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

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" );
    err = MPI_Finalize();
    return 0;
}
```

All C functions return an error message

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 calls return an error message as argument

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.
- Use the job.slurm script to submit to the workload manager slurm.
- Try to run directly without submitting to slurm (but don't tell me).

Better Hello (C)

```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;
}
```

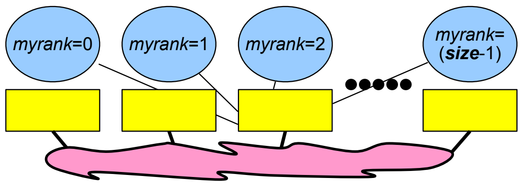

Better Hello (Fortran)

```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

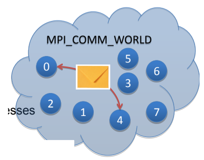
- The rank identifies different processes within a communicator
- The rank is the basis for any work and data distribution.



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

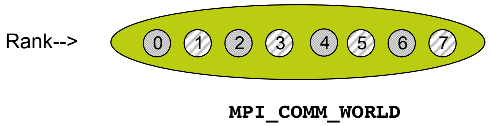
Some Basic Concepts

- 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)



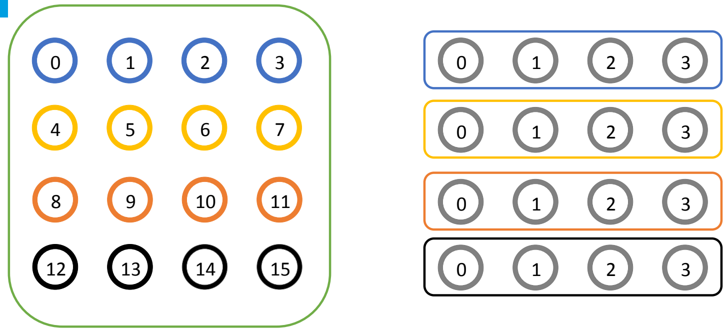
Communicator

- 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



Split communicator

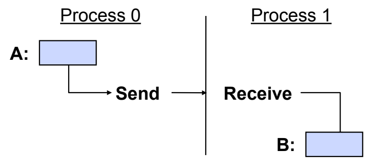
Split a Large Communicator Into Smaller Communicators



Point to Point communication

MPI Basic Send/Receive

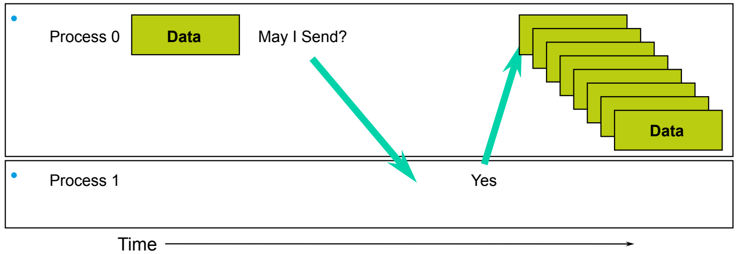
- Basic message passing process. Send data from one process to another



- 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

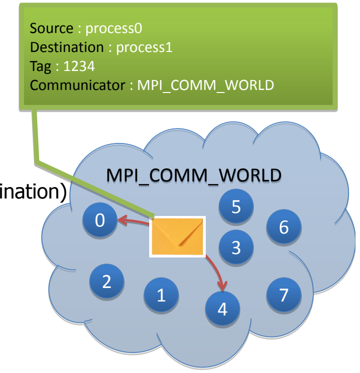
- Data transfer plus synchronization



- 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/destination)
  - 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

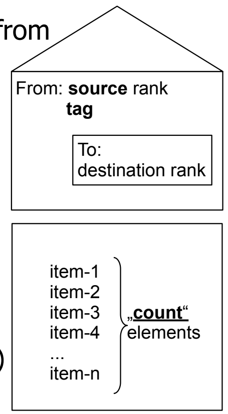
MPI FORTRAN	FORTRAN datatypes
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL*8
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

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: SendRecv (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 the 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

Instructions

```
if (myid > 0){
   MPI_Recv(MPI_Get_count(&status,MPI_INT,&count);?I_ANY_TAG,
               MPI_COMM_WORLD,&status);1,myid+1,MPI_COMM_WORLD);
```

MPI-task	0	1	2	3
-1-	MPI_send	MPI_send	MPI_send	
-2-		MPI_recv	MPI_recv	MPI_recv
-3-		MPI_Get_c	MPI_Get_c	MPI_Get_c

Is MPI Large or Small?

- Is MPI large (300+ functions) or small (6 functions)?
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
  - Many programs can be written with just 6 basic functions

MPI_INIT	**MPI_COMM_SIZE**	**MPI_SEND**
MPI_FINALIZE	**MPI_COMM_RANK**	**MPI_RECV**

- MPI is just right
  - A small number of concepts
  - Large number of functions provides flexibility, robustness, efficiency, modularity, and convenience
  - One need not master all parts of MPI to use it

You just learned MPI

- In principle we could stop here, all you need to know about MPI is covered.
- The other topics covered during the course are tweaks and extensions to this basic Send/Recv mechanism:
  - make it more convenient to use
  - write optimised functions for common problems (reduction)
  - avoid common mistakes
  - extension for work distribution
  - parallel IO
  - ….

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: SendRecv/deadlock (2)

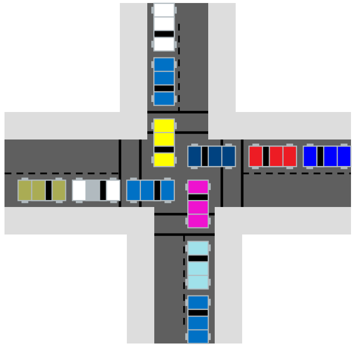
- Find out what the program deadlock (.c or .f90) is supposed to do. Run it with two processors and see what happens.

  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.

Deadlocks

Processes wait for some event or condition that will never occur

Example: Traffic jam Cars unable to turn or back up

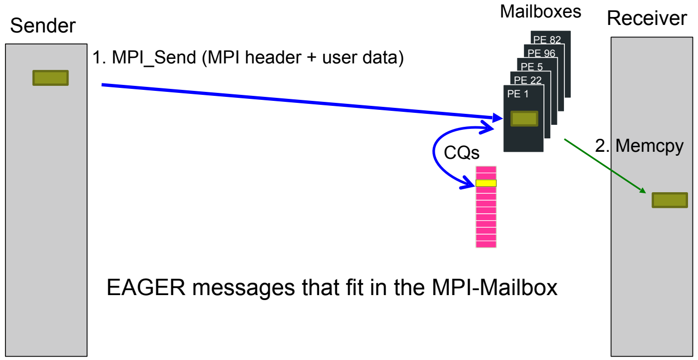


MPI Messaging Protocols

- MPI uses multiple different protocols
  - choice depends on message size

1. Eager messaging
   - Buffers used on send and receive side, opportunity to send direct to remote buffer
   - Used for small messages
   - Offers good potential for overlap
2. Segmentation And Reassembly (SAR)
   - Similar to Eager, with multiple messages/buffers each side (new for the Slingshot implementation)
3. Rendezvous messaging
   - After an initial setup communication bulk data transfer happens in the MPI_Recv
   - Used for large messages

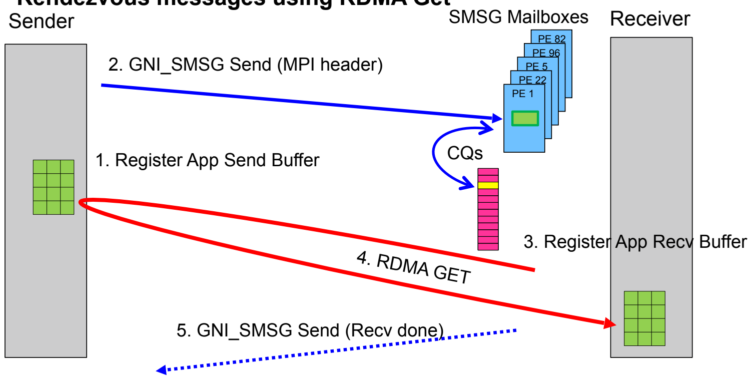
MPI Inter-Node Message type Eager



Mailbox size can changes with the number of ranks in the job

MPI Inter-Node Message type R0

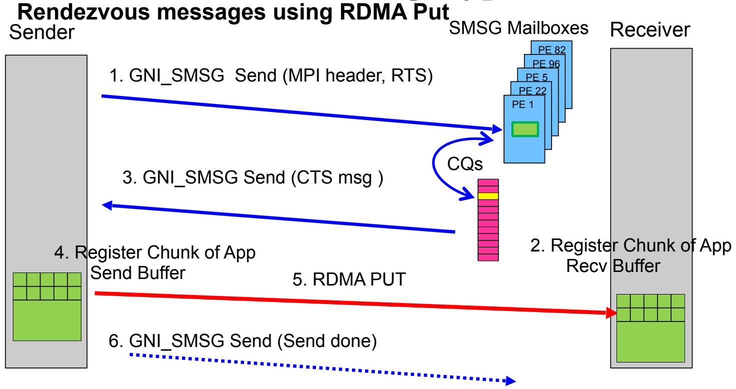
Rendezvous messages using RDMA Get



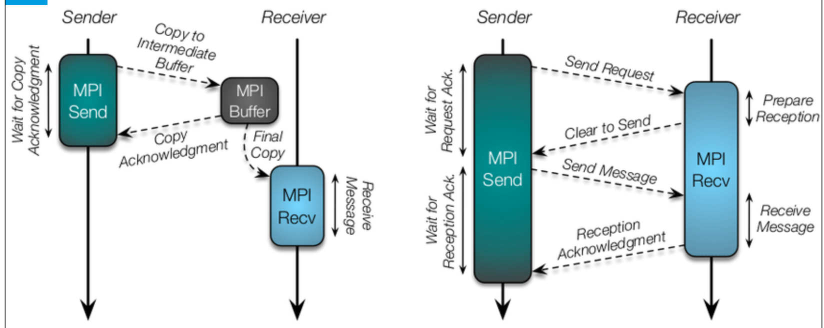
- No extra data copies
- Direct transfer from send-side user buffer to receive-side user buffer

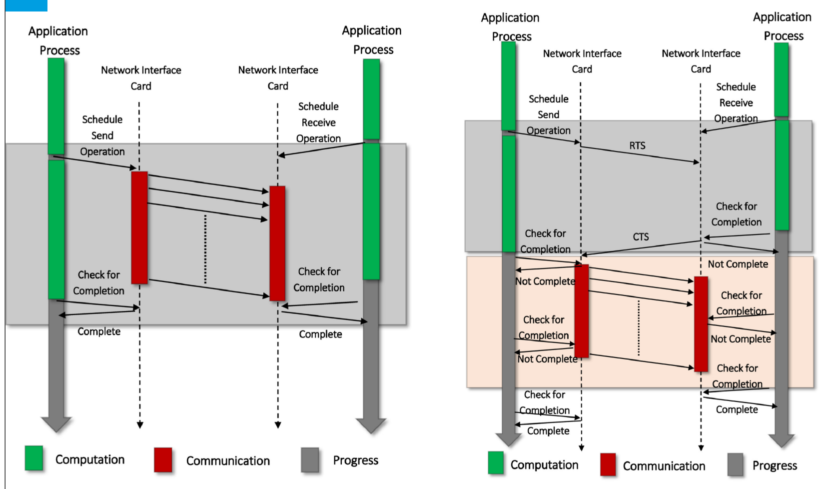
MPI Inter-Node Message type R1

Rendezvous messages using RDMA Put



- *Repeat steps 2-6 until all sender data is transferred*
- Chunksize is MPI_GNI_MAX_NDREG_SIZE (default of 4MB)





(a) Overlap Potential of Eager Protocol

(b) Overlap Potential of Rendezvous Protocol

Making Messages more eager

Eager	SAR	Rendezvous

- One way to improve performance
  - send more messages on the eager protocol; potentially more overlap
- Do this by raising the value of the eager/SAR threshold
  - set environment variable in jobscript
  - export btl_sm_eager_limit=<value>
  - value is in bytes: default is 4096 bytes. (~4 kB)
- When might this help
  - If MPI takes a significant time in a profile
  - If you have a lot of messages between 16kB and, say, 256 kB
    - CrayPAT MPI tracing can tell you this
- Also try to post `MPI_IRecv` call before the `MPI_ISend` call
  - can avoid unnecessary buffer copies

OpenMPI info and tuning

- Open MPI has many, many run-time tunable parameters (called "MCA parameters"), and usually only a handfull of them are useful to a given user.
- Application tuner
  - Generally, these are parameters that can be used to tweak MPI application performance. This even includes parameters that control resource exhaustion levels (e.g., number of free list entries, size of buffers, etc.), and could be considered "correctness" parameters if they're set too low. But, really -- they're tuning parameters.
- man-page of ompi_info

  ompi_info --param btl tcp

Tuning MPI message protocols

- btl_sm_eager_limit: If message data plus header information fits within this limit, the message is sent "eagerly" — that is, a sender attempts to write its entire message to shared buffers without waiting for a receiver to be ready. Above this size, a sender will only write the first part of a message, then wait for the receiver to acknowledge its readiness before continuing. Eager sends can improve performance by decoupling senders from receivers.
- btl_sm_max_send_size: Large messages are sent in fragments of this size. Larger segments can lead to greater efficiencies, though they could perhaps also inhibit pipelining between sender and receiver.
- btl_sm_free_list_num: This is the initial number of fragments on each (eager and max) free list. The free lists can grow in response to resource congestion, but you can increase this parameter to pre-reserve space for more fragments.

ompi_info --param btl all

The form of the environment variables that Open MPI sets is:

OMPI_MCA_<key>=<value>

export OMPI_MCA_btl_sm_eager_limit=8192

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

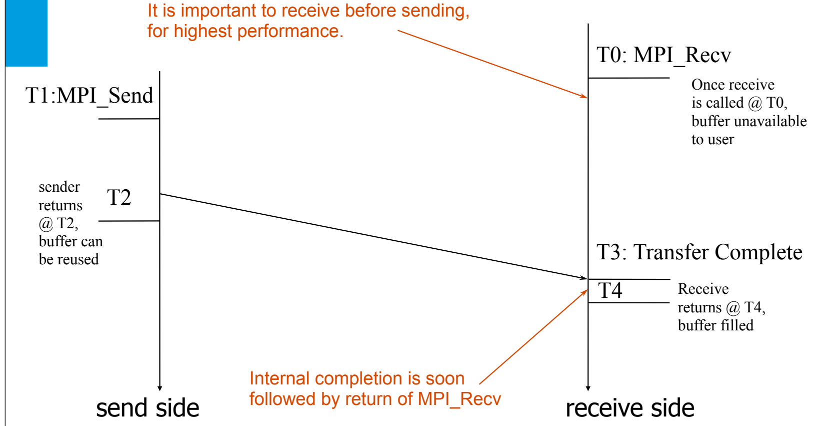
- Order the operations more carefully:

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

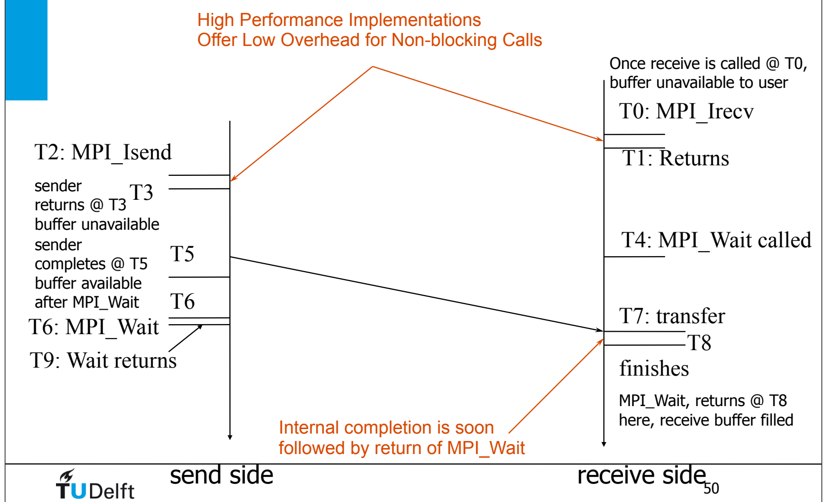
- Use non-blocking operations:

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 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

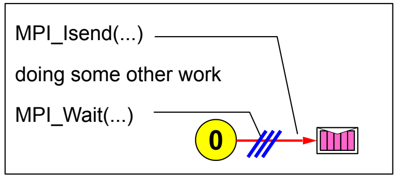
- Non-blocking (asynchronous) operations return (immediately) "request handles" that can be waited on and queried

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

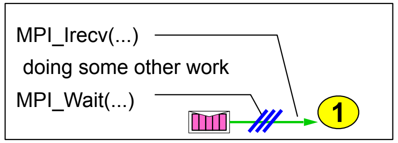
- Non-blocking operations allow overlapping computation and communication.
- Anywhere you use **MPI_Send** or **MPI_Recv**, you can use the pair of **MPI_Isend/MPI_Wait** or **MPI_Irecv/MPI_Wait**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead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

Non-Blocking Examples

- Non-blocking **send**



- Non-blocking **receive**



/// = waiting until operation locally completed

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

Fun with FLUPS (Fourier-based Library of Unbounded Poisson Solvers)

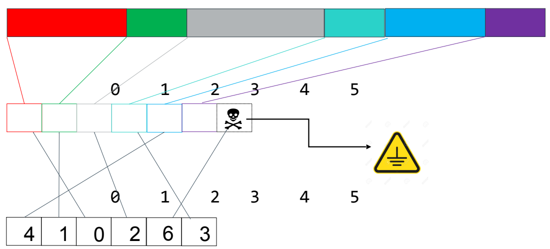
```
void exchange_data() {
  for(int ib = 0 ; ib<send_nBlock; ib++) {
    double *data;
    data = recvbuf[destTag[ib]];
    for (int i0 = 0 ; i0<nmax; i0++) data[i0] = v[i0];
    MPI_Isend(data, …); // Actually, from within MPI_Start()
  }
}
```

- Failing randomly with:
  - SIGSEGV in data[i0] = v[i0]
  - PMPI_Start: Invalid MPI_Request
  - xmpem_attach error: No such file or directory

buffer[big]

recvbuf[6]

destTag[6]



```
void gather_tags()
{
  int inb=onb=6;
  for (int ib=0;ib<inb;ib++) {
    MPI_Irecv(&destTag[ib],1,MPI_INT,isrc[ib],0,comm,&irequest[ib]);
  }

  for (int ib=0;ib<onb;ib++) {
    MPI_Isend(&ib,1,MPI_INT,idst[ib],0,comm,&orequest[ib]);
  }

  MPI_Waitall(irequest,inb,MPI_STATUSES_IGNORE);
  MPI_Waitall(orequest,onb,MPI_STATUSES_IGNORE);
}
```



```
void gather_tags()
{
  int inb=onb=6;
  for (int ib=0;ib<inb;ib++) {
    MPI_Irecv(&destTag[ib],1,MPI_INT,isrc[ib],0,comm,&irequest[ib]);
  }

  for (int ib=0;ib<onb;ib++) {
    MPI_Isend(&ib,1,MPI_INT,idst[ib],0,comm,&orequest[ib]);
  }

  MPI_Waitall(irequest,inb,MPI_STATUSES_IGNORE);
  MPI_Waitall(orequest,onb,MPI_STATUSES_IGNORE);
}
```



```
void gather_tags()
{
  int inb=onb=6;
  for (int ib=0;ib<inb;ib++) {
    MPI_Irecv(&destTag[ib],1,MPI_INT,isrc[ib],0,comm,&irequest[ib]);
  }

  for (int ib=0;ib<onb;ib++) {
    MPI_Isend(&ib,1,MPI_INT,idst[ib],0,comm,&orequest[ib]);
    // Sometimes, a 6 (onb) was being sent!
  }

  MPI_Waitall(irequest,inb,MPI_STATUSES_IGNORE);
  MPI_Waitall(orequest,onb,MPI_STATUSES_IGNORE);
}
```


- **MPI 3.1, section 3.7.2**
  - **The sender should not modify any part of the send buffer after a on-blocking send operation is called, until the send completes.**
- Maybe worth checking with
  - grep –r MPI_Isend src
  - and make sure that buffers that should not change do not change

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

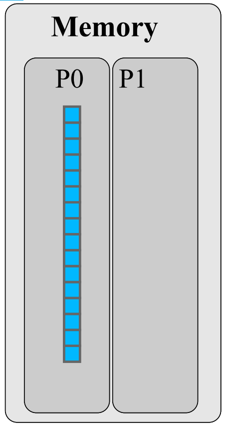
Send Modes

- 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

Work distribution example

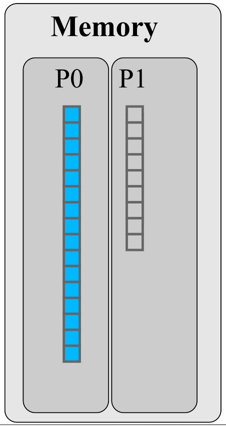
- Parallel sum

Parallel Sum

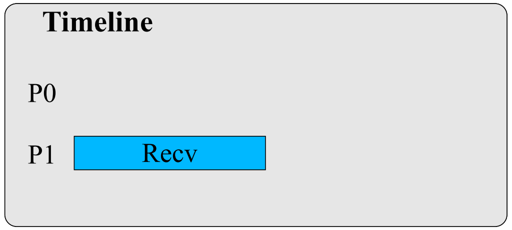


- Array is originally on process 0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segment
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

Parallel Sum

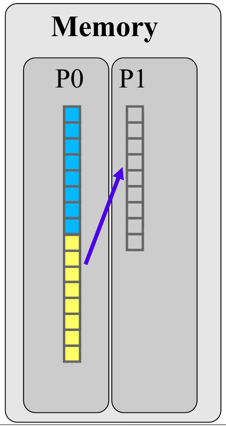


Step 1: Receive operation in scatter

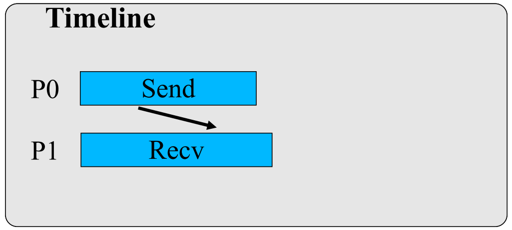


P1 posts a receive to receive half of the array from process 0

Parallel Sum

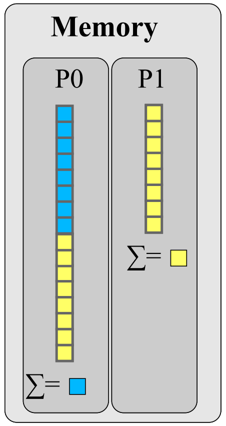


Step 2: Send operation in scatter

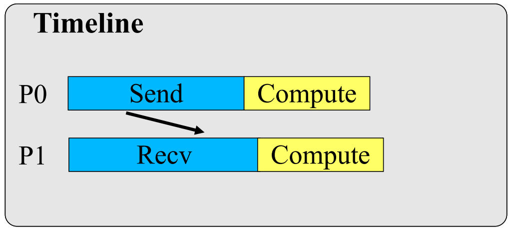


P0 posts a send to send the lower part of the array to P1

Parallel Sum

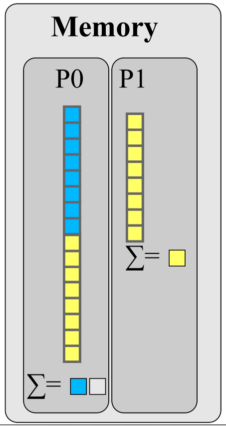


Step 3: Compute the sum in parallel

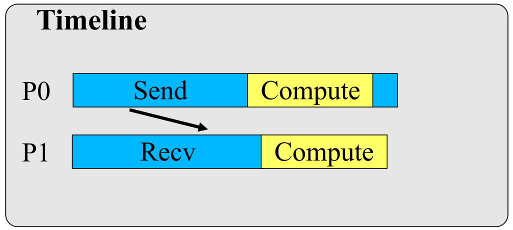


P0 & P1 computes their partial sums and store them locally

Parallel Sum

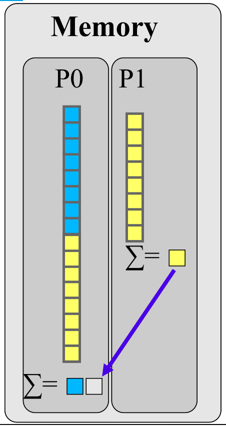


Step 4: Receive operation in reduction

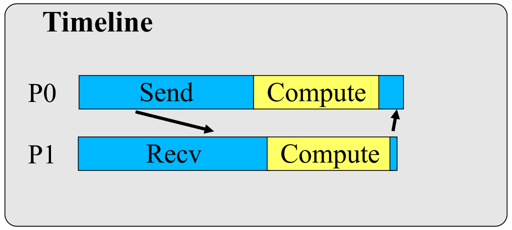


P0 posts a receive to receive partial sum

Parallel Sum

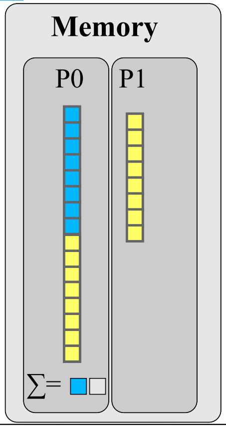


Step 5: Send operation in reduction

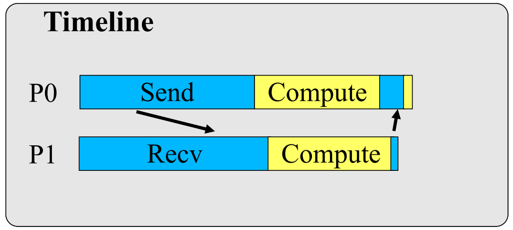


P1 posts a send with partial sum

Parallel Sum



Step 6: Compute final answer



P0 sums the partial sums

Exercise: ParallelSum

```
#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: ParallelSum

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** (details on next slide) 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

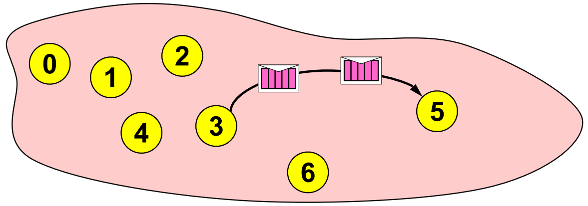
```
MPI_PROBE ( source, tag, communicator, status )
```

- **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: PingPong/Basic

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

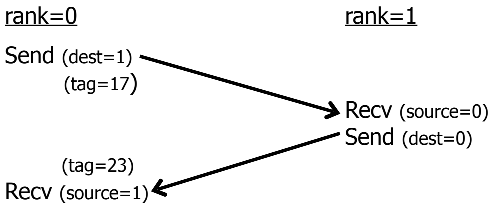
Write a program that sends a 'float' array of fixed length, say, ten times back (ping) and forth (pong) to obtain an average time for one ping-pong.

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




```
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
```

Timing MPI Programs

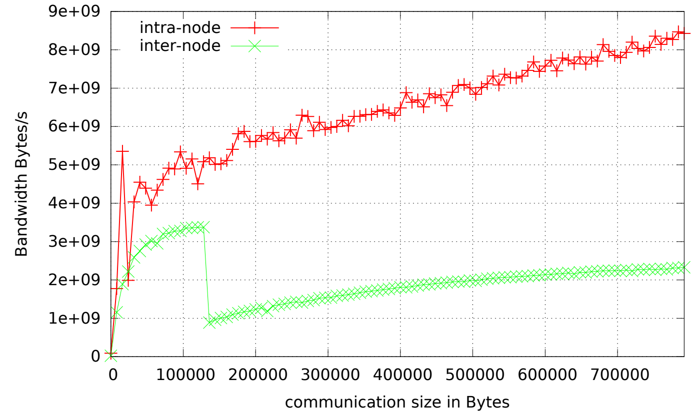
- **MPI_WTIME** returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past

```
double MPI_Wtime( void )
DOUBLE PRECISION MPI_WTIME( )
```

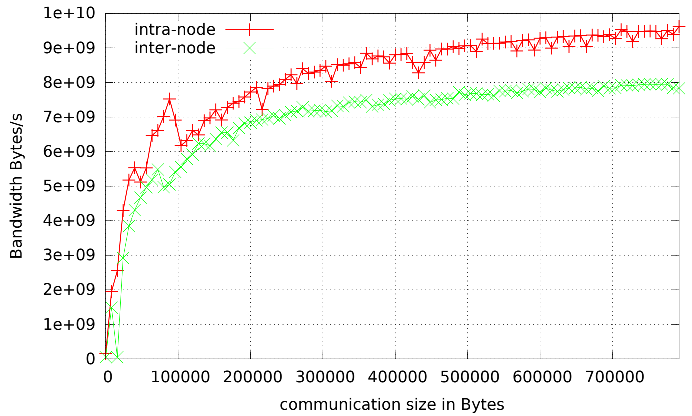
- **MPI_WTICK** returns the resolution of **MPI_WTIME** in seconds. It returns, as a double precision value, the number of seconds between successive clock ticks.

```
double MPI_Wtick( void )
DOUBLE PRECISION MPI_WTICK( )
```

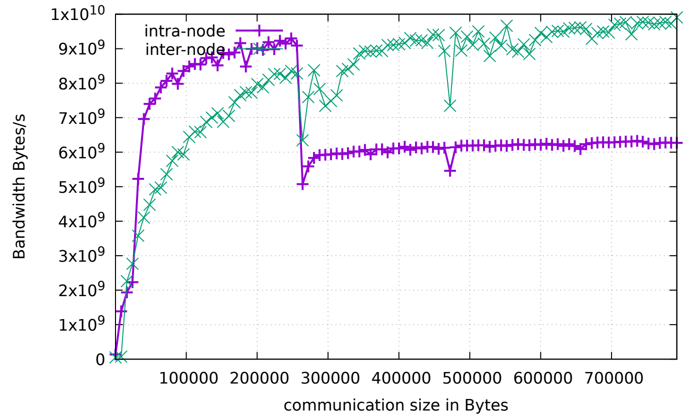

Basic Ping Pong BW ameland



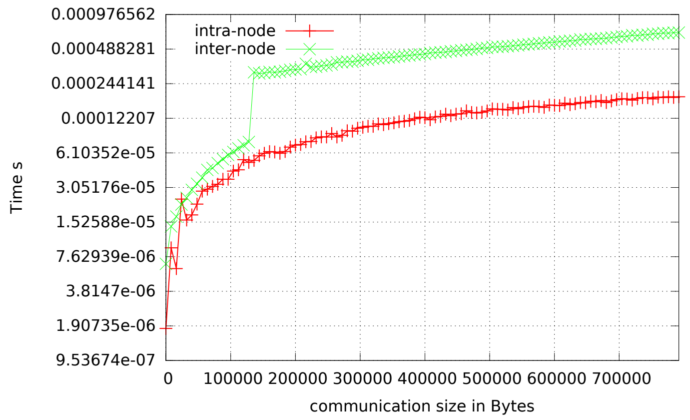
Basic Ping Pong BW XC40



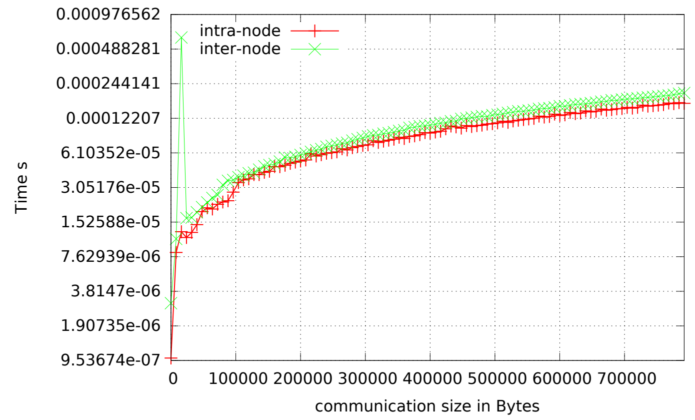
Basic Ping Pong BW DelftBlue



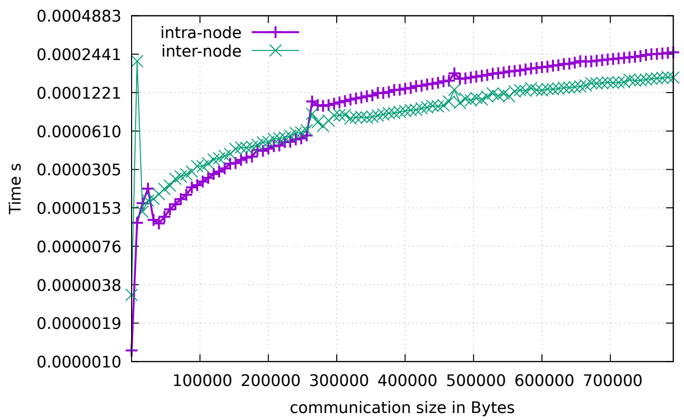
Ping-Pong time ameland



Ping-Pong time XC40



Ping-Pong time DelftBlue



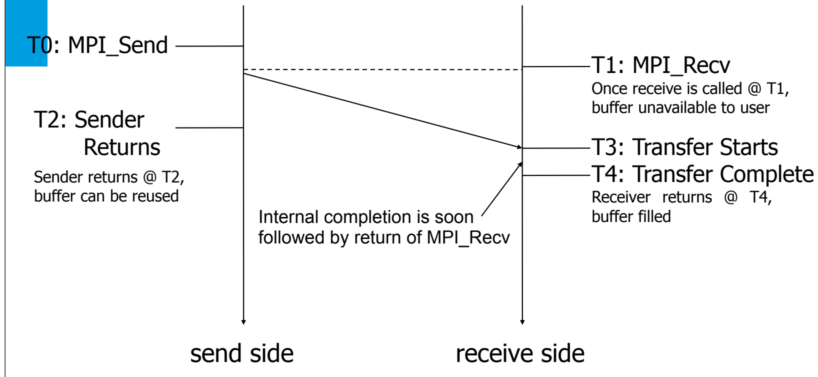
Exercise: PingPong/Advanced

- latency = transfer time for zero length messages
- bandwidth = message size (in bytes) / transfer time
- No need to program yourself.
- Print out message transfer time and bandwidth
  - for following send modes:
    - standard send (MPI_Send: **ping_pong_advanced2_send.c**)
    - synchronous send (MPI_Ssend: **ping_pong_advanced2_ssend.c**)
  - Compare the following message sizes for send and ssend:
    - 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.

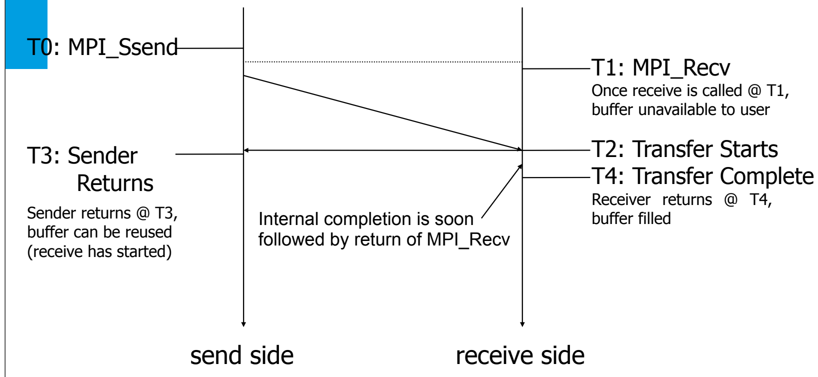
Standard Send-Receive Diagram



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



ping-pong advanced DelftBlue

[jthorbecke@login03 Advanced]$ more send.dat

message size	transfertime	bandwidth
8 bytes	0.182850 usec	43.751709 MB/s
512 bytes	1.666010 usec	307.321085 MB/s
32768 bytes	6.379230 usec	5136.670100 MB/s
2097152 bytes	201.958690 usec	10384.064187 MB/s

[jthorbecke@login03 Advanced]$ more ssend.dat

message size	transfertime	bandwidth
8 bytes	0.932230 usec	8.581573 MB/s
512 bytes	0.618070 usec	828.385134 MB/s
32768 bytes	4.292960 usec	7632.961872 MB/s
2097152 bytes	205.652840 usec	10197.534836 MB/s

mpi4py: MPI use in Python

- Python Interface to MPI
- https://mpi4py.readthedocs.io/en/stable/
- Examples (4) are in directory Python/
  - module load openmpi
  - module load py-mpi4py
  - README how to run the examples

Functionality

- Particular to mpi4py: no need to call MPI_Init() or MPI_Finalize
- run the Python script that uses mpi4py with the MPI launcher
  - srun -n …
  - mpirun -np …
  - mpiexec -np …
- The same as you would do with a regular MPI program,

Example 1

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

print "Hello! I'm rank %02d from %02d" % (comm.rank, comm.size)

print "Hello! I'm rank %02d from %02d" % (comm.Get_rank(),
      comm.Get_size())

print "Hello! I'm rank %02d from %02d" %
      (MPI.COMM_WORLD.Get_rank(), MPI.COMM_WORLD.Get_size())
```

- srun -n 5 python hello-mpi.py

Data Communication

- Python objects can be communicated with the send and receive methods of the communicator

send(data, dest, tag)

- data: Python object to send
- dest: destination rank
- tag: id given to the message

data = receive(source, tag)

- source: source rank
- tag: id given to the message
- data is provided as return value

- Destination and source ranks as well as tags have to match

Example 2

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
assert comm.size == 2

if comm.rank == 0:
    sendmsg = 123
    comm.send(sendmsg, dest=1, tag=11)
    recvmsg = comm.recv(source=1, tag=22)
    print "[%02d] Received message: %s" % (comm.rank, recvmsg)
else:
    recvmsg = comm.recv(source=0, tag=11)
    print "[%02d] Received message: %d" % (comm.rank, recvmsg)
    sendmsg = "Message from 1"
    comm.send(sendmsg, dest=0, tag=22)
```