Introduction to Point-to-Point Communication

Joachim Hein (LUNARC, Lund University)
Xin Li (PDC, KTH)

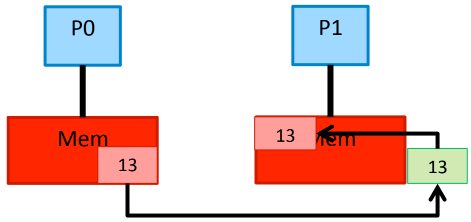
Overview

- Concept: Message passing point-to-point
- The MPI application interface for sending and receiving messages

Double sided point-to-point communication

- Most basic form of communication in message passing:

Program on P0:	Program on P1:
`a = 13`	
`send(a,1)`	
	`recv(b,0)`
	`c = c + b`



Standard send in C: MPI_Send

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

- `buf:` address of send buffer
- `count:` number of elements to be sent
- `datatype:` date type of buffer (explained further down)
- `dest:` rank of receiver
- `tag:` message tag (put **0** if you don't need)
- `comm:` communicator

Standard send Fortran 90: MPI_SEND

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, &
  COMM, IERROR)
```

```
<TYPE>::  BUF
INTEGER:: COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- `BUF:` (address of) send buffer
- `COUNT:` number of elements to be sent
- `DATATYPE:` date type of buffer (explained further down)
- `DEST:` rank of receiver
- `TAG:` message tag (put **0** if you don't need)
- `COMM:` communicator

Standard send Fortran 2008: MPI_SEND

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, &
  COMM, IERROR)
TYPE(*),DIMENSION(..),INTENT(IN)::  BUF
INTEGER,INTENT(IN):: COUNT, DEST, TAG
TYPE(MPI_Datatype),INTENT(IN) :: DATATYPE
TYPE(MPI_Comm),INTENT(IN) :: COMM
INTEGER,INTENT(OUT),OPTIONAL :: IERROR
```

- `BUF:` (address of) send buffer
- `COUNT:` number of elements to be sent
- `DATATYPE:` date type of buffer (explained further down)
- `DEST:` rank of receiver
- `TAG:` message tag (put **0** if you don't need)
- `COMM:` communicator

Standard send in Python: send

```
comm.send(obj, dest=dest, tag=tag)
```

- `obj:` The Python object being sent
- `dest:` The rank number of the rank the data will be sent to
- `tag:` Message tag (optional)

Example:

```
comm.send(obj, dest=1, tag=0)
```

Receiving data in C

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

- `buf:` address of receive buffer (output)
- `count:` number of elements to be received
- `datatype:` date type of buffer (explained further down)
- `source:` rank of sender (data origin)
- `tag:` message tag (needs to match the send!)
- `comm:` communicator
- `status:` status (output), info on: sender, tag, error

Receiving data in Fortran 90

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG,&
  COMM, STATUS, IERROR)
```

```
<TYPE>:: BUF
INTEGER:: COUNT, DATATYPE, SOURCE, TAG, COMM,
 STATUS(MPI_STATUS_SIZE), IERROR
```

- `BUF:` address of receive buffer (output)
- `COUNT:` number of elements to be received
- `DATATYPE:` date type of buffer (explained further down)
- `SOURCE:` rank of sender (data origin)
- `TAG:` message tag (needs to match the send!)
- `COMM:` communicator
- `STATUS:` status (output), info on: sender, tag, error

Receiving data in Fortran 2008

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG,&
  COMM, STATUS, IERROR)
```

```
TYPE(*), DIMENSION(..) :: BUF
INTEGER, INTENT(IN) :: COUNT, SOURCE, TAG
TYPE(MPI_Datatype), INTENT(IN) :: DATATYPE
TYPE(MPI_Comm), INTENT(IN) :: COMM
TYPE(MPI_Status) :: STATUS
INTEGER, OPTIONAL, INTENT(OUT) :: IERROR
```

- `BUF:` address of receive buffer (output)
- `COUNT:` number of elements to be received
- `DATATYPE:` date type of buffer (explained further down)
- `SOURCE:` rank of sender (data origin)
- `TAG:` message tag (needs to match the send!)
- `COMM:` communicator
- `STATUS:` status (output), info on: sender, tag, error

Receiving data in Python

```
obj = comm.recv(source=source, tag=tag)
```

- `obj`: The Python object being received
- `source`: rank of sender (data origin)
- `tag`: message tag (optional; needs to match the send)

Example:

```
myobj = comm.recv(source=0, tag=0)
```

Predefined data types in C (selection)

MPI datatype	C datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

- choose the MPI datatype matching the send/receive buffer

Predefined data types in Fortran (selection)

MPI datatype	Fortran datatype
`MPI_INTEGER`	`INTEGER`
`MPI_REAL`	`REAL`
`MPI_DOUBLE_PRECISION`	`DOUBLE PRECISION`
`MPI_COMPLEX`	`COMPLEX`
`MPI_DOUBLE_COMPLEX`	`DOUBLE COMPLEX`
`MPI_LOGICAL`	`LOGICAL`

- choose the MPI datatype matching the send/receive buffer

Predefined data types in Python (selection)

mpi4py datatype	MPI datatype
`MPI.CHAR`	`MPI_CHAR`
`MPI.SHORT`	`MPI_SHORT`
`MPI.INT`	`MPI_INT`
`MPI.LONG`	`MPI_LONG`
`MPI.FLOAT`	`MPI_FLOAT`
`MPI.DOUBLE`	`MPI_DOUBLE`
`MPI.LONG_DOUBLE`	`MPI_LONG_DOUBLE`

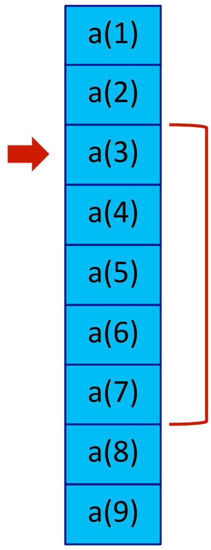
- no need to specify the datatype for sending/receiving standard Python objects

The argument count (Fortran 77 style example)

- sample code:

```
integer :: a(9)
...
call mpi_send(a(3), 5, MPI_INTEGER, &
     2, 0, my_comm, merror)
```

- Where will it send?
- Sends to rank 2 of **my_comm**
- Which elements will it send?
- Sends elements: **a(3),a(4),a(5),a(6),a(7)**



Fortran kind syntax (F90 syntax) Real data

MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)

INTEGER P, R, NEWTYPE, IERROR

- `P:` precision decimal digits
- `R:` exponent range
- `NEWTYPE:` requested MPI datatype (handle)

- Use to send real data declared with `selected_real_kind`
- Either `P` or `R` may be `MPI_UNDEFINED`

Example for receiving real data with selected_real_kind

```
integer :: dptype
real(selected_real_kind(15, 307)) :: x

call MPI_TYPE_CREATE_F90_REAL(15, 307, &
dptype, ierror)

call MPI_Recv(x, 1, dptype, 1, 0, my_comm, &
             stat, merror)
```

- Arguments of **selected_real_kind** and **MPI_TYPE_CREATE** need to match exactly!

Fortran kind syntax (F90 syntax) Integer data

```
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
```

```
INTEGER P, R, NEWTYPE, IERROR
```

- `R:` decimal exponent range
- `NEWTYPE:` requested MPI datatype (handle)

- Use to send integer data declared with `selected_integer_kind`

Fortran kind syntax (F90 syntax) Complex data

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)

INTEGER P, R, NEWTYPE, IERROR
```

- `P:` precision decimal digits
- `R:` exponent range
- `NEWTYPE:` requested MPI datatype (handle)

- Use to send complex data declared with `selected_real_kind`
- Either `P` or `R` may be `MPI_UNDEFINED`

When do MPI calls return

- **`MPI_Send`**
  - **returns:** send-buffer is safe to be overwritten
  - Whether data has or hasn't arrived depends – no idea!!
- **`MPI_Recv`**
  - **returns:**
    - receive buffer contains the data
    - you can now use that data
    - in mpi4py: the Python object is returned
- Both can involve significant waiting time!!

Timing MPI code

- MPI offers a very nice command to measure performance
- In C:

```
double MPI_Wtime(void)
```

- In Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

  no **`IERROR`**
- In Python:

```
comm.Wtime()
```

- **`MPI_WTIME`** returns **seconds** since some time in the past
- Don't use standard Fortran system clock

Summary

- This lecture has introduced
  - Sending point-to-point messages in MPI
  - Most used predefined MPI data types
  - No need to handle datatypes for standard Python objects
- Discussed timing in MPI codes

Exersise π^2

- We have $$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$
- Write a serial program to calculate π^2 in C or Fortran
- Parallelise it using MPI
  - When using ***p*** processors, each task should do **1/*p*** of the number of terms in the sum
  - Each task should work out, the total number of task and the range of terms he needs to work on
  - Once done with his terms, each task sends its partial result to rank 0
  - Rank 0 receives the partial results, calculates the final result and prints it to the screen