Foundations of
High Performance Computing

Lecture 05: Message Passing programming
Using MPI

"Foundation of HPC" course
DATA SCIENCE &
SCIENTIFIC COMPUTING
2022-2023   Stefano Cozzini

Agenda

- Recap:
  - Message Passing Paradigm
- Basic on Message Passing Interface

- Point-to-Points operation
- Collective operations

Message Passing paradigm

- Parallel programs consist of separate processes, each with its own address space

- Programmer manages memory by placing data in a particular process

- Data sent explicitly between processes

- Programmer manages
  - Memory motion
  - Data distribution

Shared nothing approach

Address Spaces

Process 0

A(10)

Process 1

A(10)

DIFFERENT VARIABLES
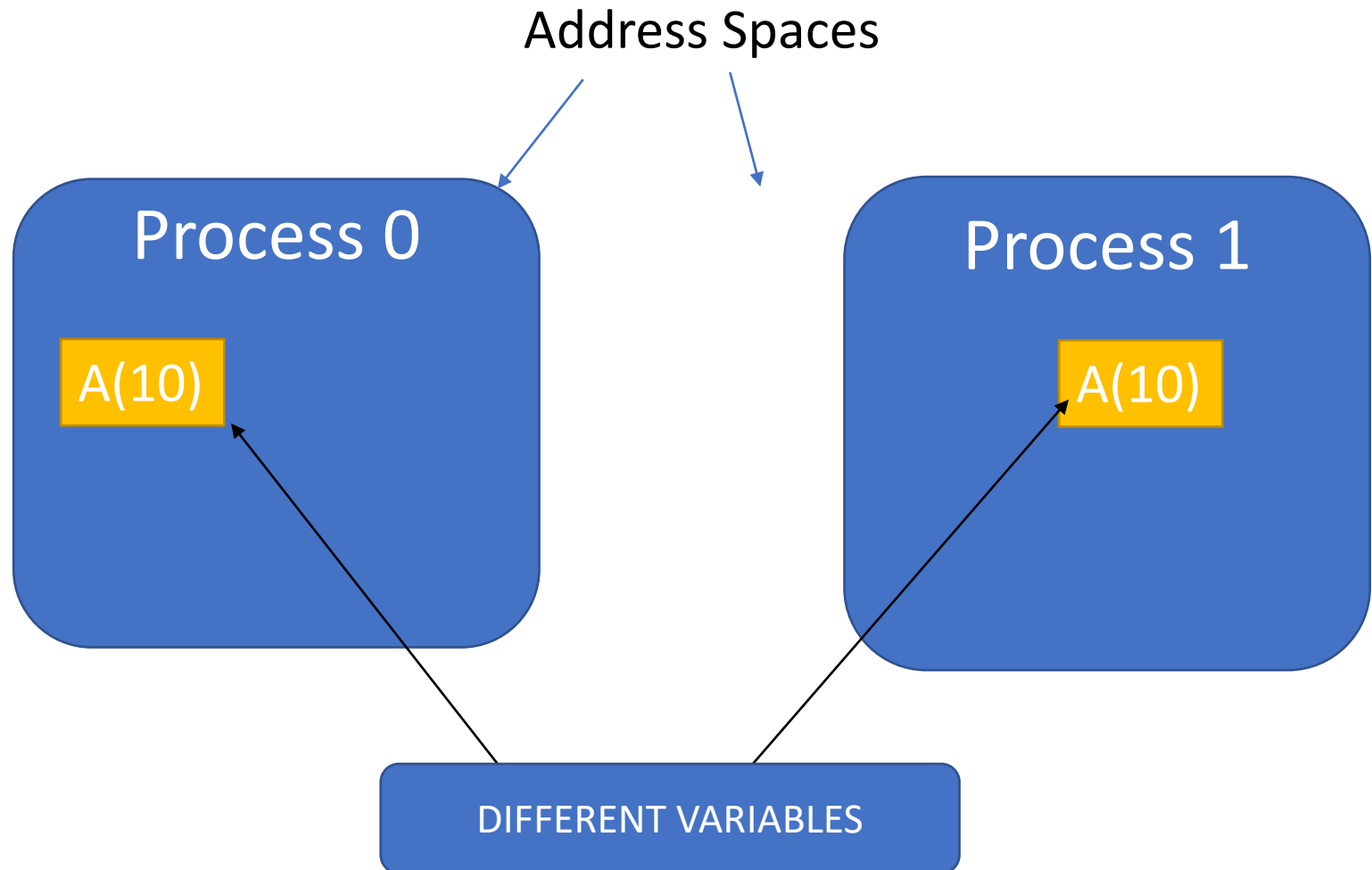
Message Passing Pro&Cons

- Pros
  - Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.
- Cons
  - Data is scattered on separated address spaces
  - The programmer is responsible for many of the details associated with data communication between processors.
  - Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

Message Passing approach= MPI

- MPI is a "standard by consensus," originally designed in an open forum that included hardware vendors, researchers, academics, software library developers, and users, representing over 40 organizations.

- This broad participation in its development ensured MPI's rapid emergence as a widely used standard for writing message-passing programs.

- MPI is not a true standard; that is, it was not issued by a standards organization such as ANSI or ISO.

What is MPI ?

A STANDARD...

- The actual implementation of the standard is demanded to the software developers of the different systems

- Standard is discussed at the www.mpi-forum.org

- In all systems MPI has been implemented as a library of subroutines over the network drivers and primitives

- many different implementations
  - MPICH (the original one)
  - OpenMPI
  - IntelMPI

MPI evolution

- MPI "standard" was introduced by the MPI Forum in May, 1994 and updated in June, 1995.

- MPI-2.0 was completed in 1997,

- MPI-3.0 was finalized in 2012.

- MPI-3.1 was finalized in 2015, which provides small additions and fixes to MPI-3.0.

- MPI 3 is still the most widely used standard (implanted in both Intel and openMPI

- MPI-4.0 was released on June 9, 2021.

- Current efforts focus on MPI 4.1 and MPI 5.0.
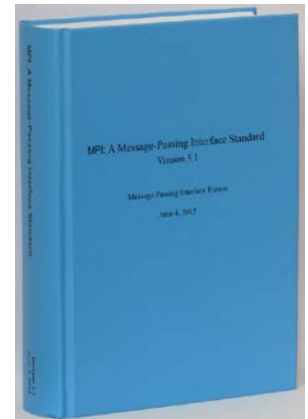
Some reasons to use MPI

- INTERNATIONAL STANDARD

- MPI evolves: MPI 1.0 was first introduced in 1994, most current version is MPI 4.0 (June 2021)

- Available on almost all parallel systems (free MPICH, Open MPI used on many clusters), with interfaces for C/C++ and Fortran

- Supplies many communication variations and optimized functions for a wide range of needs

- Works both on distributed memory (DM) and shared memory (SM) hardware architectures

- Supports large program development and integration of multiple modules

How to program with MPI ?

- MPI is a library:
  - All operations are performed with subroutine calls

- Basic definitions are in
  - mpi.h for C/C++
  - mpif.h for Fortran 77 and 90
  - MPI module for Fortran 90 (optional)

How to compile MPI Programs

- NO STANDARD: left to the implementations:
- Generally:
  - You should specify the appropriate include directory (i.e. -I/mpidir/include)
  - You should specify the mpi library (i.e. -L/mpidir/lib -lmpi)
- Usually MPI compiler wrappers do this job for you. (i.e. mpicc)

How to run MPI programs ?

- The MPI Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.

- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.

- Many implementations provided `mpirun -np 4 a.out` to run an MPI program

- The specific commands to run a program on a parallel system are defined by the environment installed on the parallel computer

How to write an MPI program ?

- Modify your serial program to insert MPI routines which distribute data and loads to different processors.

WHICH MPI ROUTINES DO I NEED ?

Basic Features of MPI routines

- Calls may be roughly divided into four classes:

  - Calls used to initialize, manage, and terminate communications

  - Calls used to communicate between pairs of processors. (Pair communication)

  - Calls used to communicate among groups of processors. (Collective communication)

  - Calls to create data types.

Minimal set of MPI routines

- MPI_INIT: initialize MPI

- MPI_COMM_SIZE: how many Processors?

- MPI_COMM_RANK: identify the Processor

- MPI_SEND : send data

- MPI_RECV: receive data

- MPI_FINALIZE: close MPI

(Almost) All you need
is to know this 6 calls

Our first program

```fortran
PROGRAM hello

 INCLUDE 'mpif.h'

 INTEGER err

 CALL MPI_INIT(err)

 call  MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)

 call  MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)

 print *, 'I am ', rank, ' of ', size

 CALL MPI_FINALIZE(err)

END
```

C

```c
#include <stdio.h>

#include <mpi.h>

int main (int argc, char * argv[])

{

  int rank, size;

  MPI_Init( &argc, &argv );

  MPI_Comm_rank( MPI_COMM_WORLD,&rank);

  MPI_Comm_size( MPI_COMM_WORLD,&size );

  printf( "I am %d of %d\n", rank, size );

  MPI_Finalize();
```

Some notes/observations

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process including the printf/print statements
- I/O not part of MPI-1 (MPI-IO part of MPI-2)
- print and write to standard output or error not part of MPI-1 or MPI-2 or MPI-3
- output order is undefined (may be interleaved by character, line, or blocks of characters),
- A consequence of the requirement that non-MPI statements execute independently

Type signatures

- All MPI routines have some similarities in their naming and in the parameters that they require. However, there are differences between C and Fortran, so let's look at these separately.

- C bindings
  - For C, the general type signature is

    rc = MPI_Xxxxx(parameter, ...)

  - Note that case is important here, as it is with all C code. For example, MPI must be capitalized, as must the first character after the underscore. Everything after that must be lower case. All C MPI functions return an integer return code, which can be used to determine if the call succeeded.
    - If rc == MPI_SUCCESS, then the call was successful.

- C programs should include the file "mpi.h". This contains definitions for MPI functions and constants like MPI_SUCCESS.

- Fortran bindings
  - For Fortran, the general type signature is

    Call MPI_XXXXX(parameter,..., ierror)

  - Note that case is not important here. So, an equivalent form would be

    call mpi_xxxxx(parameter,..., ierror)

- Instead of having a function that returns an error code, as in C, the Fortran versions of MPI calls are subroutines that usually have one additional parameter in the argument list, ierror, which is the return code. Upon success, ierror is set to MPI_SUCCESS.

Inizialing/Finalizing MPI program

- Initializing the MPI environment
  - C:

    ```
    int MPI_Init(int *argc, char ***argv);
    ```
  - Fortran:

    ```
    INTEGER IERR
    CALL MPI_INIT(IERR)
    ```

- Finalizing MPI environment
  - C:

    ```
    int MPI_Finalize()
    ```
  - Fortran:

    ```
    INTEGER IERR
    CALL MPI_FINALIZE(IERR)
    ```

This two subprograms should be called by all processes, and no other MPI calls are allowed before `mpi_init` and after `mpi_finalize`

MPI Communicator

- The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.

- It identifies the group of all processes.

- All MPI communication subroutines have a communicator argument. The Programmer could define many communicators at the same time

There is a default communicator (automatically defined):

MPI_COMM_WORLD

Communicator size and processor rank

How many processors are associated with a communicator?

C:

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

Fortran:

```
INTEGER COMM, SIZE, IERR

CALL MPI_COMM_SIZE(COMM, SIZE, IERR)

OUTPUT:  SIZE
```

What is the ID of a processor in a group?

C:

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

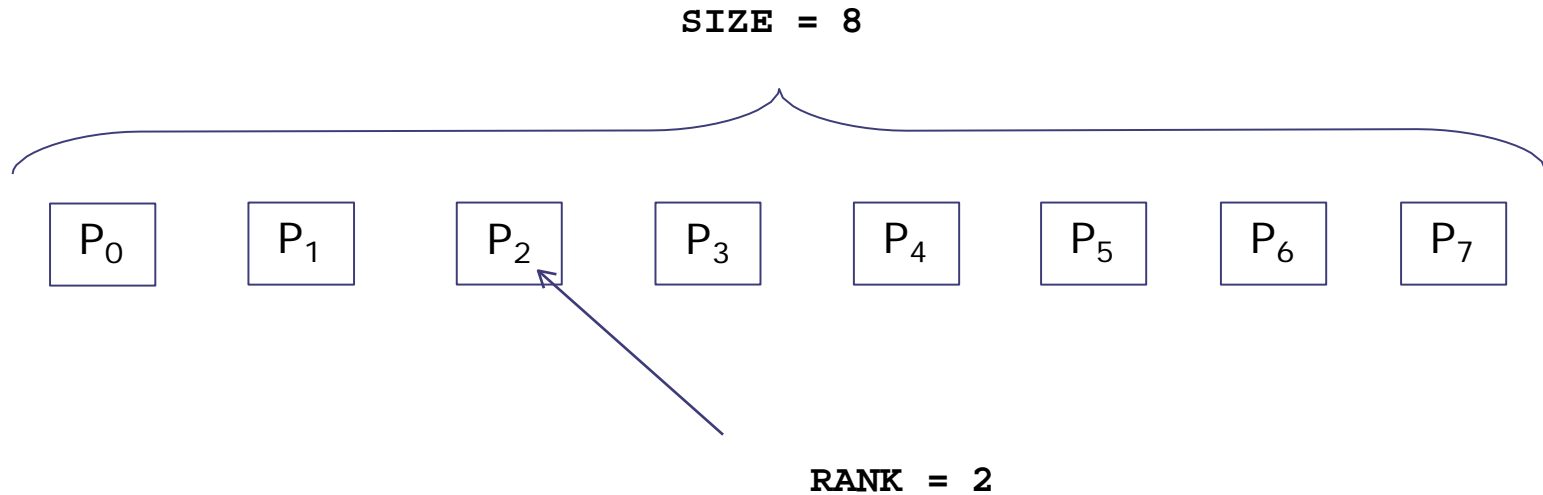Fortran:

```
INTEGER COMM, RANK, IERR
CALL MPI_COMM_RANK(COMM, RANK, IERR)

OUTPUT:  RANK
```

Communicator size and processor rank

SIZE = 8

P_0   P_1   P_2   P_3   P_4   P_5   P_6   P_7

RANK = 2

`Size` is the number of processors associated to the communicator

`rank` is the index of the process within a group associated to a communicator (`rank` = 0,1,…,N-1). The rank is used to identify the source and destination process in a communication

Basic elements of a message

- To send a message via mail we typically have:
  - An envelope (with possibly some hints on the content itself… i.e., advertisement, bills, greetings….)
  - A message
  - A destination address
  - A sender address
  - A tools to send the message (phone/mailer/ messenger)

For MPI it is exactly the same thing...

Basic Send/Receive

Process 0   Process 1

Send(data) ⟍
                    ⟍
                          ↘
                              Receive(data)
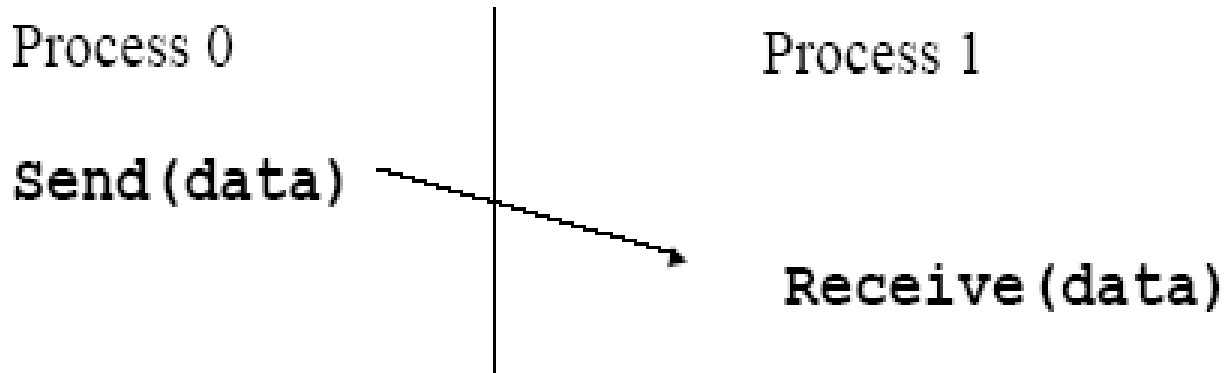
- How will "data" be described? datatypes
- How will processes be identified? rank/comm
- How will the receiver recognize messages? tag
- What will it mean for these operations to complete? blocking/non-blocking ()

Describing data

- The data in a message to send or receive is described by a triple (address, count, datatype), where an MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

MPI data type: Fortran language

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	

MPI data type: C language

MPI Data type	C Data type
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`Signed log 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`	

MPI data tag

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message

- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive

Our first send message…

The simplest call:

```
MPI_send(buffer, count, data_type,
destination,tag,communicator)
```

where:

BUFFER: data to send

COUNT: number of elements in buffer .

DATA_TYPE : which kind of data types in buffer ?

DESTINATION the receiver

TAG: the label of the message

COMMUNICATOR set of processors involved

And our first receiver..

The simplest call:

```
MPI_recv( buffer, count, data_type, source, tag, communicator, status)
```

Similar to send with the following differences:

- SOURCE is the sender ; can be set as MPI_any_source (receive a message from any processor within the communicator)
- TAG the label of message: can be set as MPI_any_tag: receive any kind of message
- STATUS integer array with information on message in case of error

The status array

Status is a data structure allocated in the user's program.

In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

A fortran example

```fortran
Program MPI
  Implicit None
!
  Include 'mpif.h'
!
  Integer    :: rank
  Integer    :: buffer
  Integer, Dimension( 1:MPI_status_size ) :: status
  Integer                            :: error
!
  Call MPI_init( error )
  Call MPI_comm_rank( MPI_comm_world, rank, error )
!
  If( rank == 0 ) Then  buffer = 33
    Call MPI_send( buffer, 1, MPI_integer, 1, 10, &
             MPI_comm_world, error )
  End If
!
  If( rank == 1 ) Then
    Call MPI_recv( buffer, 1, MPI_integer, 0, 10, &
             MPI_comm_world, status, error )  Print*, 'Rank
    ', rank, ' buffer=', buffer
    If( buffer /= 33 ) Print*, 'fail'  End If
  Call MPI_finalize( error )  End Program MPI
```

Questions for you:

- How many processors should I run this program on ?
- Can I run this program on 1000 processors ?

Blocking vs Non blocking calls

Q: When is a SEND instruction complete?
A: When it is safe to change the data that we sent.

Q: When is a RECEIVE instruction complete?
A: When it is safe to access the data we received.

Blocking vs Non blocking calls

With both communications (send and receive) we have two choices:

Start a communication and wait for it to complete:
BLOCKING approach

Start a communication and return control to the main program:
NON-BLOCKING approach

The Non-Blocking approach REQUIRES us to check for completion
before we can modify/access the sent/received data!!!

MPI_send/MPI_recv

MPI_SEND() and MPI_RECV()
are blocking operations.

Communication mode and MPI routines

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	**Message sent (receive state unknown)**	`MPI_SEND`	`MPI_ISEND`
receive	**Completes when a message has arrived**	`MPI_RECV`	`MPI_IRECV`
Synchronous send	**Only completes when the receive has completed**	`MPI_SSEND`	`MPI_ISSEND`
Buffered send	**Always completes, irrespective of receiver**	`MPI_BSEND`	`MPI_IBSEND`
Ready send	**Always completes, irrespective of whether the receive has completed**	`MPI_RSEND`	`MPI_IRSEND`

Message overhead

- How much time your process will spend waiting for the blocking send (or receive) to return.

- Each mode has different overhead characteristics

- the overhead as having two sources:
  - System overhead: the time spent transferring buffer contents
  - Synchronization overhead' the time spent waiting for another process

- We can classify all of the time spent waiting as either system or synchronization overhead.

Synchronous send
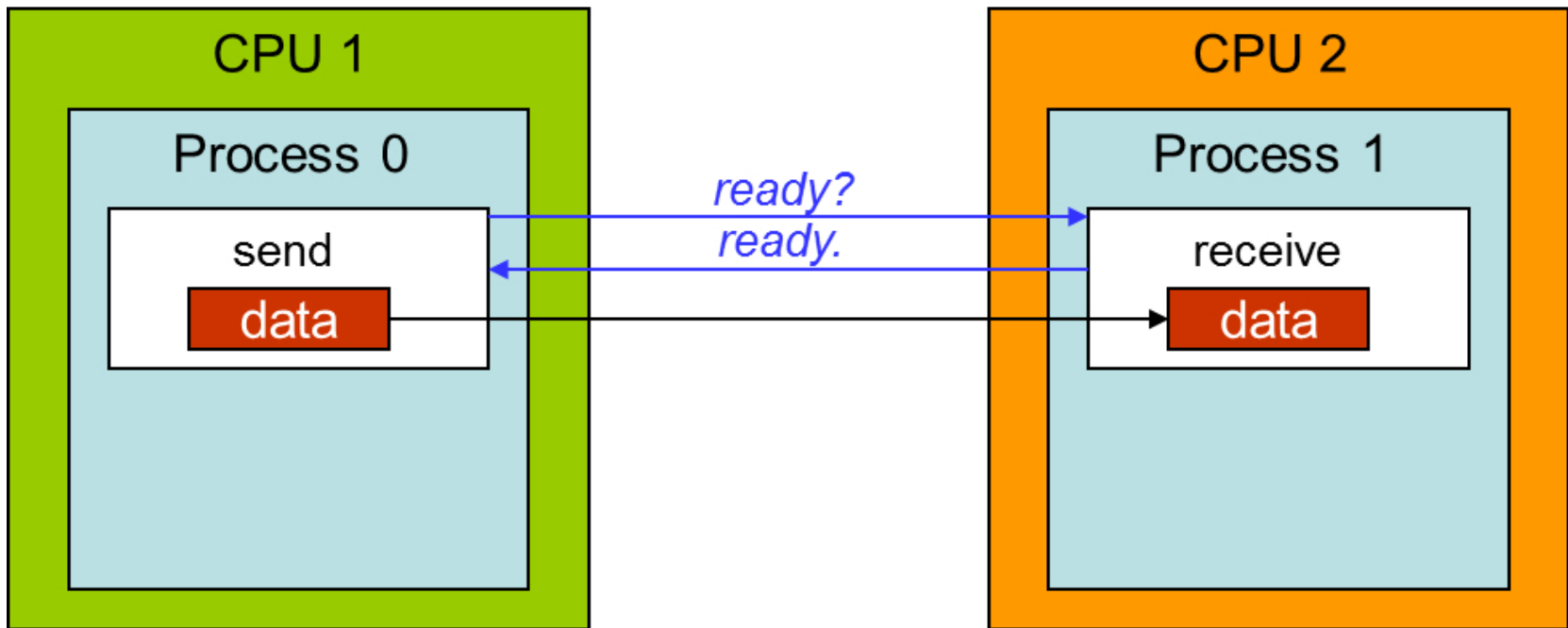
the safest Point-To-Point communication method, as the sending process requires the receiving process to provide a "ready" signal, or the ***matching receive***, in order to initiate the send; therefore, the receiving process will always be ready to receive data from the sender.

Buffered Send

it copies the data from the message buffer to a user-supplied buffer, and then returns. The data will be copied from the user-supplied buffer over the network once the "ready to receive" notification has arrived.
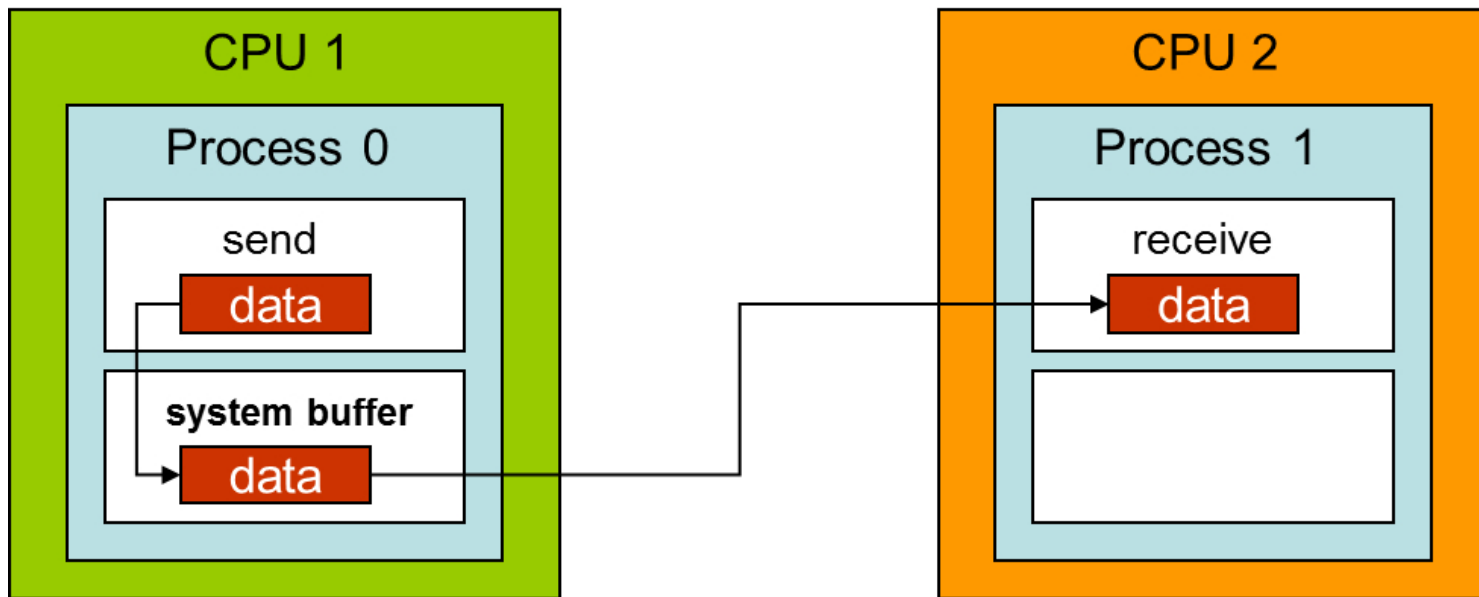

Implications:
1. timing of the corresponding receive is irrelevant
2. data in the original buffer can be modified
3. synchronization overhead on the sender is eliminated
4. system overhead is added

Ready Send

It attempts to reduce system and synchronization overhead by assuming that a ready-to-receive message has already arrived. Conceptually this results in a diagram that is identical to that first used to describe the simple view of point-to-point communication
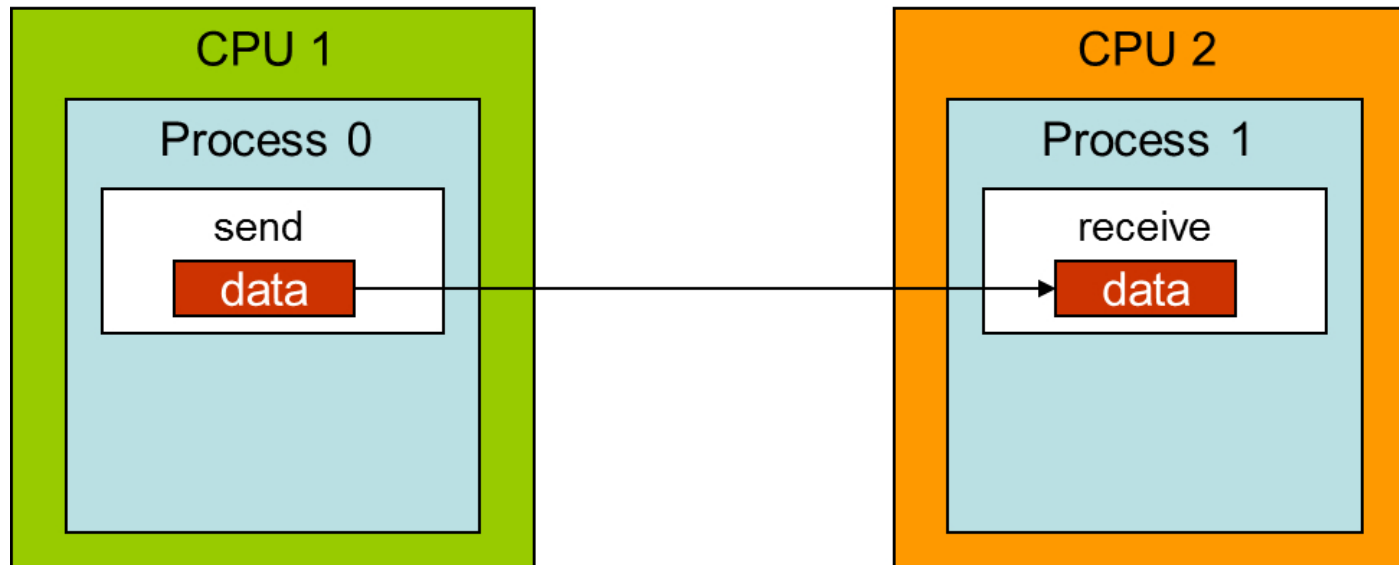

The idea is to have a blocking send that only blocks long enough to send the data to the network. However, if the matching receive has not already been posted when the send begins, an error will be generated.
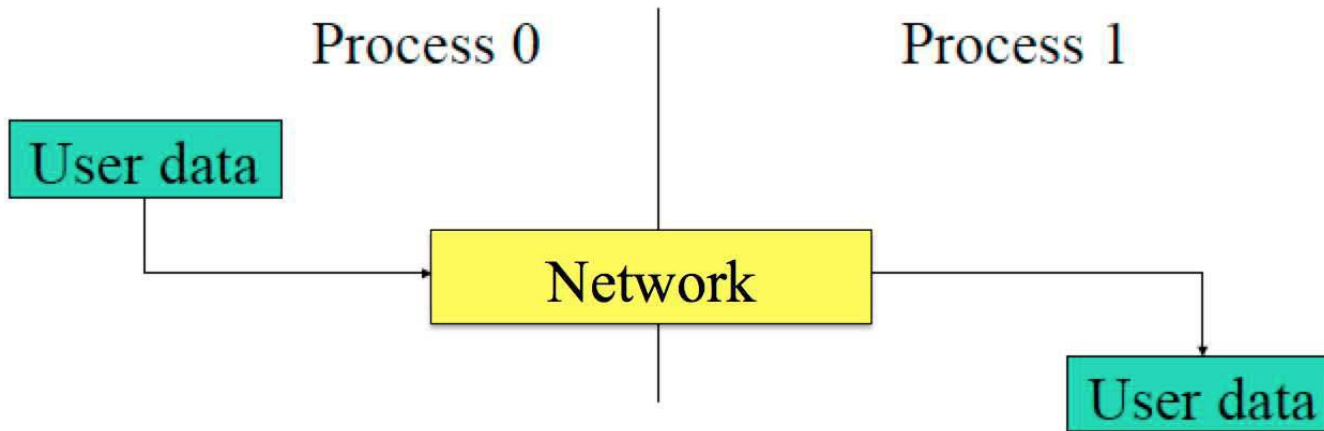
Standard Send

- the hardest to define.
- Its functionality is left open to the implementer in the MPI-1 and -2 specifications.
- Standard send, however, is intended to take advantage of specific optimizations that may be available via system enhancements.
- The strategy is to treat large and small messages differently.
- message buffering helps to reduce synchronization overhead, but it comes at a cost in memory.
- If messages are large, the penalty of putting extra copies in a buffer may be excessive and may even cause the program to crash
- Generally there is a threshold:
  - for intel MPI set by this env variable I_MPI_EAGER_THRESHOLD.

To make it clear:

Process 0 | Process 1

User data → Local buffer → Network → Local buffer → User data
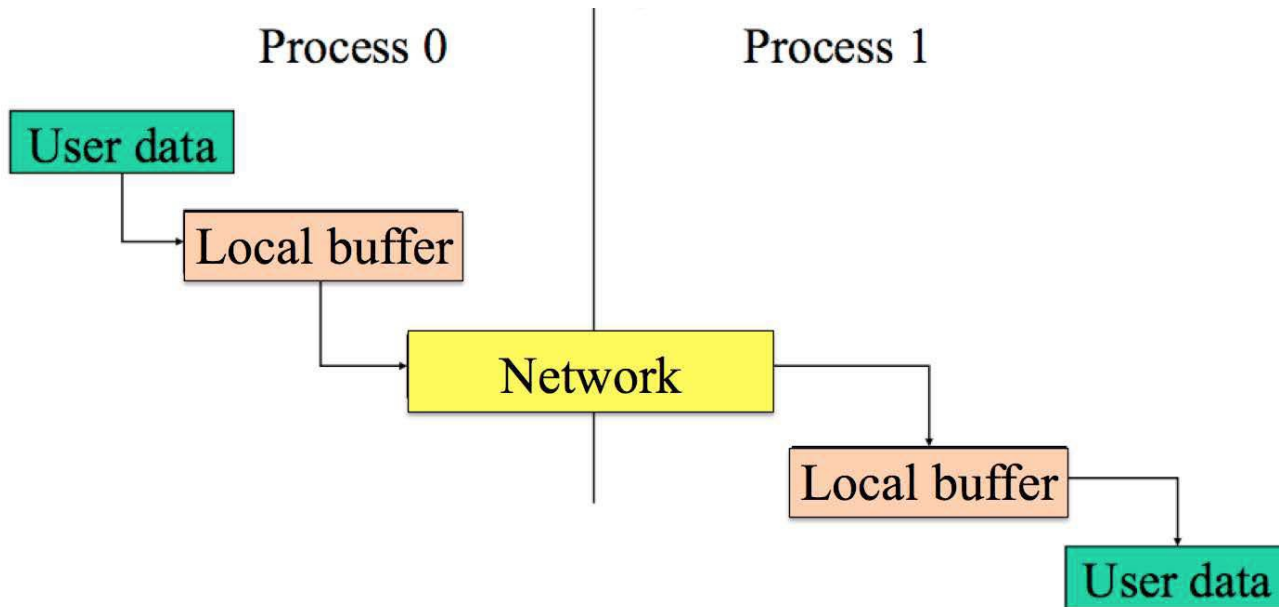
Small messages make use of system-supplied buffer

To make it clear:

Large message are really blocking

Choosing a communication mode

Mode	Advantages	Disadvantages
Synchronous	- Safest, therefore most portable - No need for extra buffer space - SEND/RECV order not critical	- Can incur substantial synchronization overhead
Ready	- Lowest total overhead - No need for extra buffer space - SEND/RECV handshake not required	- RECV *must* precede SEND
Buffered	- Decouples SEND from RECV - no sync overhead on SEND - Programmer can control size of buffer space - SEND/RECV order irrelevant	- Copying to buffer incurs additional system overhead
Standard	- Good for many cases - Compromise position	- Protocol is determined by MPI implementation

Non blocking communication

- Nonblocking calls merely initiate the communication process.
- The status of the data transfer, and the success of the communication, must be verified at a later point in the program.
- The purpose of a nonblocking send is mostly to notify the system of the existence of an outgoing message: the actual transfer might take place later.
- It is up to the programmer to keep the send buffer intact until it can be verified that the message has actually been copied someplace else.
- Likewise, a nonblocking receive signals the system that a buffer is prepared for an incoming message, without waiting for the actual data to arrive.
- In either case, before trusting the contents of the message buffer, the programmer must check its status

Non blocking send

- Not really a "fire and forget" mechanism.

- Three reasons to check the status
  - You may want to modify/re-use the buffer
  - You may want to release resources involved.
  - You may want to check if the communication was successful

Non blocking call syntax

- Sa syntax as the blocking ones, with two exceptions:

- The letter "I" (think of "initiate") appears in the name of the call, immediately following the first underscore: e.g., MPI_Irecv.

- The final argument is a handle to an opaque (or hidden) request object that holds detailed information about the transaction. The request handle can be used for subsequent Wait and Test calls.

Non blocking send/receive (Fortran)

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

tag   (INTEGER) number identifying the message

comm   (INTEGER) communicator of the sender and receiver

req   (INTEGER) output, identifier of the communications handle

ierr   (INTEGER) output, error code (if **ierr=0** no error occurs)

Non blocking send/receive (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 dest, int tag, MPI_Comm comm, MPI_Request *req);
```

Request handle

- All nonblocking calls in MPI return a *request handle* in lieu of a status variable.

- The purpose of the handle is to let you check on the status of your message at a later time.

- In MPI, the status variable becomes defined only after your message data is ready.

- Analogy: buzzer at the fastfood
  - You submit your request at the counter and a buzzer is given to you..

Function for non blocking communication

- MPI_Test - You go to the counter every two minutes ask the guy if your buzzer is not working.

- MPI_Wait - You hang around at the table with your friends till the buzzer sounds

- MPI_Request_free - You give back the buzzer without learning the status of your food. (Perhaps you've canceled your order; or perhaps the order is obviously ready and you already know what's in it, so you can just grab it without further checking.)

Waiting for completion…

Fortran:

`MPI_WAIT(req, status, ierr)`

- A call to this subroutine cause the code to wait until the communication pointed by req is complete.

- **Req (**INTEGER) input/output, communication handler (initiated by **MPI_ISEND** or **MPI_IRECV**).

- **Status** (INTEGER) array of size **MPI_STATUS_SIZE**,

- if **Req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

- **ierr**   (INTEGER) output, error code (if **ierr=0** no error occours).

C:

`int MPI_Wait(MPI_Request *req, MPI_Status *status);`

Testing for completition

Fortran:

MPI_TEST(req, flag, status, ierr)

- A call to this subroutine sets flag to .true. if the communication pointed by req is complete, sets flag to .false. otherwise.

- **Flag**(LOGICAL) output, .true. if communication req has completed .false. otherwise

- **Req (**INTEGER) input/output, communication handler (initiated by **MPI_ISEND** or **MPI_IRECV**).

- **Status** (INTEGER) array of size **MPI_STATUS_SIZE**,

- **ierr** (INTEGER) output, error code (if **ierr=0** no error occours).

C:

int MPI_Wait(MPI_Request *req, int *flag, MPI_Status *status);

Pros and Cons of Non-Blocking Send/ Receive

- Non-Blocking communications allows the separation between the initiation of the communication and the completion.

- Advantages:
  - between the initiation and completion the program could do other useful computation (latency hiding).

- Disadvantages:
  - the programmer has to insert code to check for completion.

Deadlock

- Often encountered in parallel processing..
- In communications, a typical scenario involves two processes wishing to exchange messages: each is trying to give a message to the other, but neither of them is ready to accept a message.
- the deadlock phenomenon is most common with blocking communication.
- It is relatively easy to get into a situation where multiple tasks are waiting for events that haven't been initiated yet—and never can be.

MPI: a case study

Problem: exchanging data between two processes

```
If( rank == 0 ) then
    Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                    MPI_comm_world, status, error )
  Else If( rank == 1 )then
    Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                    MPI_comm_world, status, error )
End If
```

DEADLOCK

Solution A

```
If( rank == 0 ) then
    Call MPI_Bsend( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                    MPI_comm_world, status, error )
  Else If( rank == 1 )then
    Call MPI_Bsend( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                    MPI_comm_world, status, error )
End If
```

Requires a copy therefore is not efficient
for large data set memory problems

Solution B

Use non blocking SEND : isend
send go back but now is not safe to change the buffer

```
If( rank == 0 ) then
    Call MPI_Isend( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                    MPI_comm_world, status, error )
  Else If( rank == 1 )then
    Call MPI_Isend( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
    Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                    MPI_comm_world, status, error )
 End If
Call MPI_wait( REQUEST, status ) ! Wait until send is complete
```

1 A handle is introduced to test the status of message.
2. More efficient of the previous solutions

Solution C

Just rearrange the order of call and all is done

```
If( rank == 0 ) then
     Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
     Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                    MPI_comm_world, status, error )
  Else If( rank == 1 )then
     Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                    MPI_comm_world, status, error )
     Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
End If
```

the most efficient one and the recommended one

Strategies to avoid deadlock

1. Go with buffered mode

Use buffered sends so that computation can proceed after copying the message to the user-supplied buffer. This will allow the receives to be executed. This method resolves deadlock problems the same way that strategy 2 does, though with somewhat different performance characteristics.

2. Use nonblocking calls

Have each task post a nonblocking receive before it does any other communication. This allows each message to be received, no matter what the task is working on when the message arrives or in what order the sends are posted.

3. Arrange for a different ordering of calls between tasks

Have one task post its receive first and the other post its send first. That clearly establishes that the message in one direction will precede the other.

4. Try MPI's coupled Sendrecv methods

MPI_Sendrecv and MPI_Sendrecv_replace can be elegant solutions. In the _replace version, the system allocates some buffer space to deal with the exchange of messages.

Often, deadlock is simply a result of not carefully thinking out your parallelization scheme so solution 1 should always be your first stop.

SendRecv / SendRecv_replace

- functions that combine a blocking send and receive into a single API call.

- This is useful for "swap" communications:
  - messages must be exchanged between processes;
  - "chain" communications, where messages are being passed down the line.

- There are two versions of these combined calls which differ in the number of buffers.

Sendrecv

int MPI_Sendrecv (\
  const void *sendbuf, int sendcount, \
   MPI_Datatype sendtype, int dest, int sendtag, \
   void *recvbuf, int recvcount, MPI_Datatype recvtype, \
   int source, int recvtag, \
   MPI_Comm comm, MPI_Status *status)

Parameters:
*sendbuf: The address of the send buffer.
sendcount: Number of elements in the send buffer.
sendtype: Data type of the send buffer.
dest: Destination process number (rank number).
sendtag: Sent message tag.
*recvbuf: The memory address of the receiving buffer.
recvcount: Number of elements in the receive buffer.
recvtype: Data type of the receiving buffer.
source: Source process rank number.
recvtag: Receiving message tag.
comm: MPI communicator.
*status: Status of object.

MPI Tutorial: part 1

1. Compile/Run and understand usage of MPI programs
   - mpi_pi.c
   - hello_world.c/f90

2. relative performance of the four communication modes (synchronous, ready, buffered, and standard)
   -

3. reduce the synchronization overhead replacing blocking receive with a nonblocking receive

4. fix deadlock problems on a ring

5. play with different MPI_send call on mpi_pi.c