Programming with MPI

Basic send and receive

Jan Thorbecke

Acknowledgments

- This course is partly based on the MPI course developed by
  - Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
    http://www.hlrs.de/home/
- CSC – IT Center for Science Ltd.
  https://www.csc.fi

CSC-IT CENTER FOR SCIENCE

CSC

HLRS

High-Performance Computing Center | Stuttgart

Contents

- Initialisation of MPI
  - exercise: HelloWorld
- Basic Send & Recv
  - exercise: Sum
  - exercise: SendRecv
- more Send Receive messages
  - exercise: PingPong (optional)
  - exercise: Ring (optional)

A Minimal MPI Program (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    err = MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

A Minimal MPI Program (Fortran 90)

```
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

All MPI fortran call return an error message

Starting the MPI Environment

- **MPI_INIT ( )**

  Initializes MPI environment. This function must be called and must be the first MPI function called in a program (exception: **MPI_INITIALIZED**)

  Syntax

```
int MPI_Init (  int *argc, char ***argv )

MPI_INIT ( IERROR )
INTEGER IERROR
```

NOTE: Both C and Fortran return error codes for all calls.

Exiting the MPI Environment

- **`MPI_FINALIZE (   )`**

  Cleans up all MPI state. Once this routine has been called, no MPI routine (even **`MPI_INIT`**) may be called

Syntax

```
int MPI_Finalize ( );

MPI_FINALIZE  ( IERROR )
INTEGER IERROR
```

MUST call MPI_FINALIZE when you exit from an MPI program

C and Fortran Language Considerations

- Bindings
  - C
    - All MPI names have an **`MPI_`** prefix
    - Defined constants are in all capital letters
    - Defined types and functions have one capital letter after the prefix; the remaining letters are lowercase
  - Fortran
    - All MPI names have an **`MPI_`** prefix
    - No capitalization rules apply
    - last argument is an returned error value

MPI Function Format

- C:

```
#include <mpi.h>

   error = MPI_Xxxxxx(parameter, …);
```

- Fortran:

```
INCLUDE 'mpif.h'

   CALL MPI_XXXXXX( parameter, ..., IERROR )
```

don't forget

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?

- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

MPI Rank

- MPI runtime assigns each process a **rank**, which can be used as an ID of the processes
  - ranks start from 0 and extent to N-1
- Processes can perform different tasks and handle different data based on their **rank**

```
...
if ( rank == 0 ) {
   …
   }
if ( rank == 1) {
   …
   }
...
```

Exercise: Hello World

- README.txt
  - Try to answer the questions in the README
  - How is the program compiled?
  - How do you run the parallel program?
- There is a C and Fortran version of the exercise.

Better Hello (C)

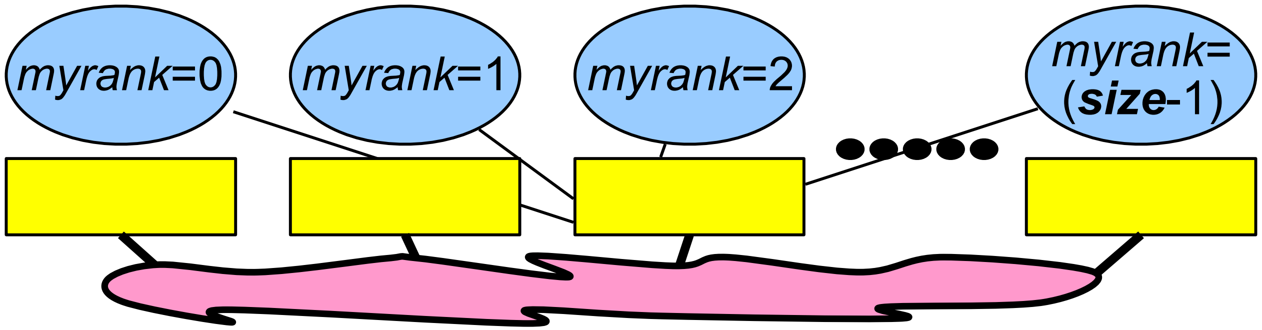
```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Better Hello (Fortran)

```
program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Rank

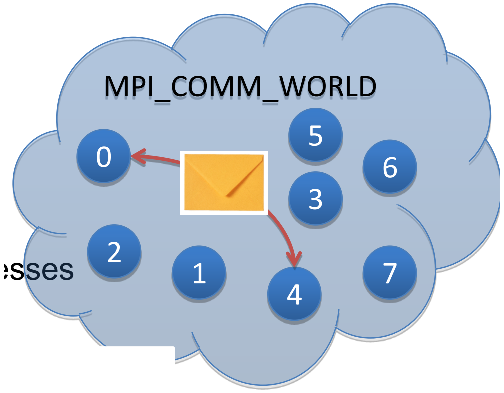
- The rank identifies different processes within a communicator
- The rank is the basis for any work and data distribution.
- C: int MPI_Comm_rank(MPI_Comm comm, int *rank)
- Fortran: MPI_COMM_RANK(comm, rank, ierror)
  INTEGER comm, rank, ierror



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

Some Basic Concepts

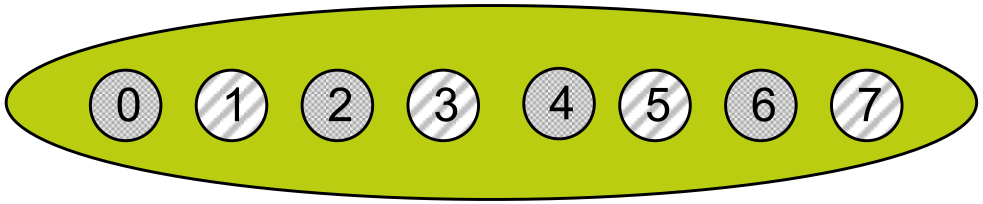
- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.
- Each process has it own number
  - starts with 0
  - ends with (size-1)



Communicator

- Communication in **`MPI`** takes place with respect to communicators
- **`MPI_COMM_WORLD`** is one such predefined communicator (something of type **“`MPI_COMM`”**) and contains group and context information
- **`MPI_COMM_RANK`** and **`MPI_COMM_SIZE`** return information based on the communicator passed in as the first argument
- Processes may belong to many different communicators

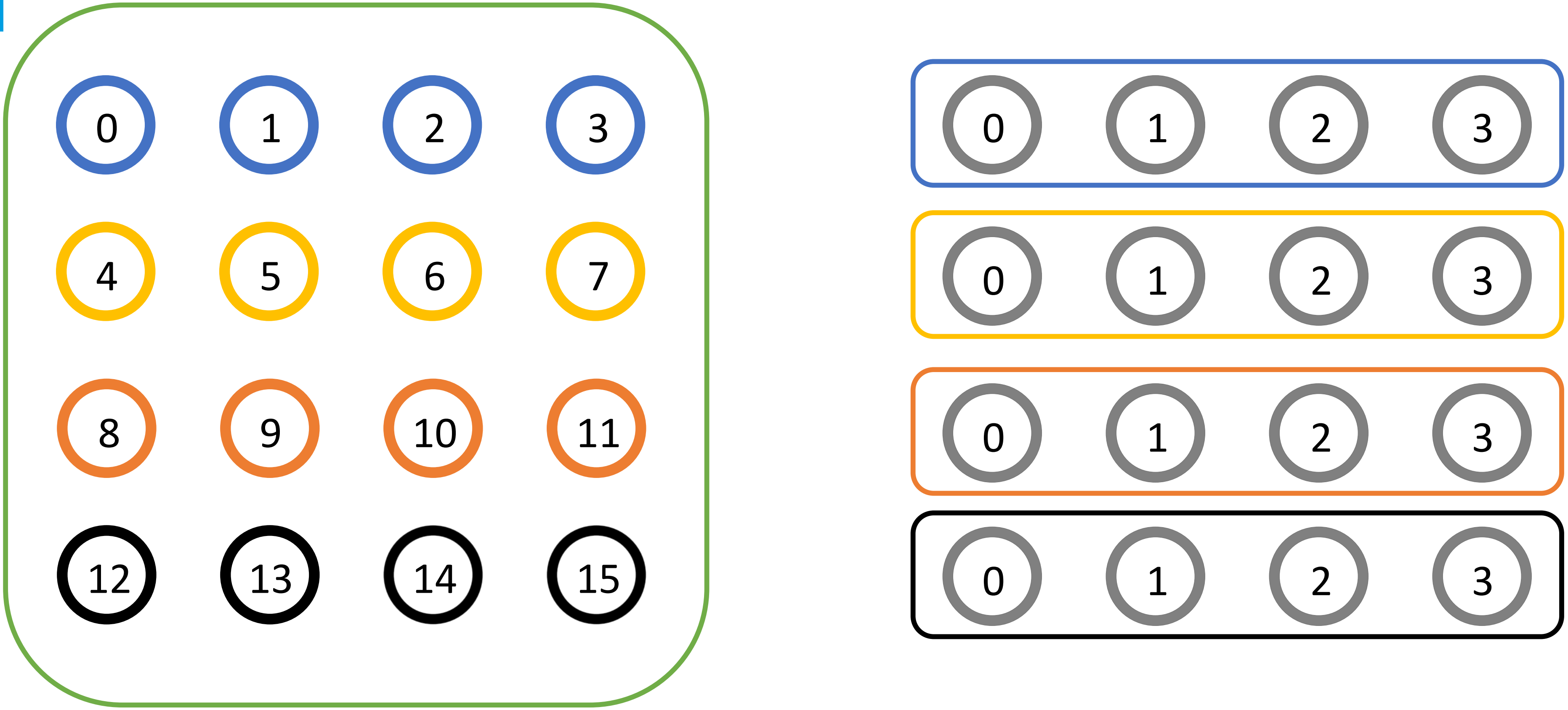
Rank-->



`MPI_COMM_WORLD`

Split communicator

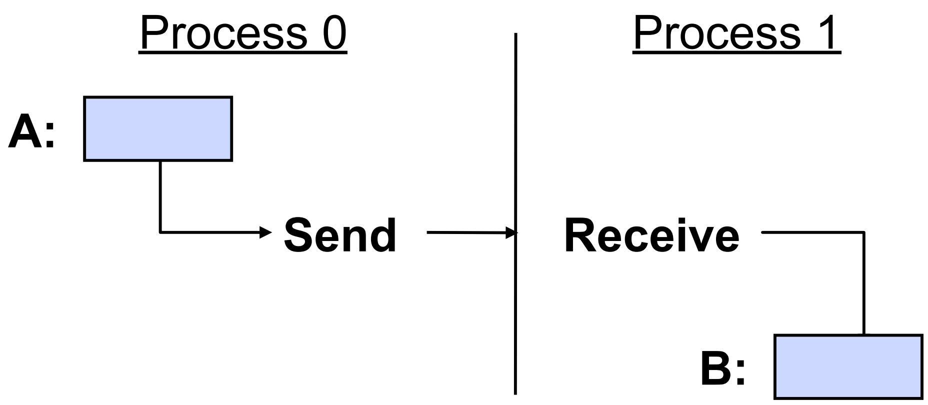
Split a Large Communicator Into Smaller Communicators



Point to Point communication

MPI Basic Send/Receive

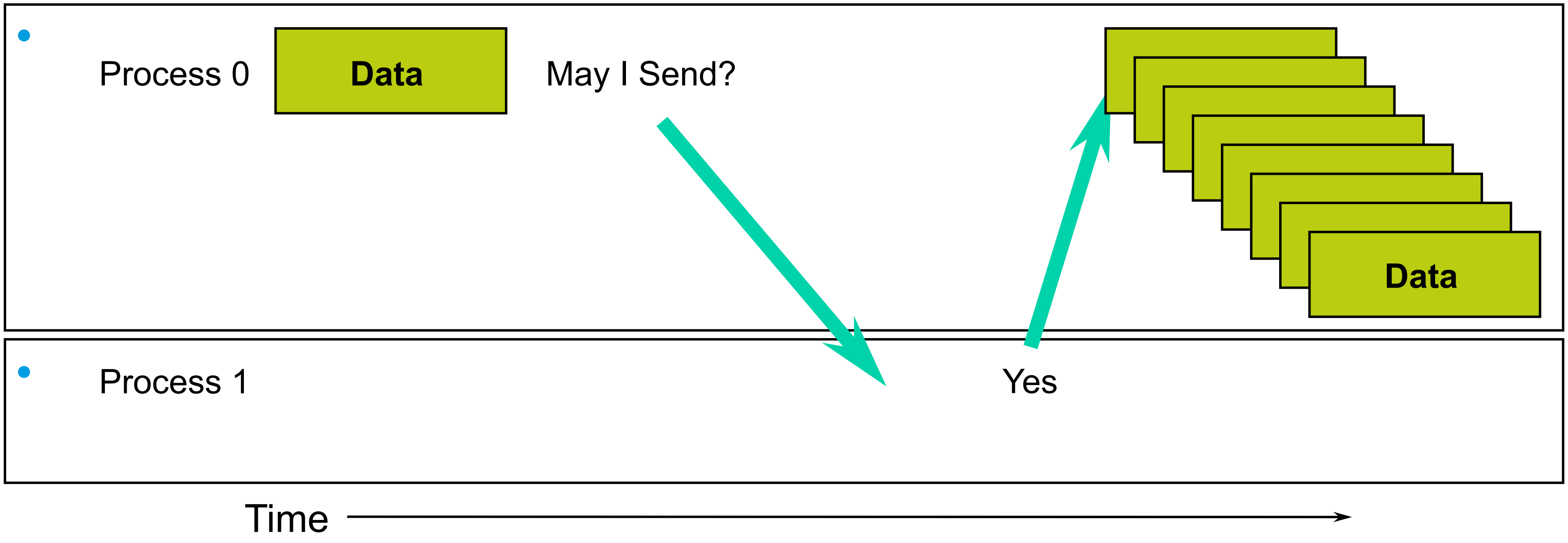
- Basic message passing process. Send data from one process to another



- Questions
  - To whom is data sent?
  - Where is the data?
  - What type of data is sent?
  - How much of data is sent?
  - How does the receiver identify it?

MPI Basic Send/Receive

- Data transfer plus synchronization



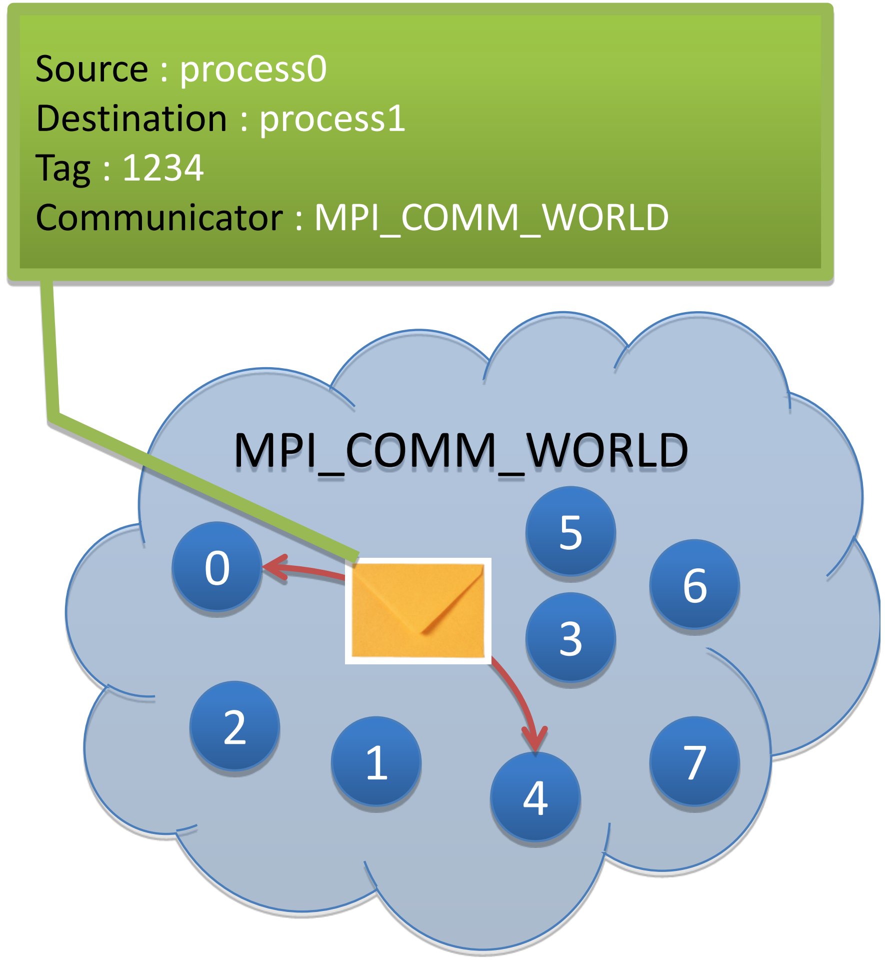
- Requires co-operation of sender and receiver
- Co-operation not always apparent in code
- Communication and synchronization are combined

Message Organization in MPI

- Message is divided into data and envelope

- data
  - buffer
  - count
  - datatype

- envelope
  - process identifier (source/des
  - message tag
  - communicator



MPI Basic Send/Receive

- Thus the basic (blocking) send has become:

  `MPI_Send ( buf, count, datatype, dest, tag, comm )`

  – Blocking means the function does not return until it is safe to reuse the data in buffer. The message may not have been received by the target process.

- And the receive has become:

  `MPI_Recv( buf, count, datatype, source, tag, comm, status )`

  - The source, tag, and the count of the message actually received can be retrieved from `status`

MPI C Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long_int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Fortran Datatypes

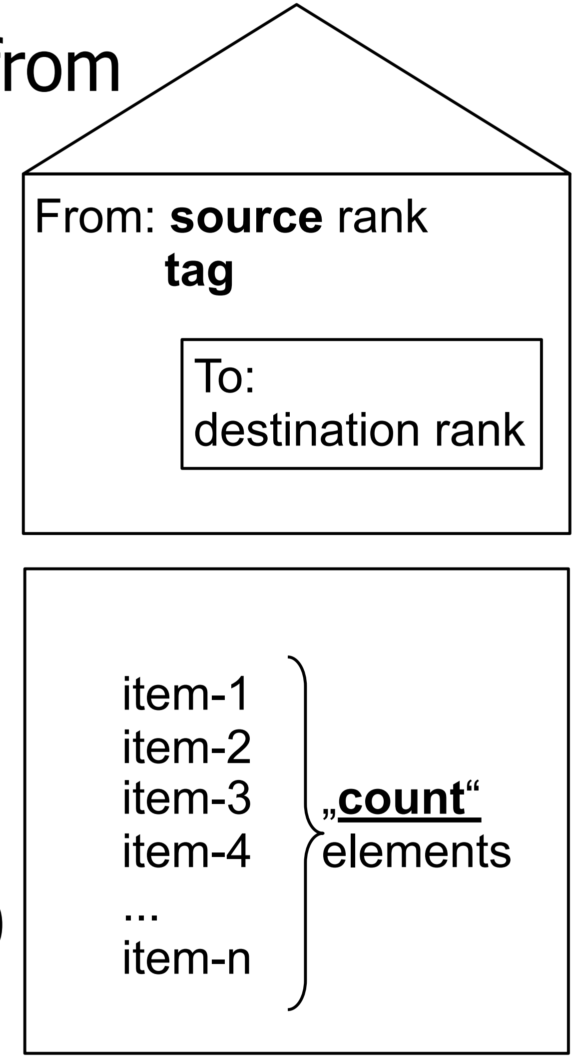
MPI FORTRAN	FORTRAN datatypes
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL*8
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Process Naming and Message Tags

- Naming a process
  - **destination** is specified by **`( rank, group )`**
  - Processes are named according to their rank in the group
  - Groups are defined by their distinct "communicator"
  - **`MPI_ANY_SOURCE`** wildcard rank permitted in a receive Tags are integer variables or constants used to uniquely identify individual messages
- Tags allow programmers to deal with the arrival of messages in an orderly manner
- MPI tags are guaranteed to range from 0 to 32767 by MPI-1
  - Vendors are free to increase the range in their implementations
- **`MPI_ANY_TAG`** can be used as a wildcard value

Communication Envelope

- Envelope information is returned from MPI_RECV in status.
- C:
  status.MPI_SOURCE
  status.MPI_TAG
  count via MPI_Get_count()
- Fortran:
  status(MPI_SOURCE)
  status(MPI_TAG)
  count via MPI_GET_COUNT()



Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.
- Receiver's buffer must be large enough.

Exercise: Send - Recv (1)

Write a simple program where every processor sends data to the next one. You may use as a starting point the basic.c or basic.f90. The program should work as follows:

- Let ntasks be the number of the tasks.
- Every task with a rank less than ntasks-1 sends a message to task myid+1. For example, task 0 sends a message to task 1.
- The message content is an integer array of 100 elements.
- The message tag is the receiver's id number.
- The sender prints out the number of elements it sends and the tag number.
- All tasks with rank ≥ 1 receive messages. You should check the MPI_SOURCE and MPI_TAG fields of the status variable (in Fortran you should check the corresponding array elements). Also check then number of elements that the process received using MPI_Get_count.
- Each receiver prints out the number of elements it received, the message tag, and the rank.
- Write the program using MPI_Send and MPI_Recv

Blocking Communication

- So far we have discussed *blocking* communication
  - **`MPI_SEND`** does not complete until **buffer** is empty (available for reuse)
  - **`MPI_RECV`** does not complete until **buffer** is full (available for use)

- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

Exercise: Send - Recv (2)

- Find out what the program deadlock (.c or .f90) is supposed to do. Run it with two processors and see what happens.

  a) Why does the program get stuck ?
  b) Reorder the sends and receives in such a way that there is no deadlock.
  c) Replace the standard sends with non-blocking sends (MPI_Isend/ MPI_Irecv) to avoid deadlocking. See the man page how to use these non-blocking
  d) Replace the sends and receives with MPI_SENDRECV.
  e) In the original program set maxN to 1 and try again.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

- What happens with

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This is called "unsafe" because it depends on the availability of system buffers.

Some Solutions to the “unsafe” Problem

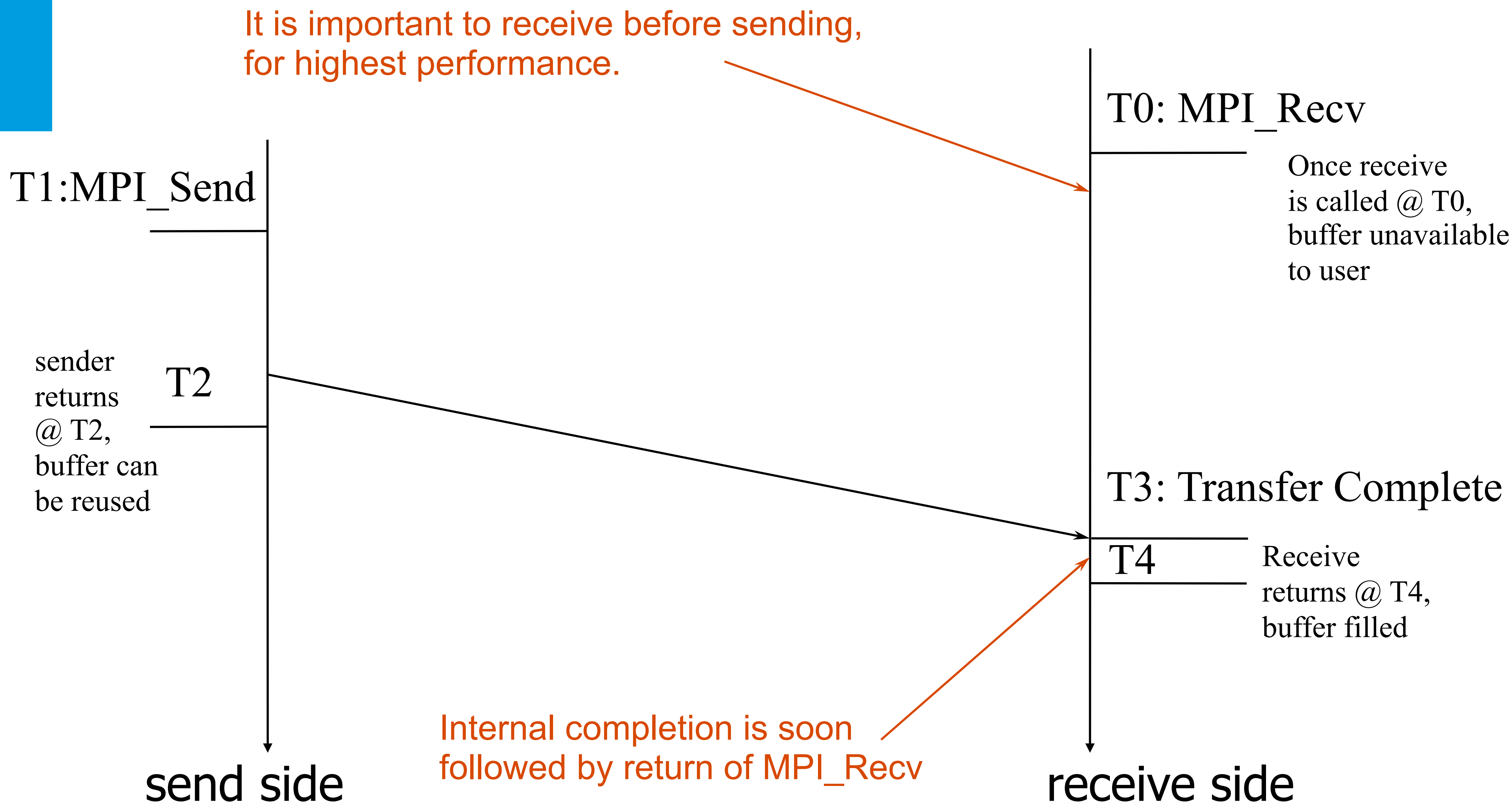
- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

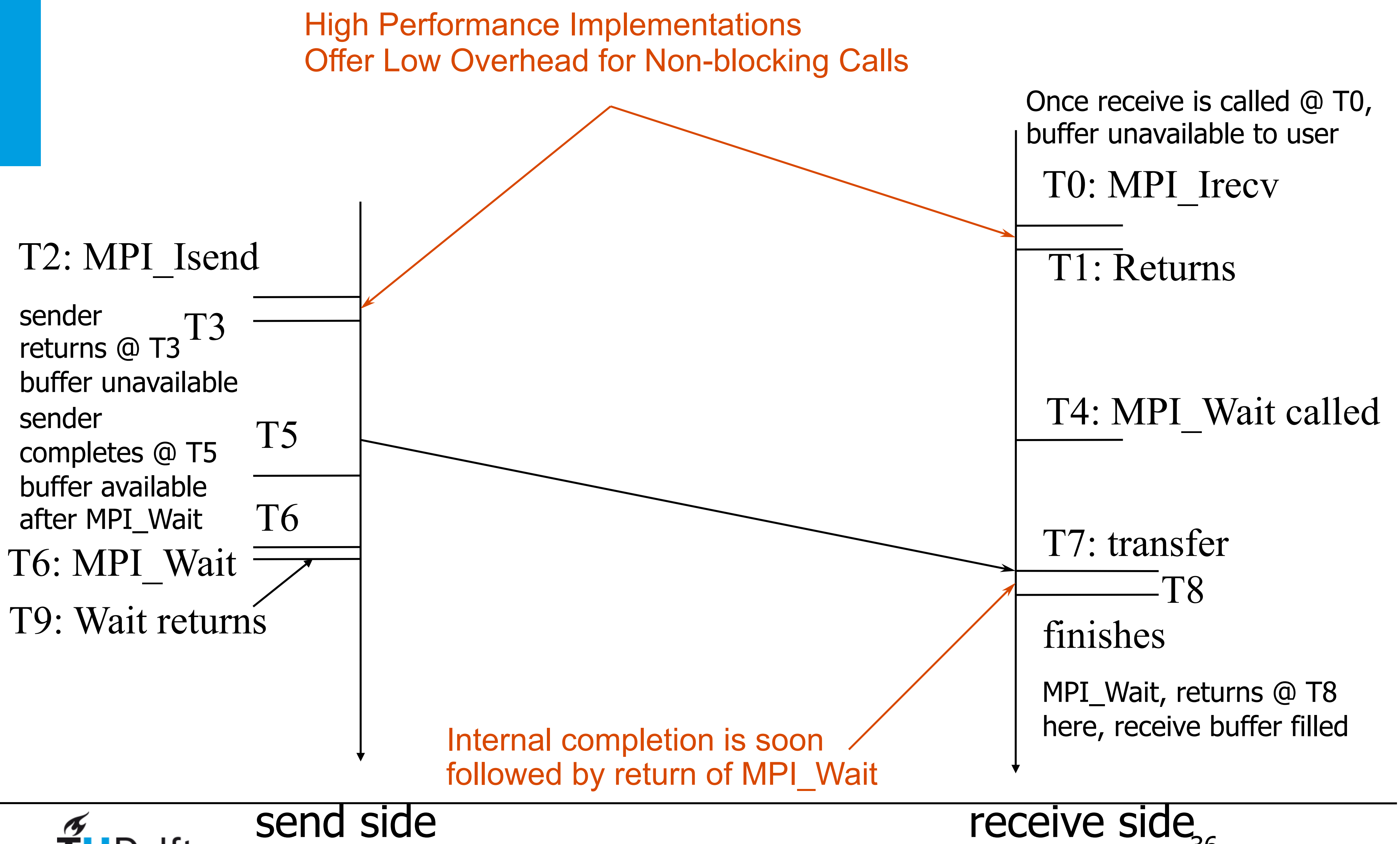
- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Blocking Send-Receive Diagram (Receive before Send)



Non-Blocking Send-Receive Diagram



Non-blocking Receive

- C:

  MPI_Irecv(buf, count, datatype, source, tag, comm,
  OUT &request_handle);

  MPI_Wait(INOUT &request_handle, &status);

- Fortran:

  CALL MPI_IRECV (buf, count, datatype, source, tag, comm,
  OUT request_handle, ierror)

  CALL MPI_WAIT(INOUT request_handle, status, ierror)

- buf must not be used between Irecv and Wait (in all progr. languages)

Message Completion and Buffering

- A send has completed when the user supplied buffer can be reused

```
*buf = 3;
MPI_Send ( buf, 1, MPI_INT, ... );
*buf = 4; /* OK, receiver will always receive 3 */

*buf = 3;
MPI_Isend(buf, 1, MPI_INT, ...);
*buf = 4; /* Undefined whether the receiver will get 3 or 4 */
MPI_Wait ( ... );
```

- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

Non-Blocking Communications

- Separate communication into three phases:
- Initiate non-blocking communication
  - returns Immediately
  - routine name starting with MPI_I…
- Do some work
  - “latency hiding”
- Wait for non-blocking communication to complete

Non-Blocking Communication

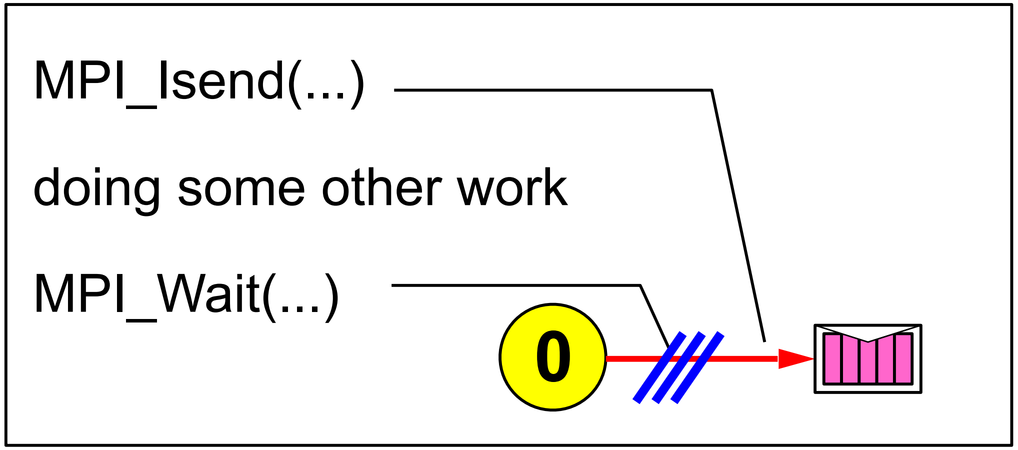
- Non-blocking (asynchronous) operations return (immediately) "request handles" that can be waited on and queried

```
MPI_ISEND( start, count, datatype, dest, tag,
  comm, request )
MPI_IRECV( start, count, datatype, src, tag,
  comm, request )
MPI_WAIT( request, status )
```

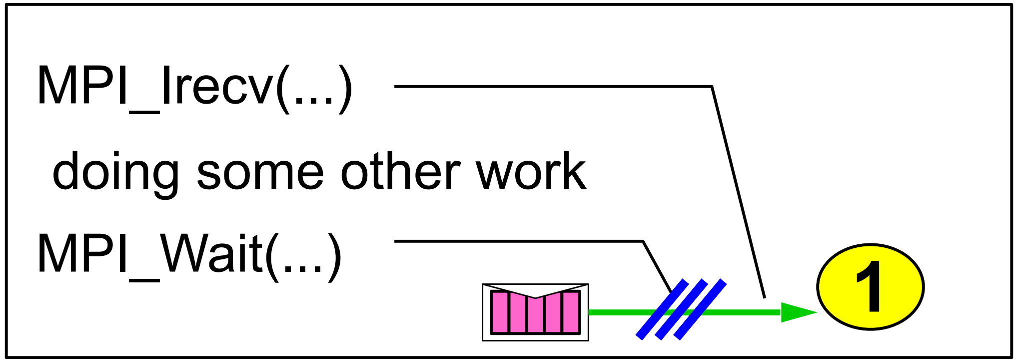
- Non-blocking operations allow overlapping computation and communication.
- Anywhere you use `MPI_Send` or `MPI_Recv`, you can use the pair of `MPI_Isend/MPI_Wait` or `MPI_Irecv/MPI_Wait`
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

Non-Blocking Examples

- Non-blocking **send**



- Non-blocking **receive**



= waiting until operation locally completed

Completion

- C:

  MPI_Wait(&request_handle, &status);

  MPI_Test(&request_handle, &flag, &status);

- Fortran:

  CALL MPI_WAIT(request_handle, status, ierror)
  completes if request is finished

  CALL MPI_TEST(request_handle, flag, status, ierror)
  test if request_handle is finished without waiting

- one must use
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == 1 or .TRUE.

Multiple Completion's

- It is often desirable to wait on multiple requests
- An example is a worker/manager program, where the manager waits for one or more workers to send it a message

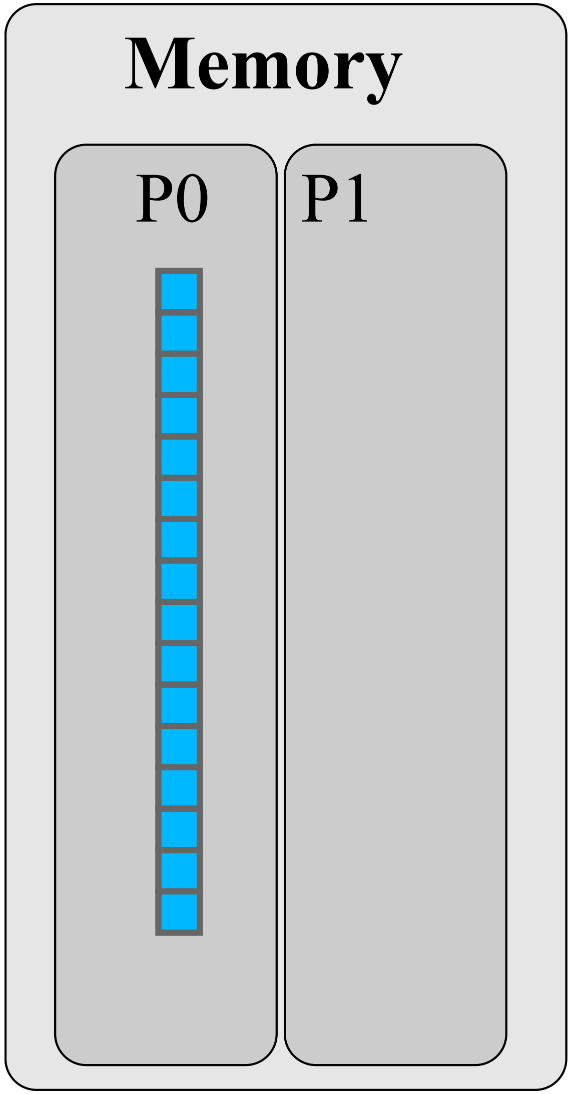
```
MPI_WAITALL( count, array_of_requests, array_of_statuses )

MPI_WAITANY( count, array_of_requests, index, status )

MPI_WAITSOME( incount, array_of_requests, outcount,
  array_of_indices, array_of_statuses )
```

- There are corresponding versions of **`TEST`** for each of these

Other Send Modes

- Standard mode **(MPI_Send, MPI_Isend)**
  - The standard **MPI_Send**, the send will not complete until the send buffer is empty
- Synchronous mode **(MPI_Ssend, MPI_Issend)**
  - The send does not complete until after a matching receive has been posted
- Buffered mode **(MPI_Bsend, MPI_Ibsend)**
  - User supplied buffer space is used for system buffering
  - The send will complete as soon as the send buffer is copied to the system buffer
- Ready mode **(MPI_Rsend, MPI_Irsend)**
  - The send will send eagerly under the assumption that a matching receive has already been posted (an erroneous program otherwise)

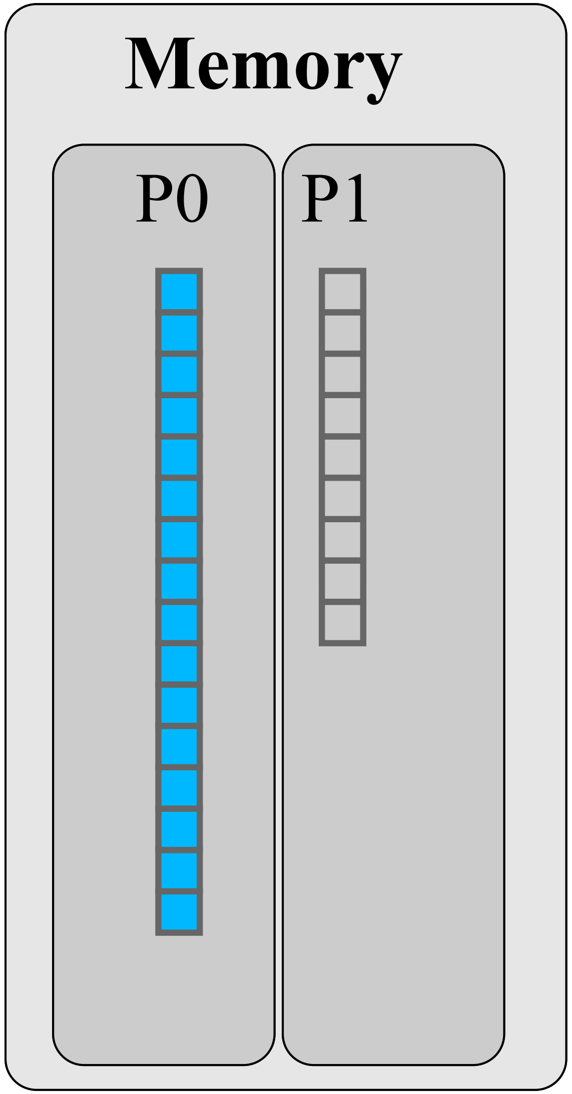
Parallel Sum



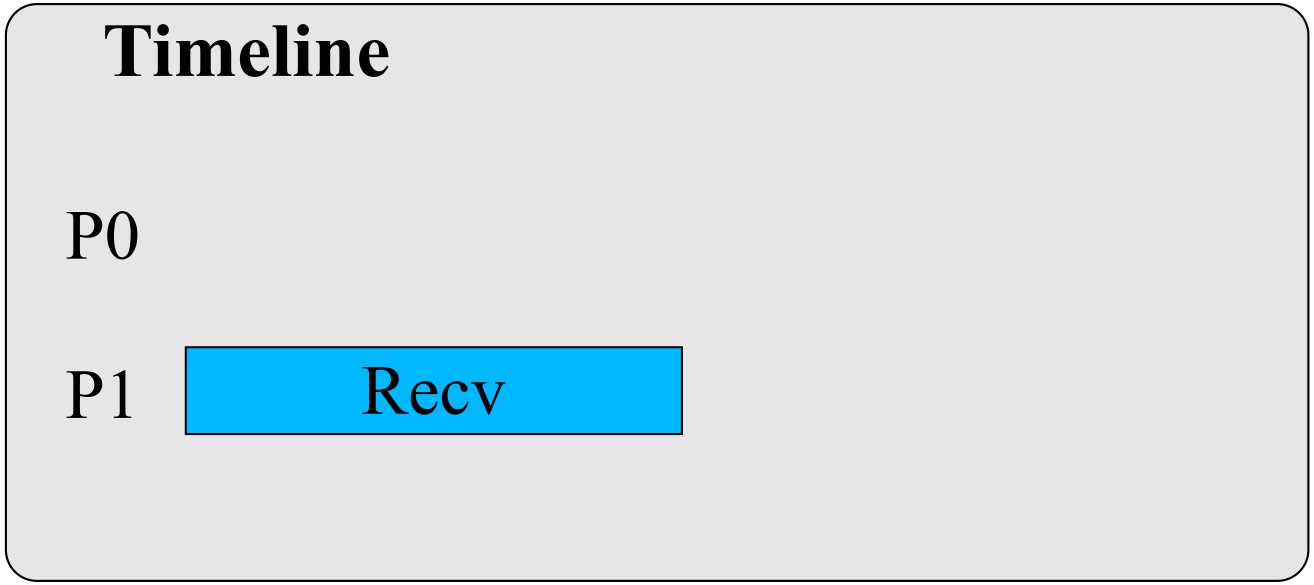
- Array is originally on process 0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segment
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

TU Delft

Parallel Sum

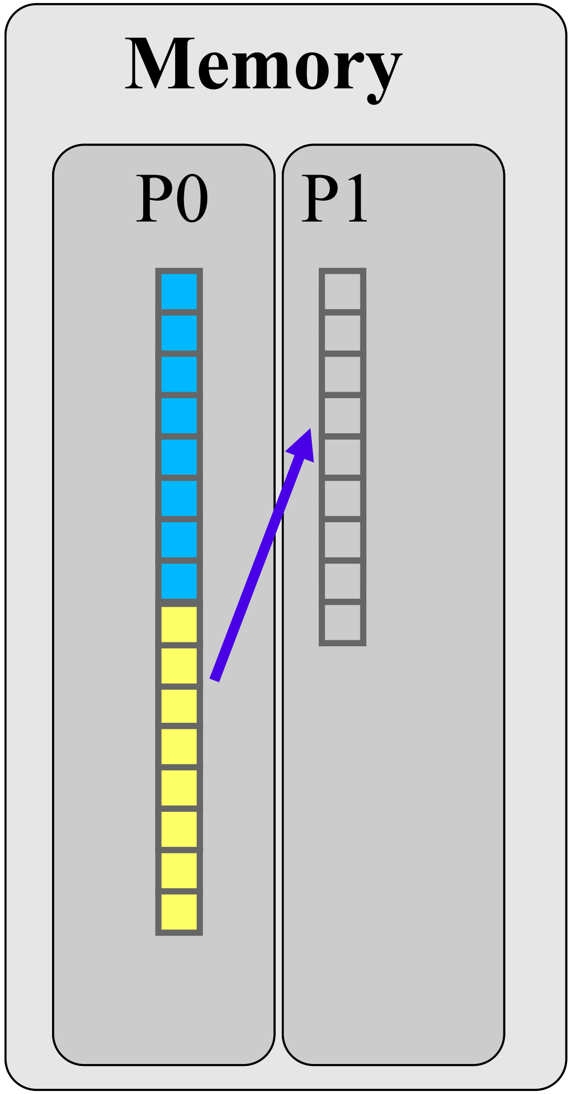


Step 1: Receive operation in scatter

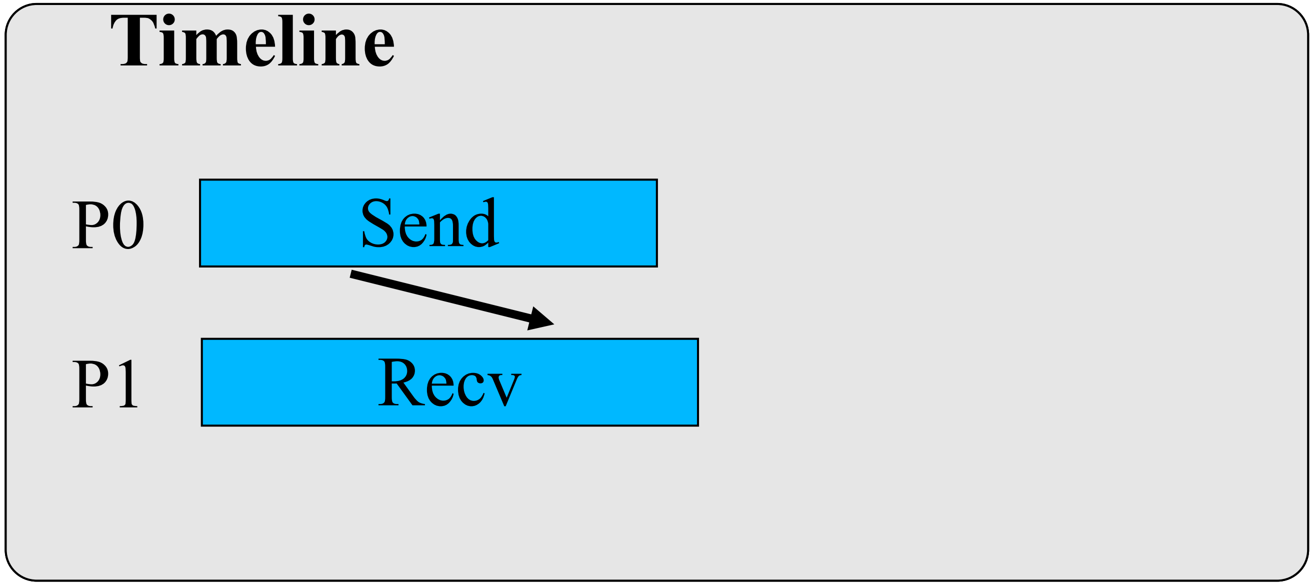


P1 posts a receive to receive half of the array from process 0

Parallel Sum

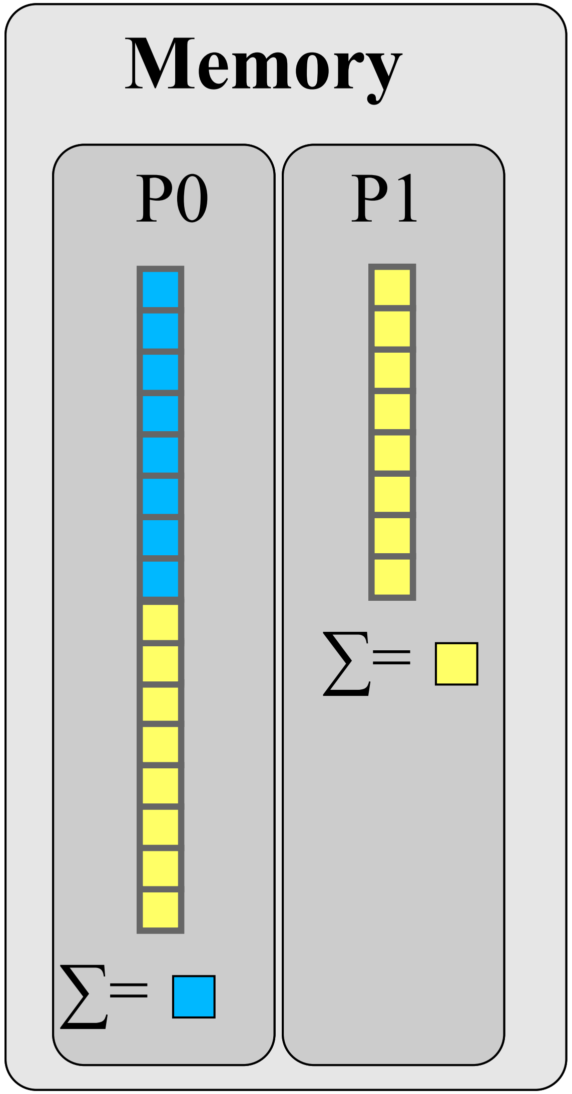


Step 2: Send operation in scatter

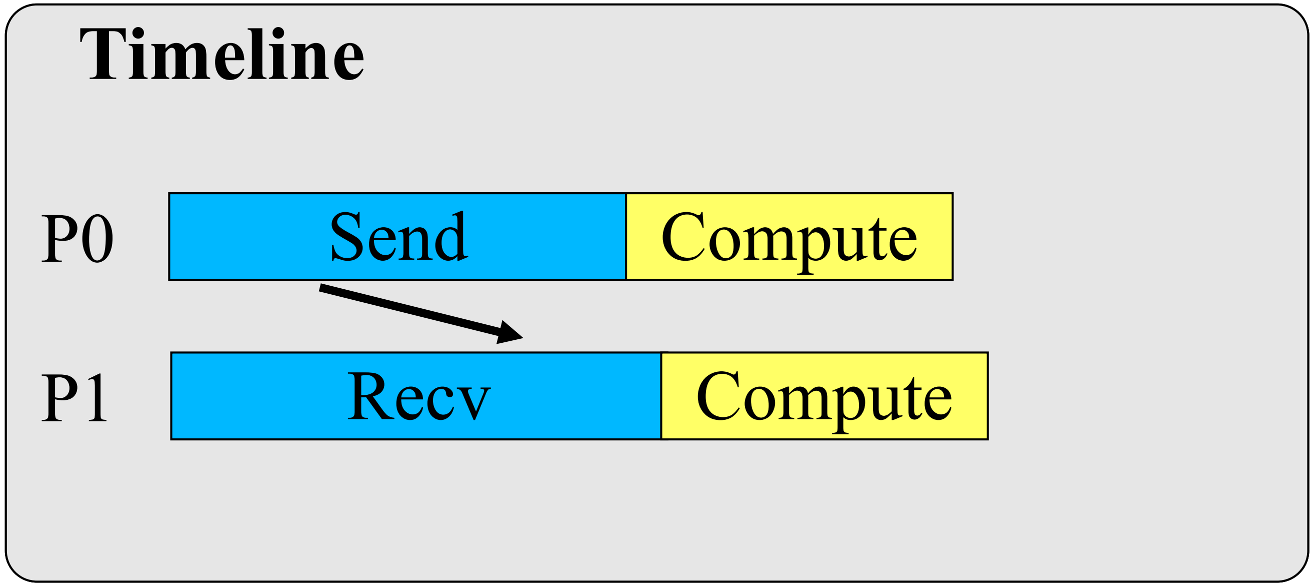


P0 posts a send to send the lower part of the array to P1

Parallel Sum

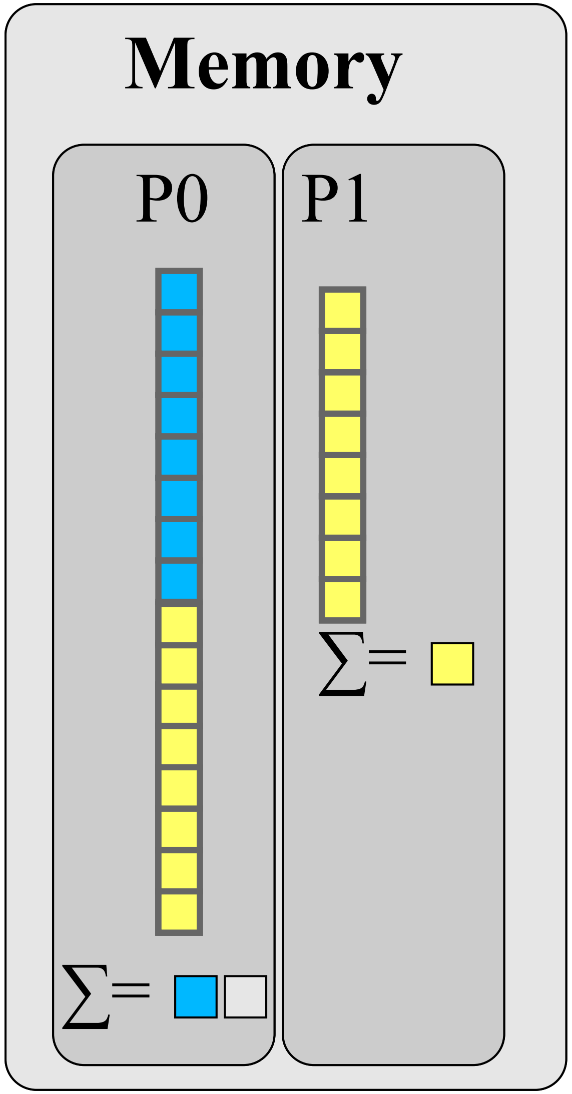


Step 3: Compute the sum in parallel

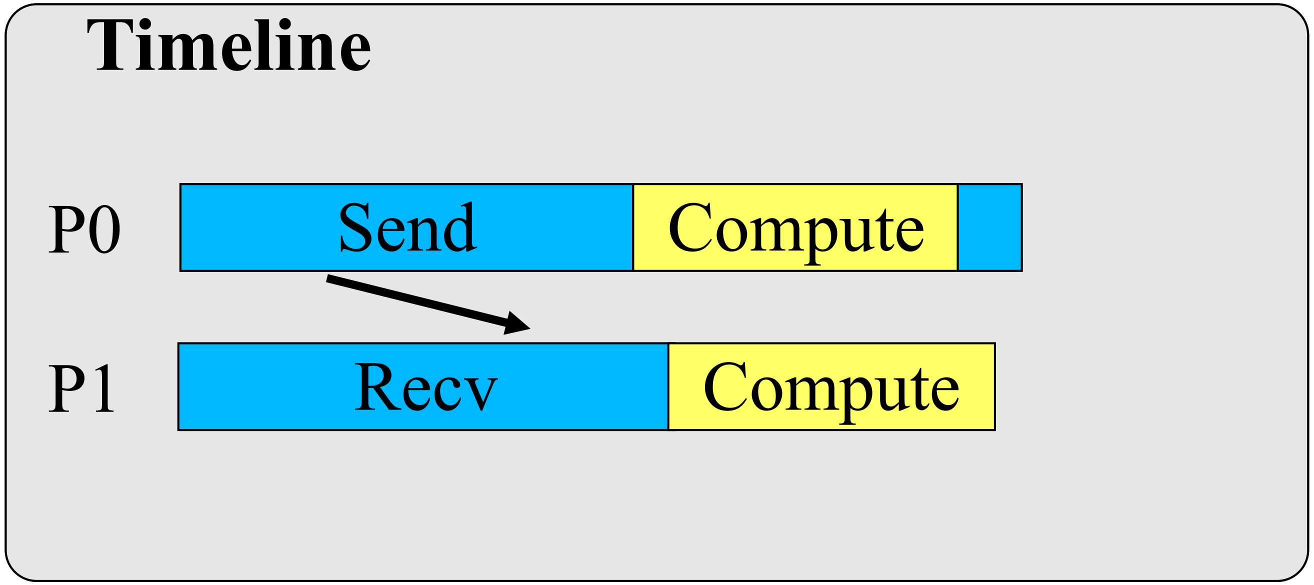


P0 & P1 computes their partial sums and store them locally

Parallel Sum

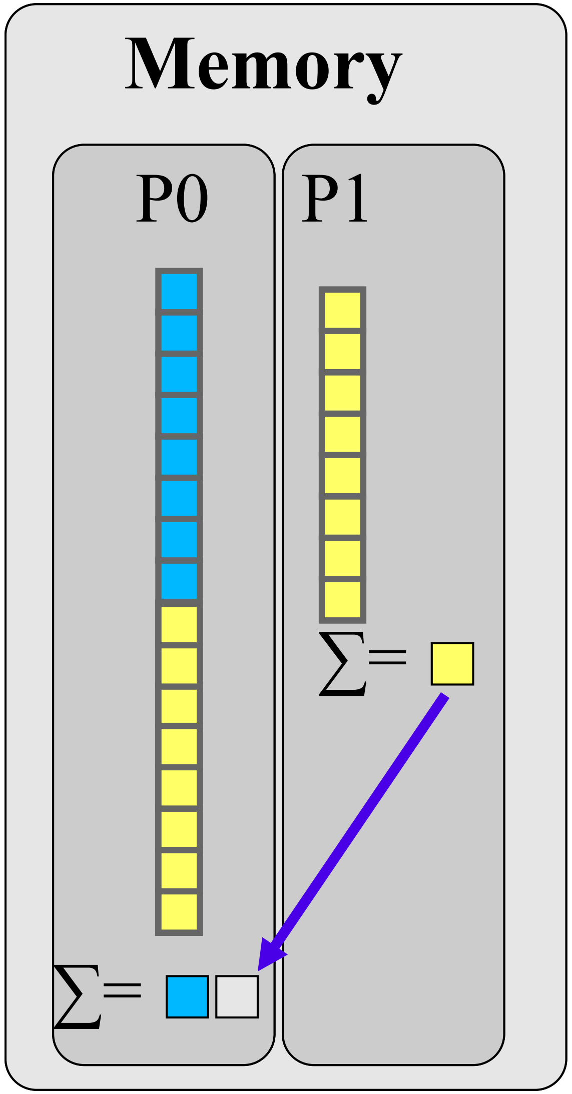


Step 4: Receive operation in reduction

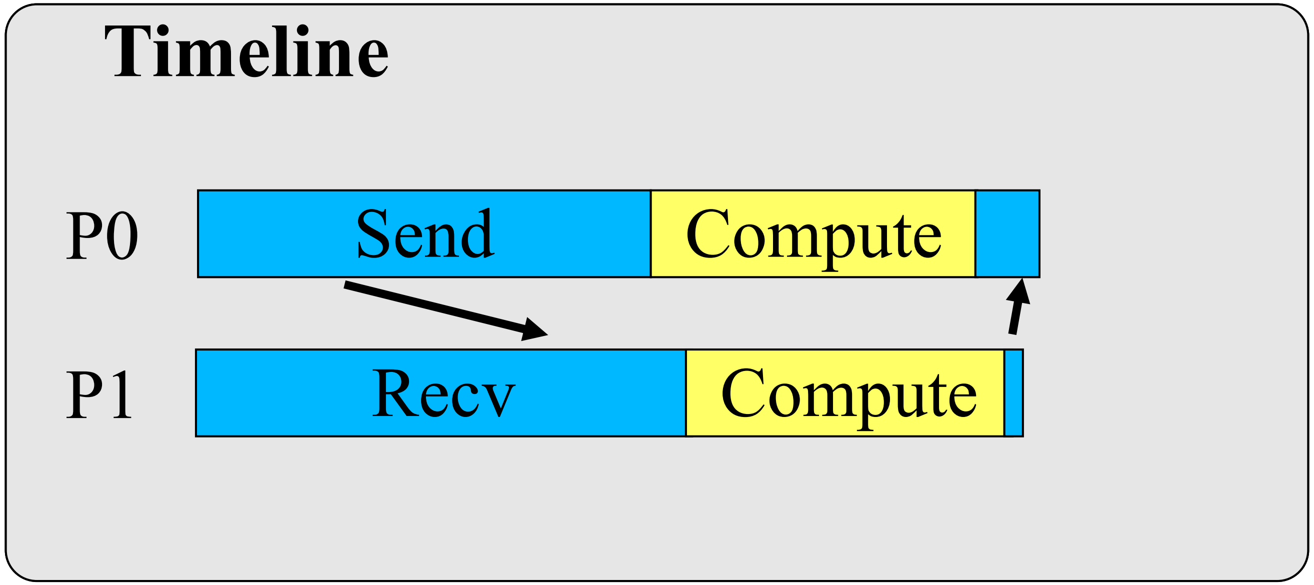


P0 posts a receive to receive partial sum

Parallel Sum

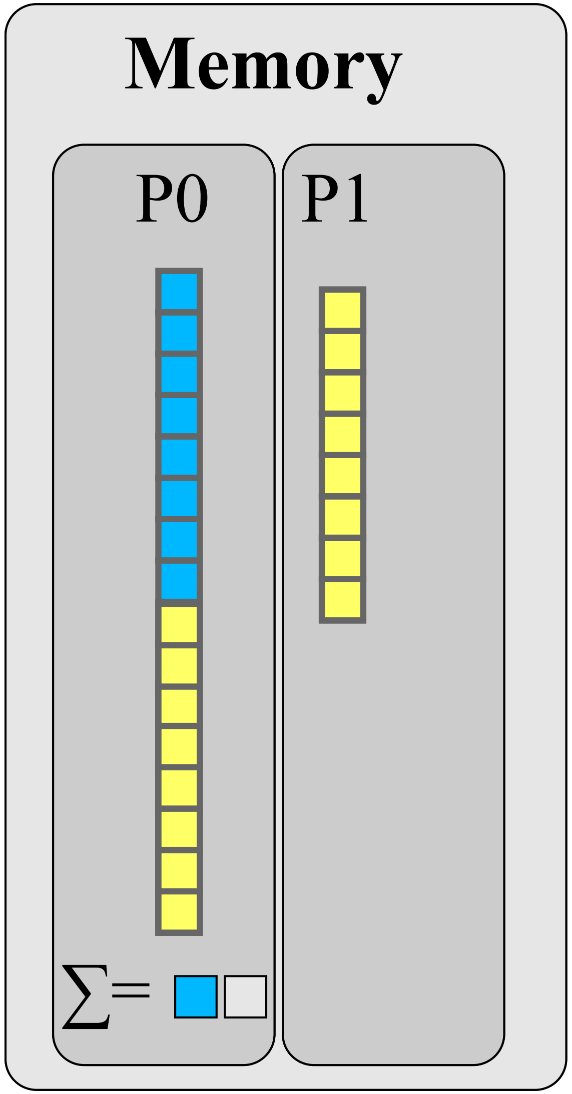


Step 5: Send operation in reduction

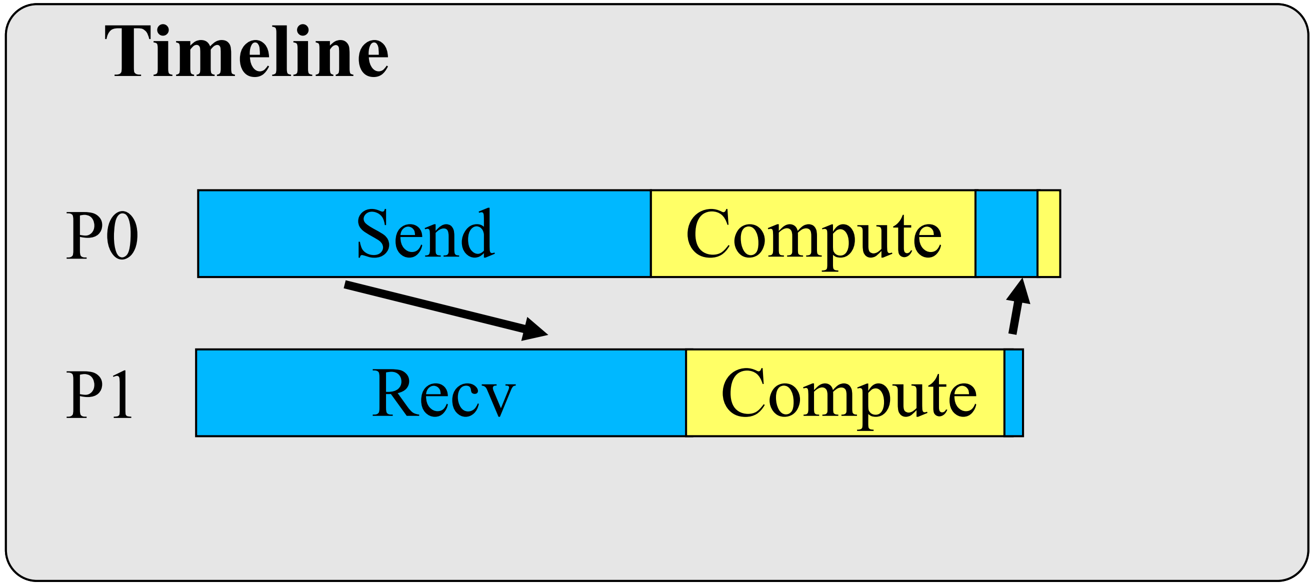


P1 posts a send with partial sum

Parallel Sum



Step 6: Compute final answer



P0 sums the partial sums

Exercise: Parallel sum

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int i,N;
    double *array;
    double sum;
    N=100;
    array=malloc(sizeof(double)*N);
    for(i=0;i<N;i++){
        array[i]=1.0;
    }
    sum=0;
    for(i=0;i<N;i++){
        sum+=array[i];
    }
    printf("Sum is %g\n",sum);
}
```

Exercise: Parallel sum

1. Parallelize the sum.c program with MPI
   - The relevant MPI commands can be found back in the README
   - run this program with two MPI-tasks

2. Use MPI_status to get information about the message received
   - print the count of elements received

3. Using MPI_Probe to find out the message size to be received
   - Allocate an arrays large enough to receive the data
   - call MPI_Recv()

*_sol.c contains the solution of the exercise.

Probing the Network for Messages

- **MPI_PROBE** and **MPI_IPROBE** allow the user to check for incoming messages without actually receiving them

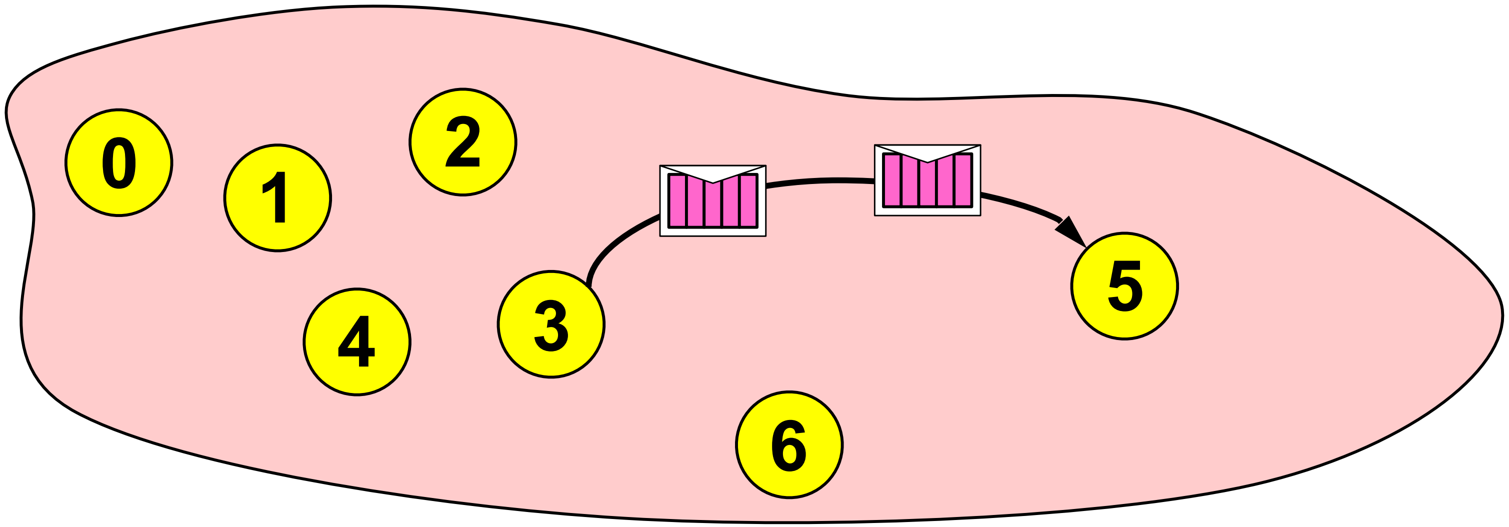
```
MPI_PROBE ( source, tag, communicator, status )
```

- **MPI_IPROBE** returns **“flag == TRUE”** if there is a matching message available. **MPI_PROBE** will not return until there is a matching receive available

```
MPI_IPROBE (source, tag, communicator, flag, status)
```

Message Order Preservation

- Rule for messages on the same connection,
  i.e., same communicator, source, and destination rank:
- Messages do not overtake each other.
- This is true even for non-synchronous sends.



- If both receives match both messages, then the order is preserved.

Exercise: Basic Ping-Pong

Ping-pong is a standard test in which two processes repeatedly pass a message back and forth.

Write a program that sends a 'float' array of fixed length, say, ten times back (ping) and forth (pong) to obtain an average time for one ping-pong.

Time the ping-pongs with MPI_WTIME() calls.

You may use pingpong.c or pingpong.f90 as a starting point for this exercise.

Investigate how the bandwidth varies with the size of the message.

Basic Ping Pong

rank=0 | rank=1

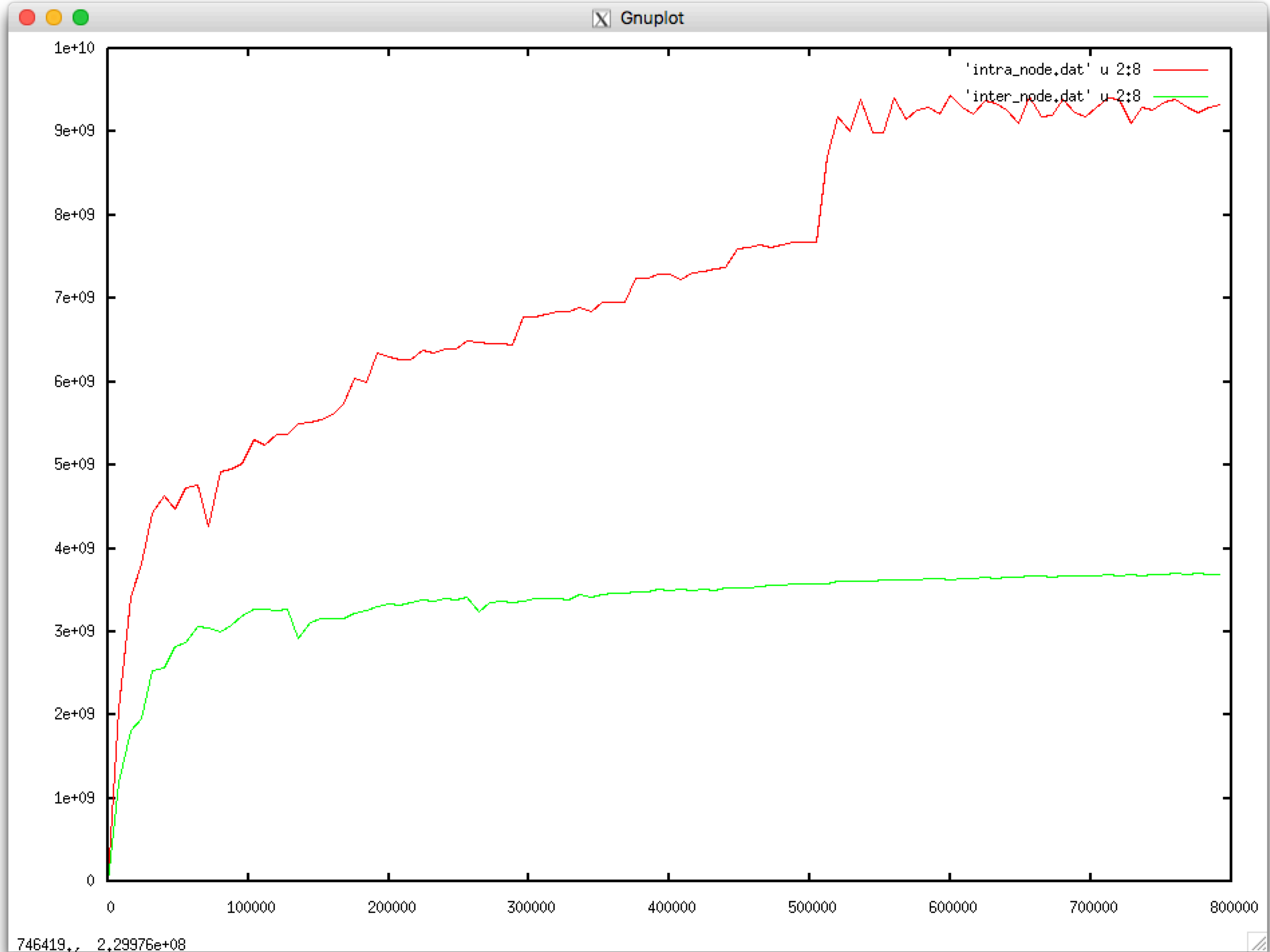
Send (dest=1)
(tag=17)

Recv (source=0)
Send (dest=0)

(tag=23)
Recv (source=1)

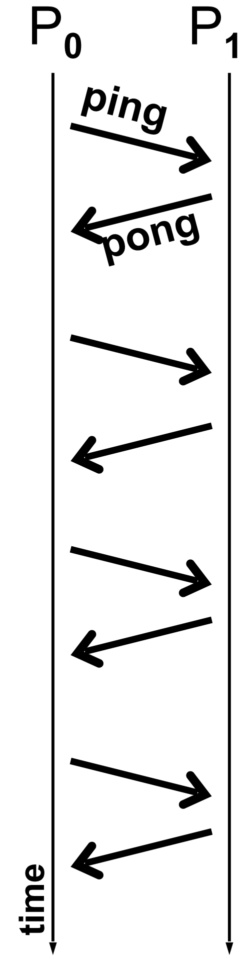
```
if (my_rank==0)              /* i.e., emulated multiple program */
     MPI_Send( ... dest=1 ...)
     MPI_Recv( ... source=1 ...)
else
     MPI_Recv( ... source=0 ...)
     MPI_Send( ... dest=0 ...)
fi
```

Basic Ping Pong output



Advanced-1 Ping Pong

- A program is written according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message,
    process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Use the timing calls before and after the loop:
- At process 0, print out the transfer time of one message
  - in seconds
  - in µs.
- Use program ping_pong_advanced1.
  no need to program yourself.



Advanced-2 Ping-Pong
Measure latency and bandwidth

- latency = transfer time for zero length messages
- bandwidth = message size (in bytes) / transfer time

- Print out message <u>transfer time</u> and <u>bandwidth</u>

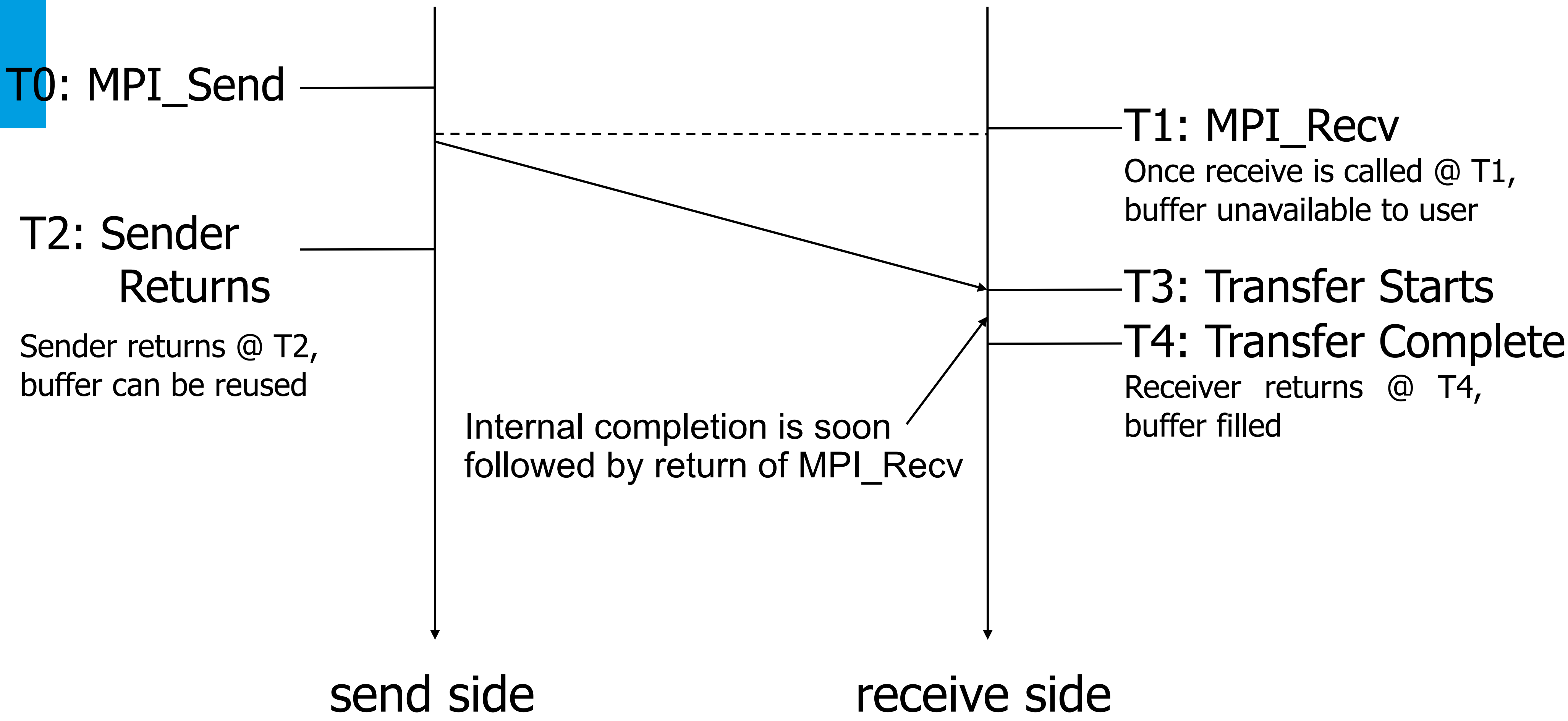
  - for following send modes:
    - standard send (MPI_Send: ping_pong_advanced2_send.c)
    - synchronous send (MPI_Ssend: ping_pong_advanced2_ssend.c)

  - for following message sizes:
    - 8 bytes (e.g., one double or double precision value)
    - 512 B (= 8*64 bytes)
    - 32 kB (= 8*64**2 bytes)
    - 2 MB (= 8*64**3 bytes)

Standard mode

- Corresponds to the common send functions
  - Blocking: MPI_Send
  - Non-blocking: MPI_Isend
- It's up to MPI implementation whether communication is buffered or not
- Buffered
  - Can be buffered either locally or remotely
  - The send (blocking) or the completion of a send (non-blocking) may complete before a matching receive
- Non-buffered
  - The send (blocking) or the completion of a send (non-blocking) only complete once it has sent the message to a matching receive
- Standard mode is non-local
  - Successful completion of the send operation may depend on the occurrence of a matching receive.

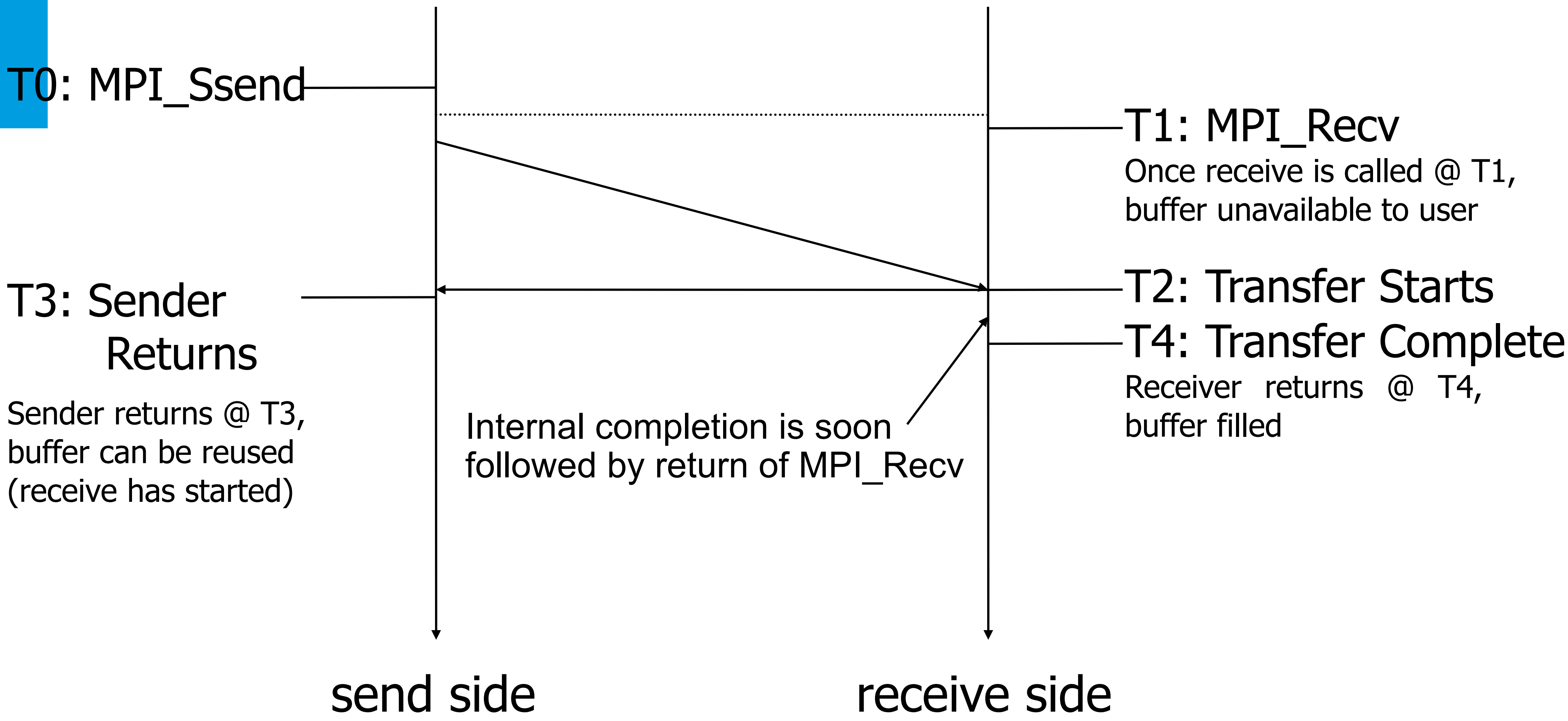
Standard Send-Receive Diagram



Synchronous mode

- Blocking: MPI_Ssend
  - Blocking send only returns once the corresponding receive has been posted
  - Same parameters as for standard mode send MPI_Send
- Uses
  - Debugging - potential deadlocks in the program are found by using synchronous sends
  - If many processes send messages to one process its unexpected message buffer can run out if it doesn't pre-post receives. By using MPI_Ssend this can be avoided! Typical example is IO where single process writes data
- Non-blocking: MPI_Issend
  - The completion (wait/test) of the send only returns once the corresponding receive has been posted
  - Same parameters as for standard mode send MPI_Isend
  - Useful for debugging - can be used to measure worst case scenario for how long the completion command has to wait

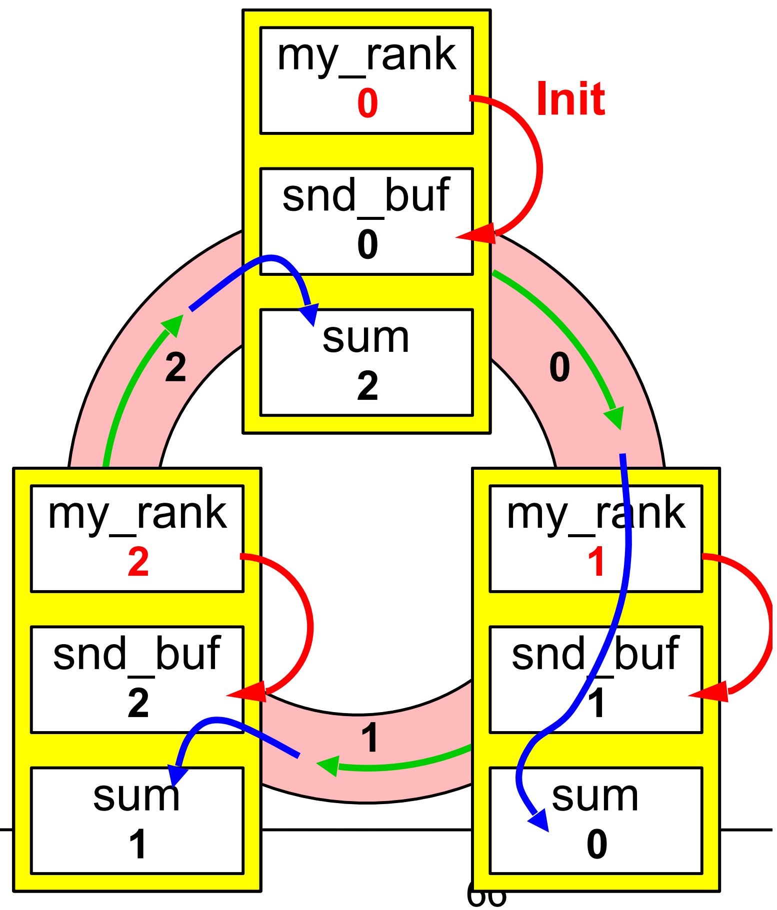
Synchronous Send-Receive Diagram



Exercise Ring

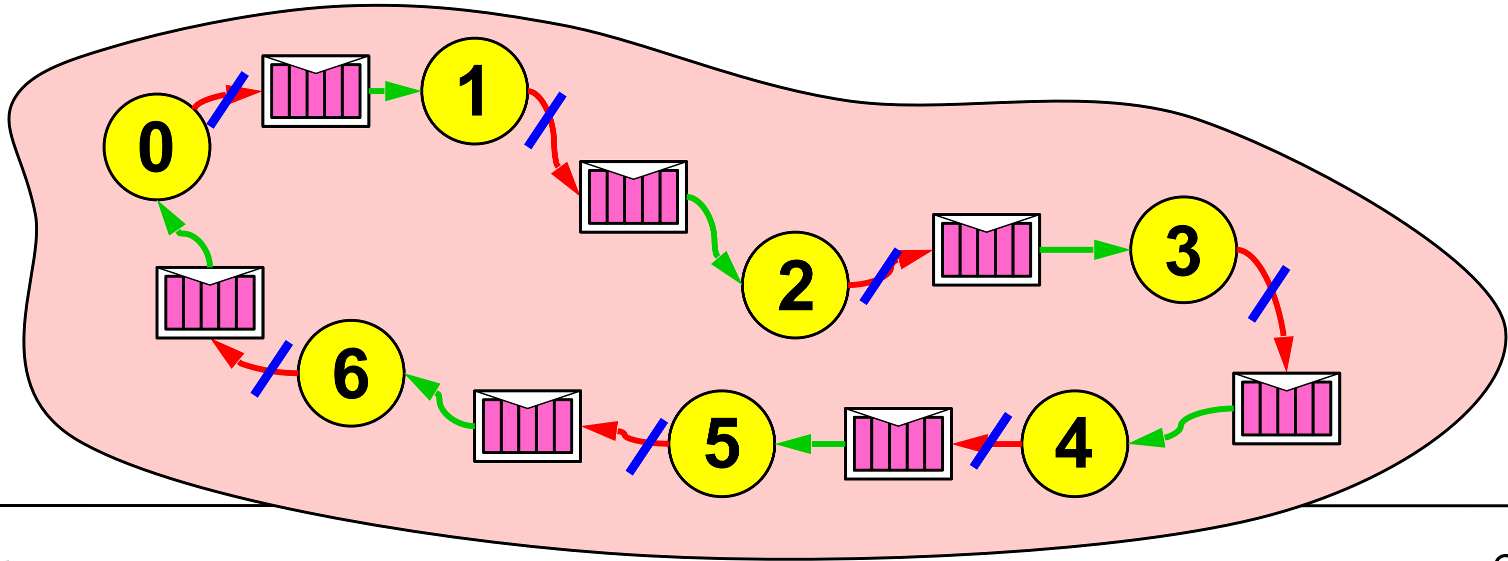
Exercise: Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable snd_buf.
- Each process passes this on to its neighbour on the right.
- Each processor calculates the sum of all values.
- Keep passing it around the ring until the value is back where it started, i.e.
- each process calculates sum of all ranks.
- Use non-blocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock



Non-Blocking Send

- Initiate non-blocking send
  in the ring example: Initiate non-blocking send to the right neighbor
- Do some work:
  in the ring example: Receiving the message from left neighbour
- Now, the message transfer can be completed
- Wait for non-blocking send to complete



Non-Blocking Receive

- Initiate non-blocking receive
  in the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
  in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete