Session 12: Introduction to MPI (4PY)

October 10th 2017, Lena Oden
Overview

• Introduction
  • Basic concepts
  • mpirun

• Hello world
• Wrapping numpy arrays
• Common Pitfalls
Introduction

• MPI de facto standard for parallel programming in HPC systems since 1994 (MPI 1.0)
• Currently, MPI 3.1
• MPI is a standard with different implementation
  • OpenMPI
  • MPICH
  • Mvapich
  • ....
• Distributed memory systems (process parallel)
• Message-passing
• Goals: performance, scalability, portability
  • Shared memory, sockets, Infiniband…
• Standard is C (C++ bindings exists)
• MPI4PY: Layer above in Python

09/10/2017
Getting started

- Requires a MPI Installation + mpi4py
- **Communicator**: The “context” processes talk to each other
  - groups processes
  - Separation of concern
- Process can be in more than on communicator
  rank = comm.Get_rank()
  size = comm.Get_size()
- **MPI_COMM_WORLD (MPI.COMM_WORLD)**
  - Basic communicator, created at start time
Introduction: mpirun

- MPI programs are started with a specialized runner application
  - Sets up environment and start the instances
  - Distributes processes across nodes

mpirun –np 2 python hello_world.py <args>

mpirun : MPI runner applications
-np 2 : number of parallel mpi processes to start
python hello_world.py : Your application
<args> : Arguments (argv and argc stay the same.)

Note: On Clusters with SLURM use srun instead on mpirun
Hello world

from mpi4py import MPI

# Communicator that contains all mpi processes
comm = MPI.COMM_WORLD

rank = comm.Get_rank()
size = comm.Get_size()
name = MPI.Get_processor_name()

print("Rank {0} out of {1} on {2}".format(rank, size, name))

$ mpirun -np 2 python3 hello_world.py
Rank 0 out of 2 on ANDREASPC
Rank 1 out of 2 on ANDREASPC
Blocking Point-to-Point

- Simple principle:
  - One Process Sends Message (comm.send)
  - another process receives the Message (comm.recv)
- Blocking, until `locally` completed
- Tag for matching (Should always be set, if possible)

```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=1)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(data)
```
MPI4Py: *pickle based vs. arrays*

- MPI4Py Supports both:
  - generic Python objects
  - buffer-like objects (e.g. numpy)

- Generic Objects
  - Data are pickled before transfer
  - Needs time and memory

- Buffer like Objects
  - `Send()`, `Recv()`
  - `Tuple/triple` for the data
    - `[data, MPI.DOUBLE]`
    - `[data, count, MPI.DOUBLE]`
Expample: Send/Recv with numpy

```python
from mpi4py import MPI
import numpy as np

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

if rank is 0:
    data = np.array([1, 2, 4, 5], dtype='int')
    comm.Send([data, MPI.INT], dest=1, tag=1)

elif rank is 1:
    data = np.zeros(4, dtype='int')
    comm.Recv([data, MPI.INT], source=0, tag=11)
    print(data)
```
Performance Comparison

MPI4Py Latency

- Python Array
- Numpy Array + send()
- Numpy Array + Send()
Non-Blocking Point to Point

- Non-blocking version of Send/Recv
- Start a send/recv operations
- Completed later (wait)
- Used to overlap computation and communication
- Avoiding Deadlocks

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
req = {}  

if rank == 0:
    data = np.array([1, 2, 4, 5], dtype='int')
    req[0] = comm.Isend([data, MPI.INT], dest=1, tag=11)
elif rank == 1:
    data = np.zeros(4, dtype='int')
    req[0] = comm.Irecv([data, MPI.INT], source=0, tag=11)
    req[0].wait()
#MPI.Request.waitall(req)
```
Exercise 1: Deadlocks

```python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
# modify this function so we don't have a deadlock
send_data1 = np.array([1, 2, 4, 5], dtype='int')
send_data2 = np.array([5, 7, 8, 9], dtype='int')
recv_data1 = np.zeros(4, dtype='int')
recv_data2 = np.zeros(4, dtype='int')
next = (rank + 1) % size
prev = (size + rank - 1) % size
comm.Send([send_data1, MPI.INT], dest=next, tag=11)
comm.Recv([recv_data2, MPI.INT], source=next, tag=12)
comm.Send([send_data2, MPI.INT], dest=prev, tag=12)
comm.Recv([recv_data1, MPI.INT], source=prev, tag=11)
```
Caution

- Send/Recv are only locally blocking
- Send may return, before the other process has received the data
- Depends on the Message size and the MP-Implementatio
- (buffered send vs. rendezvous protocol)
- Using non-blocking communication does NOT necessarily mean, that communication is handled in the background
- May require “pocking” of the MPI Progress Engine
- Depends on MPI Implementation and Message Size
- Req.test()
- Wait is usually busy wait (HPC, we don’t want but our threads to sleep)
Collective Operations

- A communication call to collective send/recev messages in a communicator
  - Barrier
  - Bcast
  - Scatter
  - Gather
  - Allgather
  - Reduce/Allreduce
- Forces a synchronization between Processes
  - Can also be a reason for slow-down
  - Usually, a busy waiting Modell (HPC)
Barrier

```python
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
comm.Barrier()
comm.barrier()
```
Bcast and Scatter

```python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank is 0:
    data = np.array([2,2,3,4,], dtype='int')
else:
    data = np.zeros(1, dtype='int')

comm.Bcast([data, MPI.INT], root=0)
print("bcast", rank, data)
if rank is 0:
    comm.Scatter([data, MPI.INT], [data, MPI.INT], root=0)
else:
    comm.Scatter(None, [data, MPI.INT], root=0)
print("scatter", rank, data)
```
Gather and Allgather

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

data = (rank+1)**2
data = comm.gather(data, root=0)

print(rank, data)

ata = np.array(rank, dtype='int')
gather = np.zeros(4, dtype='int')
comm.Allgather([data,MPI.INT], [gather, MPI.INT])
print(rank, gather)
```
Reduce/Allreduce

```python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
data = np.array(rank, dtype='int')
result = np.zeros(1, dtype='int')
comm.Reduce([data, MPI.DOUBLE],
             [result, MPI.DOUBLE],
             op=MPI.SUM, root=0)
print("reduce", rank, result)
comm.Allreduce([data, MPI.DOUBLE],
               [result, MPI.DOUBLE],
               op=MPI.SUM)
print("allreduce", rank, result)
```
Exercise 2: Computing Pi in Parallel

If rank is 0
  N = np.array(10000, 'i')

#Distribute N Across all nodes

start = time.time()
hi = 1.0 / N; s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
Pi = np.array(s * h, dtype='d')

#collect result with the reduce function
end = time.time()
if rank is 0:
    print("I get for Pi \{0\}".format(Pi_ALL))
    print("I needed \{0\} seconds").format(end-start)

09/10/2017
Exercise 3: 2-D Stencil
Tipps: Send/recv partial Arrays

req[0] = comm.Isend([grid1[1]::], MPI.DOUBLE, dest=top, tag=2)

req[1] = comm.Irecv([grid1[my_m+1]::], MPI.DOUBLE, source=btm, tag=2)
Further reading and resources used

https://en.wikipedia.org/wiki/Message_Passing_Interface

Thank you for your attention

References and further reading: