02616

Large-scale Modelling

one library to rule MPI

`mpi4py`

MPI

Welcome to MPI the fundamental Message Passing Interface.

MPI is a *standard* with official C and fortran bindings

All other language API’s are ***NOT*** official!

including `mpi4py`!

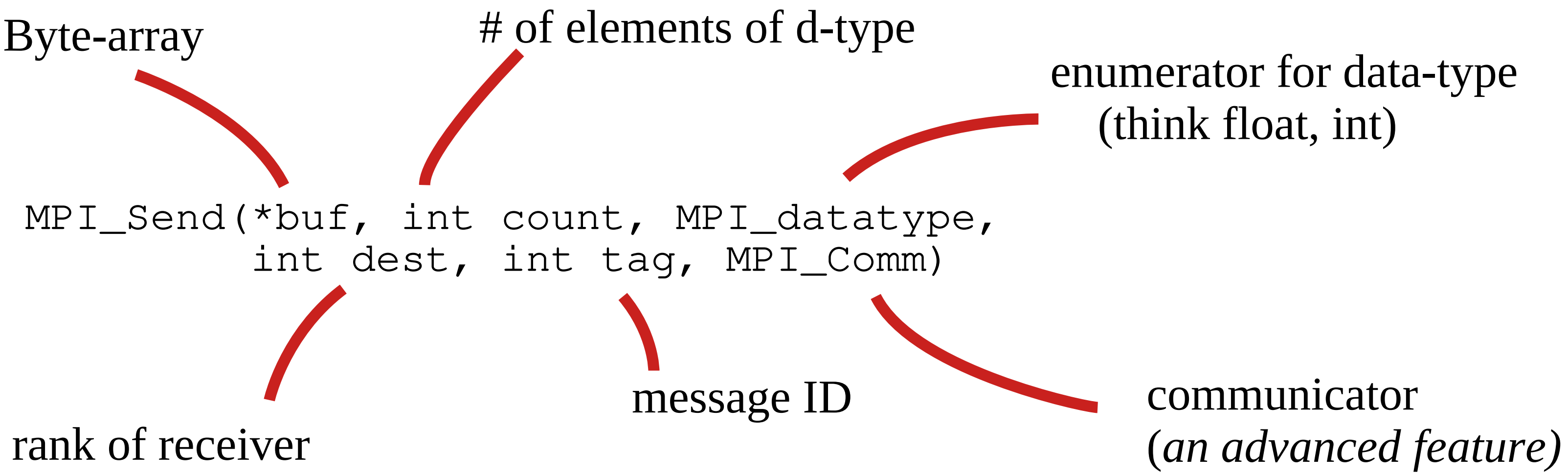
MPI implementations:

- OpenMPI
- MPICH
- IntelMPI (MPICH)
- … (vendors)

used extensively!

https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html
https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man3/index.html

MPI – behind the scenes



Byte-array

# of elements of d-type

enumerator for data-type (think float, int)

```
MPI_Send(*buf, int count, MPI_datatype,
         int dest, int tag, MPI_Comm)
```

rank of receiver

message ID

communicator (*an advanced feature)*

`MPI_Rsend`	ready-send (`Recv` *must* have been issued!)
`MPI_Bsend`	buffered send (manual setup of a buffer in MPI)
`MPI_Ssend`	synchronous send (blocking)
`MPI_Send`	implementation specific *standard* send (typically `Bsend` or `Ssend`)
`MPI_I?send`	non-blocking variants!

Any combination

```
MPI_Recv(*buf, int max_count, MPI_datatype,
         int source, int tag, MPI_Comm, *MPI_Status)
MPI_Irecv(*buf, int max_count, MPI_datatype,
          int source, int tag, MPI_Comm, *MPI_Request)
```

MPI → mpi4py

mpi4py documentation is, …, lacking…
(an open-source project ready for huge impact from contributors!)

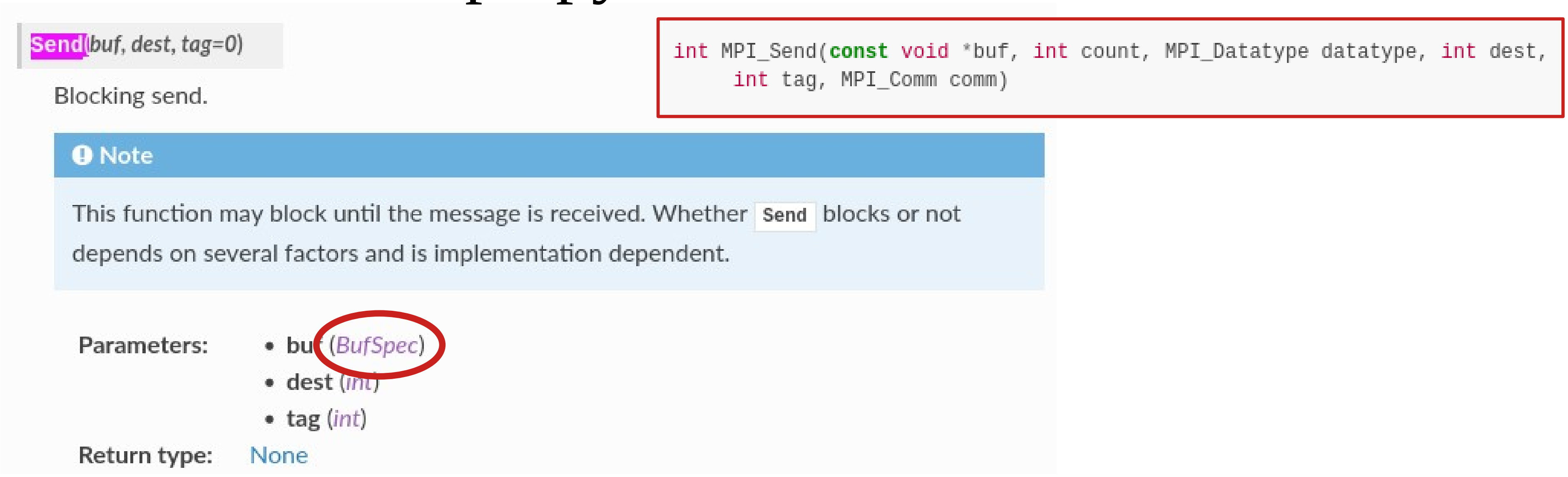
`MPI_X(…, MPI_Comm, …) → mpi4py.Comm.X(…)`

Typically, there are *sane* defaults:

- `count` defaults to number of elements in array
- `datatype` defaults to array type
- `source` defaults to `MPI_ANY_SOURCE`
- `tag` defaults to `MPI_ANY_TAG`|`0`

When in doubt, look at the documentation of both
OpenMPI + mpi4py!

mpi4py – the basics



Send(*buf, dest, tag=0*)

Blocking send.

Note

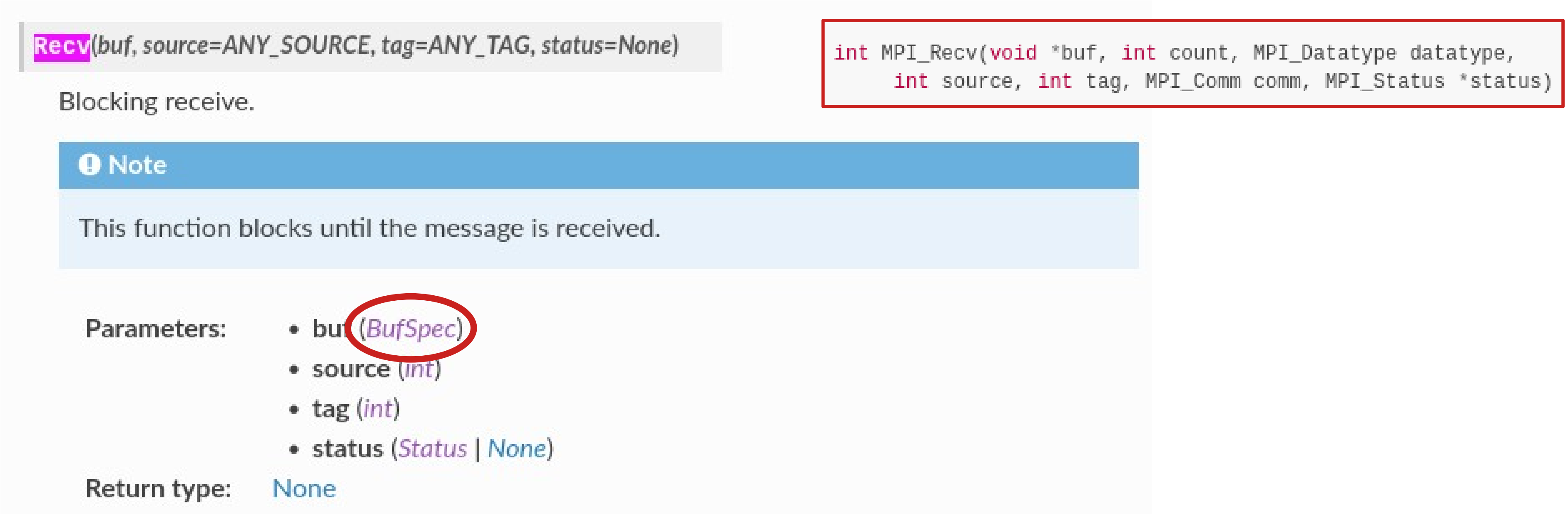
This function may block until the message is received. Whether `Send` blocks or not depends on several factors and is implementation dependent.

Parameters:

- **buf** (*BufSpec*)
- **dest** (*int*)
- **tag** (*int*)

Return type: None

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
```



Recv(*buf, source=ANY_SOURCE, tag=ANY_TAG, status=None*)

Blocking receive.

Note

This function blocks until the message is received.

Parameters:

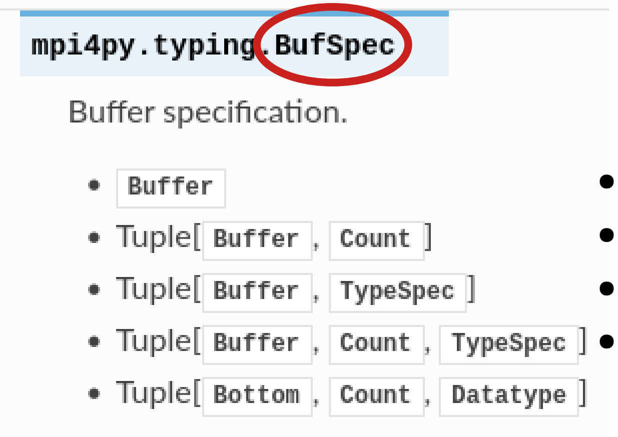
- **buf** (*BufSpec*)
- **source** (*int*)
- **tag** (*int*)
- **status** (*Status* | None)

Return type: None

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)
```

mpi4py – the basics

```
from mpi4py import MPI
C = MPI.COMM_WORLD
a = np.array([1., 2.])
```



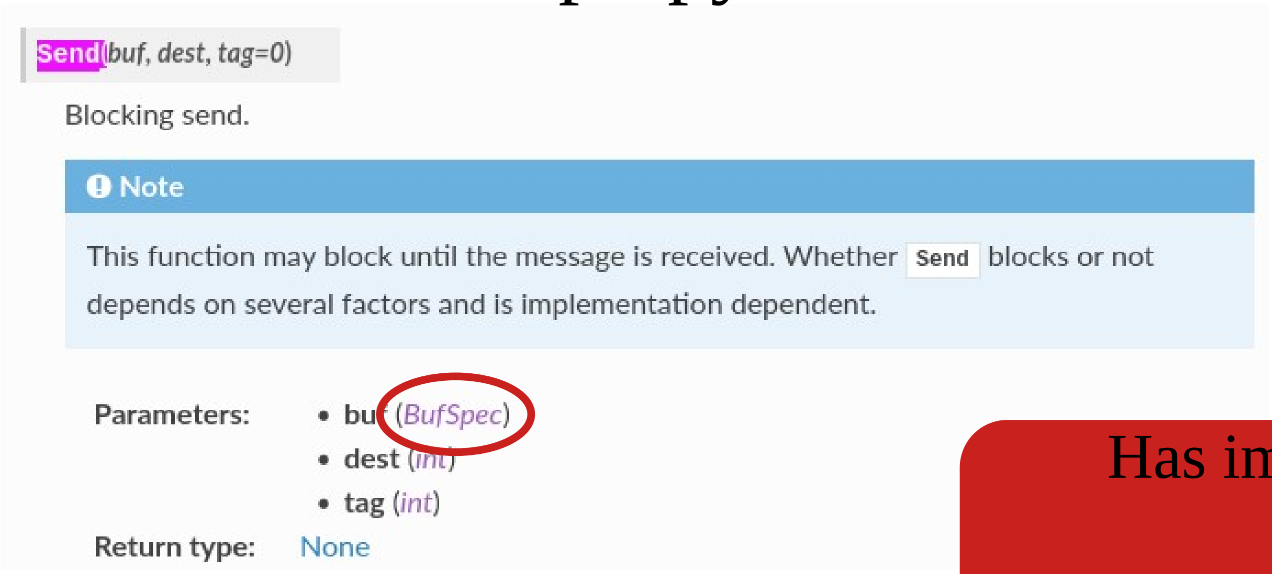
`mpi4py.typing.BufSpec`

Buffer specification.

- `Buffer`
- Tuple[`Buffer`, `Count`]
- Tuple[`Buffer`, `TypeSpec`]
- Tuple[`Buffer`, `Count`, `TypeSpec`]
- Tuple[`Bottom`, `Count`, `Datatype`]

- `C.Send(a, 1)`
- `C.Send((a, 1), 1)`
- `C.Send((a, MPI.DOUBLE_PRECISION), 1)`
- `C.Send((a, 2, MPI.DOUBLE_PRECISION), 1)`

mpi4py – the basics



Send(*buf, dest, tag=0*)

Blocking send.

Note

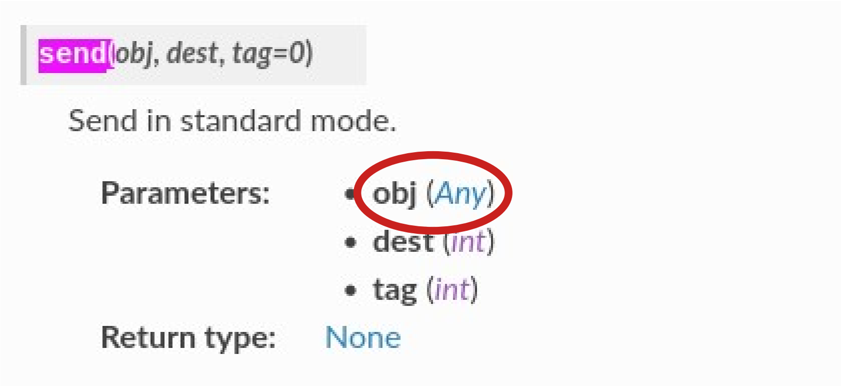
This function may block until the message is received. Whether `Send` blocks or not depends on several factors and is implementation dependent.

Parameters:
- **buf** (*BufSpec*)
- **dest** (*int*)
- **tag** (*int*)

Return type: None

Has implications!

Use only for non-critical sections!

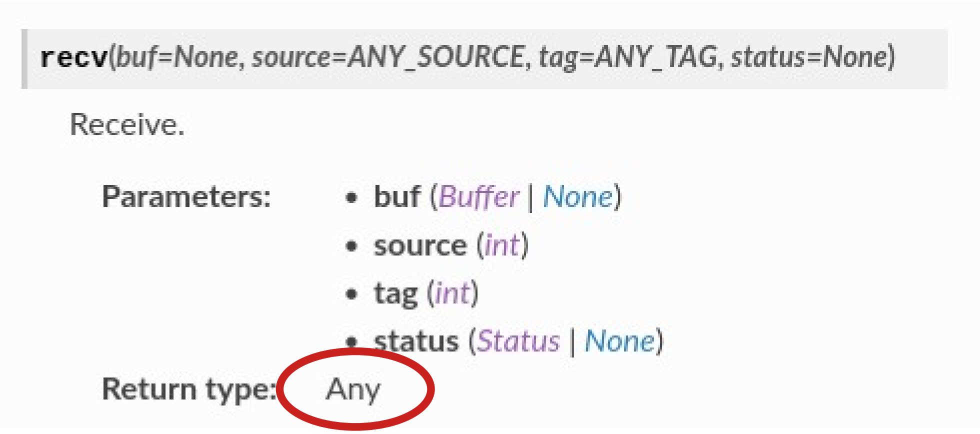


send(*obj, dest, tag=0*)

Send in standard mode.

Parameters:
- **obj** (*Any*)
- **dest** (*int*)
- **tag** (*int*)

Return type: None



recv(*buf=None, source=ANY_SOURCE, tag=ANY_TAG, status=None*)

Receive.

Parameters:
- **buf** (*Buffer* | *None*)
- **source** (*int*)
- **tag** (*int*)
- **status** (*Status* | *None*)

Return type: Any

MPI – behind the scenes

```
MPI_Recv(*buf, int max_count, MPI_datatype,
          int source, int tag, MPI_Comm, *MPI_Status)
```

`MPI_Recv` is *magic*:

- Buffer size >= sent message
- Data-type needs not be the same (`MPI_Send`: float + `MPI_Recv`: complex)
- Can receive from arbitrary source (`MPI_ANY_SOURCE`)
- Can receive any message ID (`MPI_ANY_TAG`)
- The `MPI_Status` is your gateway to information!

  Only use a status if you:

  - Use `MPI_ANY_*`, or
  - Are not sure of how many bytes received
  - *Check API documentations for details of data-extraction!*
    - `MPI_Get_count / status.Get_count`
    - `status.source` or `status.tag`

  Else use `MPI_STATUS_IGNORE` (the default)!

02616 – Week 4

MPI non-blocking

- Allows overlapping *communication* and *computation!*
- Requires manual completion!

```
Comm.Isend(…, request=request)
# do computation
Comm.wait(request)
```

- All non-blocking variants returns a *request* handle (not a status!)
- Completes only after one of the following succeeds:

  - `MPI_Test`
  - `MPI_Testany`
  - `MPI_Testall`

  May return true/false

  - `MPI_Wait`
  - `MPI_Waitany`
  - `MPI_Waitall`

  Ensures completion!

`see sample.py`

They all return a `MPI_Status` (or list thereof).

MPI collectives

Collectives are messages done with *all* the ranks participating in the distribution.

Collectives:

- `Bcast` a value to all ranks (one → all)
- `Scatter` values from a list to all ranks (one → all)
- `Gather` values into a list on a single rank (all → one)
- `Reduce` values on a single rank (all → one)

Global collectives:

- `Allgather` equivalent to Gather + Bcast
- `Allreduce` equivalent to Reduce + Bcast
- `Alltoall?` equivalent to all calling Scatter/Gather for every rank

MPI sample code

Let’s go through (some of) it together!

Volunteers for next week?