

# Avoiding deadlock: Non blocking communication

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

1

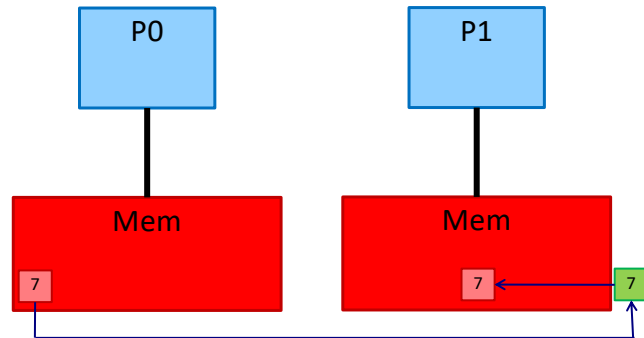
## Overview

- Eager vs. rendezvous protocol
- Deadlock
- Non-blocking communication
- Performance implications from non-blocking communication

2

## Small messages: Eager protocol

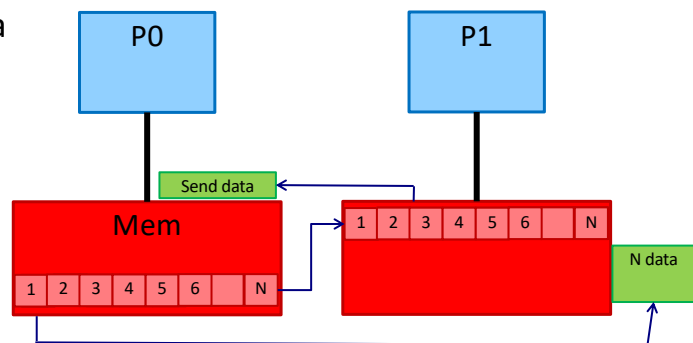
- Eager sending:
- When send is issued, data transferred to buffer on receiver
- There it gets buffered until matching receive is posted
- After receive is posted, copied into process memory
- Eager sending is not suitable for large messages:
- Receiver or Sender: not enough buffer memory to hold transferred data



3

## Large messages: Rendezvous protocol

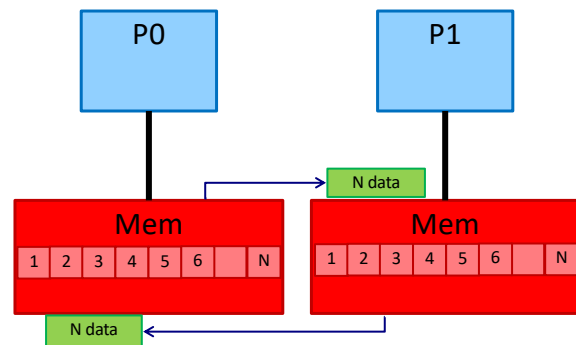
- When send is issued, meta-data transferred, e.g. N integers to come
- Matching receive is posted
  - Memory on receiver available (arg. of recv)
  - Data request is sent to sender
- Sender sends actual data
  - Sender must not overwrite send buffer until copied
  - Sender will wait forever if no matching receive is posted



4

## Deadlock

- Both tasks use a send operations which goes “rendezvous”
- Both tasks **wait for ever** for the request to send data
- **MPI\_Send** can not return, buffer can not be overwritten
- You have to assume that any **MPI\_Send** waits for the receive
  - Different system in future
  - Future use more data



5

## Synchronous send

In C:

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

In Fortran 90:

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

In Python:

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

- Same arguments as MPI\_Send
- Call returns only **after** receive has been posted
- Always uses **rendezvous** or similar protocol
- **Useful for testing for deadlock problem in a code**

6

## Non-blocking communication (overview)

- Avoid deadlock by using non-blocking send and/or receive
- Non-blocking calls return directly, in general:
  - Not safe to overwrite a send buffer
  - Receive buffer does not contain data
- Additional calls: **MPI\_Wait**, **MPI\_Test**, **MPI\_Waitall** to check communication has finished
- A “checker” has to be called, otherwise
  - **nasty memory** or **resource** leak
  - code may behave **non-deterministic**

7

## No deadlock by non-blocking receive

1. Both tasks issue non-blocking receive
    - Receive buffer becomes available
    - Program continues executing
  2. Both tasks post send “rendezvous” operations
    - Since receive buffers available, send transfers data & returns
  3. Both tasks check on receive (e.g. **MPI\_Wait**):
    - Ensures data has arrived in receive buffer
    - Checking operation required – it completes operation
  4. Continue with calculation and use received data
- **Rem:** The non-blocking call continues work, when program is doing other work

8

## Non-blocking receive in C

```
int MPI_Irecv(void* buf, int count, MPI_Datatype
    datatype, int source, int tag, MPI_Comm comm,
    MPI_Request *request)
```

- buf: address of receive buffer ([output](#))
- count: number of elements to be received
- datatype: data type of buffer (explained further down)
- source: rank of sender (data origin)
- tag: message tag (needs to match the send!)
- comm: communicator
- request: request handle ([output](#)), pass into “checker”

9

## Non-blocking receive in Fortran 90

```
MPI_IRecv(BUF, COUNT, DATATYPE, SOURCE, TAG, &
    COMM, REQUEST, IERROR)
```

```
<TYPE>:: BUF
```

```
INTEGER:: COUNT, DATATYPE, SOURCE, TAG, &
    COMM, REQUEST, IERROR
```

- BUF: address of receive buffer ([output](#))
- COUNT: number of elements to be received
- DATATYPE: data type of buffer (explained further down)
- SOURCE: rank of sender (data origin)
- TAG: message tag (needs to match the send!)
- COMM: communicator
- REQUEST: request handle ([output](#)), pass into checker

10

## Non-blocking receive in Python

```
req = comm.irecv(source=source, tag=tag)
```

- `source`: rank of sender (data origin)
- `tag`: message tag (needs to match the send!)
- A `Request` object is returned
- Python object is returned by the `wait` method of `Request` object

- Example:

```
req = comm.irecv(source=0, tag=0)  
obj = req.wait()
```

11

## Non-blocking receive in Python (large data)

```
req = comm.irecv(buf, source=source, tag=tag)
```

- When sending large data via `irecv`, you may see the “message truncated” error.
- In such case a buffer will be needed by `irecv`

- Example:

```
# 1 MB buffer; make it larger if needed  
buf = bytearray(1<<20)  
req = comm.irecv(buf, source=0, tag=0)  
obj = req.wait()
```

12

## Non-blocking send in C: MPI\_Isend

```
int MPI_Isend(void* buf, int count, MPI_Datatype
             datatype, int dest, int tag, MPI_Comm comm,
             MPI_Request *request)
```

- buf: address of send buffer
- count: number of elements to be send
- datatype: data type of buffer (explained further down)
- dest: rank of receiver
- tag: message tag (put 0 if you don't need)
- comm: communicator
- request: request handle ([output](#)), pass into checker

13

## Non-blocking send Fortran 90: MPI\_ISEND

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

- ```
<TYPE>:: BUF
INTEGER:: COUNT, DATATYPE, DEST, TAG, COMM
INTEGER:: REQUEST, IERROR
```
- BUF: address of send buffer
  - COUNT: number of elements to be send
  - DATATYPE: data type of buffer (explained further down)
  - DEST: rank of receiver
  - TAG: message tag (put 0 if you don't need)
  - COMM: communicator
  - REQUEST: request handle ([output](#)), pass into checker

14

## Non-blocking send in Python: isend

```
req = comm.isend(obj, dest=dest, tag=tag)
```

- `obj`: the Python object to be sent
- `dest`: rank of receiver
- `tag`: message tag (optional; default is 0)
- A `Request` object is returned
- The `Request` object has the `wait` method
- Example:
 

```
req = comm.isend(data, source=0, tag=0)  
req.wait()
```

15

## Waiting on non-blocking communication

- In C:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- In Fortran 90:

```
MPI_WAIT(REQUEST, STATUS, IERROR)
```

```
INTEGER:: REQUEST, STATUS(MPI_STATUS_SIZE)
```

```
INTEGER:: IERROR
```

- `request`: handle from `MPI_Irecv` or `MPI_Isend`
- `status`: status info from complete operation (recv)
- In Python:

```
req.wait()
```

- Call waits (blocks) until communication has finished

16



## Example in C

```
int b, a=17;
MPI_Request s_req, r_req;
MPI_Status  cstat;

MPI_Irecv(&b, 1, MPI_INT, npr, 0, myc, &r_req);
MPI_Isend(&a, 1, MPI_INT, npr, 0, myc, &s_req);

/* don't touch a or b here, can do other work */

MPI_Wait(&s_req, &cstat);
a = 9;    /* now safe to change a */
MPI_Wait(&r_req, &cstat);
b = 2*b;  /* b can now be used */
```

17

## Example in Fortran 90

```
integer:: b, a=17
integer:: s_req, r_req, ierror
integer, dimension(MPI_STATUS_SIZE) :: cstat

MPI_Irecv(b, 1, MPI_INTEGER, npr, 0, myc, &
  r_req, ierror)
MPI_Isend(a, 1, MPI_INTEGER, npr, 0, myc, &
  s_req, ierror)
! don't touch a or b here, can do other work
MPI_Wait(s_req, cstat, ierror)
a = 9      ! now safe to change a
MPI_Wait(r_req, cstat, ierror)
b = 2*b    ! b can now be used
```

18

## Example in Python

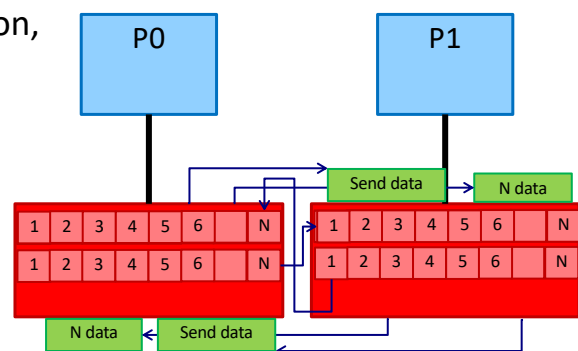
```
from mpi4py import MPI

comm = MPI.COMM_WORLD
a = 17
# for large messages supply irecv with a buffer
r_req = comm.irecv(source=0, tag=myc)
s_req = comm.isend(a, dest=0, tag=myc)
# don't touch a or b here, can do other work
s_req.wait()
a = 9
b = r_req.wait()
b = 2*b
```

19

## Example: resolving deadlock: MPI\_Isend

- Both tasks use an **MPI\_Isend** operation
- **Isend** publishes the send operation, e.g. transfers meta data
- **MPI\_Isend** calls return
- Don't modify sendbuffer yet!!
- Both task post **MPI\_Recv**
  - Publish recv-buffers
  - Match of send & recv
  - Request data transfer
  - Receives data
- Tasks **must** call **MPI\_Wait**
  - Finalises the **Isend** and frees resources



20

## Resolved deadlock with **MPI\_Isend** in C

```
MPI_Isend(p_send_buf, n, MPI_INT, neigh, 1, comm, &req);

p_recv_buf = (int*)malloc(n * sizeof(int));

MPI_Recv(p_recv_buf, n, MPI_INT, neigh, 1, comm, &stat);

MPI_Wait(&req, &stat);    // wait required

for (int i=0; i<n; i++)
    p_send_buf[i] = p_recv_buf[i];

free(p_recv_buf);
```

21

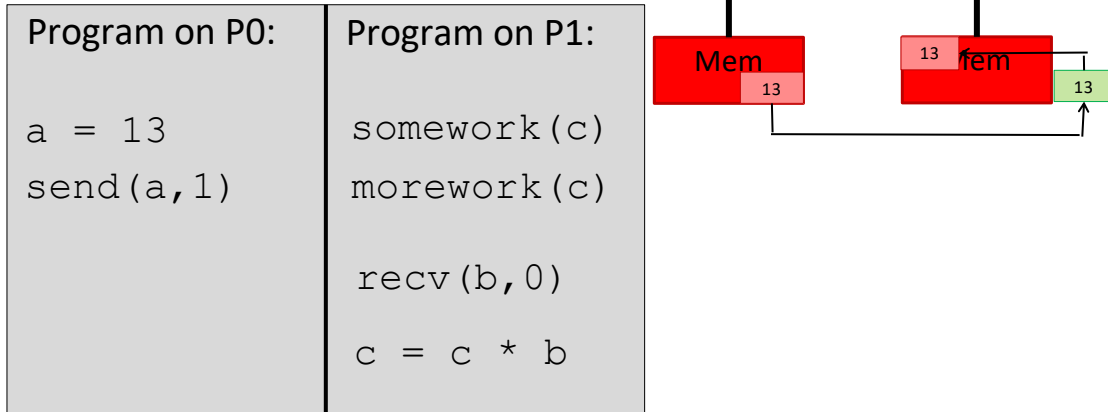
## Non-blocking communication and performance

- Non blocking communication can be used to **hide communication time**:
  - Issue communication calls early
  - Do other work, not associated with the communication buffers
  - Delay the Wait/Test for the communication calls until data is needed
  - Particular useful on modern RDMA hardware (communication not done by processor)
- Pre-posting non-blocking receives can reduce data traffic
  - Data goes directly into application buffer instead of buffer in MPI library

23

## Late receive

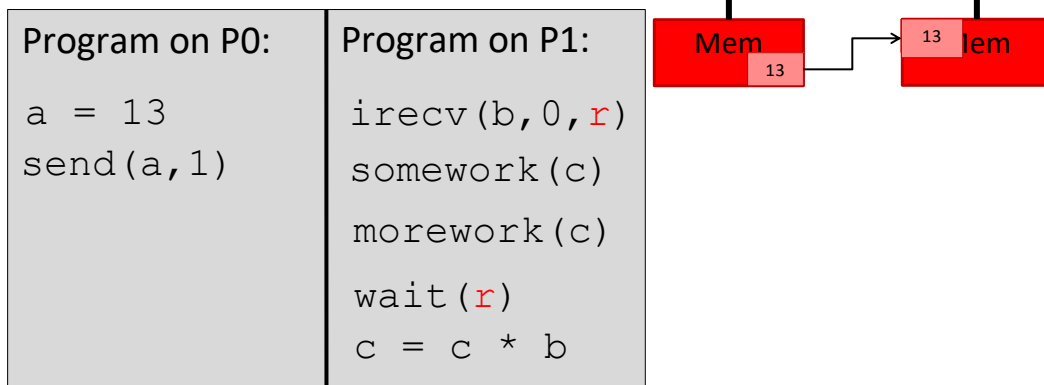
- If receive not posted, data needs storing somewhere



24

## Pre-post receive

- If non-blocking receive pre-posted, can go directly into application space



25

## Summary

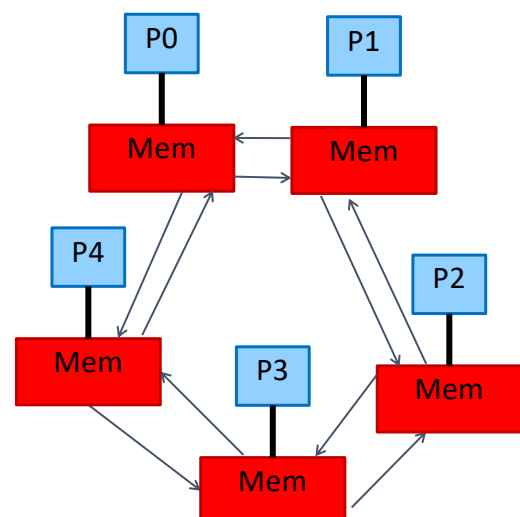
- Explained the danger of deadlock in p-2-p communication
- Non-blocking communication
  - to avoid deadlock
  - to overlap communication and calculation
  - preposted receives: performance boost on certain architectures
- **Important:** Every non-blocking call (`MPI_Isend`, `MPI_Irecv`, ...) **requires** a matching `MPI_Wait` or a true `MPI_Test`

26

## Exercise

### Messages around a ring

- Each processor should
  - Initialise 2 integer arrays
    - set one to rank number
    - other to 1000x rank number
  - Send one up and down
  - Receive from neighbours
  - Sum up received data in separate (up/down) sums
  - Pass the received data on, to continue in the direction they came
- Checks after final exchange:
  - Last receive (up & down) equal 1x and 1000x own rank
  - The two sums equal:  $\frac{1}{2}n(n-1)$  and  $500n(n-1)$



28