HPC Programming

Message Passing Interface (MPI) with Python

Peter-Bernd Otte, 3.12.2019

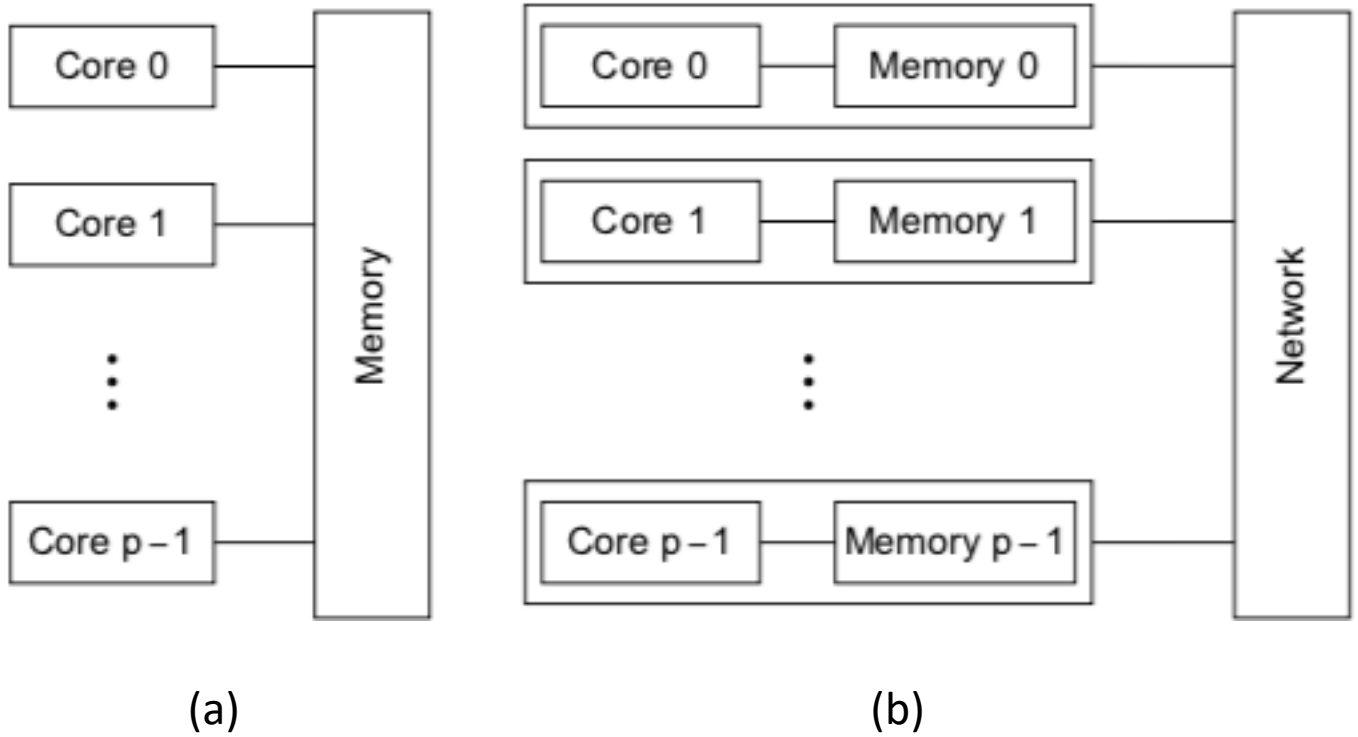
Shared System: Why two C extension

Recap

(a) Shared-Memory system:
- Each core can read/write each memory location
- Coordination of cores via shared-memory locations
- Use OpenMP
- Small projects. HIMster2: up to 32 cores/node

(b) Distributed-Memory system:
- Each core has private memory
- Cores explicitly sending messages for data exchange and coordination
- MPI
- Several nodes of a cluster

- Hybrid-Programming:
  - OpenMP+MPI



(a) (b)

MPI: Getting Started (2)

Recap

„MPI = Sending and receiving messages“

rank 0

rank 1

Single Program, Multiple Data (SPMD

Recap

- Standard MPI programming:
  - Write single executable
  - behaviour depends on its rank
    - eg rank=0: message collecting master,
      ranks>0: computing
  - Number of ranks from 1 to $O(10^4)$ on Himster2
    - $O(10^6)$ on extreme machines
  - called “Single Program, multiple Data”
- ⇔ Multiple-Program Multiple-Data (MPMD)
  - even mixture of different software possible with MPI:
    Fortran and C executable communicating fine

```
MPI_Comm_rank(MPI COMM WORLD, &my_rank);

if (my_rank == 0) {
  ...
} else {
  ...
}
```

MPI: Communicators

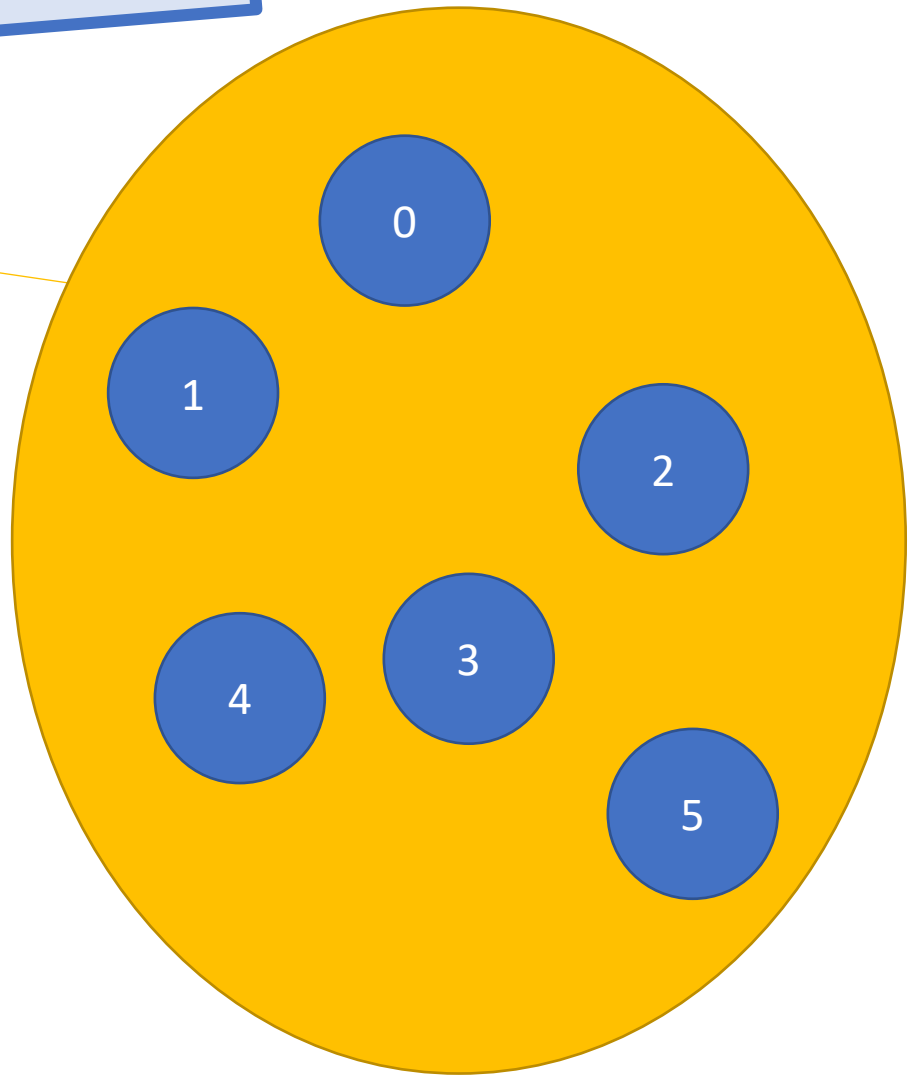
Recap

- MPI Communicator
  = group of processes that can send messages to each other.
- All processes are in `MPI_COMM_WORLD` communicator
  - Defining sub groups → see future lecture
- Number of members in communicator with

```
int MPI_Comm_size (
      MPI_Comm comm    /*in*/,
      int *comm_size_p /*out*/)
```

- Get rank of sub_process with

```
int MPI_Comm_rank (
      MPI_Comm comm   /*in */,
      int * my_rank_p /*out*/)
```



MPI: Make a match

Recap

- rank s calls: `MPI_Send(send_buf, send_buf_size, send_type, dest, send_tag, send_comm);`
- rank q calls: `MPI_Recv(recv_buf, recv_buf_size, recv_type, src, recv_tag, recv_comm, &status);`

- All 5 “green” parameters need to match to get message successfully through.
  - all mandatory to be equal, except :
  - recv_buf_size >= send_buf_size
  - dest = rank of receiving process, src = rank of sending process

MPI: different communications modes

Recap

	Blocking	Non-Blocking	note
standard send	MPI_Send	MPI_ISend	synchronous or asynchronous send (depending on message size and implementation) uses internal buffer.
synchronous send	MPI_SSend	MPI_ISSend	Only completes when the receive has started
asynchronous (buffered) send	MPI_BSend	MPI_IBSend	Completes after buffer copy (always).
ready send	MPI_RSend	MPI_IRSend	problematic: mandatory to have matching receive already listening. Not discussed in this lecture. Might be fastest solution.

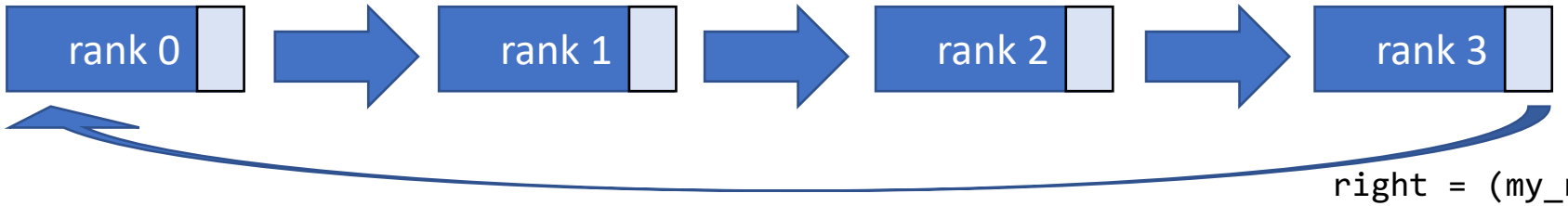
„i" stands for immediate return

	Blocking	Non-Blocking	note
standard receive	MPI_Recv	MPI_IRecv	works for all sending routines.

MPI: Non-Blocking Send & Receive

Recap

- to a 1D ring with 1 piece of data passing in one direction




```
right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
```

- cyclic: MPI_Send(…to right…)
  MPI_Recv(…from left…)

deadlock!
All are waiting for a receiver

- non-cyclic: for rank<size-2: MPI_Send(…to right…)
  for rank>0: MPI_Recv(…from left…)

serialisation!
highest rank starts, rank 0 last

(hint: all this only true if MPI calls are synchronous sends)

MPI: Non-Blocking communication

Recap

This can be accomplished by:

- non-blocking send
  1. MPI_Isend();
  2. Different_Work();
  3. MPI_Wait(); //Waits until MPI_Isend completed / send buffer is read out

- non-blocking receive
  1. MPI_Irecv();
  2. Different_Work();
  3. MPI_Wait(); //Waits until MPI_Irecv completed / receive buffer is filled

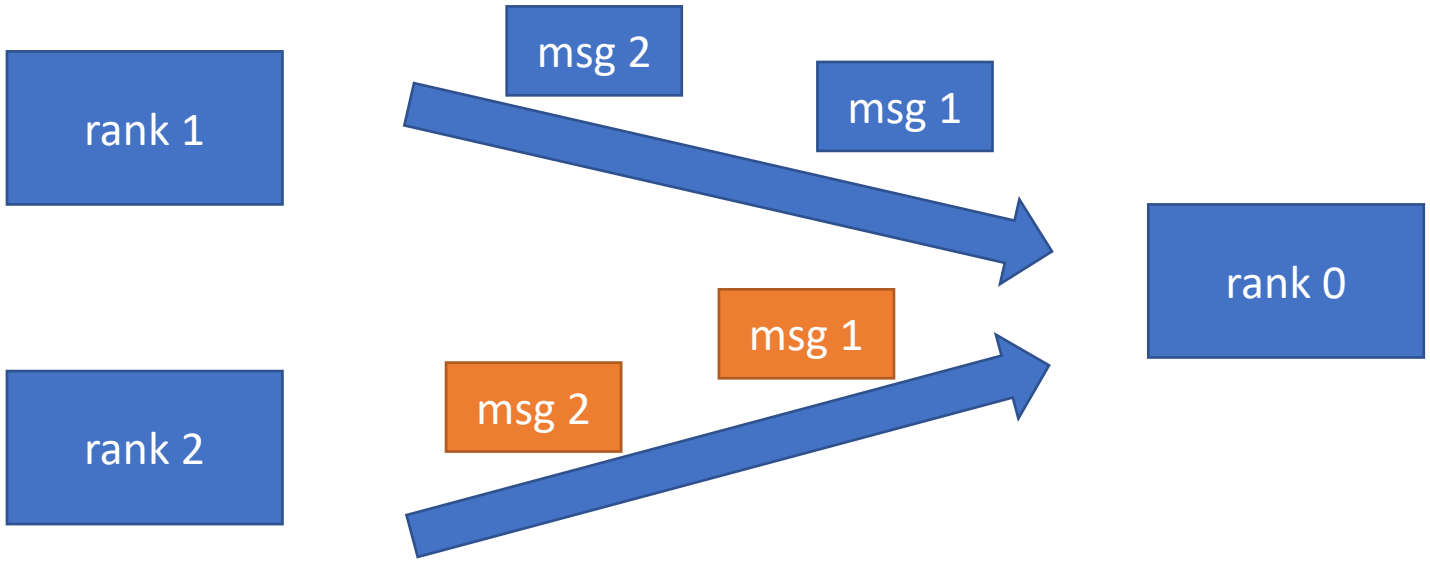
Golden MPI rule:
always <=3 lines of MPI_* calls per task

otherwise:
check MPI reference or wrong coding

MPI: Message Order Preservation

Recap

- Messages do not overtake, if same:
  - communicator (eg MPI_COMM_WORLD),
  - source rank and
  - destination rank
- true for: synchronous and asynchronous communications
- messages from different senders can overtake

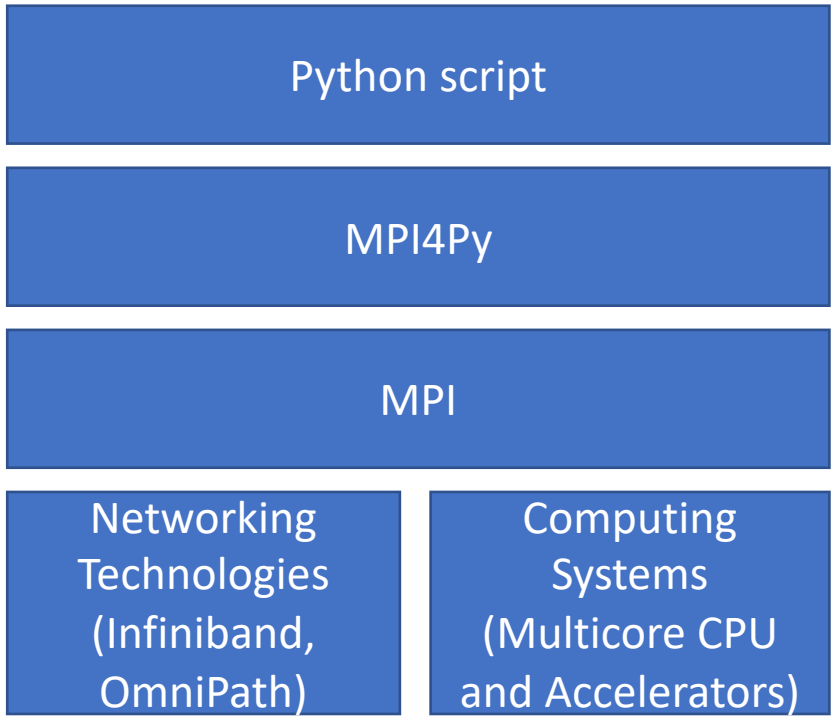


Introduction MPI4Py

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Collective Communication
5. Dealing with I/O
6. Groups & Communicators
7. MPI Derived Datatypes
8. Common pitfalls and good practice (“need for speed”)

What is MPI4Py?

- MPI4Py = Intermediate Layer, MPI-1/2 bindings for Python (“calls MPI C-functions”)
- Supported: MPI-1
  - Groups & Communicators,
  - Point-to-point Communication (send, recv, isend, irecv, wait),
  - Collective Communication (broadcast, scatter, gather, reduce, …)
  - and features from MPI-2: MPI-IO, one-sided operations, …
  - But no MPI-3
- multiple processes: no problem with Python Global Interpretator Lock (GIL)
- Written in cython (= “compiled python”)
- docs: https://mpi4py.readthedocs.io
- code: https://github.com/mpi4py/mpi4py
- lack of documentation: MPI standard and code mandatory



Gain information

- Recap the last MPI lectures on C, check the MPI-standard
- Search in https://github.com/mpi4py/mpi4py for function name
  - Note on github search:
    „At most, search results can show two fragments from the same file, but there may be more results within the file.“
  - Most of the python communication functions today are defined in:
    mpi4py/src/mpi4py/MPI/Comm.pyx
    https://github.com/mpi4py/mpi4py/blob/18c52a947443765dea0adaee0b702c044d9f150c/src/mpi4py/MPI/Comm.pyx
  - Search for „def ssend(“

MPI: MPI.send

- Sending a message to another receiving rank
- Syntax:

```
def send(self, obj, int dest, int tag=0):
        """Send"""
        cdef MPI_Comm comm = self.ob_mpi
        return PyMPI_send(obj, dest, tag, comm)
```

MPI: Elementary datatypes

- Python/NumPy/C types can't be passed
  → use MPI datatypes
- Advantage: interoperability with other software and hardware

MPI Python datatype	C equivalent	NumPy
MPI.SHORT	short int	h
MPI.INT	int	i
MPI.LONG	long int	l
MPI.LONG_LONG	long long int	q
MPI.UNSIGNED_CHAR	unsigned char	B
MPI.UNSIGNED_SHORT	unsigned short int	H
MPI.UNSIGNED_INT	unsigned int	I
MPI.UNSIGNED_LONG	unsigned long int	L
MPI.UNSIGNED_LONG_LONG	unsigned long long int	Q
MPI.FLOAT	float	f
MPI.DOUBLE	double	d
MPI.LONG_DOUBLE	long double	g
MPI.SIGNED_CHAR	char	b
MPI.C_DOUBLE_COMPLEX		D

Communication functions 1/2

- Can communicate any built-in or user-defined Python object (using pickle module)

- Transmission of python objects:
  - lowercase functions (send, recv, …)
  - high level and convenient, but slow for large data
  - pickle serialisation:

    object → pickle.dump() → MPI.send()

    object ← pickle.load() ← MPI.recv()

Communication functions 2/2

- Supports direct communication of any object exporting the single-segment buffer interface.

- Transmission of array data (as binary data)
  - uppercase functions (Send, Recv, …)
  - Fast, but low level and more verbose
    - message = [object memory, count, datatype]
  - communicate memory buffers
  - allocate send/receive buffer prior usage!!

Use considerations

- When to use:
  - existing python code: with multple processes
  - benefit from python libraries (eg numpy)

- … and not:
  - Memory intense programs, often used, core functionality → use C

Usage, Initialization, Exit, Timer

- `module load lang/Python/3.6.6-foss-2018b`
- Python code:

```
try:
  from mpi4py import MPI
except ImportError:
  print("mpi4py module not loaded.")
```

- MPI initialization and finalization: `MPI.Init(), MPI.Finalize()`
- MPI module:
  - `MPI_Init()` is called on import
  - calls `MPI_Finalize()` just before the Python process terminates
- MPI timer: `MPI.Wtime(), MPI.Wtick()`

Usage: Communicators and Ranks

„Hello World“: import, create a *communicator* and get the *rank* :

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print('My rank is ',rank)
```

- Save this to a file call comm.py and then run it:
  `mpirun -n 4 python filename.py`

Usage: Point-to-Point Communication

With pickle / lowercase functions:

- Send any pickable object: dictionary, integer, string, user object…
- with lowercase functions

- Blocking:
```
comm.send(data, dest=1, tag=1)
data = comm.recv(source=0, tag=1)
```

- Non-Blocking with isend:
```
req = comm.isend(data, dest=1, tag=1)
req.wait()
```

- Non-Blocking with irecv:
```
req = comm.irecv(source=0, tag=1)
data = req.wait()
```

Usage: Send numpy arrays (fast)

With capital function names (without pickle), for NumPy arrays:

- Blocking:

```
# automatic MPI datatype discovery
if rank == 0:
  data = numpy.arange(100, dtype=numpy.float64)
  comm.Send(data, dest=1, tag=13)
elif rank == 1:
  data = numpy.empty(100, dtype=numpy.float64)
  comm.Recv(data, source=0, tag=13)
```

Collective Communication

- `data = comm.bcast(data, root=0)`
- `comm.Scatter(data, recvbuf, root=0)`
- `comm.Gather(sendbuf, recvbuf, root=0)`
- `comm.Reduce(value, value_sum, op=MPI.SUM, root=0)`

MPI4Py exercises

- Setup your workspace: https://gitlab.rlp.net/pbotte/learnhpc/tree/master/mpi4py
- Try out exercises 1-6
- Optional: advanced examples: https://github.com/jbornschein/mpi4py-examples