A quick introduction to MPI (Message Passing Interface)

Julien Braine Laureline Pinault

École Normale Supérieure de Lyon, France

M1IF - APPD
2019-2020

ENS DE LYON

Introduction

ENS DE LYON

- Standardized and portable message-passing system.
- Started in the 90's, still used today in research and industry.
- Good theoretical model.
- Good performances on HPC networks (InfiniBand ...).

ENS DE LYON

De facto standard for communications in HPC applications.

ENS DE LYON

APIs:

- C and Fortran APIs.
- C++ API deprecated by MPI-3 (2008).

Environment:

- Many implementations of the standard (mainly OpenMPI and MPICH)
- Compiler (wrappers around gcc)
- Runtime (mpirun)

ENS DE LYON

Basics

ENS DE LYON

Compiling:

mpicc -std=c99 <fichier.c> -o <executable>

Executing:

mpirun -n <nb procs> <executable> <args>

ENS DE LYON

Compiling:

mpicc -std=c99 <fichier.c> -o <executable>

Executing:

mpirun -n <nb procs> <executable> <args>

Exercise

Write a hello world program. Compile it and execute it with mpi with 8 processes. What do you get ?

ENS DE LYON

Program structure

```
1  #include <mpi.h>
2
3  int main(int argc, char *argv[])
4  {
5      // Serial code
6
7      MPI_Init(&argc, &argv);
8
9      // Parallel code
10
11     MPI_Finalize();
12
13     // Serial code;
14 }
```

ENS DE LYON

Rank and number of processes

Getting the number of processes:

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

Getting the rank of a process:

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

ENS DE LYON

Rank and number of processes

Getting the number of processes:

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

Getting the rank of a process:

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

For now: `comm` will always be `MPI_COMM_WORLD`

ENS DE LYON

Hello World

Recap of basic MPI

```
#include <mpi.h>
int MPI_Init(int argc, char**argv);
int MPI_Finalize();
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
MPI_Comm MPI_COMM_WORLD;
```

Exercise

Write a program such that each process prints:

```
Hello from process <rank>/<number>
```

Test it. In what order do they print ?

ENS DE LYON

Point-to-point communication

ENS DE LYON

Introduction

Communication between two identified processes: a sender and a receiver.

- a process performs a sending operation
- the other process performs a matching receive operation

There are different types of send and receive routines used for different purposes.

- Synchronous send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Combined send/receive

ENS DE LYON

Introduction

Communication between two identified processes: a sender and a receiver.

- a process performs a sending operation
- the other process performs a matching receive operation

There are different types of send and receive routines used for different purposes.

- Synchronous send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Combined send/receive

ENS DE LYON

Sending data (blocking asynchronous send)

```
int MPI_Send(const void*  data,
             int          count,
             MPI_Datatype datatype,
             int          destination,
             int          tag,
             MPI_Comm     communicator);
```

- data : adress space of the process that send the data
- count : number of data elements of a particular type to be sent
- datatype : type of the data, such as `MPI_CHAR`, `MPI_INT`,...
- destination : the rank of the receiving process
- tag : identify a message (for us : use 0 most of the time)
- communicator : use `MPI_COMM_WORLD`

ENS DE LYON

Examples

```
int MPI_Send(const void*  data,
             int          count,
             MPI_Datatype datatype,
             int          destination,
             int          tag,
             MPI_Comm     communicator);
```

Examples:

- Send your rank to the process number 0:

```
MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

- Send a float array `A` of size `n` to the process number 1:

```
MPI_Send(A, n, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
```

ENS DE LYON

Receiving data (blocking asynchronous receive)

```
int MPI_Recv(void*          data,
             int            count,
             MPI_Datatype   datatype,
             int            source,
             int            tag,
             MPI_Comm       communicator,
             MPI_Status*    status);
```

- data, count, datatype, communicator : as for `MPI_Send`
- source : rank of the originating process (`MPI_ANY_SOURCE`)
- tag : identifier of the message you are waiting (`MPI_ANY_TAG`)
- status : a predefined structure `MPI_Status` that contains some information about the received message

ENS DE LYON

Example

```
int MPI_Recv(void*         data,
             int           count,
             MPI_Datatype  datatype,
             int           source,
             int           tag,
             MPI_Comm      communicator,
             MPI_Status*   status);
```

Example:

- Receive an integer array of size `n` from process number `0` and store it into a buffer:

```
MPI_Recv(buffer, n, MPI_INT, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

ENS DE LYON

Exchange data

Recap of basic MPI

```
int MPI_Recv(void* data, int n, MPI_Datatype t,
             int src, int tag, MPI_Comm comm, MPI_Status* s);
int MPI_Send(const void* data, int n, MPI_Datatype t,
             int dest, int tag, MPI_Comm comm);

MPI_Datatype MPI_INT; MPI_Status MPI_STATUS_IGNORE;
int MPI_ANY_SOURCE; int MPI_ANY_TAG;
```

Exercise

- Let each process generate a random number and print "<process rank> : <random value>".
- Have each process receive from the previous process the sum of the random values of the previous processes (i.e process 0 sends to process 1 it's random value, process 1 sends the sum of the value received by process 0 and its random value, ...)
- The last process prints the total sum

Remark : There is a simpler more efficient way to do this in MPI

Other functions

- `MPI_Ssend` : Synchronous blocking send
- `MPI_Isend` : Asynchronous non-blocking send
- `MPI_Irecv` : Asynchronous non-blocking receive
- `MPI_Sendrecv` : Simultaneous send and receive
- `MPI_Wait` : Blocks until a specified non-blocking send or receive operation has completed
- `MPI_Probe` : Performs a blocking test for a message
- `MPI_Get_count` : Returns the source, tag and number of elements of datatype received
- ...

ENS DE LYON

Collective communications

ENS DE LYON

Introduction

Types of Collective Operations:

- Synchronization
- Data Movement (eg. broadcast, scatter, gather)
- Collective Computation (eg. reduce)

ALL processes within a communicator **must** participate in any collective operation (programmer's responsibility).

ENS DE LYON

Synchronization

```
int MPI_Barrier(MPI_Comm communicator);
```

Blocks until all process have reached this routine.

Synchronization

```
int MPI_Barrier(MPI_Comm communicator);
```

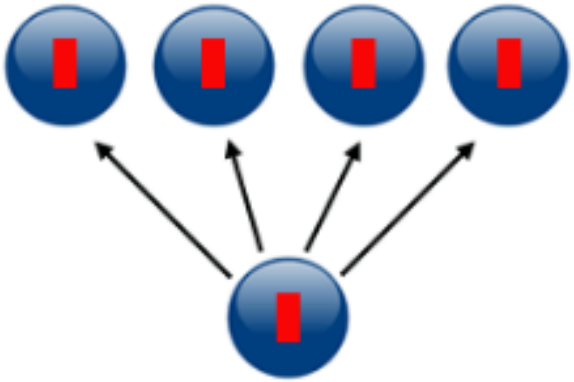
Blocks until all process have reached this routine.

Exercise

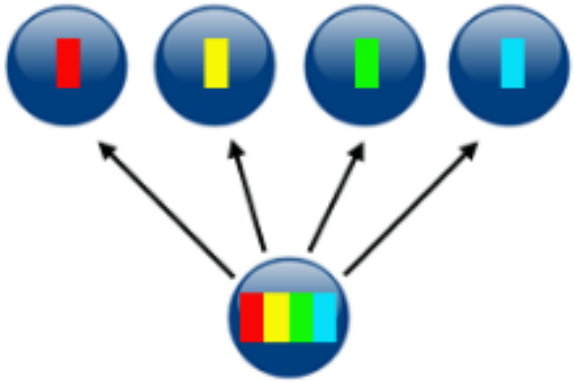
Write a program that produces such an output when executed with 4 processes (i.e. they each print "step 1" and "step 2", but each "step 2" is after the "step 1"of all processes) :

```
[0] step 1
[3] step 1
[2] step 1
[1] step 1
[3] step 2
[0] step 2
[2] step 2
[1] step 2
```

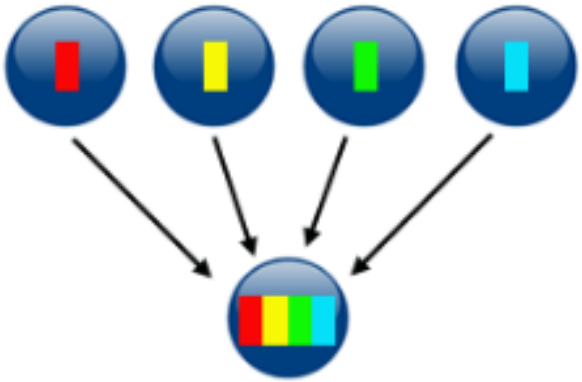
Collective operations



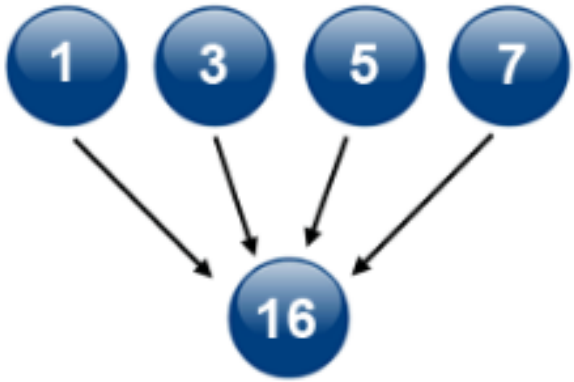
broadcast



scatter



gather



reduction

ENS DE LYON

Broadcast

```
int MPI_Bcast(void*         data,
              int           count,
              MPI_Datatype  datatype,
              int           root,
              MPI_Comm      communicator);
```

Broadcasts a message from the process with rank `root` to all other processes of the group.

ENS DE LYON

Scatter

```
int MPI_Scatter(const void*  sendbuf,
                int          sendcount,
                MPI_Datatype sendtype,
                void*        recvbuf,
                int          recvcount,
                MPI_Datatype recvtype,
                int          root,
                MPI_Comm     communicator);
```

Distributes distinct messages from a single source task to each task in the group.

ENS DE LYON

Gather

```
int MPI_Gather(const void*   sendbuf,
               int           sendcount,
               MPI_Datatype  sendtype,
               void*         recvbuf,
               int           recvcount,
               MPI_Datatype  recvtype,
               int           root,
               MPI_Comm      communicator);
```

Gathers distinct messages from each task in the group to a single destination task.

ENS DE LYON

Reduce

```
int MPI_Reduce(const void*   sendbuf,
               void*         recvbuf,
               int           count,
               MPI_Datatype  datatype,
               MPI_Op        operator,
               int           root,
               MPI_Comm      communicator);
```

Applies a reduction operation on all tasks in the group and places the result in one task.

Operators: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`, ...

ENS DE LYON

Other functions

- `MPI_Allgather` : Each task in the group, in effect, performs a one-to-all broadcasting operation within the group
- `MPI_Allreduce` : Applies a reduction operation and places the result in all tasks in the group
- `MPI_Reduce_scatter` : Same but split the result
- `MPI_Alltoall` : Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group
- ...

ENS DE LYON

To go further

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

ENS DE LYON