UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

Part I

Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

Agenda

- Introduction to MPI
- What is MPI?
- Environment management

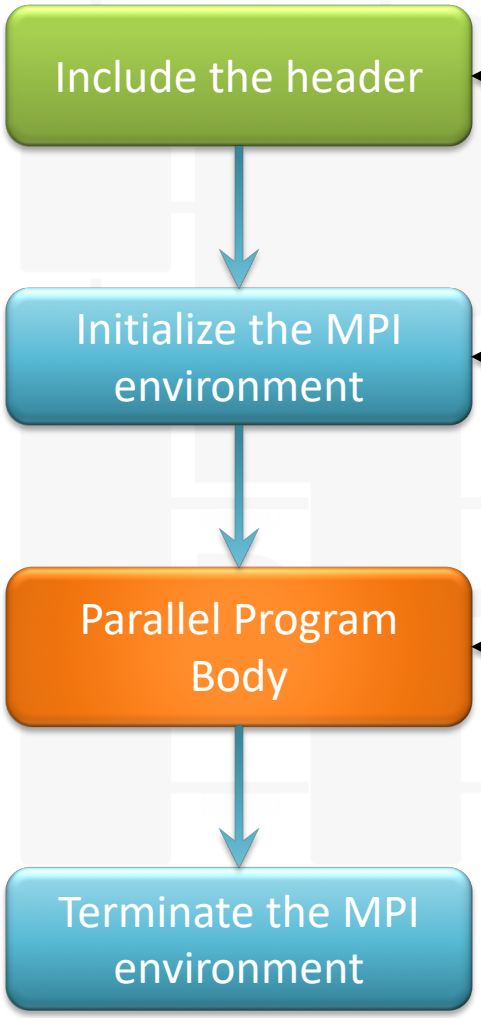
- Point-to-point communication
  - Blocking communication
  - Non-blocking communication
- MPI collective communication (Basic)
  - Synchronization
  - Broadcast, Scatter, Gather, Reduce

- Examples
- Exercises
- References

What is MPI?

- The Message Passing Interface (MPI) Standard is a message passing system
- MPI is a language-independent communication protocol used to program parallel computers
- MPI's goals are:
  - High performance
  - Scalability
  - Software Portability: Interface specifications have been defined for C/C+, Fortran, Java, Python, …
  - Hardware Portability: Intel, AMD, ARM, and PowerPC chips
- MPI is a standard for communication among processes that model a parallel program running on a distributed memory system
- The principal MPI model has no explicit shared memory concept but MPI programs can take advantage of shared memory to improve memory locality and communications.
- Many parallel scientific applications today use MPI and run successfully on the most of supercomputers.

General MPI Program Structure



#include “mpi.h” in C/C++

COMMUNICATOR and RANK

- MPI relies on a number of library routines. It provides over a hundred routines...
- But many programs require only 6 basic routines

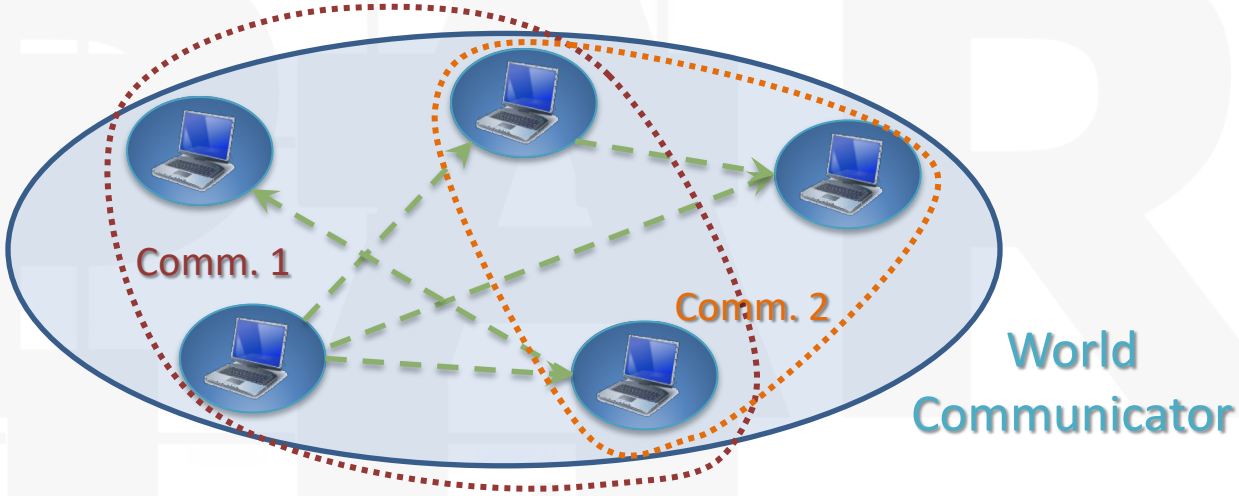
MPI in C++

```
MPI::<Func_name>(<param1>, <param2>, …)
```

- The functions are defined within the namespace MPI
- Arguments are declared with *references* instead of *pointers*
- Most MPI functions are methods of MPI C++ classes
- MPI class names are derived from the language neutral MPI types by dropping the MPI prefix and scoping the type within the MPI namespace: MPI::Datatype
- C++ functions do not return error codes. In the C++ language, error-handling is performed using EXCEPTIONS

Communicators and rank

- MPI uses objects called ***communicators*** to define which collection of processes (also called tasks) may communicate with each other.



- Processors can communicate only if they share a common communicator.
- MPI::COMM_WORLD is the predefined communicator that includes all MPI processes.
- Within a communicator, each process has its own unique integer identifier (rank) assigned by the system when the process initializes.
  - Ranks are contiguous and start with 0
  - Ranks are used by the programmer to specify the source and the destination of a message

Environment management Methods (I)

- MPI environment management methods are used for a wide array of purposes, ranging from initializing and terminating the MPI environment to querying it

```
MPI::Init(&argc, &argv)
```

- Initializes the MPI environment
  - It must be called in every MPI program, before any other MPI function and only once in a MPI program
  - It may be used to pass command line arguments to all processes

```
int MPI::COMM::Get_rank();
```

- Gets the rank of the calling process within communicator comm
  - If a process becomes associated with other communicators, it will have a unique rank within each of these as well

```
int MPI::COMM::Get_size()
```

- Gets the number of processes in the group associated with communicator COMM_WORLD

Environment management routines (II)

```
MPI::Get_processor_name(char* name, int& resultlength)
```

- Provides the processor name, as well as its length (in characters) of the name

```
MPI::Finalize()
```

- Terminates the MPI execution environment.
  - This function should be the last MPI routine called in every MPI program
  - No other MPI routines may be called after it

```
MPI::COMM::Abort( int errorcode )
```

- Terminates all MPI processes associated with communicator COMM_WORLD providing errorcode as error code

Environment management routines (III)

```
bool MPI::Is_initialized()
```

- Return true if MPI_INIT has been called, false otherwise

```
double MPI::Wtime()
```

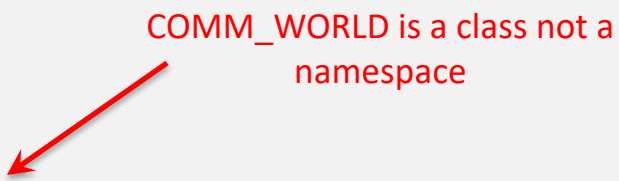
- Returns the elapsed wall clock time in seconds (double precision)

```
double MPI::Wtick()
```

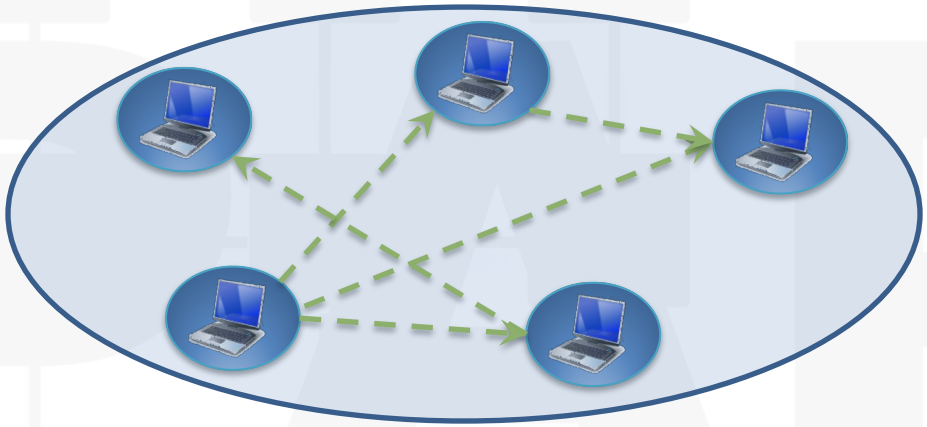
- Returns the resolution of MPI::WTIME() in seconds (double precision)

Hello World Example (I)

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

int main(int argc, char* argv[]) {
        // Initialize the MPI environment
        MPI::Init(argc, argv);

        // Get the number of processes
        int world_size =  MPI::COMM_WORLD.Get_size();

        // Get the rank of the process
        int world_rank = MPI::COMM_WORLD.Get_rank();

        // Get the name of the processor
        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI::Get_processor_name(processor_name, name_len);

        // Print off a hello world message
        printf("Hello world from processor %s, rank %d out of %d processors\n",
         processor_name, world_rank, world_size);

        // Finalize the MPI environment.
        MPI::Finalize();
}
```

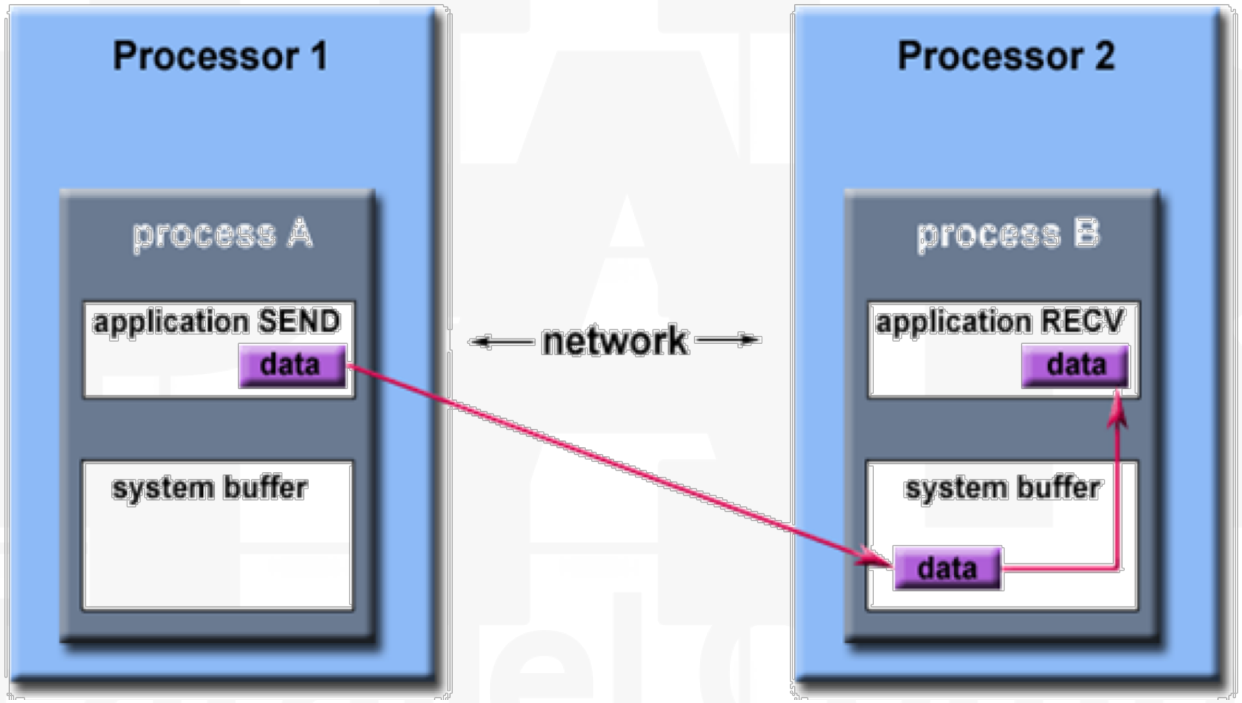


Point-to-point communication



- MPI point-to-point operations involve message passing between only two different MPI tasks
  - One task is performing a SEND operation and the other task is performing a matching RECEIVE operation
- Point-to-point Communication: Blocking and Non-Blocking
- There are different types of SEND and RECEIVE routines

Application and system buffer



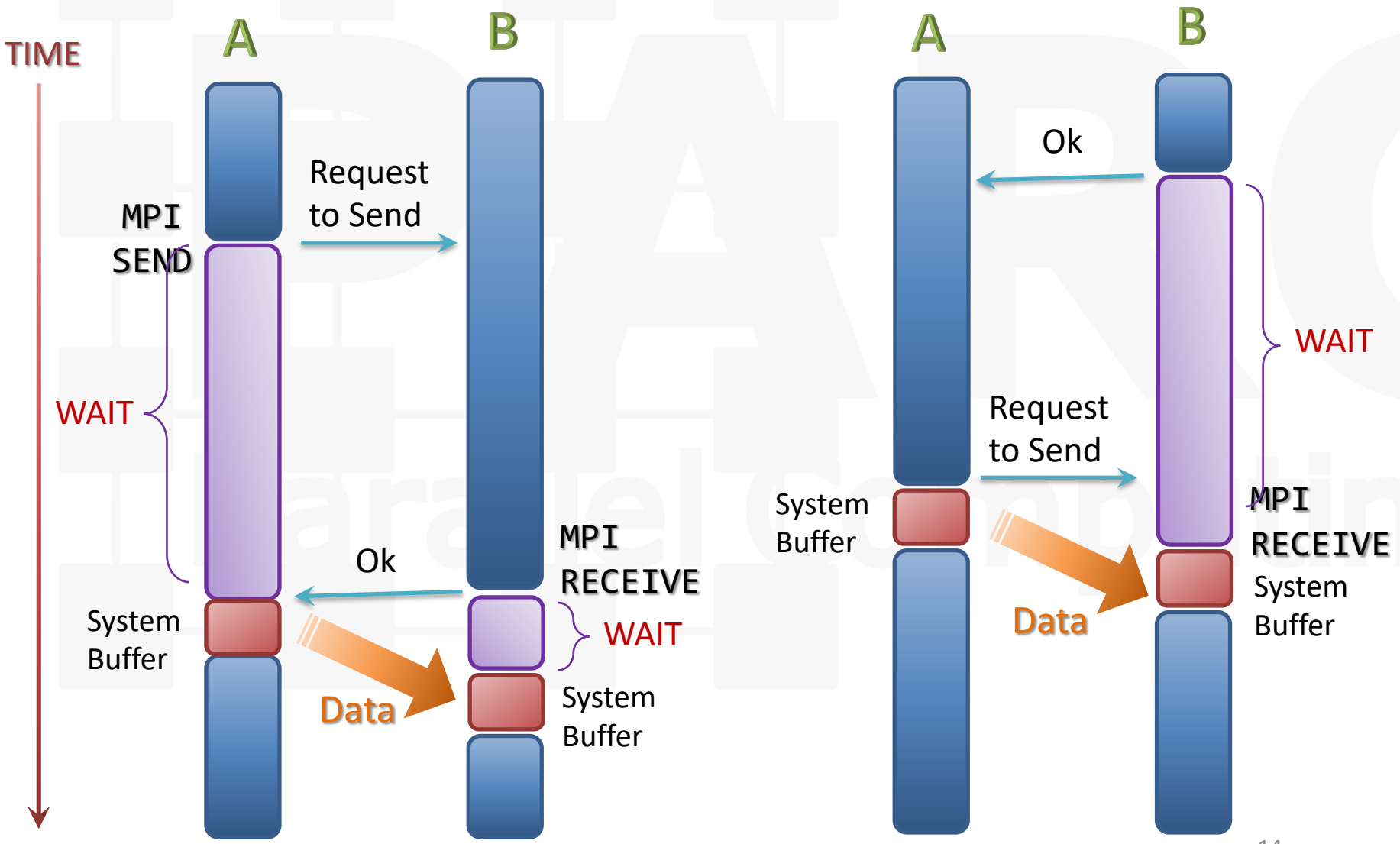
- A *system buffer* area is reserved to hold data in transit
  - This area is opaque to the programmer and managed entirely by the MPI library
- The address space managed by the user (i.e. your program variables) is called *application buffer*
  - MPI also provides support for user managed send buffer

Blocking communication (I)

- A blocking communication suspends the execution of the program
- Call does not return until the operation has been completed
- Allows you to know when it is safe to use the data received or reuse the data sent

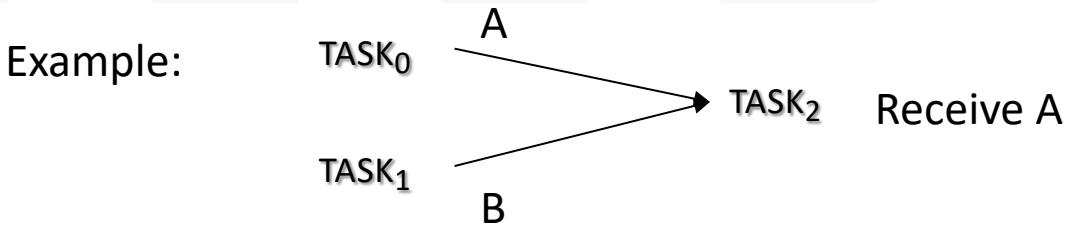
- A blocking SEND will only return after it is safe to modify the application buffer (program variables) for reuse
  - It does not imply that the data were actually received
- A blocking RECEIVE only returns after the data has arrived and is ready for use by the program

Blocking communication (II)



Blocking communication
Order and fairness

- MPI guarantees that messages will not overtake each other
  - If a sender sends MESSAGE_1 and then MESSAGE_2 to the same destination, and both messages match the same receive, MESSAGE_1 will be received before MESSAGE_2
  - If a receiver performs RECEIVE_1 and then RECEIVE_2, and both are looking for the same message, RECEIVE_1 will receive the message before RECEIVE_2
- MPI does not guarantee fairness
  - It is up to the programmer to prevent starvation
  - Example: TASK_0 sends a message to TASK_2, but TASK_1 sends a competing message to TASK_2, matching the receive of TASK_2 → only one of the send operations will complete



Arguments of message passing method (I)

Buffer	Pointer to the application address space that references the data to be sent or received
Data count	The number of data elements of a particular type to be sent/received
Data Type	For reasons of portability, MPI predefines its elementary data types
Source/Dest	The rank of the receiving/sending process
Tag	Tags are a way to identify types of messages. So, a receiving process can use these tags to decide what messages it wants to receive at a given time
Communicator	The set of processes for which the source or destination arguments are valid
Status	Pointer to a MPI_Status struct, which contains the source, the tag of the message and the length of the message (and other two)

MPI elementary data types

- MPI class names are derived from the language neutral MPI types within the MPI namespace MPI::<DATATYPE>

MPI data type	C/C++ datatype	MPI datatype	C/C++ data type
CHAR	signed char	MPI_FLOAT	float
WCHAR	wchar_t (wide char type)	MPI_DOUBLE	double
SHORT	signed short int	MPI_C_BOOL	bool
INT	signed int	MPI_INT8_T MPI_UINT8_T	int8_t / uint8_t
LONG	signed long		
LONG_LONG	signed long long	MPI_INT16_T MPI_UINT16_T	int16_t / uint16_t
UNSIGNED_CHAR	unsigned char		
UNSIGNED_SHORT	unsigned short	MPI_INT32_T MPI_UINT32_T	int32_t / uint32_t
UNSIGNED	unsigned int	MPI_INT64_T MPI_UINT64_T	int64_t / uint64_t
2INT	{ int , int }		

MPI Operator

- MPI class names are derived from the language neutral MPI operator within the MPI namespace MPI::<OP>

MPI Operator	C/C++ Operator	Symbol
SUM	Sum	+
PROD	Product	*
MAX	Maximum	Max
MIN	Minimum	Min
MAXLOC	max value and location	
MINLOC	min value and location	
LAND	logical and	&&
LOR	logical or	||
LXOR	logical xor	!=
BAND	bit-wise and	&
BOR	bit-wise or	|
BXOR	bit-wise xor	^

Status Methods

In C++, the `MPI::Status` object is handled through the following methods:

```
int Status.Get_source()
```

Return the id of processor sending the message

```
int Status.Get_tag()
```

Return the message tag

```
int Status.Get_error()
```

Return the id of error status

```
int Status.Get_count(MPI::Datatype& datatype)
```

Return the number of received elements
Notes: If the amount of data in status is not an exact multiple of the size of datatype, a count of MPI_UNDEFINED is returned instead.
[implemented in MPICH]

Useful Macro

MPI_ANY_SOURCE

In a receive, accept a message from anyone.

MPI_ANY_TAG

In a receive, accept a message with any tag value.

MPI_PROC_NULL

This rank may be used to send or receive from no-one

MAX_PROCESSOR_NAME

Maximum length of name returned by MPI::GET_PROCESSOR_NAME()

MPI_MAX_ERROR_STRING

Maximum length of string return by MPI::GET_ERROR_STRING()

MPI_WTIME_IS_GLOBAL

If is defined and true, then the time is synchronized across all processes in COMM_WORLD

NOTE: There is no way to change MPI_WTIME_IS_GLOBAL

Blocking communication

MESSAGE PASSING METHOD (I)

```
MPI::COMM::Send(void* buf, int count, MPI::Datatype& datatype,
                int dest, int tag)
```

- Basic blocking send operation
- It returns only after the application buffer in the sending task is free for reuse

```
MPI::COMM::Recv(void* buf, int count, MPI::Datatype& datatype,
                int source, int tag, [&status])
```

- Basic blocking receive operation
- It receives a message and blocks until the requested data is available in the application buffer in the sending task
  - The call store, in status, information on the completed operation

```
MPI::COMM::Sendrecv( void* sendbuf, int sendcount,
          MPI::Datatype& senddatatype, int dest, int sendtag,
          void* recvbuffer, int recvcont,
          MPI::Datatype& recvtype, int source, int recvtag,
          [&status] )
```

- Blocking send-receive operations combine in one call

Blocking communication

MESSAGE PASSING METHOD (II)

```
MPI::COMM::Ssend(void* buf, int count, MPI::Datatype& datatype,
          int dest, int tag)
```

- Synchronous blocking send operation
- It returns only after the application buffer in the sending task is free for reuse **and** the destination process has started to receive the message

```
MPI::COMM::Rsend(void* buf, int count, MPI::Datatype& datatype,
          int source, int tag)
```

- Started only if the matching receive is already posted
- Otherwise, the operation is erroneous and its outcome is undefined
- In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance

Self-messaging
PASSING METHOD (III)

- Self-messaging will work and is part of the MPI standard.
- There is even a pre-defined convenience communicator MPI_COMM_SELF
- Send/receive calls do not cause deadlock (for example, non-blocking calls are used)
- MPI Specification:

  Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock.)

- The behavior is implementation dependent and must be avoided

Non-blocking communication

- A non-blocking SEND and a non-blocking RECEIVE will return almost immediately, they do not wait for any communication event to complete
- Non-blocking operations simply request the MPI library to perform the operation when it is able (the user cannot predict when that will happen)
- It is unsafe to modify the application buffer until you know that the requested operation has actually been performed by the MPI library
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Extra arguments of message passing routines:

Request	Pointer to a MPI::Request struct, which contains a unique request number in order to allow the completion of a non-blocking operation

Non-Blocking communication

MESSAGE PASSING METHOD (I)

```
Request MPI::COMM::Isend(void* buf, int count,
                MPI::Datatype& datatype, int dest, int tag)
```

- Is the basic non-blocking send operation
- Processing continues immediately; a communication REQUEST HANDLE is provided for handling the pending message status
- The application buffer should not be modified until subsequent calls to MPI::WAIT or MPI::TEST indicate that the non-blocking send has completed

```
Request MPI::COMM::Irecv(void* buf, int count,
              MPI::Datatype& datatype, int source, int tag)
```

- Is the basic non-blocking receive operation
- Processing continues immediately without waiting for the message to be received and copied into the application buffer
- The program must call MPI::WAIT or MPI::TEST to determine when the non-blocking receive operation completes and the message is available in the application buffer

Non-Blocking communication

MESSAGE PASSING METHOD (II)

```
Request MPI::COMM::Issend(void* sendbuf, int sendcount,
          MPI::Datatype& datatype, int dest, int tag)
```

- Is the synchronous non-blocking send operation
- It is similar to MPI::ISEND, except that MPI::WAIT or MPI::TEST indicate when the destination process has received the message

```
  Request.Wait( [&status] )
```

- The MPI::WAIT routine blocks processing until a specified non-blocking send or receive operation has completed
  - The call store, in status, information on the completed operation

```
  bool Request.Test( [&status] )
```

- The MPI::TEST routine checks the status of a specified non-blocking send or receive operation
  - It return true if the operation has completed, false otherwise
  - The call store, in status, information on the completed operation

C++ Error handling (I)

- C++ functions do not return error codes
- If the default error handler is set to MPI::ERRORS_THROW_EXCEPTIONS, then the C++ exception mechanism will be used

```
Comm.Set_errhandler(const MPI::Errhandler& errhandler)
```

Attaches a new error handler to a communicator

- The class MPI::Exception is basically a wrapper around an INT, it also provides a way to return an error description string:

```
int Exception.Get_error_code()
```

Converts an error class into an error code

```
int Exception.Get_error_class()
```

Converts an error code into an error class

```
const char* Exception.Get_error_string()
```

Returns a string for a given error code

C++ Error handling (II)

Exception handling example:

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

int main(int argc, char* argv[]) {
    MPI::Init(argc, argv);
    MPI::COMM_WORLD.Set_errhandler(MPI::ERRORS_THROW_EXCEPTIONS);
    try{
        int rank = MPI::COMM_WORLD.Get_rank();
        std::cout<< "I am " << rank << std::endl;
    }
    catch (MPI::Exception e) {
        std::cout << "MPI ERROR: " << e.Get_error_code()
              << " -" << e.Get_error_string() << std::endl;
    }
    MPI::Finalize();
}
```

Collective communication

- MPI collective communication must involve all processes in the scope of a communicator
  - ALL TASKS MUST REACH THE COLLECTIVE CALL, otherwise deadlock occurr.
- There are different types of collective operations:
  - Synchronization: processes wait until all members of the group have reached the synchronization point
  - Data movement: broadcast, scatter/gather
  - Collective computation (reductions): one member of the group collects data from the other members and performs an arithmethic operation on those values
- Collective operations are blocking

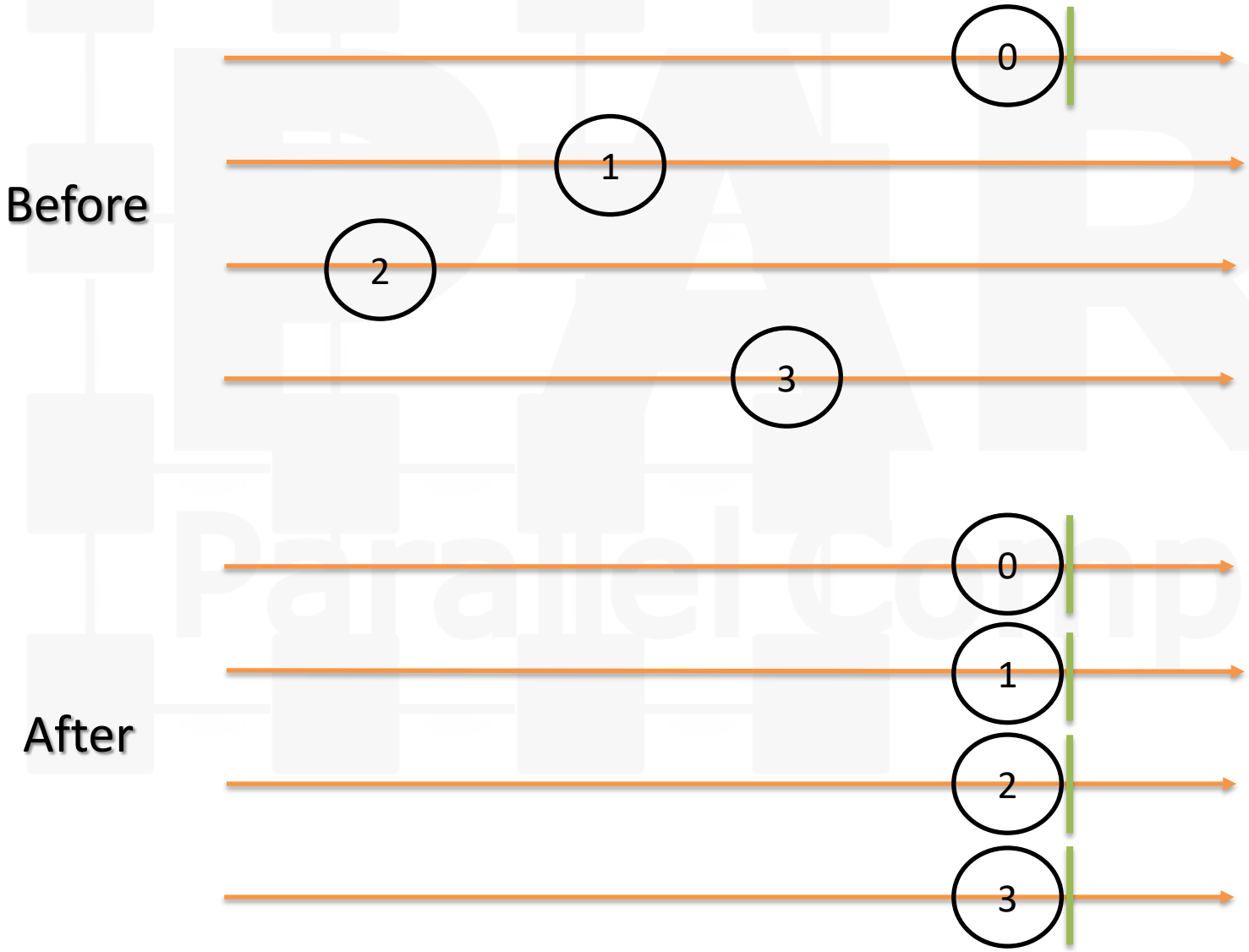
Collective communication

Basic Methods (I)

```
MPI::COMM::Barrier()
```

- Creates a synchronization barrier in a group
- Each task blocks when reaching the call wait until all tasks in the group reach the same call
- Barrier() is not necessary before/after collective operations, if all buffers are valid already.
- Barrier() does not magically wait for non-blocking calls. If you use a non-blocking send/recv and both processes wait at an Barrier() after the send/recv pair, it is not guaranteed that the processes sent/received all data after the Barrier(). Use Wait() instead.
- Can be useful to timing the code

Barrier



Collective communication

Basic Methods (II)

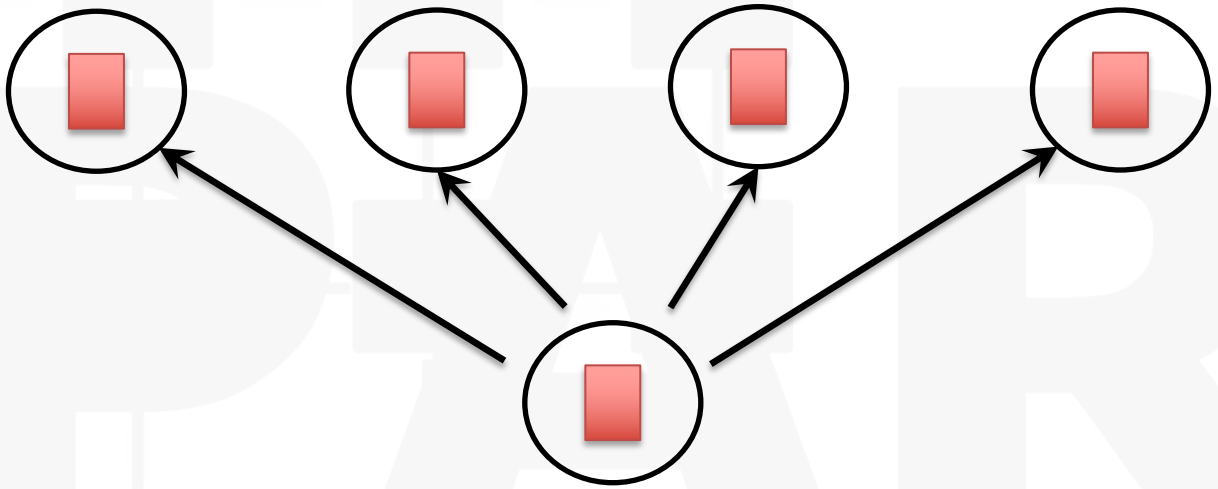
```
MPI::COMM::Bcast(void* buffer, int count,
                 MPI::Datatype& datatype, int root)
```

- Broadcasts a message from the process having ROOT as rank to all the other processes in the group

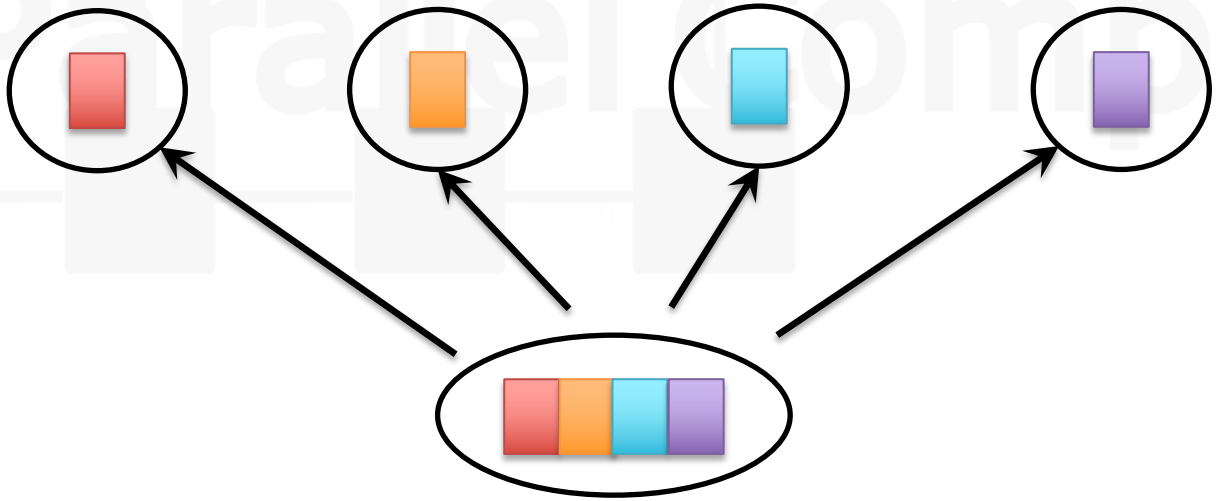
```
MPI::COMM::Scatter(void* sendbuf, int sendcount,
          MPI::Datatype& sendtype, void* recvbuf,
       int recvcount, MPI::Datatype& recvtype, int root)
```

- Distributes distinct messages from the task having ROOT as rank to each task in the group
- The first parameter, SENDBUF, is an array of data that resides on the ROOT process
- On the task ROOT, all arguments are relevant
- On the other tasks all arguments, except the first one, are relevant
- SENDCOUNT/RECVCOUNT is often equal to the number of elements in the array divided by the number of processes
- SENDBUF and RECVBUF must not overlap

Broadcast



Scatter



Collective communication

Basic Methods (II)

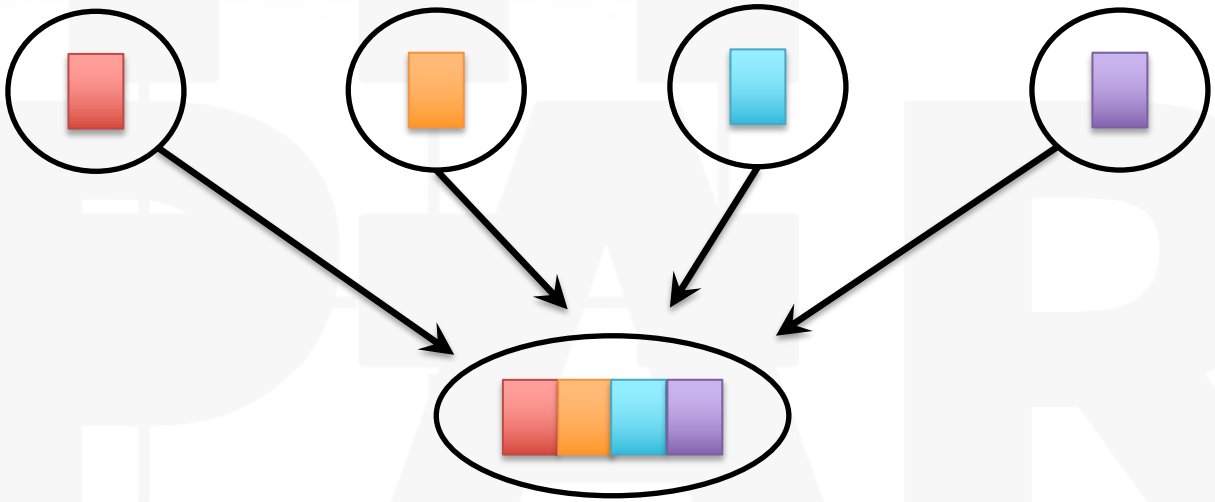
```
MPI::COMM::Gather(void* sendbuf, int sendcount,
        MPI::Datatype& sendtype, void* recvbuf,
        int recvcount, MPI::Datatype& recvtype, int root)
```

- Gathers distinct messages from each task in the group to the task having ROOT as rank
- On the task ROOT, all arguments are relevant
- On the other tasks all arguments, except RECVBUF and RECVCOUNT, are relevant
- GATHER() is the inverse of SCATTER()
- The received datas are ordered respect to task rank

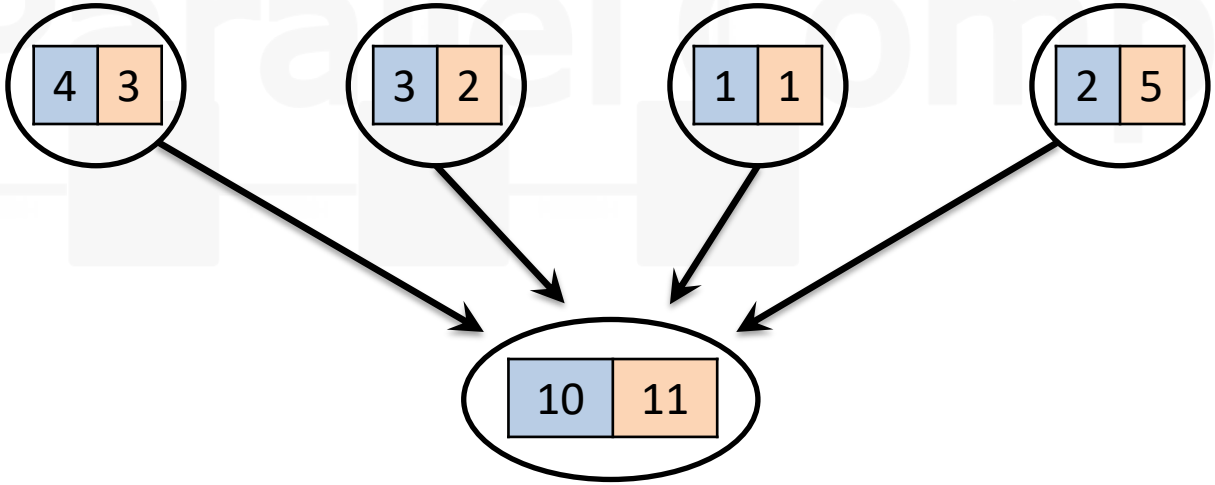
```
MPI::COMM::Reduce(void* sendbuf, void* recvbuf, int count,
        MPI::Datatype& datatype, MPI:OP, int root)
```

- Applies a reduction operation to the send buffer in each task in the group and places the result in the receive buffer in the task having ROOT as rank

Gather



Reduce <+>



Run a MPI Program
Run with Mpich

1) To compile a MPI program you have to use the wrapper compiler

```
mpicxx source.cpp –o out.x
```

2) Start the MPI program mpiprogram

```
mpiexec –n <number_of_processes> ./<mpi_program>
```

Examples

- Example 1: HelloWorld

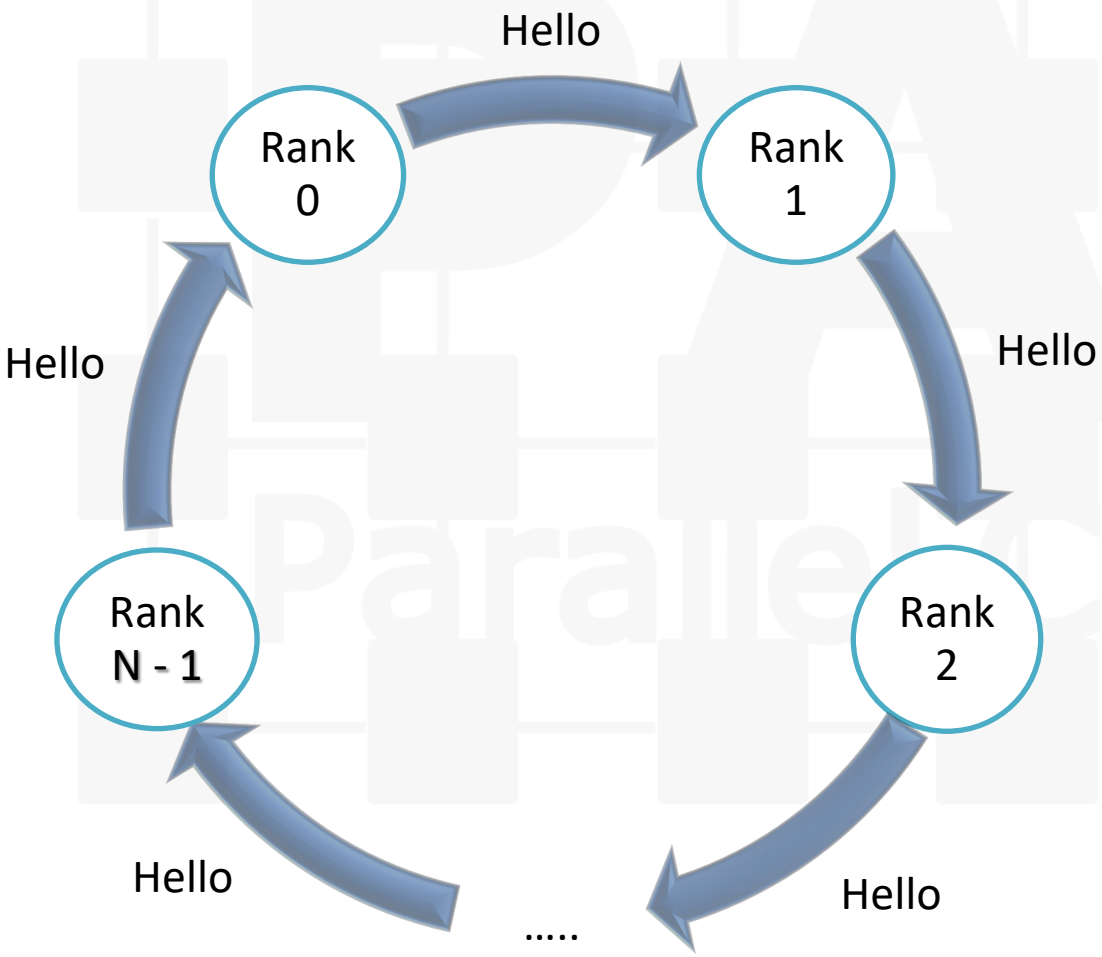
```
Hello world from processor <host>, rank 2 out of 4 processors
Hello world from processor <host>, rank 0 out of 4 processors
Hello world from processor <host>, rank 1 out of 4 processors
Hello world from processor <host>, rank 3 out of 4 processors
```

- Example 2: SendReceive


```
Message Received: "Hello! from rank 0" of size 18
```

Exercises (I)

RING



1) For each step print rank, source and dest of the message.
2) Use a collective communication to optimize the code

Exercises (II)
LATENCY AND BANDWIDTH

1) What is the latency between two nodes in the same host? and in different hosts?
2) What is the latency in the ring?
3) What is the bandwidth between two nodes on the same host? and in different hosts?

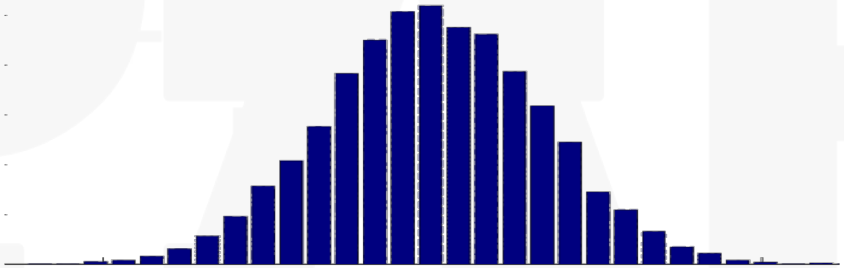
$$latency = \frac{receive_{time} - send_{time}}{2}$$

$$bandwitdh = \frac{number\ of\ bytes}{elapsed\ seconds}$$

Exercises (III)

Basic Collective Communication

Random Char Histogram



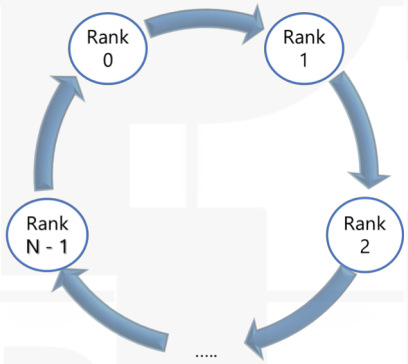
- Every task generate a random sequence of N chars (from ‘A‘ to ‘Z’).

1) N (problem size) is broadcast from root
2) Random seeds are scattered from root
3) Each task compute local histogram
4) The root takes the total histogram (sum of histogram)

What is the speedup again the sequential version for large values?

Exercises (IV)

- Modify RING exercise with non-blocking communication



- RC4 Decrypt

References

- MPICH Home page

  http://www.mpich.org/

- Official MPI specifications

  http://www.mpi-forum.org/docs/docs.html

- Good API References:
  - OpenMPI (another open source MPI implementation)

    https://www.open-mpi.org/doc/v1.8/
  - Microsoft MPI References

    https://msdn.microsoft.com/en-us/library/dn473458%28v=vs.85%29.aspx

- Tutorials

  http://mpitutorial.com/

  https://computing.llnl.gov/tutorials/mpi/