CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Message Passing Interface (MPI)

Summer School 2016 – Effective High Performance Computing

Maxime Martinasso, CSCS

July 20 – 21, 2016

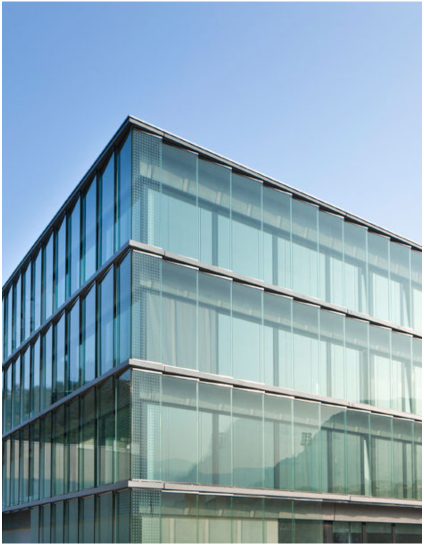
Previous course summary

- MPI, message passing paradigm
- Distributed memory (but also shared memory)
- Write a simple program, no communication

Course Objectives

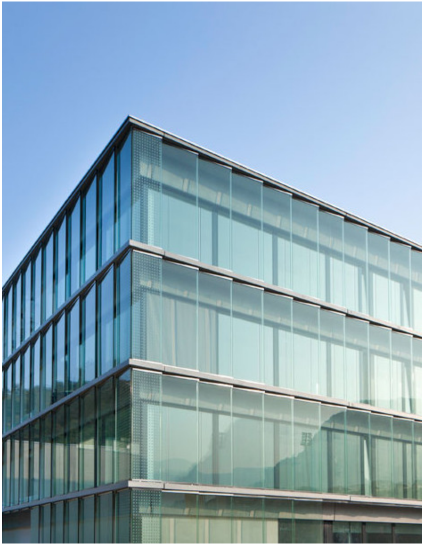
- The understanding of a point-to-point communications
- The understanding of their different flavors
- The importance of buffer availability and scope

General Course Structure



- An introduction to MPI
- Point-to-point communications
- Collective communications
- Topology
- Datatypes

General Course Structure



- An introduction to MPI
- Point-to-point communications
  - Communication
  - Message
  - Data types
  - Standard Send/Recv
  - Communication status
  - Wildcards
  - Synchronization
  - Blocking and non-blocking
  - Transfer modes
  - Summary
- Collective communications
- Topology
- Datatypes

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Point-to-point communications

Point-to-Point communication

It is the fundamental communication facility provided by a MPI library.

Communication between 2 processes:

- It is conceptually simple: source process A sends a message to destination process B, B receives the message from A.
- Communication takes place within a communicator
- Source and Destination are identified by their rank in the communicator

Message

Data is exchanged in the buffer, an array of count elements of some particular MPI data type

- One argument that usually must be given to MPI routines is the type of the data being passed.
- This allows MPI programs to run automatically in heterogeneous environments

Messages are identified by their envelopes. A message could be exchanged only if the sender and receiver specify the correct envelope.

body			envelope			
buffer	count	datatype	source	destination	communicator	tag

Message ordering

Each process has a FIFO receipt (queue), incoming messages never overtake each other.

If process A does multiple sends to process B those messages arrive in the same order.

Data types

MPI provides its own data type for send and recv buffers.

- Handle type conversion in a heterogeneous collection of machines

General rule

MPI datatype must match datatypes among pairs of send and recv

MPI defines ”handles” to allow programmers to refer to data types.

Fortran - MPI Intrinsic Datatypes

MPI Data type	Fortran Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	-
MPI_BYTE	-

C - MPI Intrinsic Datatypes

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

Basic Send/Recv

C/C++

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int
    dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype type, int
    source, int tag, MPI_Comm comm, MPI_Status *status);
```

Fortran

```
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, source, tag, comm, status, ierr)
```

buf	array of type type see table
count	(INTEGER) number of element of buf to be sent
type	(INTEGER) MPI type of buf
dest	(INTEGER) rank of the destination process
source	(INTEGER) rank of the source process
tag	(INTEGER) number identifying the message
comm	(INTEGER) communicator of the sender and receiver
status	(INTEGER) array containing communication status
ierr	(INTEGER) error code (if ierr=0 no error occurs)

Basic Send/Recv

C/C++

```c
int MPI_Send(void *buf, int count, MPI_Datatype type, int
    dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype type, int
    source, int tag, MPI_Comm comm, MPI_Status *status);
```

Fortran

```fortran
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, source, tag, comm, status, ierr)
```

buf	
count	BODY
type	
source/dest	
tag	
comm	ENVELOPE
status	
ierr	(INTEGER) error code (if ierr=0 no error occurs)

MPI Status

`MPI_Status` structures are used by the message receiving functions to return data about a message. It is an `INTEGER` array of `MPI_STATUS_SIZE` elements in Fortran. The array contains the following info:

- `MPI_SOURCE` - id of processor sending the message
- `MPI_TAG` - the message tag
- `MPI_ERROR` - error status

There may also be other fields in the structure, but these are reserved for the implementation.

Wildcards

Both in Fortran and C, `MPI_Recv` accepts wildcards:

- To receive from any source: `MPI_ANY_SOURCE`
- To receive with any tag: `MPI_ANY_TAG`
- Actual source and tag are returned in the receivers status parameter

Synchronization

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is actually never the case. A send resp. a receive can be triggered before or after its corresponding receive resp. send.
- Is the receiving rank prepared to receive the data?...

Rendez-vous protocol

Internal to MPI not expose to the user

C/C++ Pseudo-code

```
if (size_to_send < CSTE ) {
    /* EAGER protocol
    copy the send buffer to a kernel buffer
    post a send request using kernel buffer */
} else {
    /* RENDEZ-VOUS protocol
    prepare a rendez-vous buffer (send buffer size)
    post a send request using the rendez-vous buffer
    wait for the send request to be acknowledged
    post a send request using buffer */
}
```

Practicals

Exercise: 02.MPI_pt2pt

1. Standard Send/Recv communication
2. Ping-pong
3. Measuring bandwidth

Blocking and non-blocking communications

A blocking communication means...

That the buffer can be re-used after the function as returned.

A non-blocking communication means...

That the buffer can only be re-used after a wait function as returned.
Until then, the buffer must not be overwritten (and even read).

Blocking and non-blocking do not mean...

That the communication is completed after the function as returned.

Blocking and non-blocking communications

A blocking communication means...

That the buffer can be re-used after the function as returned.

A non-blocking communication means...

That the buffer can only be re-used after a wait function as returned.
Until then, the buffer must not be overwritten (and even read).

Blocking and non-blocking do not mean...

That the communication is completed after the function as returned.

$\Rightarrow$ **It is all about buffer availability!**

Blocking and non-blocking examples

Blocking example:

C/C++
```
int buf[120];
MPI_Send(buf,MPI_INT
    ,120,...);
// from here buffer can be
    re-used
but[3]=5
```

Non-blocking example:

C/C++
```
int buf[120];
MPI_Isend(buf,MPI_INT
    ,120,...);
// do not overwrite buf
// buf[3]=5 illegal
// leads to undefined
    behaviour
MPI_Wait(...);
// from here buffer can be
    re-used
but[3]=5
```

Each rank does not necessary know if the communication has been completed by the time the MPI send function returns.

The `I` in `MPI_Isend` means immediate, the function returns ”immediately” after its call.

Non-blocking MPI_Isend and MPI_Irecv

C/C++

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int
    dest, int tag, MPI_Comm comm, MPI_Request *req);
int MPI_Irecv(void *buf, int count, MPI_Datatype type, int
    source, int tag, MPI_Comm comm, MPI_Request *req);
```

Fortran

```
MPI_ISEND(buf, count, type, dest, tag, comm, req, ierr)
MPI_IRECV(buf, count, type, source, tag, comm, req, ierr)
```

buf	array of type type see table
count	(INTEGER) number of element of buf to be sent
type	(INTEGER) MPI type of buf
dest	(INTEGER) rank of the destination process
source	(INTEGER) rank of the source process
tag	(INTEGER) number identifying the message
comm	(INTEGER) communicator of the sender and receiver
req	(INTEGER) identifier of the communications handle
ierr	(INTEGER) error code (if ierr=0 no error occurs)

Waiting for completion

C/C++
```
int MPI_Wait(MPI_Request *req, MPI_Status *status)
int MPI_Waitall(int count, MPI_Request array_of_requests[],
    MPI_Status array_of_statuses[])
```

Fortran
```
MPI_WAIT(req, status, ierr)
MPI_WAITALL(count, array_of_requests, array_of_statuses,
    ierr)
```

req	(INTEGER) identifier of the communications handle
status	(INTEGER) array containing communication status
count	(INTEGER) number of element of in arrays
ierr	(INTEGER) error code (if ierr=0 no error occurs)

Testing for completion

C/C++

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *
    status)
int MPI_Testall(int count, MPI_Request array_of_requests[],
    int *flag, MPI_Status array_of_statuses[])
```

Fortran

```fortran
MPI_TEST(req, flag, status, ierr)
MPI_TESTALL(count, array_of_requests, flag,
    array_of_statuses, ierr)
```

req	(INTEGER) identifier of the communications handle
flag	(LOGICAL) true if req has completed false otherwise
status	(INTEGER) array containing communication status
count	(INTEGER) number of element of in arrays
ierr	(INTEGER) error code (if ierr=0 no error occurs)

Transfer modes

Five different communication modes are supported for the Send:

- Standard Mode
- Synchronous Mode
- Buffered Mode
- Ready Mode

All of them can be Blocking or Non-Blocking.

Transfer modes: Standard Mode

- A send operation can be started whether or not a matching receive has started
- Can be buffered or synchronous. It is up to the implementation (and not MPI standard) to decide whether outgoing messages will be buffered
- May complete before a matching receive is posted
- Non-local operation (in general)

Transfer modes: Synchronous Mode

- A send operation can be started whether or not a matching receive has started
- The send will complete successfully only if a matching receive was posted and the receive operation has reached a certain point in its execution
- The completion of a synchronous send not only indicates that the send buffer can be reused but also indicates that the receiver has reached a certain point in its execution (usually it has received all data)
- Non-local operation

Transfer modes: Buffered Mode

- A send operation can be started whether or not a matching receive has been posted
- It completes whether or not a matching receive has been posted (independent from the receive)
- The original buffer can be read and overwritten, it has been copied to user-supplied buffer
- Buffer space is allocated on demand `MPI_Buffer_Attach`
- Local operation

Transfer modes: Ready Mode

- A send operation may be started only if the matching receive is already started (error otherwise)
- The completion of the send operation does not depend on the status of a matching receive and merely indicates the send buffer can be reused
- Used for performance reasons
- Non-local operation

Transfer modes overview

Mode	Completion Condition	Blocking	Non-blocking
Standard send	Message sent (receive state unknown)	`MPI_Send`	`MPI_Isend`
Receive	Completes when a matching message has arrived	`MPI_Recv`	`MPI_Irecv`
Synchronous send	Only completes after a matching recv() is posted and the receive operation is at some stages.	`MPI_Ssend`	`MPI_Issend`
Buffered send	Always completes, irrespective of receiver. Guarantees the message to be buffered.	`MPI_Bsend`	`MPI_Ibsend`
Ready send	Always completes, irrespective of whether the receive has completed.	`MPI_Rsend`	`MPI_Irsend`

Communication - summary

Blocking send and recv

- Does not mean that the process is stopped during communication.
- It means that, at return, it is safe to use the variables involved in communication.

Non Blocking send and recv

- Cannot use variables involved in communication until ”wait” completion functions are called.

Transfer modes

- Define the behaviour of the various function for point to point communication. The behaviour can be implementation dependent.

Combined Send and Recv in one function

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The source and destination are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

Fortran

```fortran
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type, rcvid, tag,
                  rcvbuf, rcv_size, rcv_type, sndid, tag,
                  comm, status, ierr)
```

Other functions

- Test if a message has arrived:

  `MPI_Probe, MPI_Iprobe`

- Abort the application:

  `MPI_Abort`

- Buffer management:

  `MPI_Buffer_attach, MPI_Buffer_detach`

- Wait functions:

  `MPI_Waitany, MPI_Waitsome`

- Test functions:

  `MPI_Testany, MPI_Testsome`

- Cancel a request:

  `MPI_Cancel`

Practicals

Exercise: 02.MPI_pt2pt

4. Parallel sum using a ring:
   without using `if (rank == 0)... else ...`

Rank 0	0	1	2
Rank 1	1	2	0
Rank 2	2	0	1

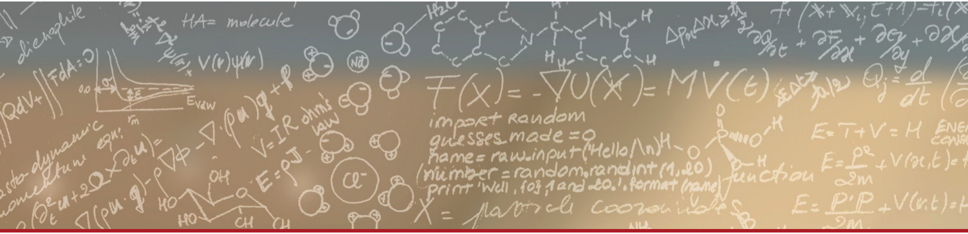
   All ranks obtain the sum.

5. Send and Recv neighbor ghost cells:
   Top to bottom for C and right to left for Fortran

6. Identify (not solve) 3 bugs in the buffer management (only in C, no Fortran version)

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Thank you for your attention.