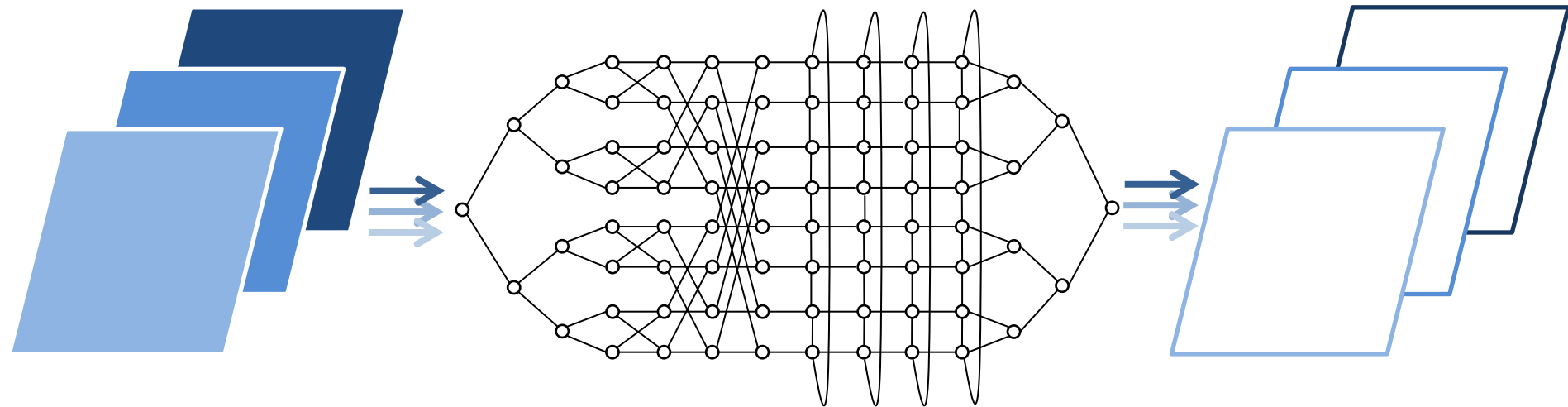


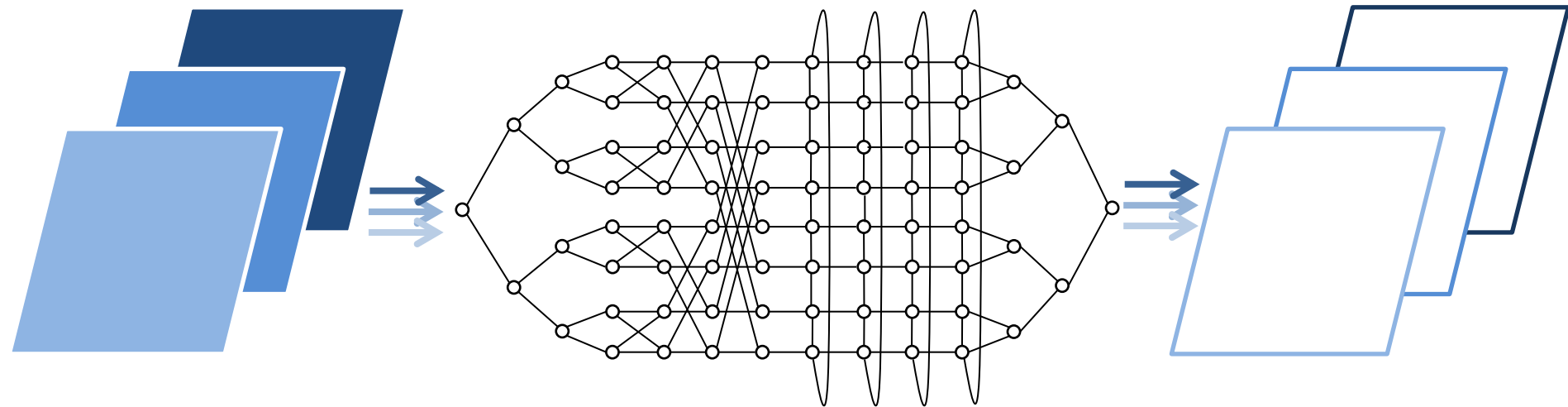
Programmation Parallèle

MPI: Message Passing Interface

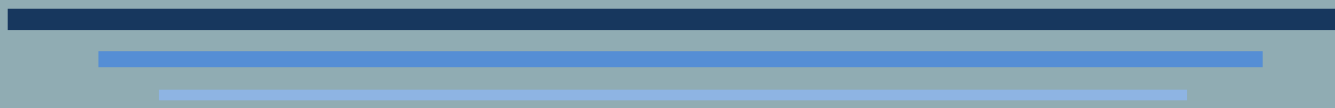


Introduction to MPI and p2p communications

ENSIIE-HPC/BigData-PP-IIP-Lecture 1



PARALLEL PROGRAMMING



Message Exchange

Message Exchange



- Message characteristics
 - Sender
 - Destination task
 - Data to exchange
- High-level protocol
 - Pair of actions will resolve message exchange
 - Sender must send the message
 - Let's consider a function called *send*
 - Recipient must receive the message
 - Let's consider a function called *recv*

Main Principle

- Two parallel tasks **T0** et **T1**
 - Distinct memory space
 - Each task has its own instructions to execute

T0 Task

```
instruction1;  
instruction2;
```

T1 Task

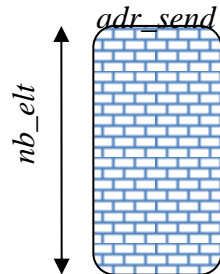
```
instruction1;  
instruction2;
```

Main Principle

- **T1** depends on **T0**
 - **T0** must send data to **T1**
 - Data are located in *adr_send* with *nb_elt* elements

T0 Task

```
instruction1;  
instruction2;  
send(adr_send, nb_elt, T1);
```



T1 Task

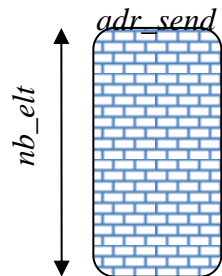
```
instruction1;  
instruction2;
```

Main Principle

- **T1** must receive data from **T0** (*recv*)
 - Size of message *nb_elt* should be known by recipient
 - Recipient may have to allocate a memory zone to get the received data (zone pointed by *adr_recv*)

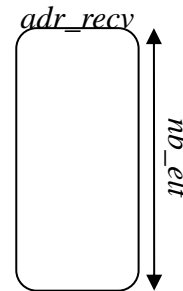
T0 Task

```
instruction1;  
instruction2;  
send(adr_send, nb_elt, T1);
```



T1 Task

```
instruction1;  
instruction2;  
recv(adr_recv, nb_elt, T0);
```

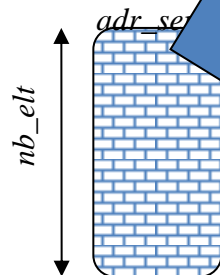


Main Principle

- Communication
 - *send* blocks **T0** until data are sent
 - *recv* blocks **T1** until data are received

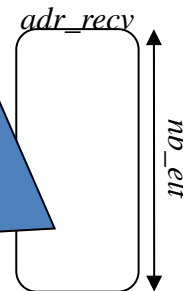
T0 Task

```
instruction1;  
instruction2;  
send(adr_send, nb_elt, T1);
```



T1 Task

```
instruction1;  
instruction2;  
recv(adr_recv, nb_elt, T0);
```



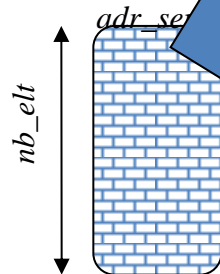
Data Transfer

Main Principle

- Communication
 - *send* blocks **T0** until data are sent
 - *recv* blocks **T1** until data are received

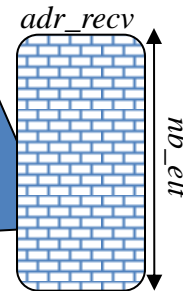
T0 Task

```
instruction1;  
instruction2;  
send(adr_send, nb_elt, T1);
```



T1 Task

```
instruction1;  
instruction2;  
recv(adr_recv, nb_elt, T0);
```



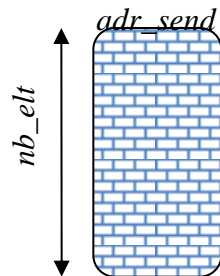
Data Transfer

Main Principle

- **T1** owns a complete copy of data sent by **T0**

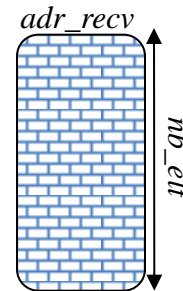
T0 Task

```
instruction1;  
instruction2;  
send(adr_send, nb_elt, T1);
```



T1 Task

```
instruction1;  
instruction2;  
recv(adr_recv, nb_elt, T0);
```

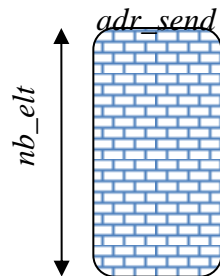


Main Principle

- Tasks **T0** and **T1** may continue their execution
- Following instructions of **T1** may access to data stored at address *adr_recv*

Tâche T0

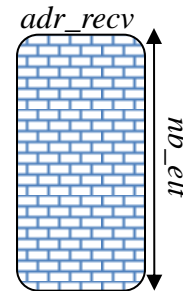
```
instruction1;  
instruction2;  
send(adr_send, nb_elt, T1);
```



```
instruction3;
```

Tâche T1

```
instruction1;  
instruction2;  
recv(adr_recv, nb_elt, T0);
```



```
instruction3;
```

Example



- Parallel sum on each element of an array
- Hypothesis
 - Array t with N floats (N is even)
 - Array t is distributed across 2 tasks T_0 and T_1
 - Parallelism type: data
- Goal
 - T_1 must print the sum of each element of t
- Code?

Example



T0 sends its partial sum to T1

T0

```
double p = 0.0;  
int i;
```

```
for( i=0 ; i<N/2 ; i++ )  
    p += tab[i];
```

```
send(&p, 1, T1);
```

T1 needs partial sum from T0

T1

```
double p = 0.0;  
double s;  
int i;
```

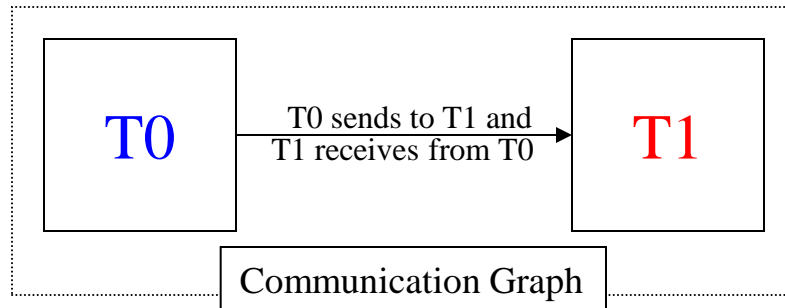
```
for( i=0 ; i<N/2 ; i++ )  
    p += tab[i];
```

```
recv(&s, 1, T0);
```

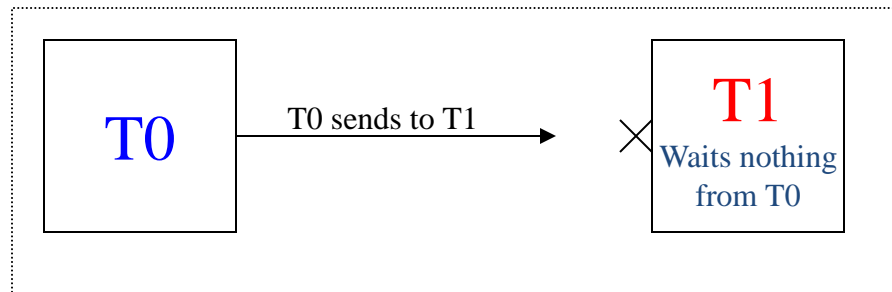
```
printf("%g",s+p);
```

Send/Recv Matching

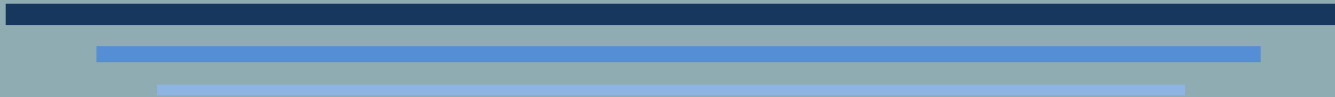
- Every *send* corresponds to one *recv* (and vice-versa)
- Model with an oriented graph
 - Vertices are tasks
 - Edges are communications



- A missing send or receive action lead to a deadlock situation



INTRODUCTION TO MPI



Introduction





- MPI: Message-Passing Interface
- High-level API (Application Programming Interface)
 - Parallel programming
 - Distributed-memory paradigm
- Implementation as a library
 - Interface through functions
- Language compatibility
 - C
 - C++
 - FORTRAN

Pourquoi utiliser MPI ?



- MPI is mostly an interface
- MPI is available in every type of parallel architectures
- MPI supports heavy parallelism
- Machine and/or network vendors often provides their own optimized version of MPI library
- MPI is also available in *open source* for most of current supercomputer architectures
 - MPICH2 : <http://www.mcs.anl.gov/research/projects/mpich2/>
 - OpenMPI : <http://www.open-mpi.org>

MPI Overview

- 
- 
- MPI includes (mainly MPI 1)
 - Execution environment
 - Point-to-point communication
 - Collective communications
 - Groups and topologies of MPI processes
 - MPI 2.0 adds
 - One-sided communications
 - Dynamic process creation
 - Multithreading
 - Parallel I/O
 - MPI 3.0 adds
 - Non-blocking collective communications
 - New one-sided communications
 - Non-blocking I/O collective
 - Neighborhood collectives

Hello World!

```
#include <stdio.h>
/* MPI function signatures */
#include <mpi.h>

int main(int argc, char **argv)
{
    /* Initialization of MPI */
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    /* Finalization of MPI */
    MPI_Finalize();
    return 0;
}
```

- Header file
 - Need to include it
 - Contains signatures of each available MPI function
 - Function bodies are located inside a library
- Syntax
 - All functions related to MPI start with MPI_
- Convention
 - No MPI calls before MPI_Init
 - No MPI calls after MPI_Finalize

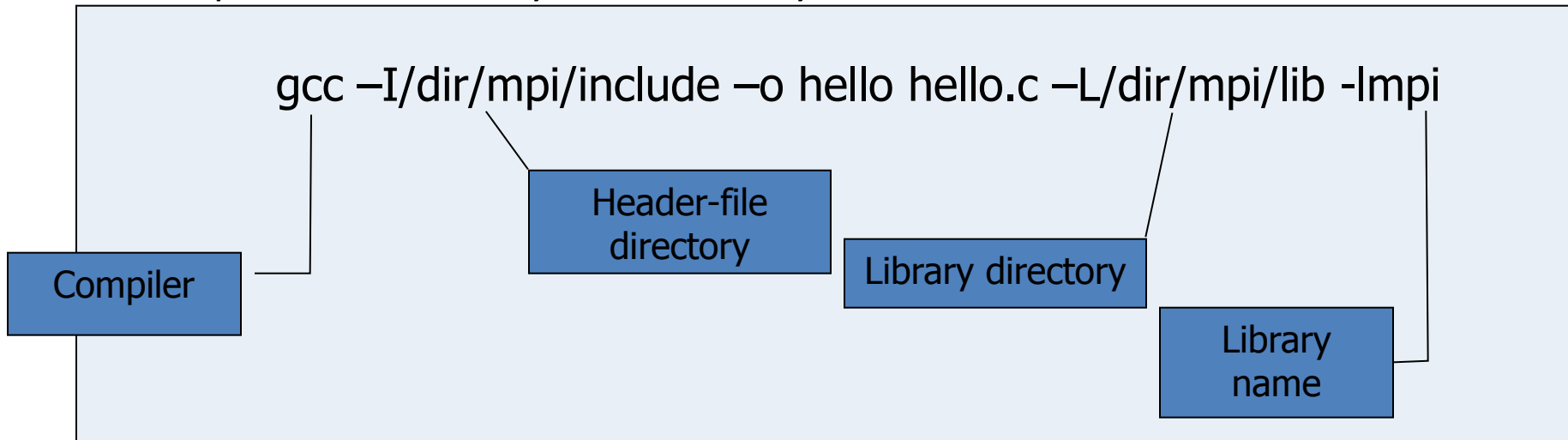
Compilation

- Basically
 - *Compilation process like any other library*
- But multiple ways to compile an MPI program
 - Simple way: rely on `mpicc` script
 - Complex way: launch regular compiler with options to specify paths to the library
- Simple way
 - Script/program that hide the library configuration details

```
mpicc -o hello hello.c
```
 - Call the default underlying compiler
 - Possible to change the compiler that will be invoked

Compilation

- Complex way
 - Without the script → pass right options for library configuration
- Generic mandatory options to use external library
 - Directory where header files are located (e.g., `mpi.h`)
 - Directory where library files are located (e.g., `libmpi.so`)
 - Name of the library to use (*linker*)
- Example: `libc` library or MPI library



Execution w/ Job Manager



- Slurm can spawn MPI processes
 - Rely on `srun` command
- If not available
 - Use of `mpirun` script (different syntax and usage)

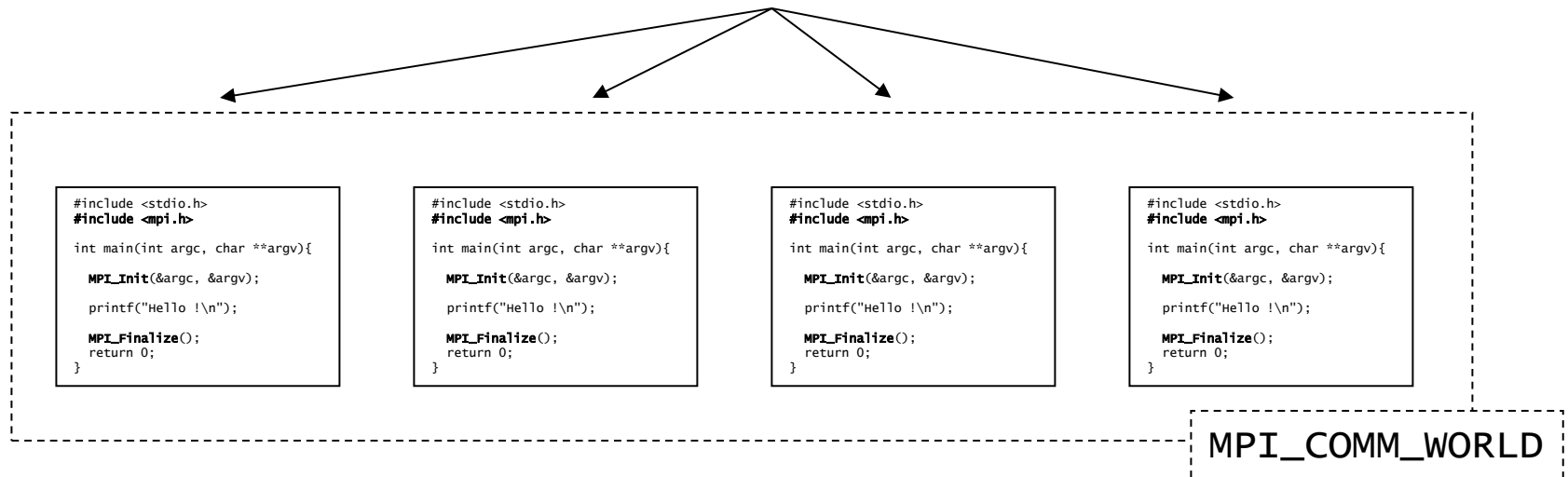
```
srun -n 4 ./hello
```

```
Hello World!  
Hello World!  
Hello World!  
Hello World!
```

- Remarks
 - Creation of 4 processes
 - Every process has the same instructions
 - Processes are independent for execution

Communicator

`srun -n 4 ./hello`



- Group of processes form a communicator
 - Predefined: `MPI_COMM_WORLD` w/ all processes
- Communicator = set of processes + communication context
 - Type: `MPI_Comm`

Total Number of Processes

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

int main(int argc, char **argv) {
    int N;
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &N);

    printf("Number of processes = %d\n", N);

    MPI_Finalize();
    return 0;
}
```

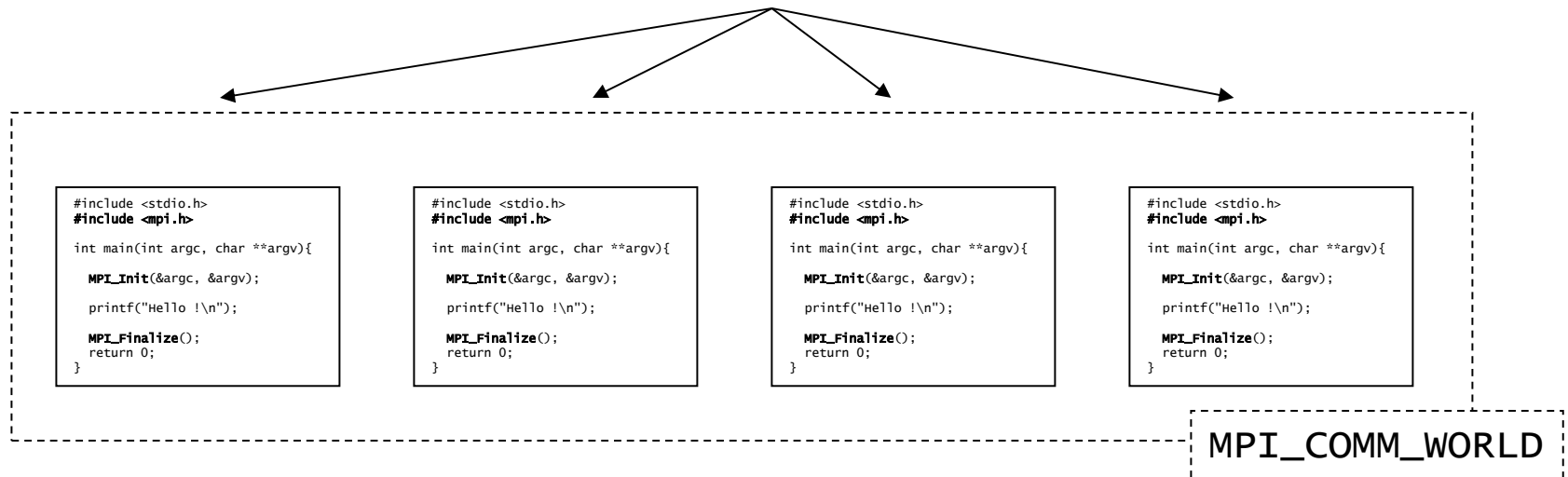
```
% srun -n 4 ./a.out
Number of processes = 4
Number of processes = 4
Number of processes = 4
Number of processes = 4
%
```

```
int MPI_Comm_size( MPI_Comm comm, int *size);
```

- Return size of communicator `comm` in `*size`
- If `comm == MPI_COMM_WORLD`, `MPI_Comm_size` returns the total number of MPI processes in the application

Process Rank

```
srun -n 4 ./hello
```



- Inside a communicator, MPI assigns rank from 0 to size-1
 - This is the rank of a process
- Function `MPI_Comm_rank` returns the rank in the communicator `comm` inside the address `*rank`:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Process Rank



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

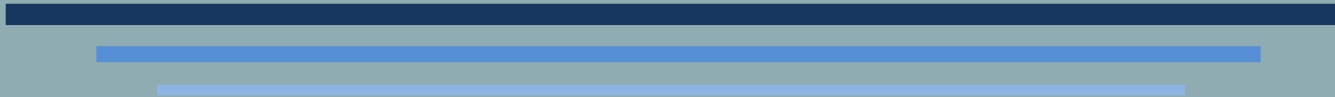
int main(int argc, char **argv) {
    int N, me;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &N);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    printf("My rank is %d out of %d\n", me, N);
    MPI_Finalize();
    return 0;
}
```

```
% srun -n 4 ./a.out
My rank is 1 out of 4
My rank is 0 out of 4
My rank is 3 out of 4
My rank is 2 out of 4
%
```

Process Rank

- Number of processes may differ from number of available cores/processors!
 - By default, Slurm binds one MPI process to one core
 - Option `-c` can be used to book multiple cores per rank
- Execution of processes is not related to their rank
 - Parallel execution
 - At the beginning, no ordering between processes
 - Only communications can imply some partial ordering
- Rank is usually used to determine
 - Which part of data should I work on?
 - What is my role (master/slave)?

MPI POINT-TO-POINT COMMUNICATIONS



Send/Recv

MPI Communication



- MPI is a parallel distributed-memory model
 - Each process accesses its own memory space
 - Based on message passing
- What is the main interface for data exchange w/ MPI?
- To send a message
 - `MPI_Send` function

Sending Messages

- Function to send a message

```
int MPI_Send (
```

```
    void *buf(in),
```

```
    int count(in),
```

```
    MPI_Datatype datatype(in),
```

```
    int dest(in),
```

```
    int tag(in),
```

```
    MPI_Comm comm(in)
```

```
);
```

} Main characteristics of
message to send

Sending Messages

- Function to send a message

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

Data
address

Data to send inside an
array pointed by **buf**
whose elements are of
type **datatype**.

MPI predefined scalar
types corresponding to
existing C types.

Sending Messages

<i>MPI_Datatype</i>	<i>C Type</i>
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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	<i>One byte</i>
MPI_PACKED	<i>Pack of non-contiguous data</i>

Sending Messages



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

Message size is **count**.

Not in bytes, but in number of elements of type **datatype** (portable way to express size).

Sending Messages



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

Communicator for message.

Communicator = (sub-)set
of processes +
communication context

MPI_COMM_WORLD
contains all processes
created during application
launch

Sending Messages



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

Recipient rank.

This rank is valid inside communicator `comm`.

For `MPI_COMM_WORLD`, `dest` should be between 0 and number of ranks (excl.).

Sending Messages



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

Label named **tag**
used to identify
messages.

Allows distinguish
messages from the
same sender and the
same recipient.

Remarks on Sending Messages





- `MPI_Send` is blocking function
 - Returning from `MPI_Send`, process can manipulate (e.g., write) the data buffer containing the message
- How to determine the message tag
 - Can use any way you want
 - Not necessary for different send/recipient pair
 - Example:
$$\text{tag} = \text{src} * N + \text{dest}$$
(N total number of MPI processes, `src` sender rank, `dest` recipient rank;)
- Be careful: the number of tags is limited!

MPI Communication



- What is the main interface for data exchange w/ MPI?
- Message reception
 - `MPI_Recv` function

Receiving Messages



```
int MPI_Recv (
```

```
void *buf(out),
```

```
int count(in),
```

```
MPI_Datatype datatype(in),
```

```
int source(in),
```

```
int tag(in),
```

```
MPI_Comm comm(in),
```

```
MPI_Status *status(out)
```

```
);
```

} Main characteristics of message
to receive

Receiving Messages

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

Address of memory zone to put the received data.

This zone should be allocated in some way BEFORE!

Max size of received message

Unit is in number of elements of type **datatype**.

The actual size of received message is less or equal to **count**.

Receiving Messages

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

Rank of sender.

Rank should be valid in
comm communicator.

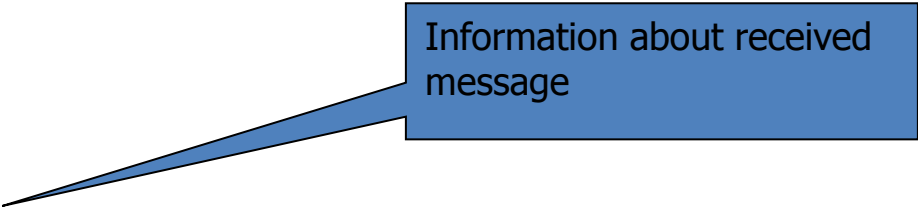
Can specify the predefined
value `MPI_ANY_SOURCE` →
May match a message from
any sender in the target
communicator

Message tag.

Should be the same of the
one put in corresponding
`MPI_Send` function call.

Receiving Messages

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



Information about received message

Information and Status

- `MPI_Status` is a C structure

```
struct MPI_Status {  
    int MPI_SOURCE; /* message sender (useful w/ MPI_ANY_SOURCE argument) */  
    int MPI_TAG; /* message tag (useful w/ MPI_ANY_TAG argument) */  
    int MPI_ERROR; /* error code */  
};
```

- If message size is unknown to the recipient, it is possible to extract the actual size with `MPI_Get_count`

```
int MPI_Get_count(  
    MPI_Status *status(in), /* status returned by MPI_Recv */  
    MPI_Datatype datatype(in), /* Type of elements in the message */  
    int *count(out) /* Size of the message (in number of elements of type datatype) */  
);
```

Simple MPI Example

```
int main(int argc, char **argv) {
    double p = 0., s0;
    int i, r;
    MPI_Status status;

    MPI_Init(&argc, &argv); /* Initialization of MPI library */
    MPI_Comm_rank(MPI_COMM_WORLD, &r); /* Get the rank of current rank */

    for( i = 0 ; i < N/2 ; i++ )
        p += tab[i];

    tag = 1000; /* Message tag */
    if (r == 0) {
        MPI_Send(&p, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&s0, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
        printf( "Sum = %d\n", s0+p );
    }
    MPI_Finalize();
    return 0;
}
```

Simple MPI Example



```
sum = 0.; /* Each process has N/P elements of distributed */
for( i = 0 ; i < N/P ; i++ ) /* array and perform a partial sum */
    sum += tab[i];

if (r == 0) {
    /* Process 0 receives P-1 messages in any order */
    for( t = 1 ; t < P ; t++ ) {

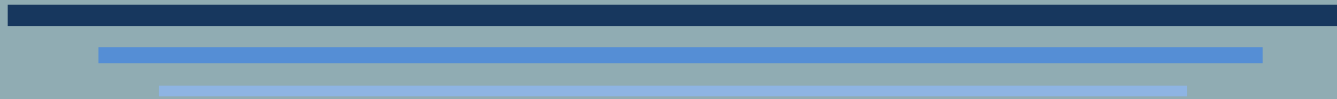
        MPI_Recv(&s, 1, MPI_DOUBLE,
                MPI_ANY_SOURCE, MPI_ANY_TAG, /* wildcards */
                MPI_COMM_WORLD, &sta);

        printf(« Message from rank %d\n", sta.MPI_SOURCE);

        sum += s; /* Contribution of process sta.MPI_SOURCE to the global sum */
    }
} else {

    /* Other processes send their partial sum to rank 0 */
    MPI_Send(&som, 1, MPI_DOUBLE, 0, rang, MPI_COMM_WORLD);
}
```

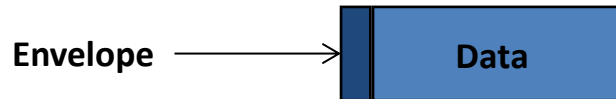
MESSAGE PASSING PROTOCOLS



Eager, Rendez-vous, Short

Message protocols

- Message consists of “envelope” and data
 - Envelope contains tag, communicator, length, source information, plus impl. private data



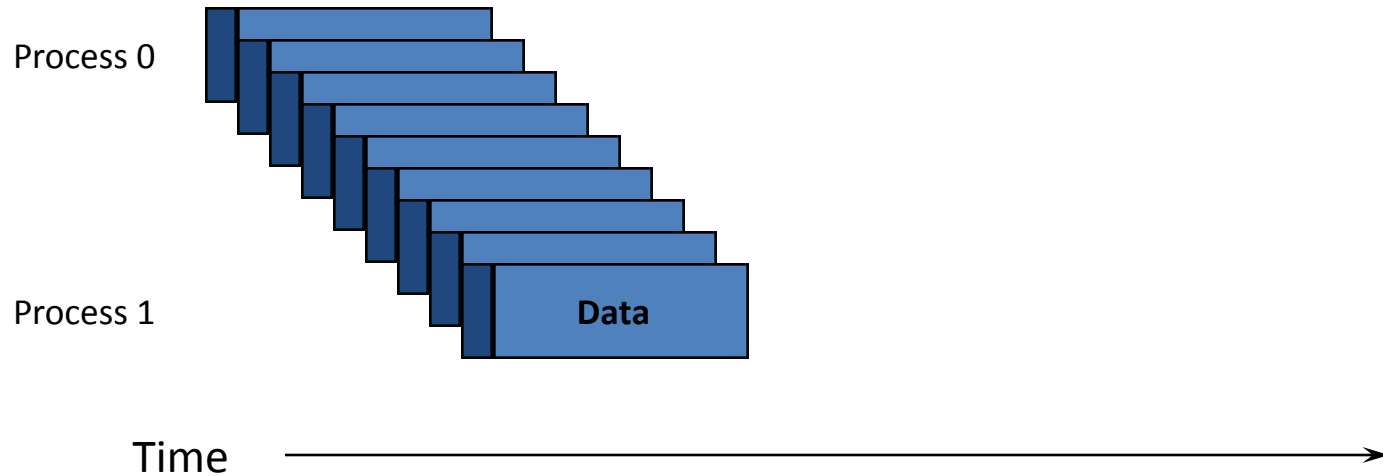
- Short
 - Message data (message for short) sent with envelope
- Eager
 - Message sent assuming destination can store
- Rendezvous
 - Message not sent until destination oks

Message Protocol Details




- User versus system buffer space
- Packetization
- Collective operations
- Datatypes, particularly non-contiguous
 - Handling of important special cases
 - Constant stride
 - Contiguous structures

Eager Protocol



- Data delivered to process 1
 - No matching receive may exist; process 1 must then buffer and copy.

Eager Features

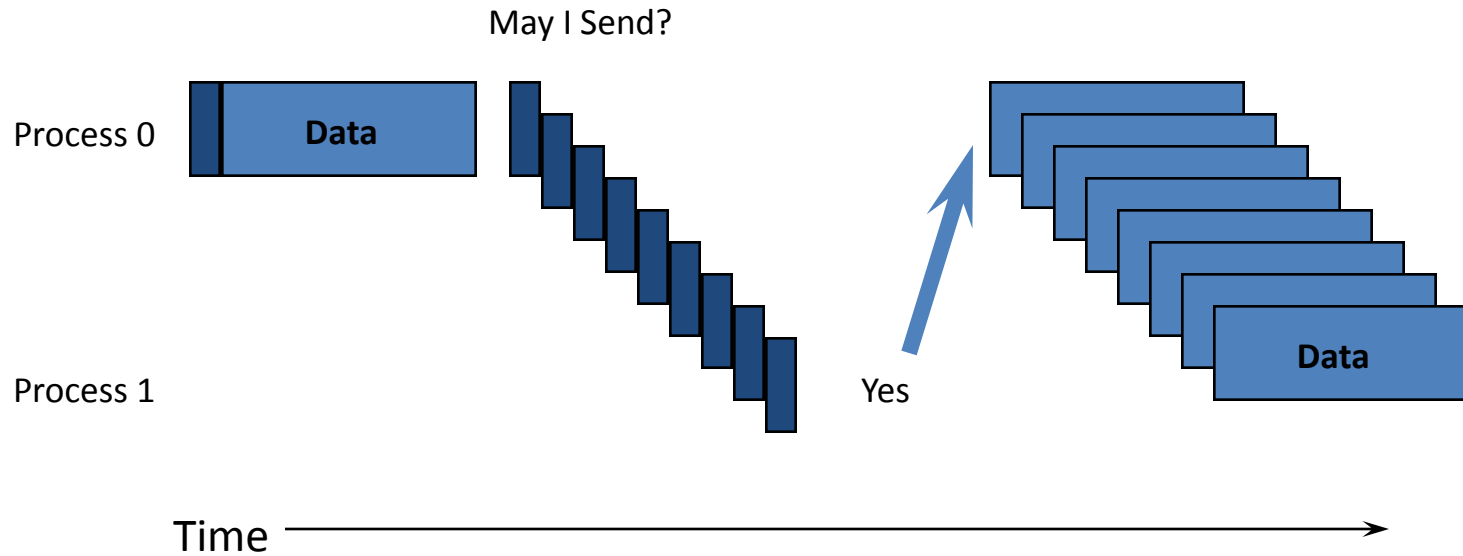
- 
- Reduces synchronization delays
 - Simplifies programming (just MPI_Send)
 - Requires significant buffering
 - May require active involvement of CPU to drain network at receiver's end
 - May introduce additional copy (buffer to final destination)

How Scalable is Eager Delivery?



- Buffering must be reserved for arbitrary senders
- User-model mismatch (often expect buffering allocated entirely to “used” connections).
- Common approach in implementations is to provide same buffering for all members of `MPI_COMM_WORLD`; this is optimizing for non-scalable computations
- Scalable implementations that exploit message patterns are possible

Rendezvous Protocol





- Envelope delivered first
- Data delivered when user-buffer available
 - Only buffering of envelopes required

Rendezvous Features



- Robust and safe
 - (except for limit on the number of envelopes...)
- May remove copy (user to user direct)
- More complex programming (waits/tests)
- May introduce synchronization delays (waiting for receiver to ok send)

Short Protocol

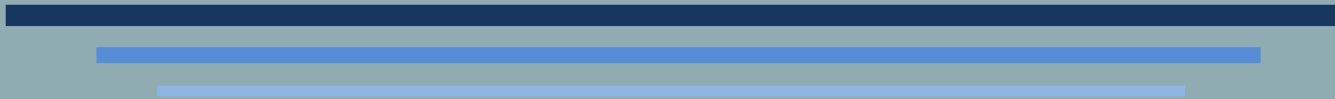
- 
- 
- Data is part of the envelope
 - Otherwise like eager protocol
 - May be performance optimization in interconnection system for short messages, particularly for networks that send fixed-length packets (or cache lines)

Special Protocols for DSM



- Message passing is a good way to use distributed shared memory (DSM) machines because it provides a way to express memory locality.
- Put
 - Sender puts to destination memory (user or MPI buffer). Like Eager.
- Get
 - Receiver gets data from sender or MPI buffer. Like Rendezvous.
- Short, long, rendezvous versions of these

MPI POINT-TO-POINT COMMUNICATIONS



Blocking, Bufferized, Ready, Non-Blocking

Blocking Communications



- Definition of operations
 - An **operation** is a set of one or more procedures leading from a well-defined input state to a well_defined output state. An operation consists in four stages: Initialization, starting, completion and freeing.
 - **Initialization** hands over the argument list to the operation but not to the content of the message buffers. For an operation, it may be specified that array arguments must not be changed until the operation is freed

Blocking Communications



- Definition of operations
 - **Starting** hands over the content of the message data buffer to the associated operation.
 - Note that **initiation** is an abbreviation for the combination of the initialization and starting stages.
 - **Completion** returns control of the content of the message data buffers and indicates that any output buffers have been updated
 - **Freeing** returns control of the rest of the argument list

Blocking Communications



- Definition

- A **procedure is blocking** if return from the procedure indicates the user is allowed to reuse resources specified in the call.
- For a **blocking operation**, all four stages are combined in a single blocking procedure call.

Blocking Communications



■ Definition

- A *send* is **blocking** if after performing *send* it is possible to manipulate (read/write) the input data buffer without corrupting the communication

■ Meaning

- A blocking *send* will not return while the communication library is not able to handle the message
- It doesn't mean that
 - Message has been received
 - Message has been sent !!!

Blocking Communications



- After *send*, T0 may modify the value of a
- T1 will receive 100 (value of a as input of *send* by T0)
- Note
 - Resolving a blocking send does not mean that the receiver has the message

Blocking Communications



- Definition

- A *recv* is **blocking** if after performing *recv* the output buffer contains the received message

- Meaning

- A blocking *recv* will not return while the message has not been received and processed

Blocking Communications



- After *send*,
 - T0 may manipulate *a* and its content
- After *recv*,
 - Content of output buffer (*a* in T1) can be manipulated (read, write, print...) without concurrency issue

Blocking Communications



- `MPI_Send` et `MPI_Recv` are blocking
 - `MPI_Send` returns when data buffer can be manipulate again by sender
 - `MPI_Recv` returns when the message arrived and has been processed

- Issue?
 - Be careful to deadlock situations!

Ring Topology

```
left = (rank + P - 1) % P;  
right = (rank + 1) % P;
```

```
if (rank == 0)  
    m = 0;
```

```
/* Receiving from left-hand side */
```

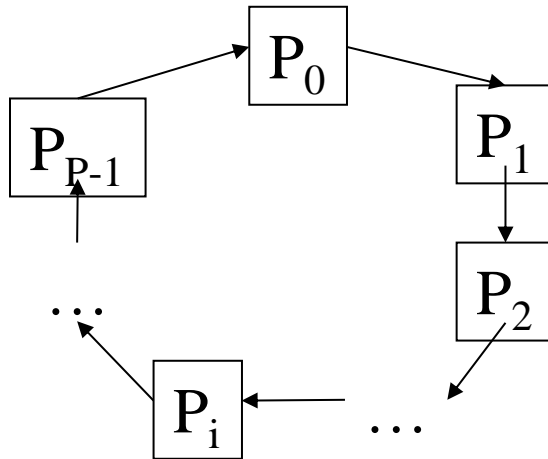
```
MPI_Recv(&m, 1, MPI_INT, left, tag1, MPI_COMM_WORLD, &sta);
```

```
/* Sending to right-hand side */
```

```
MPI_Send(&m, 1, MPI_INT, right, tag2, MPI_COMM_WORLD);
```

Ring example:

Processes pass one message in increasing rank value

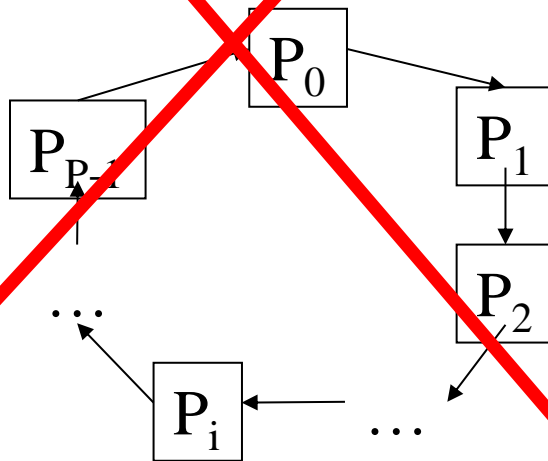


Ring Topology

```
left = (rank + P - 1) % P;  
right = (rank + 1) % P;  
  
if (rank == 0)  
    m = 0;  
  
/* Receiving from left-hand side */  
MPI_Recv(&m, 1, MPI_INT, left, tag1, MPI_COMM_WORLD, &sta);  
  
/* Sending to right-hand side */  
MPI_Send(&m, 1, MPI_INT, right, tag2, MPI_COMM_WORLD);
```

Ring example:

Processes pass one message in increasing rank value



Each process P_i waits a message from P_{i-1} before sending it to P_{i+1} . To do so, P_{i-1} should send this message, but P_{i-1} is blocked because it wait for a message from P_{i-2} ...

⇒ **deadlock**

Communication Mode



- Multiple modes for blocking communications
 1. **Synchronous mode**
 2. Buffered mode
 3. Standard mode

Synchronous Mode



- Definition

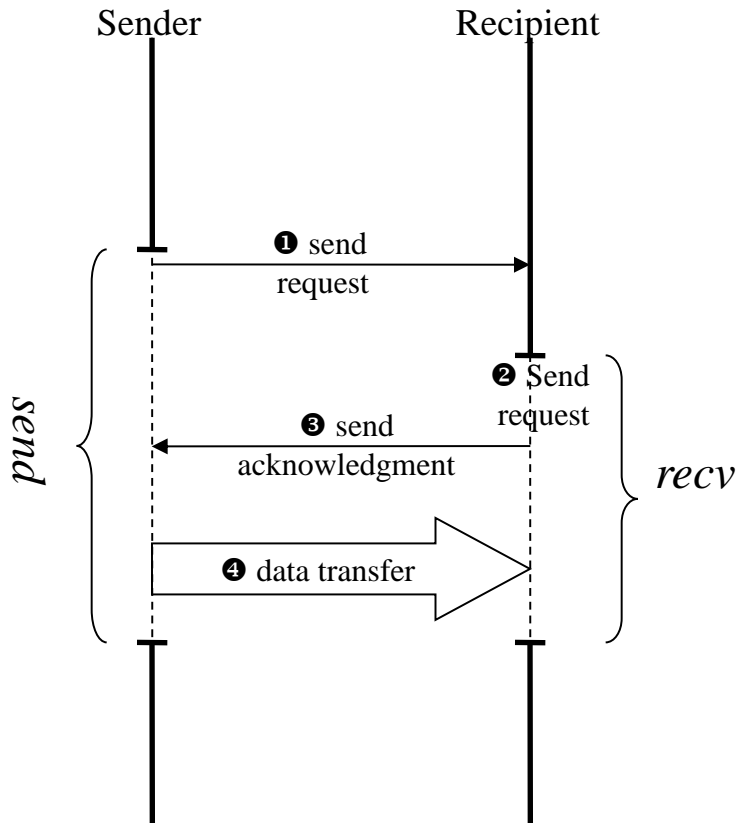
- A **synchronous send** will block while the message has not been received by the recipient

- Implementation

- Require some sort of synchronization mechanism between sender and recipient
- Design of a data-transfer protocol

Synchronous Mode

■ Synchronous communication protocol



- ❶ For a synchronous send, sender transfer a request to the receiver and waits for an answer
- ❷ When recipient starts the `recv` function, it waits for a sender request
- ❸ When recipient has the expected request, it answers with an acknowledgment message
- ❹ Sender and recipient are now synchronized leading to a safe data transfer

Synchronous Mode



■ Advantages

- No intermediate copy inside internal buffer
- May rely on optimized direct remote memory access (DMA or RDMA)

■ Drawbacks

- Involve a remote synchronization (like *rendez-vous*) between the two MPI processes
- May lead to idle overhead

■ Optimal situation

- When sender and recipient calls the corresponding function *at the same time*
- Possible in data parallelism when load is balanced between the two MPI processes

MPI Synchronous Mode



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

- Same signature as MPI_Send.
- Receive with function MPI_Recv

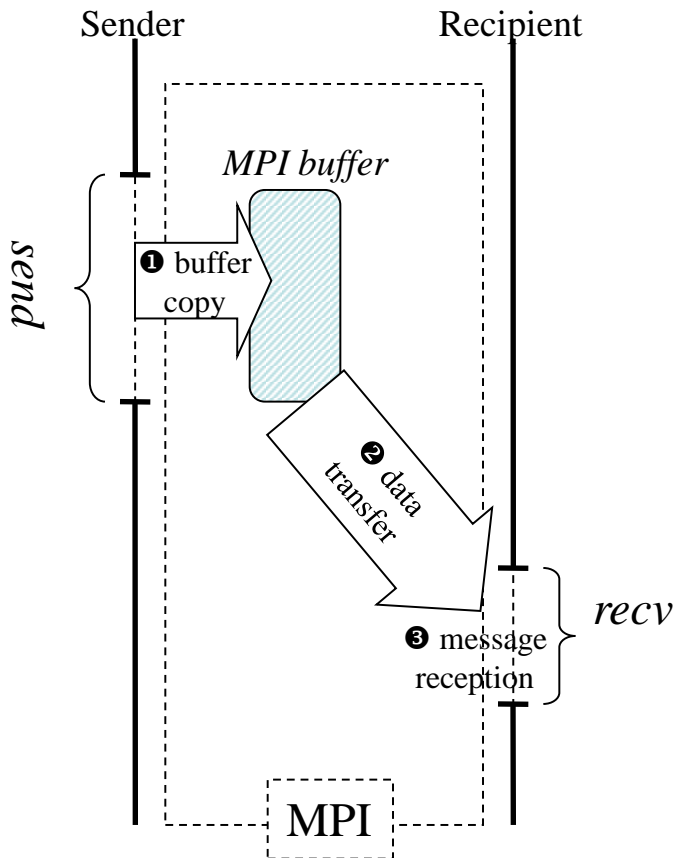
Communication Mode



- Multiple modes for blocking communications
 1. Synchronous mode
 2. **Buffered mode**
 3. Standard mode

Buffered Mode

- Waits until message has been copied to internal buffer
- Protocol:



- ➊ Sender copies incoming message inside a buffer (managed by the communication library). Send function may return
- ➋ Communication library owns a copy of the data to transfer and sends it to the recipient
- ➌ Recipient gets the message asap

Buffered Mode



- Advantages
 - Ability to decouple *send* and *recv* actions: *send* may return before recipient calls *recv* function
- Drawbacks
 - Intermediate data copy
 - CPU overhead
 - Memory consumption overhead
 - Memory bandwidth overhead
 - Limited to an upper bound (buffer size)
- Optimal situations
 - When *send* and *recv* functions are not posted *at the same time*
 - Load is not balanced between MPI processes

Buffer Allocation

- User may provide its own buffer to replace the internal one.
 - Function to attach user-allocated buffer `buf` of size `sz` bytes

```
int MPI_Buffer_attach(void *buf, int sz);
```
- Such buffer can be released and used again in the application by the user
 - Function to detach a user-allocated buffer
 - Return the buffer start address and its size

```
int MPI_Buffer_detach(void **buf_adr, int *sz);
```

Buffer Allocation

```
#define BUFFSIZE 100000
int sz;
char *buf;

MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE );
...
MPI_Bsend(msg1, ...);
MPI_Bsend(msg2, ...);
...
MPI_Buffer_detach( &buf, &sz );
free(buf);
```

- Only used in `MPI_Bsend`
- Only one buffer may be attached
- Only useful for sender

Communication Mode



- Multiple modes for blocking communications
 1. Synchronous mode
 2. Buffered mode
 3. **Standard mode**

Standard Mode



- Function for standard communication
 - `MPI_Send`
- Standard communication protocol
 - MPI includes an internal threshold T
 - If input message size is lower than T
 - Switch to buffered mode
 - If input message size is larger than T
 - Switch to synchronous mode

Standard Mode

```
if ( rang == 0 )
voisin = 1;
else if ( rang == 1 )
voisin = 0;

MPI_Send(&msg1, N, MPI_BYTE, voisin, tag1, comm);
MPI_Recv(&msg2, N, MPI_BYTE, voisin, tag2, comm);
```

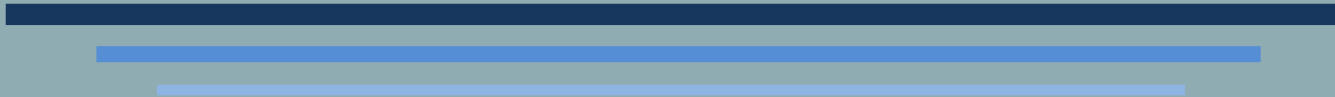
- Is this code safe?
- NO
 - If N is small enough → OK
 - If N is too large → Deadlock

Standard Mode



- Hint to detect such issues
 - Replace calls to `MPI_Send` by `MPI_Ssend`
 - Whatever the size of messages and scheduling, the applications should not deadlock
- Deadlocks means application bug!

MPI POINT-TO-POINT COMMUNICATIONS



Nonblocking Communications

Nonblocking Communication




- Definition

- A **nonblocking** communication has no guarantee when send function returns!

- Meaning

- No safe access to input message when function send returns
- To be sure that message buffer can be reused, an additional function should be called and returned

Nonblocking Send MPI_Isend



```
int MPI_Isend (  
void *buf(in),  
int count(in),  
MPI_Datatype datatype(in),  
int dest(in),  
int tag(in),  
MPI_Comm comm(in),  
MPI_Request *req(out)  
);
```

One additional argument
MPI_Request *req.

Request id is returned in *req
(MPI_Request = MPI opaque
type).

To finish the communication
MPI_Wait should be called.

Check Function MPI_Wait

```
int MPI_Wait (  
    MPI_Request *req(inout),  
    MPI_Status *sta(out)  
);
```

MPI_Wait blocks until communication represented by *req is done.

Detailed information about finished communication are store into *sta.

When MPI_Wait returns

- *req is assigned to MPI_REQUEST_NULL (invalid request)
- Input message buffer can be safely manipulated by sender

Remark:

MPI_Send ⇔ MPI_Isend + MPI_Wait

Nonblocking Example

```
MPI_Request req;
MPI_Status sta;

MPI_Isend(buf, N, MPI_BYTE,
dest, tag1, comm,
&req);

instruction1;
instruction2;
...
instructionN;

MPI_Wait(&req, &sta);
```

Instructions
between
MPI_Isend and
MPI_Wait should
not write into buf.

In the meantime,
message progresses

- Advantages
 - Recover communications and computation

Nonblocking Communication



```
int MPI_Test (  
MPI_Request *req(inout),  
int *flag(out),  
MPI_Status *sta(out)  
);
```

Write true (non-zero value) in *flag if request *req is over.

If *flag is true, *req is assigned to MPI_REQUEST_NULL and *sta is filled.

If *flag is false, values of *req and *sta are not guaranteed.

Nonblocking Communication



- Example :

```
MPI_Irecv(msg, N, MPI_BYTE, dest, tag, comm, &req);  
do {  
    instruction1;  
    ...  
    instructionN;  
    MPI_Test(&req, &flag, &sta);  
} while( !flag );
```

Nonblocking Communication

```
int MPI_Waitall (  
    int nb_req(in),  
    MPI_Request *tab_req(inout),  
    MPI_Status *tab_sta(out)  
);
```

Return when `nb_req` requests located in array `tab_req` are completed.

Status of communications are available as output in array `tab_sta`.

Remark:

Order of communication completion is not important

Nonblocking Communication

- Example: send/receive with left/right neighbors

```
MPI_Request req[4];
MPI_Status sta[4];

left = (rang + P - 1) % P;
right = (rang + 1) % P;

MPI_Isend(&x[1], 1, MPI_DOUBLE, left, tag, comm, req);
MPI_Isend(&x[N], 1, MPI_DOUBLE, right, tag, comm, req+1);
MPI_Irecv(&x[0], 1, MPI_DOUBLE, left, tag, comm, req+2);
MPI_Irecv(&x[N+1], 1, MPI_DOUBLE, right, tag, comm, req+3);

MPI_waitall(4, req, sta);
```

Other Available Functions




- MPI proposes multiple functions to complete non-blocking communications
- `MPI_Testall`
 - Test is all requests as input are completed
- `MPI_Waitany / MPI_Testany`
 - Wait/Test until at least one request is completed
 - Return index of completed request
- `MPI_Waitsome / MPI_Testsome`
 - Wait/Test until at least one request is completed
 - Return set of completed requests

Communications and modes

- Nonblocking communication is different from asynchronous
- Nonblocking communications can be done in different modes: synchronous, buffered or regular

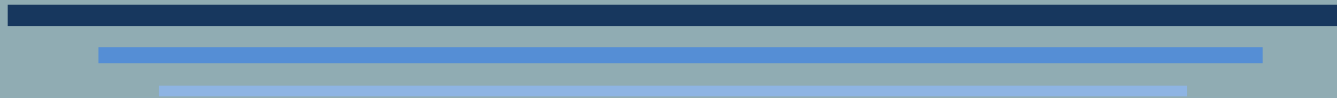
Type/Mode	Standard	Buffered	Synchronous	Receive
Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_recv
Non-Blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irecv

P2P comm \neq Protocols



- Eager not Bsend or Rsend, rendezvous not Ssend resp., but related
- Each Point-to-point communications is ultimately implemented with the three protocols

CHECKING INCOMING MESSAGES



Checking Incoming Messages



- How to receive a message without knowing the actual final size?
 - `MPI_Recv` function requires an upper bound on incoming messages
 - `MPI_Recv` is not appropriate if the message size is unknown
 - MPI proposes function to retrieve information on incoming messages before performing the receive actions: `MPI_Iprobe` and `MPI_Probe`

Checking Incoming Messages



```
int MPI_Probe (  
int source(in),  
int tag(in),  
MPI_Comm comm(in),  
int *flag(out)  
);
```

Wait for a message coming from sender **source** with label **tag** has arrived (MPI_ANY_SOURCE and MPI_ANY_TAG are allowed).

Upon return, status is written in ***sta**.

Checking Incoming Messages

```
int MPI_Iprobe (  
int source(in),  
int tag(in),  
MPI_Comm comm(in),  
int *flag(out),  
MPI_Status *sta(out)  
);
```

Check if a message coming from **source** with label **tag** has arrived (MPI_ANY_SOURCE and MPI_ANY_TAG are allowed).

Return true (non-zero value) in ***flag** such a message exists.

In such case, status of incoming message is provided in ***sta**.

Receiving messages after

MPI_Iprobe/MPI_Probe



- Calls to `MPI_Iprobe` and `MPI_Probe` checks incoming messages or wait for a specific message to come.
 - But they do not perform the actual reception
- To receive the target message:
 1. Call `MPI_Get_count` to get the message size
 2. Allocate a buffer corresponding to this size
 3. Call `MPI_Recv` to receive the message

Receiving messages after

MPI_Iprobe/MPI_Probe



```
MPI_Status sta;
int size, done;
do {
    instruction1;
    ...
    instructionN;
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &done, &sta);
} while (!done);

MPI_Get_count(&sta, MPI_BYTE, &size);
char *buf = malloc( size );
MPI_Recv(buf, size, MPI_BYTE, sta.MPI_SOURCE, sta.MPI_TAG,
MPI_COMM_WORLD, &sta);
```