Programming Machine Learning Algorithms for HPC

- MPI4py

Dr. Pierrick Pochelu and Dr. Oscar J. Castro-Lopez

UNIVERSITÉ DU LUXEMBOURG

MPI4py

Introduction

What is MPI?

- The Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures.
- The MPI standard defines the syntax and semantics of library routines that are useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.
- MPI is a communication protocol for programming[4] parallel computers. Both point-to-point and collective communication are supported.

Key Parts of an MPI Program

1. Initialize MPI Environment.

2. Get the Rank and Size:
    - Rank is the unique identifier for each process, obtained with MPI_Comm_rank. Each process receives a different rank, starting from 0 up to size - 1.
    - Size is the total number of processes, obtained with MPI_Comm_size.

3. Communication (data transfer).

4. Main Computation.

5. Finalize MPI Environment.

MPI4py

- *MPI for Python* provides Python bindings for the *Message Passing Interface* (MPI) standard, allowing Python applications to exploit multiple processors on workstations, clusters and supercomputers.

- This package builds on the MPI specification and provides an object oriented interface resembling the MPI-2 C++ bindings.

- It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communication of any *picklable* Python object, as well as efficient communication of Python objects exposing the Python buffer interface (e.g. NumPy arrays and builtin bytes/array/memoryview objects).

Communicating Python Objects and Array Data

- MPI for Python can communicate any built-in or user-defined Python object taking advantage of the features provided by the pickle module. These facilities will be routinely used to build binary representations of objects to communicate (at sending processes), and restoring them back (at receiving processes).

Communicators

- In MPI for Python, Comm is the base class of communicators.
- The number of processes in a communicator and the calling process rank can be respectively obtained with methods Comm.Get_size and Comm.Get_rank.

Point-to-Point Communications

- Point to point communication is a fundamental capability of message passing systems. This mechanism enables the transmission of data between a pair of processes, one side sending, the other receiving.

- MPI provides a set of *send* and *receive* functions allowing the communication of *typed* data with an associated *tag*.

Blocking Communications

- MPI provides basic send and receive functions that are *blocking*. These functions block the caller until the data buffers involved in the communication can be safely reused by the application program.

- In MPI for Python, the Comm.Send, Comm.Recv and Comm.Sendrecv methods of communicator objects provide support for blocking point-to-point communications within Intracomm and Intercomm instances. These methods can communicate memory buffers.

- The variants Comm.send, Comm.recv and Comm.sendrecv can communicate general Python objects.

Nonblocking Communications

- On many systems, performance can be significantly increased by overlapping communication and computation. This is particularly true on systems where communication can be executed autonomously by an intelligent, dedicated communication controller.

- In MPI for Python, the Comm.Isend and Comm.Irecv methods initiate send and receive operations, respectively. These methods return a Request instance, uniquely identifying the started operation. Its completion can be managed using the Request.Test, Request.Wait and Request.Cancel methods.

Collective Communications

- Collective communications allow the transmittal of data between multiple processes of a group simultaneously.
- The more commonly used collective communication operations are the following.
  - Barrier synchronization across all group members.
  - Global communication functions
    - Broadcast data from one member to all members of a group.
    - Gather data from all members to one member of a group.
    - Scatter data from one member to all members of a group.
  - Global reduction operations such as sum, maximum, minimum, etc.

Running Python scripts with MPI

Most MPI programs can be run with the command **mpiexec**. In practice, running Python programs looks like:

- `mpiexec -n 4 python script.py`

Before using MPI4py

1. Load MPI module (every time before running your script):
   - `module load mpi/OpenMPI`

2. Install mpi4py (just once).
   - `pip install mpi4py`

3. Allocate resources from SLURM (every time before running your script).

Example point to point message

- Allocate resources for using one node, two tasks and each using one CPU:
  - `--nodes 1`
  - `--ntasks-per-node 2`
  - `--cpus-per-task 1`

For example to run an interactive job for 30 minutes:

- `si -t 30 --nodes 1 --ntasks-per-node 2 --cpus-per-task 1`

Point to point example

```python
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(f"rank {rank} sends data")
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(f"rank {rank} receives {data}")
```

```
module load mpi/OpenMPI
```

```
mpiexec -n 2 python point_point.py
```

Point to point many tasks example

- `si -t 30 --nodes 1 --ntasks-per-node 128 --cpus-per-task 1`

```python
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    for i in range(1, size):
        comm.send(data, dest=i, tag=11)
    print(f"rank {rank} sends data")
elif rank > 0:
    data = comm.recv(source=0, tag=11)
    print(f"rank {rank} receives {data}")
```

```
mpiexec -n 2 python point_point2.py
```

```
mpiexec -n 128 python point_point2.py
```

Collective Communication Broadcasting example

```python
from mpi4py import MPI

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

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
print(f"rank {rank} has {data}")
```

```
mpiexec -n 10 python collective_comm1.py
```

Collective Communication Scattering example

```python
from mpi4py import MPI

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

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
print(f"rank {rank} has {data}")
```

```
mpiexec -n 10 python collective_comm2.py
```

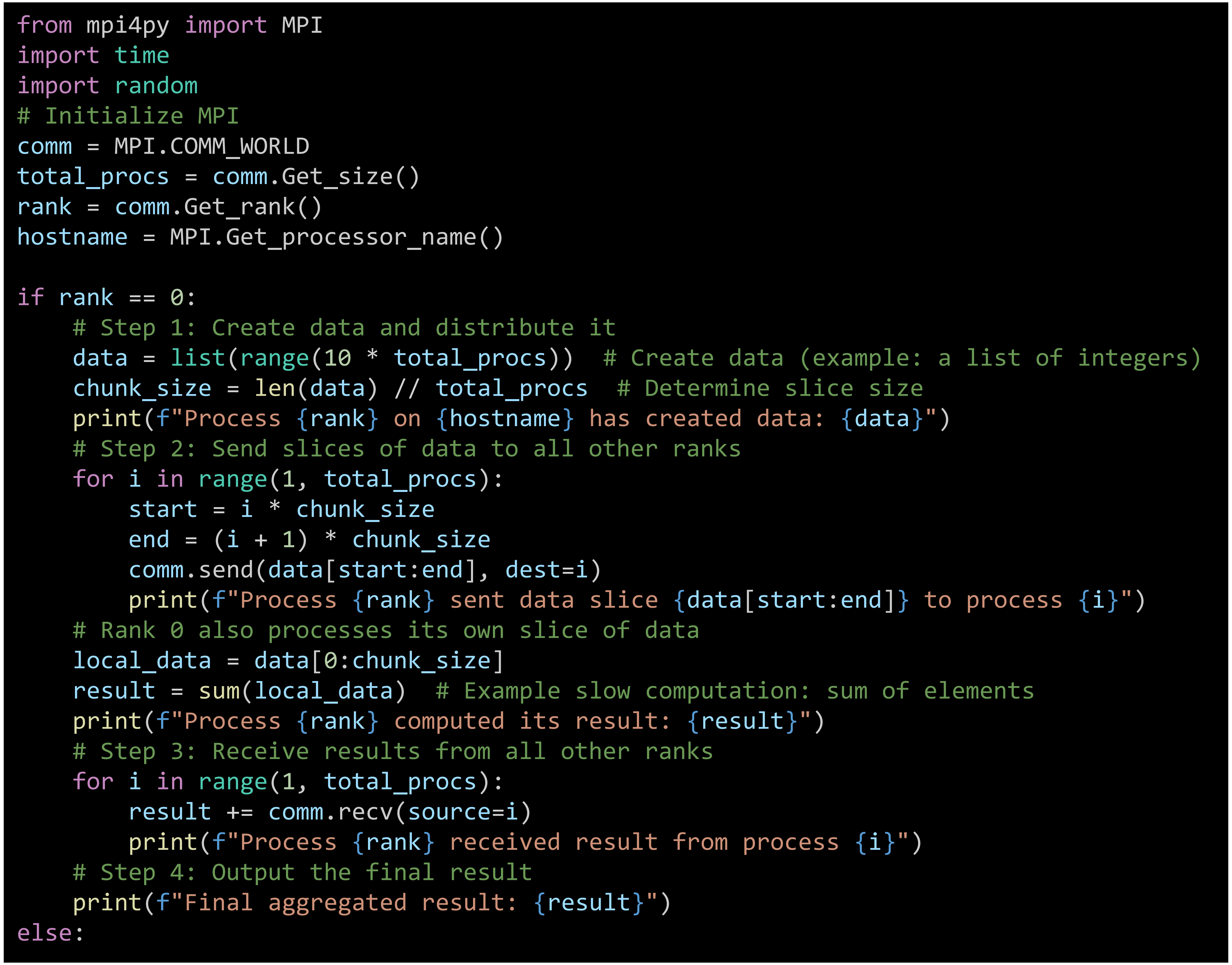
Collective Communication Gather example

```python
from mpi4py import MPI

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

data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
else:
    assert data is None
print(f"rank {rank} has {data}")
```

```
mpiexec -n 10 python collective_comm3.py
```



```python
from mpi4py import MPI
import time
import random
# Initialize MPI
comm = MPI.COMM_WORLD
total_procs = comm.Get_size()
rank = comm.Get_rank()
hostname = MPI.Get_processor_name()

if rank == 0:
    # Step 1: Create data and distribute it
    data = list(range(10 * total_procs))  # Create data (example: a list of integers)
    chunk_size = len(data) // total_procs  # Determine slice size
    print(f"Process {rank} on {hostname} has created data: {data}")
    # Step 2: Send slices of data to all other ranks
    for i in range(1, total_procs):
        start = i * chunk_size
        end = (i + 1) * chunk_size
        comm.send(data[start:end], dest=i)
        print(f"Process {rank} sent data slice {data[start:end]} to process {i}")
    # Rank 0 also processes its own slice of data
    local_data = data[0:chunk_size]
    result = sum(local_data)  # Example slow computation: sum of elements
    print(f"Process {rank} computed its result: {result}")
    # Step 3: Receive results from all other ranks
    for i in range(1, total_procs):
        result += comm.recv(source=i)
        print(f"Process {rank} received result from process {i}")
    # Step 4: Output the final result
    print(f"Final aggregated result: {result}")
else:
```

```python
else:
    # Step 5: Receive part of the data
    local_data = comm.recv(source=0)
    print(f"Process {rank} on {hostname} received data slice: {local_data}")

    # Step 6: Run the slow algorithm (e.g., sum of elements)
    local_result = sum(local_data)
    # Simulate long time
    sleep_time = random.randint(1, 5)
    time.sleep(sleep_time)
    print(f"Process {rank} computed its result: {local_result}")

    # Step 7: Send the result back to rank 0
    comm.send(local_result, dest=0)
    print(f"Process {rank} sent result back to process 0")

# Finalize MPI
MPI.Finalize()
```

Useful links

- Supercomputers | HPC @ Uni.lu
- https://mpi4py.readthedocs.io/en/stable/index.html