Training @ CINES

MPI

Johanne Charpentier & Gabriel Hautreux
charpentier@cines.fr hautreux@cines.fr
Summary

- Clusters Architecture
- OpenMP
- MPI
- Hybrid MPI+OpenMP
MPI – Message Passing Interface

1. Introduction
2. MPI Environment
3. Point to point communications
4. Collective communications
5. Communicators
MPI – Message Passing Interface

1. Introduction
2. MPI Environment
3. Point to point communications
4. Collective communications
5. Communicators
Message Passing Interface
MPI Applications:

- Astrophysics
- Fluid Dynamic
- DNA
- Mechanical Structures
- Biochemistry
- Data Mining
- Cryptanalyze
- Brute force
- Reverse engineering
- etc
What is MPI?

- An **API**: defined code to be used.
- A list of **headers** (include), provided by your mpi runtime package.
- A list of **libraries** (lib/lib64), provided by your mpi runtime package.
- A **Launcher** (mpirun/mpiexec), provided by your mpi runtime package and whose purpose is to manage MPI processes during calculations (based on a hostfile).

Note: as seen, mpif90/mpicc are only **wrappers** to your compiler with added mpi include and libs.
What is MPI?

```
program hello_world
use mpi
implicit none
integer :: rank, nb_mpi_processes, ierror, hostname_len
character (len=MPI_MAX_PROCESSOR_NAME) :: hostname

! To enhance code readability, we let MPI call or MPI native variables in capital letters in Fortran
call MPI_INIT(ierr) ! Init MPI (init MPI_COMM_WORLD communicator, set rank to each process, etc)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_mpi_processes, ierr) ! Ask the number of MPI processes running

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr) ! Ask the rank of the current process
call MPI_GET_PROCESSOR_NAME(hostname,hostname_len,ierr) ! Ask the name of the host the process is running on

print*, 'Hello world ! I am process', rank, 'on', nb_mpi_processes,'processes. I am running on',hostname

call MPI_FINALIZE(ierr) ! Close MPI
end program hello_world
```

Compilation, using provided wrapper (include/lib)

```
:~$ mpif90 hello.f90
```

Conductor and computations

```
:~$ mpirun -np 2 ./a.out
Hello world ! I am process 1 on 2 processes. I am running on occigen50
Hello world ! I am process 0 on 2 processes. I am running on occigen50
```
Vocabulary: A process with sons threads

- Private memories of threads
- Private memory
MPI is multiple processes working together by communicating through network.
Introduction

General MPI Program Structure

MPI include file

Declarations, prototypes, etc.

Program Begins

Serial code

Initialize MPI environment

Parallel code begins

Do work & make message passing calls

Terminate MPI environment

Parallel code ends

Serial code

Program Ends
How do you communicate using mails?

You have a **departing address** (the rank)
You specify a **destination address** (the rank)
You use a letter/box adapted to the size of what you send (the size of data)
You choose the support to use: Web, US Postal Service, UPS, La Poste, etc (the communicator)

Same with MPI, with also

*The type of what you send* (integer, real/float, etc)
*A communication number* (tag)
*Error* return for Fortran and *Status.*
Communicator

Predefined communicator that includes all your MPI processes
Domain decomposition
MPI – Message Passing Interface

1. Introduction

2. MPI environment

3. Point to point communications

4. Collective communications

5. Communicators
program hello_world

use mpi

implicit none

integer :: rank, nb_mpi_processes, ierror, hostname_len

character (len=MPI_MAX_PROCESSOR_NAME) :: hostname

!To enhance code readability, we let MPI call or MPI native variables in capital
letters in Fortran

call MPI_INIT(ierr) ! Init MPI (init MPI_COMM_WORLD communicator, set rank to each
process, etc)

call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_mpi_processes, ierr) ! Ask the number of MPI
processes running

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr) ! Ask the rank of the current
process

call MPI_GET_PROCESSOR_NAME(hostname,hostname_len,ierr) ! Ask the name of the host
the process is running on

print*, 'Hello world ! I am process',rank,' on',nb_mpi_processes,' processes. I am
running on',hostname ! Say hello

call MPI_FINALIZE(ierr) ! Close MPI

end program hello_world

::$ mpi MPI hello world
f90 hello.f90
::$ mpirun -np 2 ./a.out
Hello world ! I am process 1 on 2 processes. I am running on
service2

Hello world ! I am process 0 on 2 processes. I am running on
service2
MPI basic code

Load headers or modules

Fortran: use mpi
C: #include<mpi.h>

Initialize the MPI execution environment

Fortran: call MPI_INIT(ierr)
C: MPI_Init(NULL, NULL);
MPI basic code

Get the rank of the current process

Fortran : call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

C : MPI_Comm_rank(MPI_COMM_WORLD, &rank);

Get the total number of processes currently working together

Fortran : call MPI_COMM_SIZE(MPI_COMM_WORLD, nb_mpi_processes, ierror)

C : MPI_Comm_size(MPI_COMM_WORLD, &nb_mpi_processes);
Get the name of the computer/node your process is running on

Fortran: call MPI_GET_PROCESSOR_NAME(hostname,hostname_len,ierror)

C: MPI_Get_hostname(hostname,&hostname_len);

Finalize MPI context

Fortran: call MPI_FINALIZE(ierr)

C: MPI_Finalize();
Exercises 1.1, 1.2
MPI – Message Passing Interface

1. Introduction
2. MPI Environment
3. Point to point communications
4. Collective communications
5. Communicators
Two different MPI tasks: one send a message, the other perform a matching receive
- Synchronous send
- Blocking send / blocking receive
- Non_blocking send/non-blocking receive
- Buffered send
- Combined send/receive...

Buffering

Path of a message buffered at the receiving process
Blocking or non-blocking communications

Blocking send/receive: easiest, but might waste time..
Non-blocking send/receive: might be able to overlap wait with other stuff, need the use of MPI_WAIT and/or MPI_TEST
Sending data to another process

Example, 1D blur: \( A(i-1) + A(i) + A(i+1) \rightarrow B(i) \)
Sequential or OpenMP:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
 A & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
 B & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

MPI? Need to share work, and so to share memory

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
 A & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
 B & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
 A & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
 B & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]
MPI? Need to share work, and so to share memory:

Need to communicate to know neighbor's data:
Sending data to another process: use “ghosts” cells
Fortran:
call MPI_SEND ( ball, 1, MPI_INTEGER, 1, tag, MPI_COMM_WORLD, ierror )

C: MPI_Send ( &ball, 1, MPI_INTEGER, 1, tag, MPI_COMM_WORLD );

• Ball : data to be sent
• 1 : int, number of items of this data to be sent
• MPI_INTEGER : int, type of the data
• 1 : int, rank of destination
• MPI_COMM_WORLD : int, communicator to use
• ierror : int
**Fortran:**
call MPI_RECV ( ball, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD, status, ierror )

**C:**
MPI_Recv ( &ball, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
Synchronous Send/Receive data:

Theses communications are synchronous. The process will wait here until data is sent or received. In order to make it work, you have to choose an order: a process send its data while the receiver waits it. When its done, reverse: the first one waits for the data and the second one send it. This is laborious, we will see some bypass after.
program ping_pong
use mpi
implicit none
integer :: rank, nb_mpi_processes, ierror
integer :: niter = 6

call MPI_INIT( ierror )
call MPI_COMM_SIZE( MPI_COMM_WORLD , nb_mpi_processes , ierror )
call MPI_COMM_RANK( MPI_COMM_WORLD , rank , ierror )
if(nb_mpi_processes /= 2) stop 'This program is design to be run with 2 processes only'

ball = 0
do n=1,niter
    if(rank==0) then
        call MPI_SEND ( ball , 1 , MPI_INTEGER , 1 , tag , MPI_COMM_WORLD , ierror )
        ! 0 send ball to 1, and wait for transfer to be finished
        call MPI_RECV ( ball , 1 , MPI_INTEGER , 1 , tag , MPI_COMM_WORLD , MPI_STATUS_IGNORE , ierror )
        ! 0 receive ball from 1, and wait for transfer to be finished
        ball = ball + 2
    end if

    if(rank==1) then
        call MPI_RECV ( ball , 1 , MPI_INTEGER , 0 , tag , MPI_COMM_WORLD , MPI_STATUS_IGNORE , ierror )
        ball = ball + 1
        call MPI_SEND ( ball , 1 , MPI_INTEGER , 0 , tag , MPI_COMM_WORLD , ierror )
    end if

    print*, 'Process',rank,'iter',n,'ball value is :',ball
end do

call MPI_BARRIER(MPI_COMM_WORLD,ierror) ! A barrier. processes stop here, and can pass it only if ALL processes are here. Useful for debug, can impact performances
end program ping_pong

::$ mpirun -np 2 ./a.out
    Process 0 iter 1 ball value is : 3
    Process 0 iter 2 ball value is : 6
    Process 0 iter 3 ball value is : 9
    Process 0 iter 4 ball value is : 12
    Process 0 iter 5 ball value is : 15
    Process 0 iter 6 ball value is : 18
    Process 1 iter 1 ball value is : 1
    Process 1 iter 2 ball value is : 4
    Process 1 iter 3 ball value is : 7
    Process 1 iter 4 ball value is : 10
    Process 1 iter 5 ball value is : 13
    Process 1 iter 6 ball value is : 16
Synchronous Send/Receive data:

Theses communications are synchronous. The process will wait here until data is sent or received. In order to make it work, you have to choose an order: a process sends its data while the receiver waits for it. When it's done, reverse: the first one waits for the data and the second one sends it. This is laborious, we will see some bypass after.

Use Sendrecv to do both at the same time!
Fortran:
call MPI_SENDRECV ( val, 1, MPI_INTEGER, 1, tag, 
                          val0, 1, MPI_INTEGER, 1, tag, 
                          MPI_COMM_WORLD, status, ierror )

C:
MPI_Sendrecv ( &val, 1, MPI_INTEGER, 1, tag, &val0, 1, 
                MPI_INTEGER, 1, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
Fortran:
call MPI_SENDRECV ( val, 1, MPI_INTEGER, 1, tag, val0, 1, MPI_INTEGER, 1, tag, MPI_COMM_WORLD, status, ierror )
    val = val0
end if
if(rank==1) then
    call MPI_SENDRECV ( val, 1, MPI_INTEGER, 0, tag, val0, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD, status, ierror )
    val = val0
end if

C:
MPI_Sendrecv ( &val, 1, MPI_INTEGER, 1, tag, &val0, 1, MPI_INTEGER, 1, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
Exercises 1.3, 1.4
MPI – Message Passing Interface

1. Introduction
2. MPI Environment
3. Point to point communications
4. Collective communications
5. Communicators
Apply to all processes of the specified communicator

Available:

- Synchronization
- Reductions (Max, Min, SUM, PROD, etc)
- Global broadcast or gather, and derivatives
MPI collective communications

- MPI_BARRIER
- MPI_REDUCE
- MPI_ALLREDUCE
- MPI_BCAST
- MPI_SCATTER
- MPI_GATHER
- MPI_GATHERV
- MPI_ALLTOALL
Synchronization

As for OpenMP, you can use barriers to ensure all processes stay synchronize before or after a specific action.

Fortran: call MPI_BARRIER(MPI_COMM_WORLD,ierror)

C: MPI_Barrier(MPI_COMM_WORLD);
Reductions: REDUCE

get result on only one process: REDUCE
get result on all processes: ALLREDUCE
get result on only one process : REDUCE
get result on all processes : ALLREDUCE

Reductions : ALLREDUCE

b0 + b1 + b2 + b3 = a

AllReduce
Fortran:
CALL MPI_ALLREDUCE (val, sum_val, 1, MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierror)

C:
MPI_Allreduce(&val, &val, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
Collective communications

Broadcast: MPI_BCAST

Private memory

A0

b

MPI_BCAST
**Fortran:**

```fortran
CALL MPI_BCAST ( val , 1 , MPI_DOUBLE_PRECISION , 0 , MPI_COMM_WORLD , ierror)
```

**C:**

```c
MPI_Bcast(&val , &val , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```
Collective communications

Broadcast: MPI_SCATTER
Fortran:

CALL MPI_SCATTER (aval(1:8), 2, MPI_DOUBLE_PRECISION, bval(1:2), 2, MPI_DOUBLE_PRECISION, 3, MPI_COMM_WORLD, ierror)

C:

MPI_Scatter (&aval, 2, MPI_REAL, &bval, 2, MPI_REAL, 3, MPI_COMM_WORLD);
Collective communications

Broadcast: MPI_GATHER
Fortran:

CALL MPI_GATHER( val, 1, MPI_DOUBLE_PRECISION, cval(1:4), 1, MPI_DOUBLE_PRECISION, 3, MPI_COMM_WORLD, ierror)

C:

MPI_Gather (val, 1, MPI_DOUBLE_PRECISION, &cval, 1, MPI_DOUBLE_PRECISION, 3, MPI_COMM_WORLD);
Collective communications

Broadcast: MPI_ALLGATHER
**Fortran:**

```fortran
CALL MPI_ALLGATHER( val, 1, MPI_DOUBLE_PRECISION,
cval(1:4), 1 , MPI_DOUBLE_PRECISION, MPI_COMM_WORLD,
ierror)
```

**C:**

```c
MPI_Allgather(val, 1, MPI_DOUBLE_PRECISION, &cval, 1 ,
MPI_DOUBLE_PRECISION, MPI_COMM_WORLD);
```
Broadcast: MPI_GATHERV
Broadcast: MPI_ALLTOALL

Private memory

Private memory

Private memory

Private memory

MPI_ALLTOALL
Others available, not covered in this training:

MPI_SCATTERV()
MPI_GATHERV()
MPI_ALLGATHERV()
Etc.

All information can be found here:
http://www.mpich.org/static/docs/v3.0.x/www3/
Exercises 1.5
MPI – Message Passing Interface

1. Introduction
2. Acquire information
3. Point to point communications
4. Collective communications
5. Communicators
Cartesian communicator
Specific communicator dedicated to Cartesian organization

User tuned communicator
Users can defined their own communicators for specific purposes
Cartesian communicator
Cartesian communicator

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,2</td>
<td>1,2</td>
<td>2,2</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
</tr>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>MPI_PROC_NULL</td>
<td>MPI_PROC_NULL</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>MPI_PROC_NULL</td>
<td>MPI_PROC_NULL</td>
</tr>
</tbody>
</table>
### Cartesian Communicator

**Communicators**

A Cartesian communicator is a type of communicator that operates within a grid or coordinate system. The grid layout helps in structuring the communication pattern or protocol.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
### Cartesian communicator

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_PROC_NULL</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>MPI_PROC_NULL</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>MPI_PROC_NULL</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>MPI_PROC_NULL</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>MPI_PROC_NULL</td>
</tr>
</tbody>
</table>
When you communicate with MPI_PROC_NULL, communication does not occur, but does not stop computation.

   Useful to simplify source code.

Why using Cartesian communicator when you can do it manually? Reorganize!

   This function tune your ranks numbers depending on their proximity in the super computer to maximize performances.
nb_process_axe(1) = 4
cart_boundaries(1) = .false. ! Means not periodic, .true. Means periodic

call MPI_CART_CREATE( MPI_COMM_WORLD , ndim , nb_process_axe(1:ndim) , &
& cart_boundaries(1:ndim) , .true. , MPI_COMM_CART , ierror )

call MPI_CART_COORDS( MPI_COMM_CART , rank , ndim , cart_position(1:ndim) , ierror )

call MPI_CART_SHIFT (MPI_COMM_CART, 0, 1, cart_neigh(-1), cart_neigh(+1), ierror)
User tuned communicator

**Fortran**:

```
CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, color, key, MY_COM, ierror)
```

**C**:

```
MPI_Comm_split(MPI_COMM_WORLD, color, key, &MY_COMM);
```

```plaintext
integer :: key, color, MY_COMMUNICATOR
[...]
color=MPI_UNDEFINED
if (rank == 0 .OR. rank == 1) color = 1
if (rank == 2 .OR. rank == 3) color = 2
key=rank
CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, color, key, MY_COMMUNICATOR, ierror)

CALL MPI_ALLREDUCE ( rank , ranksum , 1 , MPI_INTEGER , MPI_SUM ,
MY_COMMUNICATOR , ierror)

print *,"I am proc",rank,ranksum
```

```
I am proc           2
5
I am proc           0
1
I am proc           3
5
```
Exercises 1.6, 1.7

https://computing.llnl.gov/tutorials/mpi/
http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html
http://www.mpich.org/static/docs/v3.0.x/www3/ (API)
http://www.open-mpi.org/doc/v1.6/ (API)
http://semelin.developpez.com/cours/parallelisme/ (FR)