoall.py`

Alltoall

- The `Alltoall` collective communication combines the `scatter` and `gather` functionality.
- run command: `mpiexec -n 5 python Alltoall.py`

```
1  from mpi4py import MPI
2  import numpy
3
4  comm = MPI.COMM_WORLD
5  size = comm.Get_size()
6  rank = comm.Get_rank()
7
8  senddata = (rank+1)*numpy.arange(size,dtype=int)
9  recvdata = numpy.empty(size,dtype=int)
10 comm.Alltoall(senddata,recvdata)
11 print("process {} sending {} receiving {}".format(rank , senddata , recvdata))
12
```

P0	0	1	2	3	4
P1	0	2	4	6	8
P2	0	3	6	9	12
P3	0	4	8	12	16
P4	0	5	10	15	20

Alltoall →

P0	0	0	0	0	0
P1	1	2	3	4	5
P2	2	4	6	8	10
P3	3	6	9	12	15
P4	4	8	12	16	20

reduce

- `reduce` method takes an array of input elements in each process and returns an array of output elements to the root process.
- The output elements contain the reduced result.
- `reduce` method is different from `gather` method by the op parameter, which is the operation that one wish to apply to the return. Some of the common reduction operations defined by MPI are:
  - MPI.MAX returns the maximum element.
  - MPI.MIN returns the minimum elements.
  - MPI.SUM returns the sum up of elements.
  - MPI.PROD returns the multiplication of elements.
  - ...

reduce - continued

```
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  size = comm.Get_size()
4  rank = comm.Get_rank()
5  root = 5
6
7  data = (rank + 1)**2
8
9  recv = comm.reduce(data,root=root)
10 print("rank {} received {}".format(rank,recv))
11
```

- run command: `mpiexec -n 9 python reduce.py`

allreduce

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3 size = comm.Get_size()
4 rank = comm.Get_rank()
5
6 data = (rank + 1)**2
7
8 recv = comm.allreduce(data, op=MPI.SUM)
9 print("rank {} received {}".format(rank,recv))
```

- run command: `mpiexec -n 9 python allreduce.py`

Reduce

```
1  from mpi4py import MPI
2  import numpy
3
4  comm = MPI.COMM_WORLD
5  size = comm.Get_size()
6  rank = comm.Get_rank()
7  root = 3
8  op = MPI.SUM
9
10 senddata = (rank+1)*numpy.arange(size, dtype=int)
11 recvdata = numpy.empty(size, dtype=int)
12 comm.Reduce(senddata, recvdata, root = root, op = op)
13 print("process {} is receiving {} elements after reduce".format(root , recvdata))
14
```

- run command: `mpiexec -n 5 python Reduce.py`

Allreduce

```
1  from mpi4py import MPI
2  import numpy
3
4  comm = MPI.COMM_WORLD
5  size = comm.Get_size()
6  rank = comm.Get_rank()
7  root = 3
8  op = MPI.SUM
9
10 senddata = (rank+1)*numpy.arange(size, dtype=int)
11 recvdata = numpy.empty(size, dtype=int)
12 comm.Allreduce(senddata, recvdata, op = op)
13 print("process {} is receiving {} elements after reduce".format(root , recvdata))
```

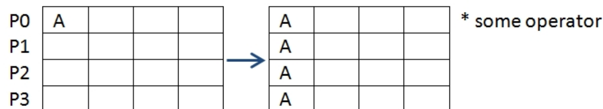
- run command: `mpiexec -n 9 python Allreduce.py`

scan

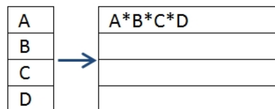
```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3 size = comm.Get_size()
4 rank = comm.Get_rank()
5
6 data = (rank + 1)**2
7
8 recv = comm.scan(data, op=MPI.PROD)
9 print("rank {} received {}".format(rank,recv))
```

- run command: `mpiexec -n 9 python scan.py`

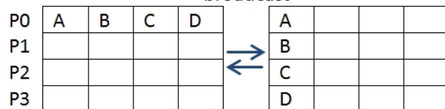
Collective Communication Summary



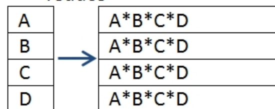
broadcast



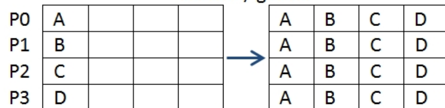
reduce



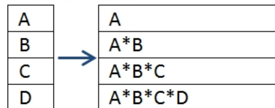
scatter / gather



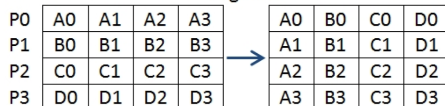
allreduce



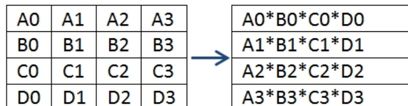
allgather



scan



alltoall



reduce scatter

Further Reading

1. MPI tutorial: https://www.uni-hildesheim.de/learnweb2016/pluginfile.php/89657/mod_resource/content/0/introMPIwithMPI4Py.pdf
2. Dalcin, L. (2012). MPI for Python.
3. Dalcín, L., Paz, R., Storti, M., & D'Elía, J. (2008). MPI for Python: Performance improvements and MPI-2 extensions. Journal of Parallel and Distributed Computing, 68(5), 655-662.
4. Dalcin, L. D., Paz, R. R., Kler, P. A., & Cosimo, A. (2011). Parallel distributed computing using python. Advances in Water Resources, 34(9), 1124-1139.
5. MPI for Python - Release 2.0.0 http://pythonhosted.org/mpi4py/mpi4py.pdf.
6. MPI for Python - User manual http://mpi4py.scipy.org/docs/usrman/index.html.
7. MPI for Python - API reference http://mpi4py.scipy.org/docs/apiref/index.html.
8. A Python Introduction to Parallel Programming with MPI http://materials.jeremybejarano.com/MPIwithPython/.
9. Distributed parallel programming in Python: MPI4PY https://www.howtoforge.com/tutorial/distributed-parallel-programming-python-mpi4py/.
10. MPI tutorial http://mpitutorial.com/tutorials/