Advanced Methods for Scientific Computing (AMSC)
Lecture title: Introduction MPI

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

What is MPI?

The Message Passing Interface (MPI) is a standard protocol for parallel programming based on the message passing paradigm. It is mainly (but not esclusively) used on distributed parallel environments.

However, it can also run on multicore architectures, but the paradigm is still based on message passing: parallel tasks are independent processes that exchange data using explicit calls to utilities that send/receive messages.

Normally you need to completely restructure a scalar version of the code to port it to MPI.

Where can I find information on MPI?

The web is plenty of information. Beware however to distinguish between the definition of the standard, provided by the MPI FORUM, and its implementation. The two main implementations of MPI are mpich and openMPI. In this course we use the latter, however the differences are minimal.

Among online reference manuals, we recommend RookieHPC, complete and with working examples.

The book by P.S.Pacheco and M. Malensek (suggested for this course) is also a great source of information. Another interesting book is Parallel programming with MPI and OpenMP by M.J.Quinn.

The courses of CINECA (the Italian supercomputing center), like this one, can also be a source of useful information (and they have been exploited for these slides).

Compilation

Both mpich and openMPI implementations provide commands to compile and run a program that uses MPI without the need of recalling all libraries that have to be linked or the path of the MPI include files. Luckily, they have the same name. To compile a MPI code written in C you do

```
mpicc [possible options] files [-c| -o executable]
```

For c++ we have

```
mpic++ [possible options] files [-c| -o executable]
```

The options are analogous to those of the corresponding compiler. Example: to compile a c++ MPI code using the c++20 standard and activating all major warnings:

```
mpic++ -std=c++20 -Wall main.cpp -o main
```

Running a MPI-based program

Also launching a MPI program requires a special command. An MPI program may run on a single multi-core machine or on clusters, even heterogeneous clusters (like PCs connected with a fast network). Therefore, launching may be complex and may require a file (the `hostfile`) that specifies the architecture and how the tasks are distributed.

However in a single multi-core machine the command may be as simple as

```
mpiexec -n <number of processes> executable
```

Often, `mpirun` may be used instead of `mpiexec`. With this command we ask to run the program `executable` in parallel using the specified number of processes.

If you run on a single multicore PC you may at have at most a number or processes equal to the number of physical cores. If the CPU supports hyper-threading, you may use the option `--use-hwthread-cpus` to run up to the max number of virtual threads.

To avoid frustrations...

Unless the algorithm is almost embarrassingly parallel, the efficiency of parallelisation via message passing can be disappointing, when not negative! The reason is due to the cost of the communication, and also the size of the problem to be solved. To see the problem let assume that our algorithm has a cost proportional to the size n of the problem, $c_p = an$, while communication cost c_c is proportional to the number of processes p,i.e. $c_p = cp$ (e very simplistic assumption), and that the code can be fully parallelised (which is almost never true but it simplifies things). The parallel efficiency will have a bound of the type

$$eff \leq \frac{1}{\frac{cp}{an} + 1}$$

So we can hope to get a reasonable efficiency only if we have a computer with good communication hardware (small c) and $n >> p$.

Types of communication

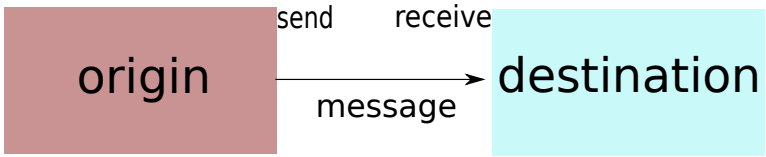
Before delving in the details of MPI, let's recall the principal ways two tasks/processes may communicate:

- Point-to-Point: the basic way where a task send a message and another receives it.
- One-to-All: a task sends data to all others.
- All-to-One: all tasks send data to a target task that has the role to collect the pieces.
- All-to-All; all tasks send data to the other tasks.

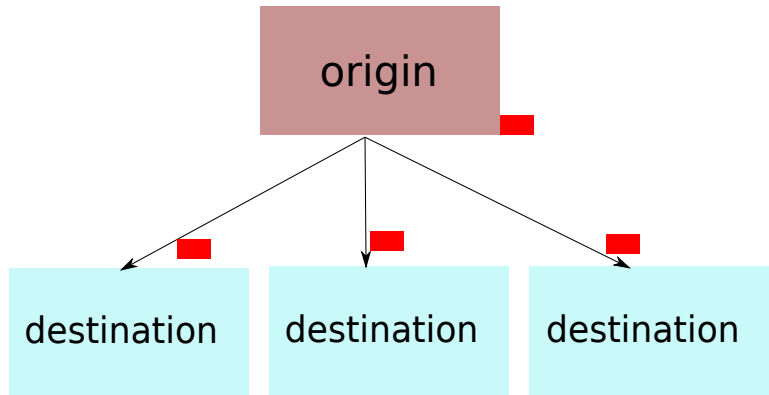
MPI spares the user the detail on how to implement those communication patterns in practice, and in particular which strategy to choose in case of collective communication.

Communication patterns

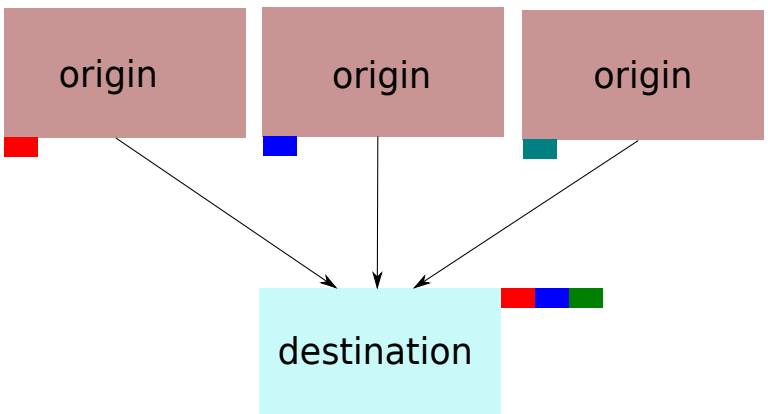
Point-to-point communication



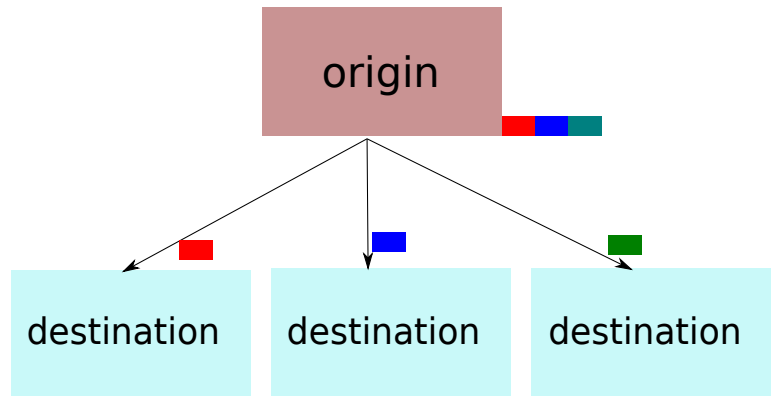
One-to-all communication
Broadcast



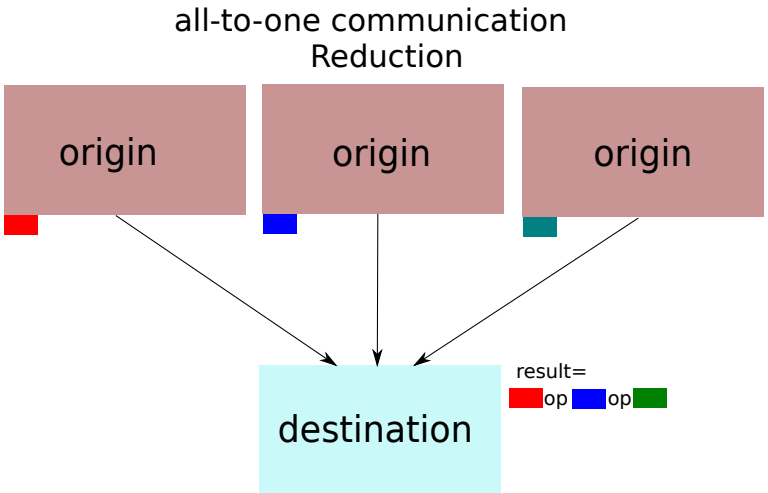
all-to-one communication
Gather

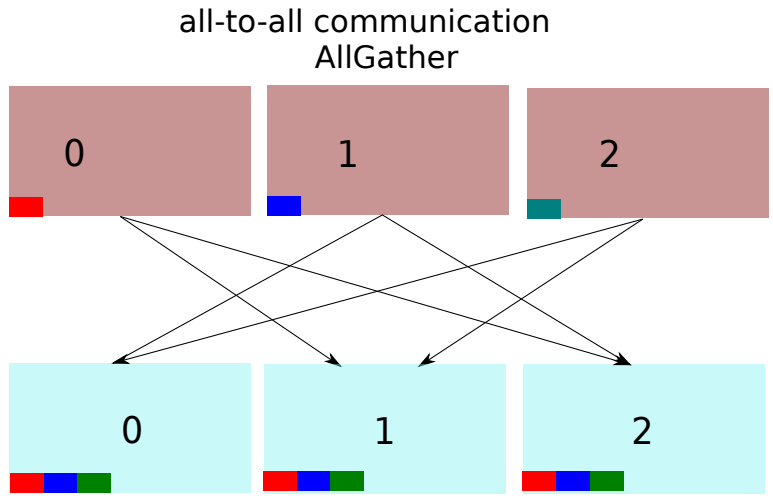


One-to-all communication
Scatter



Communication patterns





In a reduction process, the data collected from all origins is operated upon with a binary operator, a standard one (sum, multiplication, etc.) or a user defined one (advanced feature). Note that destination may coincide with one of the origins (and indeed it is usually the case).

A first MPI program

This program let each process to print on the screen

```
#include <mpi.h>
#include <iostream>
#include<string>
int main(int argc, char **argv)
//Initialize MPI and define the communicator.
MPI_Init(&argc, &argv);
MPI_Comm mpi_comm = MPI_COMM_WORLD;
//Get the number of processes.
int mpi_size;
MPI_Comm_size(mpi_comm, &mpi_size);
//Get the rank of current process.
int mpi_rank;
MPI_Comm_rank(mpi_comm, &mpi_rank);
// printout
std::cout << "Hello world! This is rank " << mpi_rank << " out of "
<< mpi_size << " processors"<< std::endl;
// Finalize MPI.
MPI_Finalize();
}
```

As you may note, all MPI functions and variables start with `MPI_`.

Single Program Multiple Data

A characteristic of a MPI program is that it consists of several identical instances of a single code, each run by a different process. If a part of the program should be executed by just one process, it is handled with an **if** statement (it is not the case of the simple program above, where every process executes the same instructions).

This is an important issue, since it may be counterintuitive: when you launch a MPI program you are in fact launching the same program on the given number of processes. Each process run the same code, independently. The only way for the processes to communicate is through messages sent using MPI functions.

In MPI, the term `rank` indicates an integer that identifies a single process: if the program is run with 4 processes, `rank` goes from 0 to 3.

Initialising and finalising MPI

The function

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

is the first command you should use in a MPI-based program. It initialises all the internal structures needed by MPI to work. The two arguments are normally the same as those of the main(), but in fact they are not normally used, and can be replaced by the NULL pointer (or **nullptr**). The function returns a status (like all MPI functions), if it is equal to MPI_SUCCESS MPI has been initialised successfully.

```
int MPI_Finalise()
```

should be the last MPI function to be called. MPI_Init and MPI_Finalize don't need to be called from the main program, even if this is often the case since it is easier to ensure that they are the first and last MPI functions called.

Communicators

In MPI a communicator is a set of processes that can send messages each other. Users can define their own communicators, but MPI provides by default the MPI_COMM_WORLD communicator, which just contains all processes, and normally is the only one needed!. The type of a MPI communicator is MPI_Comm so MPI_Comm mpi_comm = MPI_COMM_WORLD just defines mpi_comm as the default (and only) communicator.

The two functions

```
int MPI_Comm_Size(MPI_comm c, int * siz);
int MPI_Comm_Rank(MPI_comm c, int * r);
```

are used to extract the number of processes (siz) in the communicator c, and the rank of the current process (r).

Executing the program

A (slightly modified) version of the program is in Parallel/MPI/HelloWorld. If you compile it (with make) and run with

```
mpirun –n 3 ./main_hello_world
```

you get

```
Hello world! This is rank 1 out of 3 processors, on minnie.
Hello world! This is rank 2 out of 3 processors, on minnie.
Hello world! This is rank 0 out of 3 processors, on minnie.
```

The last part is linked to the use of MPI_Get_processor_name(proc_name,&name_length) that returns the name of the processor, minnie in this case (the name of my PC!).

An important note

You may note that the order of the messsages may not correspond to the rank order, and sometimes the messages may even be garbled!.

Processes are independent: there is no reason for a particular process to reach the line of the code that prints the message before any other process. And two (or more) processes may even reach it at the same time!

We will see later on how to create a critical section, i.e. a part of code accessed by one process at a time.

MPI data types

To ensure consistency across architectures MPI uses special global variables to specify the datatype. Here, a convenient table that gives the association between MPI datatypes and C/C++ datatypes:

MPI datatype	C/C++ datatype
MPI_CHAR	char
MPI_WCHAR	wchar_t
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_SIGNED_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BOOL	-/bool
MPI_COMPLEX	-/std::complex<float>
MPI_DOUBLE_COMPLEX	-/std::complex<double>
MPI_LONG_DOUBLE_COMPLEX	-/std::complex<long double>
MPI_BYTE	-/byte
MPI_PACKED	-/-

mpi_utils.hpp

In the file Parallel/Utilities/mpi_utils.hpp you have some utilities that can make the match between C++ types and MPI types easier:

- The function mpi_typeof() that returns the MPI_Datatype corresponding to a native c++ type, so you may not need to remember the table above by heart. Note that you have to pass an object as argument, maybe just a temporary created with the default constructor! Here, the mpi type is deduced from the type of the elements in the vector v:

```
MPI_SEND(v.data(),10,mpi_typeof(v[0]),0,0,MPI_COMM_WORLD);
```

- The global variable MPI_SIZE_T that represents the MPI datatype associated to std::size_t.

Point-to-point communication

The two main tools for p2p communication in MPI are MPI_Send() ans MPI_Recv(), for sending and receiving messages, respectively:

```
int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
int recipient, int tag, MPI_Comm comm);
int MPI_Recv(void* buffer, int count, MPI_Datatype datatype, int sender,
int tag, MPI_Comm comm, MPI_Status* status);
```

buffer The buffer to send/ in which receive the message.
count The number of elements in the buffer. The number of elements in the message to receive must be less than or equal to that value.
datatype The type of one buffer element (see previous slide).
recipient The rank of the recipient MPI process. sender The rank of the sender MPI process. If there is no restriction on the sender's rank, `MPI_ANY_SOURCE` can be passed
tag A tag (an integer) that identifies the message. In the reception process one may use `MPI_ANY_TAG` if not tag is required.
comm The communicator in which the communication takes place.
status The status of a receive operation. Pass `MPI_STATUS_IGNORE` if unused.

`MPI_Status` is a `struct` with three members:`MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR`.

How does it work?

```
MPI_Send(buffer, 1, MPI_INT, q, 1, MPI_COMM_WORLD);
MPI_Recv(buffer, 1, MPI_INT, r, 1, MPI_COMM_WORLD);
```

If process with rank r sends a message to process with rank q process rank q will receive the message if and only if

- The communicators are the same;
- the send tag is the same as the receive tag (or `MPI_ANY_TAG` is given as tag);
- the `MPI_Recv` sender argument is equal to r or to `MPI_ANY_SOURCE`.

This way, you can selectively address your message. Let's have a look at the example in Parallel/MPI/Blocking

Blocking communication

In the code in Blocking, two processes send each other a message (just an int) and they report the "ping-pong" between them on the screen.
The process with rank 0 sends the message with

```
MPI_Send(&ping_pong_count, 1, MPI_INT, 1, 0, mpi_comm);
```

to process 1, which is receiving it with

```
MPI_Recv(&ping_pong_count, 1, MPI_INT, 0, 0, mpi_comm, &status);
```

The tag of the message is here 0.

MPI_Send and MPI_Recv implement blocking communication: it means that they return to the calling program only when the send (resp. receive) operation has been completed.

Therefore, you must be careful not to get into a deadlock!.

Deadlock

The code main_deadlock.cpp in Parallel/MPI/NonBlocking is a program that runs with two processes and falls into a deadlock because both processes issue a MPI_Recv() before the MPI_Send().

Consequently, both processes stop, waiting to receive a message the other process is unable to send since it is itself waiting to receive!.

A possible solution is to use a non-blocking communication. In MPI non-blocking communication functions have the same name of the corresponding blocking one, prepended by an `I`, which stands for Immediate return.

For point-to-point non-blocking communication we thus have MPI_ISend() and MPI_Irecv(), with a syntax rather similar to that of the blocking counterparts.

Non-blocking point-to-point communication

```
int MPI_Isend(const void* buffer, int count, MPI_Datatype datatype,
              int recipient, int tag, MPI_Comm comm, MPI_Request* request);
int MPI_Irecv(void* buffer, int count, MPI_Datatype datatype,
              int sender, int tag, MPI_Comm comm, MPI_Request* request);
```

As you can see, there is an additional parameter, a MPI_Request. It is an handle that allows, with the use of appropriate functions, to test the status of the communication (has it been completed?) and to eventually put the process in a wait state until communication has completed.

Indeed, since the functions return immediately, we need a way to test whether the communication has completed correctly.

To the purpose, we have MPI_Test() and MPI_Wait().

MPI_Test() and MPI_Wait()

```
int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
int MPI_Testall(int count, MPI_Request requests[], int* flag,
                MPI_Status statuses[]);
```

They test if the request(s) have been completed, the result is reported in flag, which is equal to 1 if the request(s) have completed, 0 if not. The second version takes an array of requests and return a true flag if all have completed. The status may be set to MPI_STATUS_IGNORE, or, respectively MPI_STATUSES_IGNORE if it is not used.

```
int MPI_Wait(MPI_Request* request, MPI_Status* status);
int MPI_Waitall(int count, MPI_Request requests[], MPI_Status statuses[]);
```

In this case the calling process is put in a wait state until the request(s) have completed. It is the tool used to ensure synchronization when dealing with non-blocking communications.

An example

In Parallel/MPI/NonBlocking/main_non_blocking.cpp we have an example that uses non-blocking point-to-point communication. It also shows the use of the method data() of a standard vector (in fact we could have used an array). Again, it is just an example that works only with 2 processes. They send each other two messages with different tags.
We use non-blocking communication, we test if the communication has completed with MPI_Testall, and we synchronize using MPI_Waitall.

```
MPI_Irecv(to_receive.data(), to_receive.size(), MPI_DOUBLE, partner_rank,
          tag_receive, mpi_comm, &requests[0]);
MPI_Isend(to_send.data(), to_send.size(), MPI_DOUBLE, partner_rank, tag_send,
          mpi_comm, &requests[1]);
// Test for all requests to complete.
MPI_Testall(mpi_size, requests.data(), &ready, MPI_STATUS_IGNORE);
// Wait for all communications to finish.
MPI_Waitall(mpi_size, requests.data(), statuses.data());
```

Blocking or non-blocking?

We need to avoid deadlocks when doing point-to-point communications, and in more complex situations than the one seen in these first examples it may be not easy.

In general, it is better to do a send before a receive, using a non-blocking send.

Non-blocking communications are computationally advantageous if you can perform some computations before overwriting the communication buffer. If the workload is slightly unbalanced your process may do something while waiting the communication to complete. All non-blocking MPI communications are provided by functions of the form MPI_Ixxx.

Another tool that reduces the risk of a deadlock is MPI_Sendrecv(), which combines a send and a receive.

MPI_Sendrecv()

MPI_Sendrecv() is a combination of an MPI_Send() and an MPI_Recv(). It can be seen as having both functions executed concurrently.

```
int MPI_Sendrecv(const void* buffer_send, int count_send,
    MPI_Datatype datatype_send, int recipient,
    int tag_send, void* buffer_recv, int count_recv,
    MPI_Datatype datatype_recv, int sender,
    int tag_recv, MPI_Comm comm, MPI_Status* status);
```

buffer_send The buffer to send. count_send The number of elements to send. datatype_send The type of one send buffer element. recipient The rank of the recipient MPI process. tag_send The tag of the send message. buffer_recv The buffer in which receive the message. count_recv The number of elements to receive. datatype_recv The type of one receive buffer element. sender The rank of the sender process. tag_recv The tag to require from the message. MPI_ANY_TAG if no tag is required. status The variable in which store the reception status. MPI_STATUS_IGNORE can be passed.

The use of MPI_Sendrecv()

MPI_Sendrecv() is used in point-to-point communications where each process has to send and receive data from other processes. The use of this function leaves to MPI compiler how to select the best strategy to avoid deadlocks.

MPI_PROC_NULL

MPI_PROC_NULL is a special macro which, if indicated as the source or destination rank of a mpi point-to-point communication function, makes the call return without any communication.
It may be handy when we need to select "no-communication".
Example: in this piece of code

```
...
dest_rank=MPI_PROC_NULL;
MPISend(&buffer,1,MPI_DOUBLE,dest_rank,1,MPI_COMM_WORLD);
...
```

no communication is performed.

Probing

Sometimes the receiver does not know the length of the message. How can we provide it to MPI_Recv()?
A possibility is the sender sending the lenght of the message beforehand, and often this is the preferred choice, but we have another possibility: probing.
A message sent MPI_Send() for instance, can be probed before it is received with the command

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

source The rank of the sender, which can be `MPI_ANY_SOURCE` to exclude it from message filtering.
tag The tag to require from the message. If no tag is required, `MPI_ANY_TAG` can be passed.
comm The communicator.
status The variable in which store the status corresponding to the message probed (if any), which can be `MPI_STATUS_IGNORE` if unused.

Getting the length from status: MPI_get_count

The status obtained by the probing normally contains the information we want. In particular, it contains the length of the message! We can extract it by using MPI_Get_count():

```
int MPI_Get_count(const MPI_Status* status, MPI_Datatype datat,int* count);
```

status The receive operation status to query.
datat The type of elements contained in the message.
count The number of elements in the message buffer.

Thus, count is the number of elements of type datat contained in the message waiting to be received.

In folder Parallel/MPI/Probe you have an example illustrating a possible use of MPI_Probe() and MPI_Get_count().

An example of MPI_Probe

```
std::vector<int> v;
...
if(my_rank ==0)
// sending a vector of integers to processs 1, tag=0
MPI_SEND(v.data(),v.size(),MPI_INT,1,0,MPI_COMM_WORLD);
else if (my_rank==1)
int amount;
MPI_Status status;
// check arrival of a message from proces 0 with tag 0
// (probing is blocking)
MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
// get the length
MPI_Get_count(&status, MPI_INT, &amount);
v.resize(amount); //make sure v can hold the msg
MPI_Recv(v, amount, MPI_INT, 0, 0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
...
```

Some collective communications: broadcast, scatter and reduce

Sometimes of the processes has the task of setting some data, maybe by reading from a file, which must be be identically distributed to all processes: this is a broadcast. Eventually, all processes will have a copy of the broadcasted data.

When instead, each process gets a specified portion of the data, we have a scatter.

Another typical situation is when data elaborated by each process has to be collected by a single process and operated upon, for instance by summing up all the collected pieces. This is a reduction process.

MPI provides specialised functions for broadcasting, scattering and reduction.

MPI_Bcast()

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
              int emitter_rank,MPI_Comm comm);
```

buffer The buffer containing the data to be broadcasted, if the process is the emitter, it will contain the broadcasted data otherwise.
count The number of elements in the buffer
datatype The type of an element in the buffer.
emitter_rank The rank of the process broadcasting the data.
comm The communicator.

You may note that the same buffer is used as source in the emitting process and as destination in the other processes. Eventually, all processes have the same data.

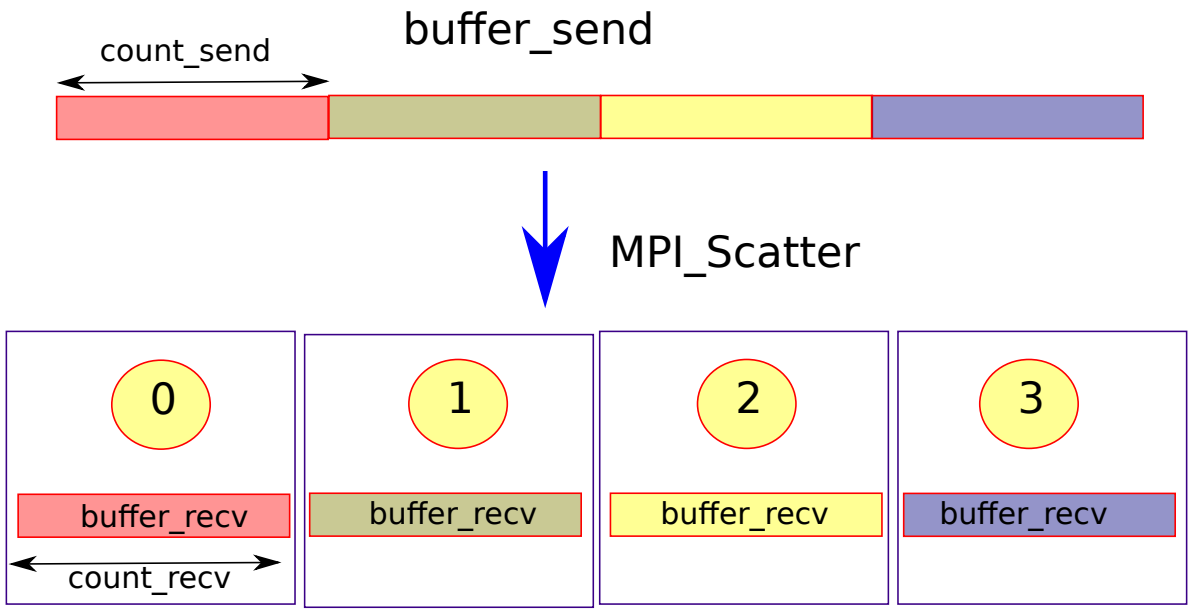
MPI_Scatter()

```
int MPI_Scatter(const void* buffer_send, int count_send,
                MPI_Datatype datatype_send, void * buffer_recv,
                int count_recv, MPI_Datatype datatype_recv, int root,
                MPI_Comm comm);
```

buffer_send The buffer containing the data to disptach from the root process.
count_sendThe number of elements to send to each process, not the total number of elements in the send buffer.
buffer_recv The buffer in which store the data dispatched.
count_recv The number of elements in the receive buffer.
datatype_recv The type of one receive buffer element.
root The rank of the process that dispatches the data to scatter.
comm The communicator.

For non-root processes, the send parameters.

How MPI_Scatter works



This form of MPI Scatter (we will see another form in a while) splits the buffer_send into p identical pieces of size count_send. Process of rank r gets the r-th piece. The reason we have to indicate a count_recv is becouse the receive data may be of a different type. In general, however, we expect count_recv to ne equal to count_send.

MPI_reduce()

```
int MPI_Reduce(void* send_buffer, void* receive_buffer,
               int count, MPI_Datatype datatype, MPI_Op oper,
               int root,MPI_Comm comm);
```

send_buffer The buffer containing the data to be reduced.
receive_buffer The buffer to store the result of the reduction.
count The number of elements in the send_buffer
datatype The type of an element in the buffer.
operation The binary associative operation to be performed for reduction.
root The rank of the process collecting the result of the reduction.
comm The communicator.

Typical operations are MPI_SUM, MPI_PROD, MPI_AND. Note that you need to initialize receive_buffer to the initial value. All precesses indicate receive_buffer, but only root gets the results.

An example: parallel composite Simpson quadrature

Simpson composite quadrature approximates the integral $\int_a^b f(x)dx$ by subdividing the interval into n subintervals of length $h = (b-a)/n$ and applying the formula

$$I = \frac{h}{6}\sum_{i=0}^{n-1}(f(x_i) + 4f(x_{i+1/2}) + f(x_{i+1})).$$

where $x_i = a + ih$ and $x_{i+1/2} = a + (i + 1/2)h$.
We want to write a parallel implementation using MPI.

Analysis of the problem

- Partition the problem into tasks: here a basic task is to find the integral on a single subinterval by computing $f(x_i) + 4f(x_{i+1/2}) + f(x_{i+1})$, another is to collect the sum.
- Aggregate tasks that can be performed independently. Clearly each process may perform a partial sum of the basic formula applied to n/p elements, being p the number of processes (if n is not a multiple of p we try to split them so that the max difference is one).
- Map tasks to processes. The root process has to perform some preprocessing, reading some data, determine how the elements are split, and distribute the info to all processes. Each process performs the partial sum. The root process does a reduction to get the global sum.

The code

The code is in the folder Parallel/MPI/Simpson/.
We have nucleated the generic formula in a function contained in Parallel/MPI/Simpson/SimpsonRule.hpp. The integrand is defined in the main() program (it can be done better, but here we keep things simple).
Some data (interval ends, total number of subintervals) is read form a file in json format, using an utility in Utilities forder.
We time the computation using the Utilities/chrono.hpp utility.

Look at the code and at the README.md file. Notable things are the use of a tuple to pack some data, and the use of the method data() of a std::vector.

Another flavour of MPI reduce: MPI_Allreduce()

In the code for Simpson composite rule, we have let the root process (0) collect the partial sum and print the result, using MPI_Reduce(). Only the root process has the reduced value. There are however situations where we want all processes to get the reduced value. We need

```
int MPI_Allreduce(const void* send_buffer, void* receive_buffer,
int count, MPI_Datatype datatype, MPI_Op operation, MPI_Comm comm);
```

As you can see, the parameters are almost the same, a part the absence of the root process, since now all processes get the reduced value.

Don't use MPI_Allreduce() when MPI_Reduce() is enough, since it is more expensive.

Gathering data

Gathering is indeed the opposite of Scattering.

Often a parallel algorithm for large scale computations is made by splitting the data (vectors/matrices) among processors, operating on the split data locally and then gathering the whole information by letting one process, typically process 0, to collect the data from all processors and build a global vector/matrix.

Let's see first the basic MPI_Gather() facility.

MPI_Gather()

```
int MPI_Gather(const void* buffer_send, int count_send,
               MPI_Datatype datatype_send, void* buffer_recv,
               int count_recv, MPI_Datatype datatype_recv,int root,
              MPI_Comm comm);
```

buffer_send The buffer containing the data to send.
count_send The number of elements in the send buffer.
datatype_send The type of one send buffer element.
buffer_recv The buffer in which store the gathered data for the root process.
count_recv The number of elements per message received, not the total number of elements to receive from all processes altogether.
datatype_recv The type of one receive buffer element.
root The rank of the process that collect the gathered data.
comm The communicator in which the gather takes place.

For non-root processes, all receiving parameters are ignored.

Scattering and Gathering with irregular data

The gather and scatter utilities seen so far assume that every local process trasmits/receives a buffer of the same lenght. But this is not always the case.

For example,let's assume that the root process has to scatter to the p processes a vector of size n. If n is not a multiple of p the set of elements to be sent to each processor is not the same. We may decide that the first $n \mod p$ processes gets $n/p + 1$ elements, and the remaining one n/p.

We cannot use the MPI_Scatter() function. A similar problem happens if we gather the local vectors, we cannot use MPI_Gather(). Luckily, MPI has the solution.

With MPI_Scatterv() and MPI_Gatherv() the number of elements dispatched from/to the root process can vary, as well as the location from which to load/store these elements in the root process buffer.

MPI_Scatterv()

```
int MPI_Scatterv(const void* buffer_send, const int counts_send[],
                 const int displacements[], MPI_Datatype datatype_send,
                 void* buffer_recv,int count_recv,
                 MPI_Datatype datatype_recv, int root,MPI_Comm comm);
```

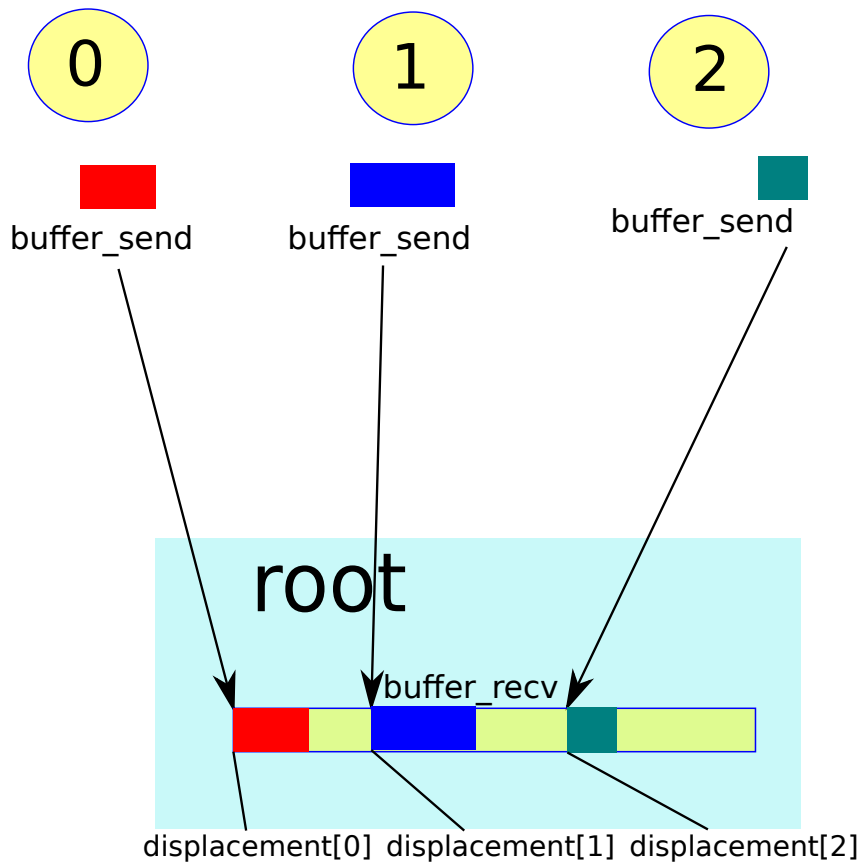
buffer_send The buffer containing the data to dispatch from the root process.
counts_send An array containing the number of elements to send to each process, not the total number of elements in the send buffer.
displacements An array containing the displacement to apply to the message sent to each process. Displacements are expressed in number of elements, not bytes.
datatype_send The type of one send buffer element.
buffer_recv The buffer in which store the data dispatched.
count_recv The number of elements in the receive buffer.
datatype_recv The type of one receive buffer element.
root The rank of the process that dispatches the data.
comm The communicator.
For non-root processes, the send parameters are ignored.

MPI_Gatherv()

```
int MPI_Gatherv(const void* buffer_send, int count_send,
MPI_Datatype datatype_send, void* buffer_recv, const int* counts_recv,
const int* displacements, MPI_Datatype datatype_recv,
int root, MPI_Comm comm);
```

buffer_send The buffer containing the data to send to the root process.
count_send The number of elements to send.
datatype_send The type of one send buffer element.
buffer_recv The buffer in which the root process stores the gathered data.
counts_recv An array containing the number of elements in the message to receive from each process, not the total number of elements to receive from all processes.
displacements An array containing the displacement to apply to the message received from each process. Displacements are expressed in number of elements, not bytes.
datatype_recv The type of one receive buffer element.
root The rank of the process that gathers the data.
comm The communicator.
For non-root processes, the receive parameters are ignored.

MPI_Gatherv


0
1
2
buffer_send
buffer_send
buffer_send
root
buffer_recv
displacement[0]
displacement[1]
displacement[2]

Explanation

MPI_Gatherv and MPI_Scatterv are very general, they allow to place the gathered elements at specified location in the receive buffer, or, conversely, to send from the send buffer non contiguous pieces. This is done by combining the information in counts_recv (respectively counts_send) and `displacements`.

However, on most cases we have displacements that define contiguos positions. Just to clarify, lets suppost the root process of rank 0 has generated a vector v with n elements, that need to be scattered to mpi_size processes. Let's sketch the program.

Scattering a vector v of size n

```
std::vector<int> counts_send;
std::vector<int> displacements;
if(my_rank ==0) {// root has to prepare some data
  counts_send.resize(mpi_size);
  displacements.resize(mpi_size,0);// init. by zero
  auto chunk=n/mpi_size; // integer division
  auto rest=n%mpi_size; // the rest of the division
  for (int i=0;i<mpi_size;++i){
    counts_send[i]=i<rest?chunk+1:chunk;
    if(i>0)displacements[i]=displacements[i-1]+counts_send[i-1];
  }
}
int local_size; // each process gets local size with normal scatter
MPI_Scatter(counts_send.data(),1,MPI_INT, &local_size,1,MPI_INT,0,mpi_comm);
//each process gets localsize its chunk of v
std::vector<double> local_v(local_size);
MPI_Scattev(v.data(),counts_send.data(),displacements.data(),
            MPI_DOUBLE, local_v.data(),local_size, MPI_DOUBLE,
            0, mpi_comm)
v.clear;v.shrink_to_fit();// we can eliminate v is not need anymore
```

Got it?

If $n = 1000$ and mpi_size=3, then

```
counts_send={334,333,333};
displacements={0,334,667};
```

The first 334 elements of v go to local_v of process 0, the second 333 to local_v of process 1, and so on...

In folder Parallel/MPI/VectorSplit you have an example of the use of MPI_Gatherv(): vectors produced locally are dispatched to root and concatenated to form a global vector.

In Parallel/Utilities/partitioner.hpp you have several utilities that help splitting vectors and matrices (as seen in an introductory lecture).

All-to-all communications

With a gather, only the root process gets the gathered data. MPI provides MPI_Allgather() and MPI_Allgatherv() as all-to-all counterpartrs of MPI_gather() and MPI_gatherv().

```
int MPI_Allgather(const void* buffer_send, int count_send,
                  MPI_Datatype datatype_send, void* buffer_recv,
                  int count_recv, MPI_Datatype datatype_recv,
                  MPI_Comm comm);int
MPI_Allgatherv(const void* buffer_send, int count_send,
               MPI_Datatype datatype_send, void* buffer_recv,
               const int* counts_recv,const int* displacements,
               MPI_Datatype datatype_recv,MPI_Comm comm);
```

The meaning of the parameters are equivalent to those of the standard gather counterparts. We do not have the root process, since here all processes get the final result, obtained by concatenating the contributions from each process.

Barriers

Sometimes it is necessary to syncronize the processes by setting a barrier: a function that blocks all MPI processes in the given communicator until they all call the function.
Indeed, we have

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

to do the job.

Packing data

It is normally better to reduce the number of communications. So if we need, for instance, to send three doubles and 2 ints we can wrap the doubles and the ints into 2 arrays and have just two sends, or, even better, create a wrapper of all data using the MPI_type_create_struct facility and have just one communication.

In C++, however, we can use also a tuple to pack elements of heterogeneous type. Provided the packed elements are either Plain Old Data (int, double, etc), aggregates or in general trivially copyable objects, the resulting tuple is trivially copyable, so can be serialized trivially.

An example using send and receive

This is just an example to show the technique, where process 0 sends some data packed in a tuple to process 1.

```
double a, b,c,
std::array<int,2> v;

if (my_rank==0)
{
  // computes a,b,c and v
  std::tuple<double,double,double,std::array<double,2>> pack=
  {a,b,c,v};
  // size in bytes
  int pack_siz=sizeof(pack);
  MPI_Send(&pack,pack_siz,MPI_BYTE,1,0,MPI_COMM_WORLD)
} else
{
 std::tuple<double,double,double,std::array<double,2>> pack;
 int pack_siz=sizeof(pack);
 MPI_Recv(&pack,pack_siz,MPI_BYTE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 std::tie(a,b,c,v)=pack;
}
```

Data packed in the tuple is serialized as a buffer of bytes.

Critical section

A critical section is a part of code which should be executed by one process at a time. MPI has no facilities to create a critical section Here we show a trick where a critical section is created to let processes print a message one at a time:

```
MPI_Comm mpi_comm = MPI_COMM_WORLD;
MPI_Comm_rank(mpi_comm, &mpi_rank);
MPI_Comm_size(mpi_comm, &mpi_size);
...
int rank=0;
MPI_Barrier(MPI_COMM_WORLD);
while (rank < mpi_size) // all processes carry the loop
    {
     if(mpi_rank==rank) //but only one enters
     { // THIS IS THE CRITICAL SECTION
       std::cout<<"Process rank="<< mpi_rank<<" is printing\n";
     }
     rank++;
     MPI_Barrier(mpi_comm);// needed to guarantee rank order
    }
```

If you take out the last MPI_Barrier() the processes may enter the critical section in any order (but the code is more efficient). See the code in Parallel/OneAtATime.

The end

With this slide we complete this overview of MPI. Of course, many more facilities are present: tools for i/o, some functions that combine 2 operations in a single function,like MPI_Reduce_scatter(), user defined communicators, tools to pack heterogeneous data in a single buffer, etc. etc.

You may find everything in a good manual, what we have seen so far is however enough in most cases.

We also mention that MPI has a FORTRAN interface as well.