Message Passing Interface for Python

MCS 507 Lecture 13
Mathematical, Statistical and Scientific Software
Jan Verschelde, 20 September 2023

Message Passing Interface for Python

Message Passing Interface

MPI = Message Passing Interface
is a standard specification for interprocess communication
for which several implementations exist.

Go to `www.open-mpi.org` for Open MPI,
an open source implementation for MPI.

MPI is a language independent commications protocol
and is the dominant model in high performance computing.

High performance computing became widespread on clusters of
workstations with open source software and commodity hardware.

Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra:
MPI: The Complete Reference. MIT Press, 1995.

Message Passing Interface for Python

MPI for Python provides bindings of MPI for Python,
allowing any Python program to exploit multiple processors.

Features:

- object oriented interface follows closely MPI-2 C++ bindings;
- supports point-to-point and collective communications
  - of any pickable Python object,
  - as well as numpy arrays and builtin bytes, strings.

`mpi4py` gives the standard MPI “look and feel” in Python scripts
to develop parallel programs.

Often, only a small part of the code needs the efficiency of a compiled language. Python handles memory, errors, and user interaction.

Available at `https://github.com/mpi4py`.
Lisandro Dalcin: MPI for Python. Current release 3.1.4, Nov 2022.
Installs well with `pip`, requires working `mpicc`.

writing parallel programs with MPI

Distributed memory parallel programming:

- All processes execute the *same* program.
- Every process has a unique identification number: its rank.
- Branch on the rank to determine which process executes what.

Types of Network Communications:

- point-to-point or collective
  - point-to-point: every send/receive has matching receive/send,
  - collective: every process participates in the communication.
- blocking or nonblocking
  - blocking: sender/receiver waits till receive/send done,
  - nonblocking: immediate send, probe if a message arrived.

Common manager-worker model:

- manager (process with rank 0) distributes jobs among the workers,
- workers (processes with rank > 0) execute jobs.

Message Passing Interface for Python

hello world with mpi4py

```
from mpi4py import MPI

SIZE = MPI.COMM_WORLD.Get_size()
RANK = MPI.COMM_WORLD.Get_rank()
NAME = MPI.Get_processor_name()

MESSAGE = "Hello from %d of %d on %s." \
  % (RANK, SIZE, NAME)
print(MESSAGE)
```

running the script

Programs that run with MPI are executed with `mpiexec`.

To run `mpi4py_hello_world.py` by 3 processes:

```
$ mpiexec -n 3 python3 mpi4py_hello_world.py
Hello from 0 of 3 on pascal.math.uic.edu.
Hello from 1 of 3 on pascal.math.uic.edu.
Hello from 2 of 3 on pascal.math.uic.edu.
$
```

Three Python interpreters are launched.

Each interpreter executes the script,
printing the hello message.

some basic MPI concepts and commands

`MPI.COMM_WORLD` is a predefined intracommunicator.
An intracommunicator is a group of processes.
All processes within an intracommunicator have a unique number.

Methods of the intracommunicator `MPI.COMM_WORLD`:

- `Get_size()` returns the number of processes.
- `Get_rank()` returns rank of executing process.

Even though every process runs the same script,
the test `if MPI.COMM_WORLD.Get_rank() == i:`
allows to specify particular code for the *i*-th process.

`MPI.Get_processor_name()`
returns the name of the calling processor.

Message Passing Interface for Python

send and receive

Process 0 sends `DATA` to process 1:

```
    MPI.COMM_WORLD.send(DATA, dest=1, tag=2)
```

Every `send` must have a matching `recv`.

For the script to continue, process 1 must do

```
    DATA = MPI.COMM_WORLD.recv(source=0, tag=2)
```

`mpi4py` uses `pickle` on Python objects.

The user can declare the MPI types explicitly.

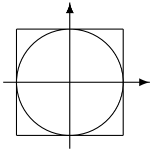
mpi4py_point2point.py

```
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)
    print(RANK, 'sends', DATA, 'to 1')
elif(RANK == 1):
    DATA = COMM.recv(source=0, tag=11)
    print(RANK, 'received', DATA, 'from 0')
```

```
$ mpiexec -n 2 python3 mpi4py_point2point.py
0 sends {'a': 7, 'b': 3.14} to 1
1 received {'a': 7, 'b': 3.14} from 0
```

a pleasingly parallel computation

The area of the unit disk is π:



Generate random uniformly distributed points with coordinates $(x, y) \in [-1, +1] \times [-1, +1]$.
We count a success when $x^2 + y^2 \leq 1$.

1. generate n points P in $[0, 1] \times [0, 1]$
2. $m := \#\{ (x, y) \in P : x^2 + y^2 \leq 1 \}$
3. the estimate is then $4 \times m/n$

With two processes:

- process i uses i as seed for random numbers;
- at end, process 1 sends to 0; process 0 adds up.

estimating π

```
import random
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

print(RANK, 'uses seed', RANK)
random.seed(RANK)

N = 10**7
k = 0
for i in range(0, N):
    x = random.uniform(0, 1)
    y = random.uniform(0, 1)
    if x**2 + y**2 <= 1:
        k = k + 1
R = float(k)/N
```

point-to-point communication in script

```
print(RANK, 'computes', R)

if(RANK == 1):
    COMM.send(R, dest=0, tag=11)
    print(RANK, 'sends', R, 'to 0')
elif(RANK == 0):
    S = COMM.recv(source=1, tag=11)
    print(RANK, 'received', S, 'from 1')
    RESULT = 2*(R + S)
    print('approximation for pi =', RESULT)
```

executing the script

```
$ mpiexec -n 2 python3 mpi4py_estimate_pi_2.py
0 uses seed 0
1 uses seed 1
0 computes 0.7853257
1 computes 0.7852532
1 sends 0.7852532 to 0
0 received 0.7852532 from 1
approximation for pi = 3.1411578
$
```

modification for any number of processes

Each worker sends an estimate.
The manager executes as many `recv` as there are workers.

```
SIZE = COMM.Get_size()

if(RANK > 0):
    COMM.send(R, dest=0, tag=11)
    print(RANK, 'sends', R, 'to 0')
elif(RANK == 0):
    R2 = 0
    for i in range(1, SIZE):
        S = COMM.recv(source=i, tag=11)
        print(RANK, 'received', S, 'from', i)
        R2 = R2 + S
    RESULT = 4*(R + R2)/SIZE
    print('approximation for pi =', RESULT)
```

Quality Up

To improve the quality of the computations:

- leave the number of samples fixed, and
- double the number of processes in each stage.

How many more correct decimal places in each stage?

Running on 12-core 3.49Ghz workstation, with p processes:

p	estimate
2	3.1411578
4	3.1413254
8	3.1413843
16	3.1415143
32	3.1415194
64	3.1415486

With $p = 64$, more than 21GB in use . . .

Message Passing Interface for Python

broadcasting data

A collective communication involves every process in the intracommunicator.

A broadcast is a collective communication in which

- one process sends the same data to all processes,
- all processes receive the same data.

In `mpi4py`, a broadcast is done with the `bcast` method.

An example:

```
$ mpiexec -n 3 python mpi4py_broadcast.py
0 has data {'pi': 3.1415926535897, 'e': 2.7182818284590}
1 has data {'pi': 3.1415926535897, 'e': 2.7182818284590}
2 has data {'pi': 3.1415926535897, 'e': 2.7182818284590}
$
```

the script `mpi4py_broadcast.py`

```
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if(RANK == 0):
    DATA = {'e' : 2.7182818284590451,
            'pi' : 3.1415926535897931 }
else:
    DATA = None # DATA must be defined

DATA = COMM.bcast(DATA, root=0)
print(RANK, 'has data', DATA)
```

scattering data

A scatter is a collective communication in which data are distributed among the processes which belong to the same intracommunicator.

Scattering the number $(i+1)^2$ to process i:

```
$ mpiexec -n 10 python mpi4py_scatter.py
9 has data 100
0 has data 1
1 has data 4
2 has data 9
3 has data 16
4 has data 25
5 has data 36
8 has data 81
6 has data 49
7 has data 64
```

the script `mpi4py_scatter.py`

```
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()

if(RANK == 0):
    DATA = [i**2 for i in range(1, SIZE+1)]
else:
    DATA = None # DATA must be defined

DATA = COMM.scatter(DATA, root=0)
print(RANK, 'has data', DATA)
```

gathering data

A gather is a collective communication in which data is sent from all processes in the same intracommunicator to one process.

An example: process i has data $(i+1)^2$.
Process 0 gather the data from all processes into a list.

```
$ mpiexec -n 10 python mpi4py_gather.py
0 has data [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

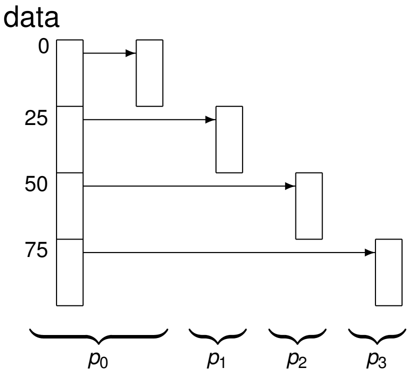
the script `mpi4py_gather.py`

```
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()

DATA = (RANK+1)**2
DATA = COMM.gather(DATA, root=0)
if(RANK == 0):
    print(RANK, 'has data', DATA)
```

Message Passing Interface for Python

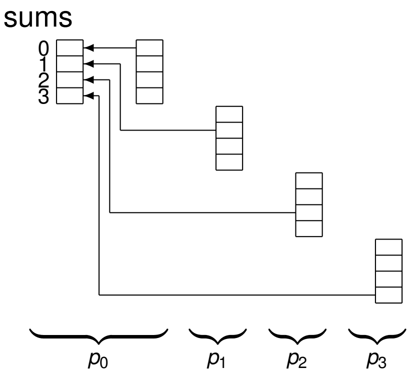
scattering data

Scattering an array of 100 numbers over 4 processors:



gathering results

Gathering the partial sums at the 4 processors to the root:



a parallel sum

To sum an array of numbers, we distribute the numbers among the processes who compute the sum of a slice. The sums of the slices are sent to process 0 who computes the total sum.

```
$ mpiexec -n 10 python mpi4py_parallel_sum.py
0 has data [0 1 2 3 4 5 6 7 8 9] sum = 45
2 has data [20 21 22 23 24 25 26 27 28 29] sum = 245
3 has data [30 31 32 33 34 35 36 37 38 39] sum = 345
4 has data [40 41 42 43 44 45 46 47 48 49] sum = 445
5 has data [50 51 52 53 54 55 56 57 58 59] sum = 545
1 has data [10 11 12 13 14 15 16 17 18 19] sum = 145
8 has data [80 81 82 83 84 85 86 87 88 89] sum = 845
9 has data [90 91 92 93 94 95 96 97 98 99] sum = 945
7 has data [70 71 72 73 74 75 76 77 78 79] sum = 745
6 has data [60 61 62 63 64 65 66 67 68 69] sum = 645
total sum = 4950
$
```

distributing slices

```
from mpi4py import MPI
import numpy as np

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()
N = 10

if(RANK == 0):
    DATA = np.arange(N*SIZE, dtype='i')
    for i in range(1, SIZE):
        SLICE = DATA[i*N:(i+1)*N]
        COMM.Send([SLICE, MPI.INT], dest=i)
    MYDATA = DATA[0:N]
else:
    MYDATA = np.empty(N, dtype='i')
    COMM.Recv([MYDATA, MPI.INT], source=0)
```

collecting the sums of the slices

```
S = sum(MYDATA)
print(RANK, 'has data', MYDATA, 'sum =', S)

SUMS = np.zeros(SIZE, dtype='i')
if(RANK > 0):
    COMM.send(S, dest=0)
else:
    SUMS[0] = S
    for i in range(1, SIZE):
        SUMS[i] = COMM.recv(source=i)
    print('total sum =', sum(SUMS))
```

Observe the following:

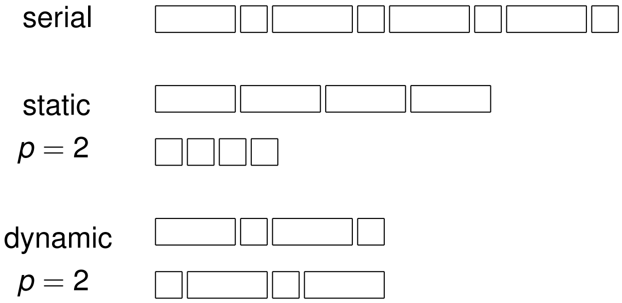
- `COMM.send` and `COMM.recv` have no type declarations.
- `COMM.Send` and `COMM.Recv` have type declarations.

Message Passing Interface for Python

1. the Message Passing Interface (MPI)
   - MPI and MPI for Python
   - hello world with mpi4py
   - point-to-point communication
2. Collective Communication
   - broadcast, scatter, and gather
   - processing numpy arrays
3. Probing for Messages
   - nonblocking communications

a motivating example

Consider scheduling 8 jobs on 2 processors:



dynamic job scheduling

Process a queue of jobs with manager/worker model:

- Manager distributes jobs to workers.
- Processing time for each job varies.
- When done, worker receives new job from manager.

Simulation of a dynamic job scheduler:

- Manager distributes random numbers to workers.
- Worker sleeps as many seconds as receive number.
- After sleeping, worker sends number back to manager.
- Manager prints received numbers from workers.

running the script

```
$ mpiexec -n 4 python3 mpi4py_dynamic.py
manager sends job 3 to 1
manager sends job 11 to 2
manager sends job 4 to 3
manager waits for jobs to return...
1 receives 3
3 receives 4
2 receives 11
1 sends 3
received 3 from 1
3 sends 4
received 4 from 3
2 sends 11
received 11 from 2
$
```

code for the workers

```
from mpi4py import MPI
from random import randint, seed
from time import sleep

seed(231112117) # for deterministic runs

COMM = MPI.COMM_WORLD
SIZE = COMM.Get_size()
RANK = COMM.Get_rank()

def worker():
    """
    A worker receives a job, sleeps as many
    seconds as the value of the job and
    then sends the job back to the manager.
    """
    job = COMM.recv(source=0, tag=RANK)
    print(RANK, 'receives', job)
    sleep(job)
    print(RANK, 'sends', job)
    COMM.send(job, dest=0, tag=RANK)
```

probing for messages

The manager probes for messages:

- Nonblocking check if message arrived.
- After arrival, receive the message.

The MPI command `Iprobe`:

```
    state = MPI.Status()
    okay = COMM.Iprobe(source=MPI.ANY_SOURCE, \
        tag=MPI.ANY_TAG, status=state)
```

Processing result of `Iprobe`:

- If `okay` is True on return, then message arrived.
- With `state.Get_source()` we get the rank of sender.

code for manager

Distribution of the jobs:

```
def manager():
    """
    Distributes jobs to the workers
    and probes for their return.
    """
    for i in range(1, SIZE):
        job = randint(2, 11)
        print('manager sends job', job, 'to', i)
        COMM.send(job, dest=i, tag=i)
    print('manager waits for jobs to return...')
```

probing for returning workers

```
count = 0
while(count < SIZE-1):
    state = MPI.Status()
    okay = COMM.Iprobe(source=MPI.ANY_SOURCE, \
        tag=MPI.ANY_TAG, status=state)
    if(okay):
        node = state.Get_source()
        data = COMM.recv(source=node, tag=node)
        print('received', data, 'from', node)
        count = count + 1
    else:
        sleep(0.5)
```

Summary and Exercises

`mpi4py` works for prototyping parallel algorithms and to learn the Message Passing Interface from within Python.

Exercises :

1. Replace the `bcast` in the script `mpi4py_broadcast` with `send` and `recv` commands to do the broadcast.
2. Replace the `scatter` in the script `mpi4py_scatter` with `send` and `recv` commands to do the scatter.
3. Replace the `gather` in the script `mpi4py_gather` with `send` and `recv` commands to do the gather.

An extra Exercise :

4. Examine `MPI.jl`, the MPI wrappers for Julia. Redo one of the examples in this lecture with `MPI.jl`.