Introduction to MPI

CDP
Shared Memory vs. Message Passing

• **Shared Memory**
  • Implicit communication via memory operations (load/store/lock)
  • Global address space

• **Message Passing**
  • Communicate data among a set of processors without the need for a global memory
  • Each process has its own local memory and communicated with others using messages
Shared Memory

• Advantages
  • **User Friendly:** global address space provides a user-friendly programming
  • **Fast and Uniform:** data sharing between tasks is both fast and uniform due to proximity of memory to CPUs

• Disadvantages
  • **Scalability:** primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can increase traffic on the shared memory CPU path.
  • **Responsibility:** Programmer responsibility for synchronization constructs that ensure “correct” access of global memory
  • **Hardware:** it becomes increasingly difficult and expensive to design and produce a shared memory machines with increasing number of processors
Message Passing

• Advantages
  • **Scalability**: adding more CPUs won’t harm CPU-memory bandwidth
  • **Responsibility**: elimination of the need for synchronization constructs such as semaphores, monitors, etc...
  • **Distributed**: naturally supports distributed computation

• Disadvantages
  • **Copy Overhead**: data exchanged among processors cannot be shared; it is rather copied (using send/receive messages, not without a cost)
  • **Complicated**: less natural transition from serial implementation
Overview - What is MPI?

Message Passing Interface

• MPI is a message-passing library
• Industry Standard
  • Developed by a consortium of corporations, government labs and universities
  • "Standard" by consensus of MPI Forum participants from over 40 organizations
• The first standard and portable message passing library with good performance
• MPI consists of 128 functions for
  • Point-to-Point message passing
  • User defined datatypes
  • Collective communication
  • Communicator and group management
  • Process topologies
  • Environmental management
• MPI v2 is now becoming the standard
  • Extends (does not change) MPI
What does MPI offer?

• **Standardization**
  • Rely on your MPI code to execute under any MPI implementation running on your architecture.

• **Portability**
  • Designed to supports most environments; very low resource requirements
  • Today your code is parallel; tomorrow it is distributed

• **Performance**
  • Meet industry’s performance demands

• **Richness**
  • 128 functions that allows many different communication methods
What is missing in MPI?

• **Dynamic process management**
  • All the processes created at initiation; cannot be changed.
  • MPI v2 already support dynamic processes

• **Shared memory operations**
  • Share data only via message passing

• **Multi-threading issues**
  • Threads are not supported by MPI (no shared memory)
  • Can use OpenMP with MPI
Design and Implement an MPI Program

• Serial
  • When possible, start with a debugged serial version
  • Much easier to debug when running serial

• Design
  • Design parallel algorithm

• Implement
  • Write code, making calls to MPI library
  • Compile

• Start Slow
  • Run with a few nodes first, increase number gradually
  • Easier to debug with small amount of processes
Basic Outline of an MPI Program

• **Initialization**
  • Initialize communications

• **Algorithm**
  • Communicate to share data between processes
  • The logic of your program

• **Finalize**
  • Exit in a "clean" fashion from the message-passing system when done communicating
Format of MPI routines

• **bindings:**
  • `xxxx(parameter, ... )`

• All MPI routines for point-to-point communication and collective communication have integer return type.

• Header file required
  • `from mpi4py import MPI`
  • for Python programs
6 Basic MPI calls

- Init
- Finalize
- Get_rank
- Get_size
- Send/send
- Recv/recv
Initializing an MPI process

• **MPI_Init**
  • Initialize environment for communication
  • The first MPI call in any MPI process
  • One and only one call to **MPI_Init** per process
  • MPI_Init is automatically called when you import the module.

• Process creation is done by the call to
  • `mpirun -np <num_processes> python <executable>`
Exiting from MPI

• **MPI_Finalize**
  • Exit in a "clean" fashion when done communicating
  • Cleans up state of MPI.
  • The last call of an MPI process
  • Must be called only when there is no more pending communications
  • MPI_Finalize is *automatically* called before the Python process ends.
Basic MPI Definitions

• **Group**
  - An ordered set of processes
  - Has its own unique identifier (handle)
    - Assigned by the system
    - Unknown to the user
  - Associated with a communicator
  - Initially, all processes are members of the group given by the predefined communicator **COMM_WORLD**

• **Rank**
  - Unique, integer identifier for a process within a group
  - Sometimes called a "process ID"
  - Contiguous and begin at zero
  - Used to specify the source and destination of messages
Communicator

• Defines the collection of processes (group) which may communicate with each other (context)
• Possesses its own unique identifier (handle)
• Most MPI subroutines require you to specify the communicator as an argument
• We can create and remove groups/communicators during the program runtime
• **COMM_WORLD** is the predefined communicator which includes all processes in the MPI application
Rank and size within a communicator

• **Process Rank**
  • Gets a process' rank within a communicator
  • `Get_rank()`

• **Cluster Size**
  • Gets the number of processes within a communicator
  • `Get_size()`
MPI Communicator Rank/Size Example

```python
import mpi4py.MPI as MPI
# MPI.Init()

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
print("Helloworld! I am process %d of %d processes." % (rank, size))

# MPI.Finalize()
```
Sending and receiving messages

• **MPI_Send**
  • Basic blocking send operation
  • Called "standard" send mode
  • `Send(obj, dest=0, tag=0)`

• **MPI_Recv**
  • Basic blocking receive operation
  • `Recv(buf, source=0, tag=0, status=None)`
Sending and receiving messages

• Send/Recv - uses Numpy arrays, fast
• send/recv - uses any python object (pickle), slow

• Communication of buffer-like objects [data, count, datatype]
  • Automatic MPI datatype discovery for NumPy arrays and PEP-3118 buffers is supported, but limited to basic C types (all C/C99-native signed/unsigned integral types and single/double precision real/complex floating types) and availability of matching datatypes in the underlying MPI implementation. In this case, the buffer-provider object can be passed directly as a buffer argument, the count and MPI datatype will be inferred.
MPI messages

- Message = data + envelope

\[\text{Send} \ (\text{[data, count, datatype]}, \ \text{dest, tag})\]
Data

- **data**: starting location of data
- **count**: number of elements
  - receiver $\geq$ sender
- **datatype**: basic or derived
  - receiver $==$ sender

\[
\text{Send} \ ( [\text{data}, \text{count}, \text{datatype}], \text{dest}, \text{tag})
\]
Envelope

• **dest**: Destination or source
  • Rank in a communicator of sender/receiver respectively
  • Must match or receiver may use **ANY_SOURCE**

• **tag**: Message identifier
  • Integer chosen by programmer
  • Must match or receiver may use **ANY_TAG**

\[
\text{Send} \left( \left[ \text{data}, \text{count}, \text{datatype} \right], \text{dest}, \text{tag} \right)
\]

DATA  ENVELOPE
MPI Status

• `Get_count()` - returns message size in Bytes
• `Get_elements(datatype)` - returns number of elements of type `datatype`
• `Get_source()` - returns message source
• `Get_tag()` - returns message tag
• `Get_error()` – returns the error code
MPI Send/Recv Simple 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("Message sent, data is: ", data)

elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print("Message Received, data is: ", data)
```
MPI Send/Recv Numpy Arrays

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

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

# pass explicit MPI datatypes
if rank == 0:
    data = np.arange(1000, dtype=np.int32)
    comm.Send([data, 1000, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = np.empty(1000, dtype=np.int32)
    comm.Recv([data, 1000, MPI.INT], source=0, tag=77)
```

from mpi4py import MPI
import numpy as np

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

# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=np.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=np.float64)
    comm.Recv(data, source=0, tag=13)
Multiplying a Dense Matrix with a Vector

• Reminder
  • The computation of each cell in the output vector is independent of the others
  • We can divide the output cells between the processes
from mpi4py import MPI
import numpy as np

RES_VECTOR_TAG = 5

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

matrix = np.ones((128, 128))
vector = np.ones(128)

chunk_size = 128//size
result = np.empty(chunk_size, dtype=np.int32)

for i in range(0, chunk_size):
    result[i] = np.matmul(matrix[i+rank*chunk_size], vector)

if rank != 0:
    comm.Send([result, chunk_size, MPI.INT], dest=0, tag=1)
else:
    for i in range(1, size):
        recv_res = np.empty(chunk_size, dtype=np.int32)
        comm.Recv(recv_res, source=i, tag=1)
        result = np.append(result, recv_res)

print(result)