Introduction to MPI

Carlos E. Alvarez[1].

[1]Dep. de Matemáticas aplicadas y Ciencias de la Computación, Universidad del Rosario

June 2019

Universidad del Rosario

MACC
Matemáticas Aplicadas y
Ciencias de la Computación

MPI: Message Passing Interface

- Library specification (Bindings for C and Fortran)

MPI: Message Passing Interface

- Library specification (Bindings for C and Fortran)
- Scalable: Handles multiple machines

MPI: Message Passing Interface

- Library specification (Bindings for C and Fortran)
- Scalable: Handles multiple machines
- Portable: Handles Big-endian and Little-endian architectures

MPI: Message Passing Interface

- Library specification (Bindings for C and Fortran)
- Scalable: Handles multiple machines
- Portable: Handles Big-endian and Little-endian architectures
- Efficient: Optimized communication algorithms

MPI: Message Passing Interface

What it does:

What it does not do:

MPI: Message Passing Interface

What it does:

- Independent of low level communication interface

What it does not do:

MPI: Message Passing Interface

What it does:

- Independent of low level communication interface
- Optimized communication

What it does not do:

MPI: Message Passing Interface

What it does:

- Independent of low level communication interface
- Optimized communication
- Allow communication / computation overlap

What it does not do:

MPI: Message Passing Interface

What it does:

- Independent of low level communication interface
- Optimized communication
- Allow communication / computation overlap

What it does not do:

- Gain performance for free (communication cost)

MPI: Message Passing Interface

What it does:

- Independent of low level communication interface
- Optimized communication
- Allow communication / computation overlap

What it does not do:

- Gain performance for free (communication cost)
- Application must be adapted

MPI: Message Passing Interface

What it does:

- Independent of low level communication interface
- Optimized communication
- Allow communication / computation overlap

What it does not do:

- Gain performance for free (communication cost)
- Application must be adapted
- Manage multi-threading

Initializing MPI

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

First MPI function to be called.
By default the parameters can be set to `NULL`.

Exiting MPI

```
int MPI_Finalize()
```

Last MPI function to be called.

Communicator

A communicator defines the group of processes that participate in a communication and assigns a `rank` to each of them.

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

The number of processes in the group is obtained with:

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

Compiling and running MPI programs

Compile:

```
mpic++ -o mpi_executable mpi_source
```

Execute:

```
mpirun -np no_processes mpi_executable
```

Compiling and running MPI programs

Submit in slurm (sbatch command):

run_mpi.sh

```
#!/bin/bash

#SBATCH -J test             # Job name
#SBATCH -o job.%j.out       # Output file
#SBATCH -N 3                # No. of nodes requested
#SBATCH -n 96               # No. of mpi tasks requested
#SBATCH -t 00:30:00         # Run time (hh:mm:ss)

# Launch MPI-based executable

prog=$1
arg1=$2

prun ./${prog} ${arg1}
```

Example:

hello_world_temp.cpp

Machine name

```
MPI_Get_processor_name(char* name, int* len);
```

Gets the name of the machine running the process.

Example:

machine_name_temp.cpp

MPI basic datatypes

Compatibility between different machines.

Some examples:

C	**MPI**
char	MPI_CHAR
short	MPI_SHORT
int	MPI_INT
float	MPI_FLOAT
double	MPI_DOUBLE

Messages

Sending a message:

Messages

Sending a message:

- Where is the data to be sent?

Messages

Sending a message:

- Where is the data to be sent?
- How much data is going to be sent?

Messages

Sending a message:

- Where is the data to be sent?
- How much data is going to be sent?
- Who will receive the data?

Messages

Sending a message:

- Where is the data to be sent?
- How much data is going to be sent?
- Who will receive the data?
- Some id for the message

Messages

Receiving a message:

Messages

Receiving a message:

- Where to store the data received?

Messages

Receiving a message:

- Where to store the data received?
- How much data is going to be received?

Messages

Receiving a message:

- Where to store the data received?
- How much data is going to be received?
- Who will send the data?

Messages

Receiving a message:

- Where to store the data received?
- How much data is going to be received?
- Who will send the data?
- Some id for the message

Messages

Receiving a message:

- Where to store the data received?
- How much data is going to be received?
- Who will send the data?
- Some id for the message
- How much data was ACTUALLY received?

Point-to-point communication

Standard Send

```
MPI_Send(void* data, int count, MPI_Datatype datatype,
         int dest, int tag, MPI_Comm comm)
```

Synchronous Send

```
MPI_Ssend(void* data, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm)
```

Point-to-point communication

Receive

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

Point-to-point communication

The `status` structure holds information about the actual tag, source and size of the message received.

- `status.MPI_SOURCE`
- `status.MPI_TAG`

Count of the data is obtained from

```
MPI_Get_count(MPI_Status* status, MPI_Datatype datatype,
              int* count)
```

If there is no interest in the status `MPI_STATUS_IGNORE` can be used.

Point-to-point communication

Wildcards:

- To receive from any source use source `MPI_ANY_SOURCE`
- To receive with any tag use tag `MPI_ANY_TAG`

Point-to-point communication

For a communication to succeed:

Point-to-point communication

For a communication to succeed:

- Sender must specify a valid destination rank

Point-to-point communication

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank

Point-to-point communication

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

Point-to-point communication

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

Point-to-point communication

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
- Receiver's buffer must be large enough

Communicator and tag

Need to match them to carry out a successful communication.

- **Communicator:** Heavy object. Defines the group of processes that participate in collective communications (scope)
- **Tag:** Arbitrary integer that identifies a point-to-point message. The tag placed by the sender must match the one expected by the receiver. Useful when two processes communicate several different messages

Example:

send_receive_temp.cpp

Message order preservation

- Messages arrive in the same order that they are sent
- This can cause problems with the synchronous send, if we receive the message in a different order from the one used by the sender

Example:

syncsend_ok_temp.cpp
syncsend_notok_temp.cpp

Exercise:

Look at the exercise: exercise1.pdf

OB

Collective Communications

Collective communications

- Point-to-point communications involve pairs of processes
- Collective communications allow more than two processes to participate

Collective communications

- All processes in a communicator participate
- Operations are blocking
- No tags are used

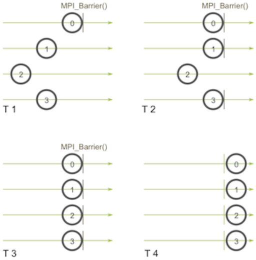
Collective communications

Examples of collective transfers

- Barrier
- Broadcast, scatter
- Gather
- Reduction

Barrier

Processes stop their execution until all get to the barrier



Barrier

Synchronize all processes.

Barrier

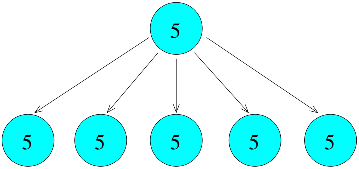
```
int MPI_Barrier(MPI_Comm comm)
```

Example:

barrier_temp.cpp

Broadcast

One process sends some data to all other processes.



Broadcast

Copy data to all.

Broadcast

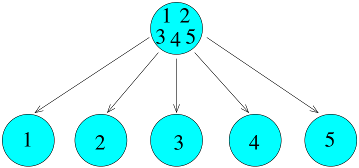
```
int MPI_Bcast(void *sendbuf, int count,
              MPI_Datatype datatype,
              int source, MPI_Comm comm)
```

Example:

broadcast_temp.cpp

Scatter

One process partitions the data and sends a part to each process.



Scatter

Distribute the data among all.

Scatter

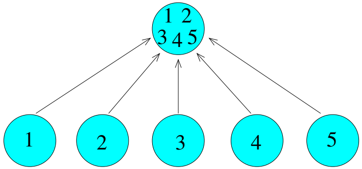
```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype,
                int source, MPI_Comm comm)
```

Example:

scatter_temp.cpp

Gather

One process collects data from each process.



Gather

Collect from all in rank order.

Gather

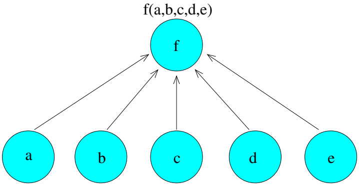
```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int dest, MPI_Comm comm)
```

Example:

gather_temp.cpp

Reduce

One process collects data from each process and performs an operation.



Reduce

Some available operations:

- MPI_MAX (Maximum)
- MPI_MIN (Minimum)
- MPI_SUM (Sum)
- MPI_PROD (Product)
- MPI_LAND (Logical AND)
- MPI_BAND (Bit level AND)
- MPI_LOR (Logical OR)
- MPI_BOR (Bit level OR)

Reduce

Perform an operation on the gathered data.

Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int dest, MPI_Comm comm)
```

If `count` is > 1, the operation will be carried out among the elements with the same index in the array, and the result will be stored in the `recvbuf` array.

Example:

reduce_sum_temp.cpp

Allreduce

One process collects data from each process, performs an operation and distributes the result back to all processes.

Reduce

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

Exercise:

Look at the exercise: exercise2.pdf