Introduction to MPI

B. Pajot

benjamin.pajot@atos.net

Copyright ©Bull S.A.S. 2016

17-10-2016
Introduction to MPI

Credits

http://www.mpi-forum.org/
https://computing.llnl.gov/tutorials/mpi/#What
http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html

Programming Model training from R. Dolbeau & G.-E. Moulard (Atos)
Formations CED – Du calcul parallèle au massivement parallèle
Plan

▶ Introduction
▶ Environment
▶ Data type
▶ Point-to-point communications
▶ Collective communications
▶ Questions
Introduction

1. Reminders about parallelism
2. Parallel programming model

17-10-2016
1. Goals of Parallel Programming

- **Parallelizing?**
  - « Reorganizing » the problem to process simultaneously data and computations while using a number of computing resources.

- **Why?**
  - Improve **performance** => computing faster
  - Process a **bigger volume of data** => using memory of several computing nodes

- **Important points:**
  - Knowing hardware **architecture**
  - Choosing a **programming model**: MPI, OpenMP, hybrid programming...
1. Architectures

- **Shared memory computer**
  - Several processors sharing the same global memory space via a fast interconnect

- **Distributed memory computer**
  - Each node with its own memory
  - Each node reaches other nodes memory via the network (call to communications routines)

- **Hybrid computer**
  - Most common case: a set of shared memory computers (eventually equipped with coprocessors or accelerators) linked by a network
1. What Matters

- From developer point of view, architecture = network of processors
  - CPU => **computing power**. Determine the FLOPS (FIoating-point Operations Per Second).
  - Several levels of memory => several levels of parallelism. Critical points: **size of memories** & IO speed.
  - Communication network => limiting factor: **bandwidth**.

- **Parallel programming**: use of a software layer to handle resources processing and access => **MPI**
2. MPI: Message Passing Interface

▶ High level API for message passing

▶ Designed for **Performance, scalability and portability**

▶ Currently, it’s the third major release:
  - 1995: v1.2  (MPI-1)
  - 1997: v2.0  (MPI-2)
  - 2008: v2.1
  - 2009: v2.2
  - 2012: v3.0  (**MPI-3**)  
  - 2015: v3.1

▶ An API with different implementations
  - Some with specific extensions...
  - ... which can break the portability of an application
2. MPI Implementations

- Open source implementation
  - MPICH (MPI-1)
  - MPICH2 (MPI-2)
  - OpenMPI (1.8.3 includes most of MPI-3)
  - LAM/MPI

- Manufacturer implementation’s
  - HP MPI
  - Intel MPI
  - BullxMPI
2. Sequential Programming Model

- The program is executed by a single process.

- This process runs on a single physical processor of the machine.

- All data (Variables and Constants) are allocated in the memory assigned for the process.

- Does not allow to exploit modern machines with several physical processors distributed on several node
  - Limited in term of computing power
  - Limited in term of problem size (1 node)
2. Programming Model by Message Passing

- A program is divided into sub-programs each executed by a process.

- The processes communicate by the interconnection network by sending or receiving messages
  - Possibility to exploit whole platform.
2. Key Notions

Key notions: process, message, synchronization

- “Virtual” process ≠ processor / physical core. Processes can execute on different or identical processors / cores.
- Each process has its own variables and does not access directly to the variables of other processes.
- Data sharing between processes is done by explicit send & receives of messages.
- Processes synchronization.
2. Programmation MPI

► In an MPI program each process runs a sub-program
  – Written in a classical language (C, C++, Fortran, Python, ...)
  – Can be different depending of the process
  – Most often, the same sub-program for all process (SPMD – Single Program Multiple Data)
    • Not required by the model, MPMD (Multiple Program Multiple Data) is also possible

► Variables of each sub-program:
  – Can have the **same name** (SPMD)
  – **Different memory locations and different values** (Distributed memory)
  – **Private to the sub-program**

► The sub-programs use routines for **sending and receiving messages** to communicate
2. Work and Data Distribution in MPI

- Create a system of N independent processes

- Each process has a unique ID: \textbf{Rank [0:N-1]}

- Data and work distribution is based on \textit{rank}

```fortran
program fortran_spmd

    implicit none

    integer :: process_rank
    integer :: var1
    real :: var2

    !! CODE (later in this pres.)
    if (process_rank == 0) call inst1()
    if (process_rank /= 0) then
        call instr2()
        call instr3()
    endif

    !! CODE (later in this pres.)
    end program fortran_spmd
```

Network
2. Messages

- Messages are blocks of data exchanged by sub-programs.

- For message sending and receiving, different information are required
  - The rank of sender/receiver
  - data location
  - data type
  - data size

![Diagram of sub-programs communicating through a network.](Image)
2. A First Simple Example

- **P0:** data read
- **P1:** data read
- **Communication via network**
- **Compute partial sum**
- **Store in variable SUM**
- **Display or write in a file**

SUM0 = SUM0 + SUM
Bases

1. Environment
2. Communicator
3. Environmental Management

17-10-2016
1. Generality

➢ Compilation unit containing MPI routines must:

- C/C++: include mpi header file "mpi.h"

- Fortran: use "MPI" module
  • Introduced in MPI-2, else use "mpi.h"

➢ The prefix "MPI_" is reserved for MPI routines and macros.

➢ MPI routines in C/C++ & Fortran have the same format:
  - “MPI” prefix and the first letter are in capital letter
    • MPI_Xxxx_xxx_xxx()
1. Initialization of MPI Program

- In MPI program, the function **MPI_Init** must be the first one called by each sub-program

  - C/C++:
    ```c
    int MPI_Init(int *argc, char **argv)
    ```

  - Fortran:
    ```fortran
    MPI_Init(mpierror)
    ```

- Initializes the **environment** of MPI execution (communicator ...)
1. Termination of a MPI Program

- Each sub-program must call `MPI_Finalize` before the end of the program
  - C/C++:
    ```c
    int MPI_Finalize()
    ```
  - Fortran:
    ```fortran
    MPI_Finalize(mpierror)
    ```

- In case where it’s necessary to stop the program before the normal end, use `MPI_Absort` function:
  - For example if memory allocation required by a process fails
2. Communicators

- A **communicator** is composed of a MPI process group.

- At the initialization of MPI program, a communicator with all MPI processes is created: **MPI_COMM_WORLD**
  - this is a global communicator.

- Each MPI process is identified by its **rank** within a communicator:
  - identifier between 0 and (number of processes in the communicator – 1)

- A process can belong to several communicators and has an associated rank (identifier) for each of these communicators

- Two MPI processes must be in the same communicator to be able to communicate together.
2. Communicator: Size and Rank

▶ **MPI_Comm_size** function provides the number of MPI processes in the communicator

- C/C++: `int MPI_Comm_size(MPI_Comm comm, int *size)`
- Fortran: `MPI_Comm_size(comm, size, mpierror)`

▶ **MPI_Comm_rank** function provides the rank of the process in the communicator:

- C/C++: `int MPI_Comm_rank(MPI_Comm comm, int*rank)`
- Fortran: `MPI_Comm_rank(comm, rank, mpierror)`
#include <mpi.h>

int main(int argc, char *argv[])
{
    /* The basic MPI Program */
    int mpierror, mpisize, mpirank;

    mpierror=MPI_Init(&argc, &argv);
    mpierror=MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    mpierror=MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);

    /* Do work here */

    mpierror=MPI_Finalize();

    return 0;
}
2. Basic Example in Fortran

```fortran
program firstmpi
! The basic MPI Program
use MPI

integer :: mpierror, mpisize, mpirank

call MPI_Init(mpierror)

call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)
call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)

! Do work here

call MPI_Finalize(mpierror)

end program firstmpi
```
2. **TP: MPI101**

- Execute the TP (in C)
  - Batch environment "SLURM"
  - `./build.sh, ./run.sh`

- Modify number of processes

```bash
srun --mpi=pmi2 -J TPMPI -N 1 -n 2 -p E52697v2 ./mpi101
```

**Compilation:**
- `icc -c mpi101.c`
- `icc -lmpi mpi101.o -o mpi101`
- `mpiicc -o mpi101`

---

**Partition Name (group of machine)**

**Job Name**

**Number of processes**

**Number of node asked**

**enables interaction between Slurm and MPI**
3. MPI Environmental Management

- **MPI_Get_processor_name**
  - Gets the processor name; format is implementation dependent

- **MPI_Get_version**
  - Gets version and sub-version of MPI

- **MPI_Initiated**
  - Gets if MPI_Init was called; for example, useful for libraries

- **MPI_Wtime**
  - Gets the time in seconds since an arbitrary point in the past
    - if MPI_WTIME_IS_GLOBAL is true (1), the value is synchronized for all processes

- **MPI_Wtick**
  - Gets the precision in second of MPI_Wtime
3. **TP: MPI101 phase 2**

- Display the name of the machine using the MPI function
- Modify the number of nodes to run the program

**MPI_Get_processor_name**
Gets the name of the processor

**Synopsis**

```
int MPI_Get_processor_name( char *name, int *resultlen )
```

**Output Parameters**

- **name** A unique specifier for the actual (as opposed to virtual) node. This must be an array of size at least MPI_MAX_PROCESSOR_NAME.
- **resultlen** Length (in characters) of the name
3. **TP: MPI101 phase 2 (solution)**

```c
#include <mpi.h>
int main(int argc, char *argv[])
{
    /* The basic MPI Program */
    int mpierror, mpisize, mpirank;
    mpierror=MPI_Init(&argc, &argv);
    mpierror=MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    mpierror=MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    /* Do work here */
    {
        char temp[MPI_MAX_PROCESSOR_NAME];
        int resultlen;
        int  r = MPI_Get_processor_name(temp, &resultlen);
        printf("I am %d out of %d running on %s\\n", mpirank, mpisize, temp);
    }
    mpierror=MPI_Finalize();
    return 0;
}
```
Data Type

17-10-2016
Data Type

▶ For portability reasons, MPI predefined **elementary data types**.

▶ Using elementary data types to build more complex types (**derived data types**)
  – Not tackled in this training

▶ MPI implementations can provide more elementary data types:
  – These types are in "mpi.h" header file.
  – They can prevent portability
# Data Type

- Predefined data types in Fortran:

<table>
<thead>
<tr>
<th>TYPE MPI</th>
<th>TYPE FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
### Data Type

#### Predefined data types in C:

<table>
<thead>
<tr>
<th>TYPE MPI</th>
<th>TYPE C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char (treated as printable character)</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char (treated as integral value)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char (treated as integral value)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>wchar_t (treated as printable character)</td>
</tr>
</tbody>
</table>
### Data Type

#### Predefined data types in C (end):

<table>
<thead>
<tr>
<th>TYPE MPI</th>
<th>TYPE C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_C_BOOL</td>
<td>_Bool</td>
</tr>
<tr>
<td>MPI_INT8_T</td>
<td>int8_t</td>
</tr>
<tr>
<td>MPI_INT16_T</td>
<td>int16_t</td>
</tr>
<tr>
<td>MPI_INT32_T</td>
<td>int32_t</td>
</tr>
<tr>
<td>MPI_INT64_T</td>
<td>int64_t</td>
</tr>
<tr>
<td>MPI_UINT8_T</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MPI_UINT16_T</td>
<td>uint16_t</td>
</tr>
<tr>
<td>MPI_UINT32_T</td>
<td>uint32_t</td>
</tr>
<tr>
<td>MPI_UINT64_T</td>
<td>uint64_t</td>
</tr>
<tr>
<td>MPI_C_COMPLEX</td>
<td>Float _Complex</td>
</tr>
<tr>
<td>MPI_C_FLOAT_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>MPI_C_DOUBLE_COMPLEX</td>
<td>double _Complex</td>
</tr>
<tr>
<td>MPI_C_LONG_DOUBLE_COMPLEX</td>
<td>long double _Complex</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Point to Point Communications

1. Description
2. Contents
3. Execution
4. Optimizations

17-10-2016
1. Principle

- A point to point communication is a communication between **two** MPI processes.

- One process is the **sender**: it sends the message.

- The other is the **receiver** or recipient: it waits the message of the senders.

- The sender and the receiver are identified by their rank.
1. Blocking Communications

- A **blocking send** blocks the process until the memory space used for the message can be re-written without any modification of the message.

- A blocking send can be **synchronous**
  - It waits for an acknowledgment of receipt.

- A blocking send can be **asynchronous**
  - If system memory space is used to send the message.

- A **blocking reception** blocks the process until data are received and ready to be used by the system.
1. Non-Blocking Communication

- **Non-blocking send and reception** return almost immediately:
  - no wait for communication event (copy in the system memory space or acknowledgement of receipt)

- They aim to **overlap** communications time and computing time.

- User can not know the exact moment when a sending or a receiving has effectively been done.

- **Synchronization routines** help ensure the sending or receiving of message.

- It is unsafe to modify the memory space used for data sending or receiving without being ensured of the end of a sending or receiving.
1. Modes

- **4 modes of send** for point to point communications:
  - Standard (MPI implementation dependant)
  - Buffered (copy in a buffer; the send is done later, asynchronously; no need to wait receiving) => should probably give better results, but requires copy in memory
  - Synchronous (with receiving; the program takes back the hand when the send is complete)
  - Ready (started only if the matching receive is already posted)

- Each mode has blocking and non-blocking implementation:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Send</strong></td>
<td>Standard</td>
<td>MPI_Send</td>
</tr>
<tr>
<td></td>
<td>Buffered</td>
<td>MPI_Bsend</td>
</tr>
<tr>
<td></td>
<td>Synchronous</td>
<td>MPI_Ssend</td>
</tr>
<tr>
<td></td>
<td>Ready</td>
<td>MPI_Rsend</td>
</tr>
<tr>
<td><strong>Receive</strong></td>
<td></td>
<td>MPI_Recv</td>
</tr>
</tbody>
</table>
2. Parameters (1/2)

Parameters of point to point communication:

- **buffer**: memory address of the data or reception buffer

- **count**: number of elements in send buffer or the maximum of elements to receive

- **type**: data type

- **comm**: communicator used

- **dest**: rank of destination

- **source**: rank of source
  - use `MPI_ANY_SOURCE` to receive a message from any source
2. Parameters (2/2)

- **tag**: a nonzero integer given by the programmer to identify a message
  - use `MPI_ANY_TAG` to receive a message without knowing the tag.

- **request**: used to associate non-blocking communication operations (Isend and Irecv) with an (MPI_request type) object used for synchronization.

- **status**: a variable containing additional information about the receive operation after it completes
  - `MPI_Status` type
  - contains the rank of the sender, the tag of the message, the error of the message.
  - Instead, we have to find out the length of the message with `MPI_Get_count`
  - which can be ignored with `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE`
2. Message Envelope & Body

- The envelope (description of the “context”) of message contains:
  - Identities of the sender and the receiver (ranks)
  - The tag
  - The communicator

- A receiving operation works with a send operation only if the envelopes match.

- Communication with the fictive process of rank \texttt{MPI_PROC_NULL} has no effect.

- The body of the message contains:
  - A buffer with data inside
  - The type of the data
  - Their size
2. Send Operations

- Standard blocking send
  
  ```c
  int MPI_Send(buffer, count, type, dest, tag, comm, status)
  ```

- Synchronous blocking send:
  
  ```c
  int MPI_Ssend(buffer, count, type, dest, tag, comm)
  ```

- Standard non-blocking send:
  
  ```c
  int MPI_Isend(buffer, count, type, dest, tag, comm, request)
  ```
2. Reception Operations

- **Blocking reception**:
  
  ```
  int MPI_Recv(buffer, count, type, source, tag, comm, status)
  ```
  
  - return only when `buffer` contains the message

- **Non-blocking reception**:
  
  ```
  int MPI_Irecv(buffer, count, type, source, tag, comm, request)
  ```
  
  - return immediately
2. A Simple Example

```fortran
program basic_sendrecv
  implicit none
  use MPI
  integer :: source, dest, tag, error, buffer, nb_elements
  integer(MPI_STATUS_SIZE) :: status
  source = 0
  dest = 1
  tag = 21
  nb_elements = 1

  ! MPI INITIALIZATION
  if (rank_process == source) then
    buffer = 12
    call MPI_SEND(buffer, nb_elements, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, error)
    print *, "I am process of rank ", rank_process, " , buffer = ", buffer
  else if (rank_process == dest) then
    buffer = 0
    print *, "I am process of rank ", rank_process, " , buffer = ", buffer
    call MPI_RECV(buffer , nb_elements, MPI_INTEGER, source, tag, MPI_COMM_WORLD, status, error)
    print *, "I am process of rank ", rank_process, " , buffer = ", buffer
  endif

  ! MPI FINALIZE

end program basic_sendrecv
```

>> I am process of rank 1, buffer = 0
>> I am process of rank 0, buffer = 12
>> I am process of rank 1, buffer = 12
3. Executing Features

- During a point to point communication, the send & the receive are 2 different operations, eventually asynchronous, done by 2 different processes.

- It raises questions:
  - What happens if no reception corresponds to the send?
  - Can we use variables send/received without impact on the message?
  - How to take back hand and do something else during the send or receive of the message?
  - …

- One important notion to be defined: completion.
3. Completion

- Of the reception: the message is arrived and the variable copied in local memory and can be used by the processor receiver.

- Of the send: the send variable (i.e. the matching memory zone) can be used safely, on read or write, in the sense that a modification of this variable by the processor sending will no more impact on the reception of the other processor.

=> Completion ≈ variables send/received can be used without risk

- A non-blocking send does not guarantee completion!
3. Sending Process

1. Beginning of the communication (posting of the send)
2. Then 2 situations possible:
   – Copy in a buffer. The send will be done later, asynchroneously. No need to wait the reception.
   – Synchronisation with reception: the routine is waiting that the process receiving is ready and that the transfer has begun. Program takes back control only when the send is complete.

» « Bufferization » will probably give better performance but requires a copy in memory (sending or receiving side, amongst the implementation). To note that the size of the buffers is necessarily limited.
3. Receiving Process

1. Initialization of the reception (**posting**): verification in the stack of the waiting messages if an envelop corresponds to the one asked by the SEND_RECV.

2. Transfer of data received in the memory zone dedicated by the SEND_RECV.

**Warning**: data type is not checked by the SEND_RECV.

**Warning 2**: In « standard » mode, using MPI_SEND and MPI_RECV is safe from the data access point of view, nevertheless a bad management in the code of the synchronization and the blocking behaviour may lead to **deadlocks**.
3. A First Example of Deadlock

```c
if (rank_process == 0) then
    ! Reception of 10 MPI_REAL in data coming from 1 with tag "tag1"
    call MPI_RECV(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, error)
    ! Send of 10 MPI_REAL of data2 towards 1 with the tag "tag2"
    call MPI_SEND(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, error)
else if (rank_process == 1) then
    ! Reception of 10 MPI_REAL in data coming from 0 with tag "tag2"
    call MPI_RECV(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, error)
    ! Send of 10 MPI_REAL of data2 towards 0 with the tag "tag1"
    call MPI_SEND(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, error)
endif
```

Waiting message from 1

![Deadlock diagram]

Waiting message from 0

Time
3. A Second Example of Deadlock

```fortran
if (rank_process == 0) then
  ! Send of 10 MPI_REAL of data2 towards 1 with the tag "tag2"
  call MPI_SEND(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, error)
  ! Reception of 10 MPI_REAL in data coming from 1 with tag "tag1"
  call MPI_RECV(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, error)
else if (rank_process == 1) then
  ! Send of 10 MPI_REAL of data2 towards 0 with the tag "tag1"
  call MPI_SEND(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, error)
  ! Reception of 10 MPI_REAL in data coming from 0 with tag "tag2"
  call MPI_RECV(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, error)
endif
```

"Bufferized" message send towards 1
"Bufferized" message send towards 0

0
Buffer
data2
Buffer
copy of data2

1
Buffer
data2
copy of data2

Time
3. A Second Example of Deadlock

```plaintext
if (rank_process == 0) then
    ! Send of 10 MPI_REAL of data2 towards 1 with the tag "tag2"
    call MPI_SEND(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, error)
    ! Reception of 10 MPI_REAL in data coming from 1 with tag "tag1"
    call MPI_RECV(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, error)
else if (rank_process == 1) then
    ! Send of 10 MPI_REAL of data2 towards 0 with the tag "tag1"
    call MPI_SEND(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, error)
    ! Reception of 10 MPI_REAL in data coming from 0 with tag "tag2"
    call MPI_RECV(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, error)
endif
```

Diagram:
- Node 0:
  - "Bufferized" message send towards 1
  - Reception of message 1
  - OK!

- Node 1:
  - "Bufferized" message send towards 0
  - Reception of message 0
  - OK!

Buffered data2 copy of data2 Buffer data2 copy of data2

Time
3. A Second Example of Deadlock

```plaintext
if (rank_process == 0) then
  ! Send of 10 MPI_REAL of data2 towards 1 with the tag "tag2"
  call MPI_SEND(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, error)
  ! Reception of 10 MPI_REAL in data coming from 1 with tag "tag1"
  call MPI_RECV(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, error)
else if (rank_process == 1) then
  ! Send of 10 MPI_REAL of data2 towards 0 with the tag "tag1"
  call MPI_SEND(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, error)
  ! Reception of 10 MPI_REAL in data coming from 0 with tag "tag2"
  call MPI_RECV(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, error)
endif
```

Synchronous message send towards 1
Synchronous message send towards 0

Time
3. A Second Example of Deadlock

```c
if (rank_process == 0) then
    ! Send of 10 MPI_REAL of data2 towards 1 with the tag "tag2"
    call MPI_SEND(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, error)
    ! Reception of 10 MPI_REAL in data coming from 1 with tag "tag1"
    call MPI_RECV(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, error)
else if (rank_process == 1) then
    ! Send of 10 MPI_REAL of data2 towards 0 with the tag "tag1"
    call MPI_SEND(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, error)
    ! Reception of 10 MPI_REAL in data coming from 0 with tag "tag2"
    call MPI_RECV(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, error)
endif
```

Synchronous message send towards 1
Synchronous message send towards 0
Waiting the reception of 1...
Waiting the reception of 0...
Deadlock!  Deadlock!

Time
3. Complements

- It exists routines allowing to perform a send and a receive at one time:

  ```
  MPI_SENDRECV(data_send, nb_elem_send, type_send, dest, tag_send,
                 data_recv, nb_elem_recv, type_recv, src, tag_recv,
                 comm, status, error)
  ```

  ```
  MPI_SENDRECV_REPLACE(data, nb_elem, type, dest, tag_send,
                        src, tag_recv, comm, status, error)
  ```

- Warning: in the first case, data_send & data_recv must be different. In the second case, the variable send is replaced by the one received.

- ```
  int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
  ```

  envelop.
4. Performance of a MPI Computation (1/2)

► What are decisive factors?
- System architecture and network between cores and nodes.
- MPI implementation.
- The code: choice of algorithms, memory management, communication/computing ratio in the code, load balancing...

► Time sharing during the execution of a MPI program
- Latency: time to begin an exchange \( \approx \) time needed to send an empty message
- Communications
- Computations

Example:
4. Performance of a MPI Computation (2/2)

How to improve the implementation?

- Use the good algorithms...
- Use specialized libraries (fftw, scalapack...).
- Overlap communications with computations.
- Change communication mode.
- Balance load between different processes.

Example:
4. Non-blocking Communications (1/2)

- How? To not wait the completion to give back control.

- Non-blocking send (MPI_ISEND): as soon as the message is posted, the program takes back control over the processor source.

- Non-blocking receive (MPI_IRecv): as soon as the reception is posted, the program takes back the hand.

=> The program can do something else during data transfers: overlap communications with computations.

Warning: The program takes back control before that the reception or the send is complete. As a consequence the variable send/received is not usable immediately.
4. Non-blocking Communications (2/2)

How to know if the reception or the send is finished? To not wait the completion to give back control?

MPI_WAIT

```call MPI_ISEND(data, nb_elements, type, dest, tag, comm, request, error)
call MPI_WAIT(request, status, error)
```

MPI_WAIT is **blocking** and gives back control as soon as the reception or the send has completed.

MPI_TEST

```logical flag
call MPI_ISEND(data, nb_elements, type, dest, tag, comm, request, error)
call MPI_WAIT(request, status, error)
```

MPI_TEST is **non-blocking** and sends back a boolean that is true if the send or the reception has completed.
Collective Communications

1. Description
2. Vector Version
3. Reduction

17-10-2016
1. Collective Communications

- A collective communication allows to realize in one call a set of point to point communications.

- A collective communication implies all processes of a communicator.

- 3 types of collective operations:
  - synchronization
  - data transfer
  - global reduction operation

- No tag needed
1. Global Synchronization

- Barrier synchronization across all members of a `comm`.
- Blocks the caller until all group members have called it
- Returns at any process only after all processes in `comm` have entered the call.

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

![Diagram showing the synchronization process with active and waiting processes across different processes P0, P1, P2, P3 over time. The diagram highlights the barrier synchronization mechanism.](image-url)

- active process
- waiting process
- barrier
1. Broadcast (1/2)

```c
int MPI_Bcast(&buffer, count, datatype, root, comm)
```
1. Broadcast (2/2)

- *root* sends data to all process in the communicator

- Others processes wait to receive the data

- Equivalent to:
  - *root* calls *MPI_Send* to all processes
  - others process call *MPI_Recv*
1. Scatter (1/2)

int MPI_Scatter(&sendbuf, sendcnt, sendtype,
&recvbuf, recvcnt, recvtype, root, comm)
1. Scatter (2/2)

- Each process receives a part of the data sent by root according to their rank:

- The message is split into n equal segments, the i-th segment is sent to the i-th process of comm.

- Equivalent to:
  - root sends to each process of rank i a part of data:
    - MPI_Send( &(sendbuf + i * sendcnt * extent(sendtype), sendcnt, sendtype, i, ...)
  - Each process receives:
    - MPI_Recv( &recvbuf, recvcnt, recvtype, root, ...)

- Only root uses sendbuf, sendcnt and sendtype arguments.

- root can use MPI_IN_PLACE to receive data:
  - root sends no data to itself
1. Gather (1/2)

int MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)
1. Gather (2/2)

- Each process (*root* process included) sends the contents of its send buffer to the root process.

- The *root* process receives the messages and stores them in rank order.

- Equivalent to:
  - each of the *n* processes (including the root process) executes a call to:
    
    ```
    MPI_Send(&sendbuf, sendcnt, sendtype, root, ...)
    ```
  - and the root executes *n* calls to:
    
    ```
    MPI_Recv(&(recvbuf + i * recvcnt * extent(recvtype)), recvcnt, recvtype, i, ...)
    ```

- *root* can use *MPI_IN_PLACE* as send buffer:
  - *root* sends no data to itself

- The *recvbuf*, *recvtype* and *recvcnt* arguments are ignored for all non-root processes.
1. Allgather (1/2)

int MPI_Allgather(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)
1. Allgather (2/2)

- MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root.

- All processes can use "MPI_IN_PLACE" in the parameter `sendbuf`,
  - process sends no data to itself
1. All-to-All Scatter/Gather (1/2)

**int MPI_Alltoall(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)**
1. All-to-All Scatter/Gather(2/2)

- It is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers.

- Equivalent to
  - each process executes a send to each process (itself included)
    \[
    \text{MPI\_Send(} \ &\{(sendbuf + i * sendcnt * extent(sendtype), sendcnt, sendtype, i, \ldots)\}
    \]
  - and a receive from every other process with a call to,
    \[
    \text{MPI\_Recv(} \ &\{(recvbuf + i * recvcnt * extent(recvtype)), recvcnt, recvtype, i, \ldots)\}
    \]

- All processes can use MPI_IN_PLACE in the parameter \textit{sendbuf} ,
  - the data to be sent is taken from the recvbuf...
  - ...and replaced by the received data
2. Vector Versions of the Collective

- Operation of collective vector (name of operation with "v" suffix)
  - MPI_Scatterv
  - MPI_Gatherv
  - MPI_Allgatherv
  - MPI_Alltoallv (et MPI_Alltoallw)

- Allows a varying count of data from each process, since recvcounts is now an array
### 2. Vector Version of Scatter (1/3)

**Vector version of MPI_Scatter**

- **sendcounts**: integer array (of length group size) specifying the number of elements to send to each *rank*
- **displs**: integer array (of length group size). Entry $i$ specifies the displacement (relative to $sendbuf$) from which to take the outgoing data to process $i$

**The send buffer is ignored for all non-root processes.**
2. Vector Version of Scatter (2/3)

▶ Equivalent to:

- the outcome is as if the root executes \( n \) send operations,

\[
\text{MPI\_Send( } \&(\text{sendbuf} + \text{displs}[i] * \text{extent}(%s), \text{sendcounts}[i], \text{sendtype}, i, ... )
\]

- and each process executes a receive

\[
\text{MPI\_Recv( } \&\text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{root}, ...
\]

▶ root can use MPI\_IN\_PLACE as recvbuf:

- root sends no data to itself
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100-i;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, recvbuf, 100-i, MPI_INT, root, comm);
2. **TP**: Example of vector version of collective communication

- The first process (rank=0) send an array to all process.
  - The size of the array depending of the rank of the receiver.
    (sends 1 element to rank=1, 2 elements to rank=2 ...
2. **TP:** Example of vector version of collective communication (solution)

```c
int *scounts = malloc(sizeof(int)* mpisize);
int *displs = malloc(sizeof(int)* mpisize);
for (i = 0 ; i < mpisize ; i++)
    scounts[i] = i;
displs[0] = 0;
for (i = 1 ; i < mpisize ; i++)
    displs[i] = displs[i-1] + scounts[i-1];

MPI_Scatterv(array, scounts, displs, MPI_INT,
             array, mpirank, MPI_INT, 0, MPI_COMM_WORLD);
```
3. Reduction (1/6)

- Performs a global reduce operation (for example sum, maximum, and logical and) across all members of a group

- Example with *SUM* operation

```
P0 3
P1 5
P2 1

P0 9
```
3. Reduction (2/6)

Predefined Reduction Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bit-wise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical exclusive or (xor)</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bit-wise exclusive or (xor)</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Min value and location</td>
</tr>
</tbody>
</table>
MPI_Reduce combines the elements provided in the input buffer `sendbuf` of each process in the communicator `comm`, using the operation `op`.

- Returns the combined value in the output buffer of the process with rank root.

Example:

```c
MPI_Reduce(&sendbuf, &recvbuf, 2, MPI_INT, MPI_MAX, 0, comm)
```

- `recvbuf[0] = max(all of sendbuf[0])`
- `recvbuf[1] = max(all of sendbuf[1])`
Another example:

\[
\text{MPI\_Reduce}(&\text{sendbuf}, &\text{recvbuf}, 3, \text{MPI\_INT}, \text{MPI\_SUM}, 0, \text{comm})
\]
3. Reduction (5/6)

- A variant of the reduce operations where the result is returned to all processes in a group.

- Example with *SUM* operation

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

![Network diagram showing the reduction process](chart.png)
3. Reduction (6/6)

The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0, ..., i.

Example with *SUM* operation

```c
int MPI_Scan(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```
3. User Defined Reduction

Just to mention that thanks to the following routines, the user can define its own reduction operations.

- `int MPI_Op_create(MPI_User_function* user_fct, int commute, MPI_Op* op)`
- `int MPI_Op_free(MPI_Op* op)`
- `void MPI_User_function(void* invec, void* inoutvec, int *len, MPI_Datatype *datatype)`
Questions?

17-10-2016
Thanks

For more information please contact:
Benjamin Pajot
T +33 4 76 29 74 57
benjamin.pajot@atos.net

Or contact R. Dolbeau & G.-E. Moulard

Atos, the Atos logo, Atos Consulting, Atos Worldgrid, Worldline, BlueKiwi, Canopy the Open Cloud Company, Yunano, Zero Email, Zero Email Certified and The Zero Email Company are registered trademarks of Atos. July 2014. © 2014 Atos. Confidential information owned by Atos, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written approval from Atos.

17-10-2016