Advanced Programming for Scientific Computing (PACS)
Introduction MPI

Alberto Artoni, Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

What is MPI?

MPI: **M**essagge **P**assing **I**nterface.
MPI is an *Application Programming Interface* (API):

- Specify how to call routines
- Specify what routines should do

...but the user don't know:

- how routines are implemented
- how routines are implemented by each vendor

Available for many languages: Fortran, C, Python.

What is MPI?

MPI is a standard for parallel programming.

- Each processor has its own memory. Memory is hence distributed.
- Data communication or synchronization is explicit and occurs via function calls

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, *An introduction to Parallel Programming (2nd Edition)* is also a great source of information.

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).

When to use MPI?

- high performance: each implementation is in principle optimized for the hardware on which it runs
- portability and standardization: MPI has been implemented for almost every architecture
- availability: many implementations are available.

- Most serial programs need to be completely re-written
- High memory overheads

Simplest paradigm: SPMD

Single **P**rogram **M**ultiple **D**ata model.

- Each task will run exactly the same code.
- MPI tasks are started when the program is executed.
- Each task has its *own local memory*.
- Communication shares the information between the tasks.
- Synchronization happens to ensure the parallel output is correct.

Example: bash arguments

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Number of arguments: %i \n", argc);

   for (int i=0; i<argc; i++)
   {
      printf("Arguments: %s \n", argv[i]);
   }

   printf("Hello World! \n");
}
```

`argc`: number of arguments.

`argv`: vector with the bash arguments.

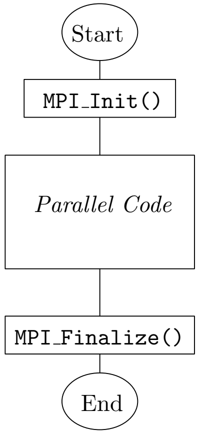
Example: serial `helloWorld.cpp`

```
#include <stdio.h>

int main()
{
   printf("Hello World! \n");
}
```

We want to write a MPI program that prints multiple Hello World.

Example: parallel helloWorld.cpp

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);

   printf("Hello World! \n");

   MPI_Finalize();
}
```



MPI_Init: initialization of the parallel run.
MPI_Finalize: finalization of the parallel run.

Compilation and execution

Compile with:

```
$ mpic++ helloWorld.cpp -o helloWorld.exe
```

Run with:

```
$ mpiexec -np 4 helloWorld.exe
```

Syntax review

MPI_Init: initialization of the parallel run.

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

- argc   length of the bash argument.
- argv   bash string.

Before MPI_Init no other parallel command is run.

MPI_Finalize: finalization of the parallel run.

```
int MPI_Finalize(void);
```

After MPI_Finalize no other parallel command is run.

Example: Hello World from processor

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);
   int rank, size;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Hello World from processor
         %i of %i! \n", rank, size);

   MPI_Finalize();
}
```

Syntax Review

`MPI_COMM_WORLD`: keyword for predefined communicator.

`MPI_Comm_rank`: returns the current processor rank.

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

• `comm`	communicator
• `rank`	current processor label

`MPI_Comm_size`: returns the number of processors in the communicator.

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

• `comm`	communicator
• `size`	number of processors in the communicator

Types of communication

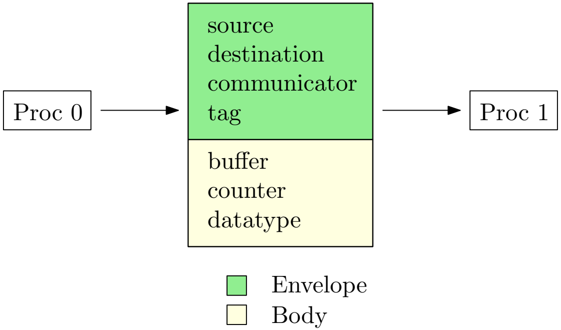
Before delving in the details of MPI, let's recall the principal ways two tasks/processes may communicate:

- 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.

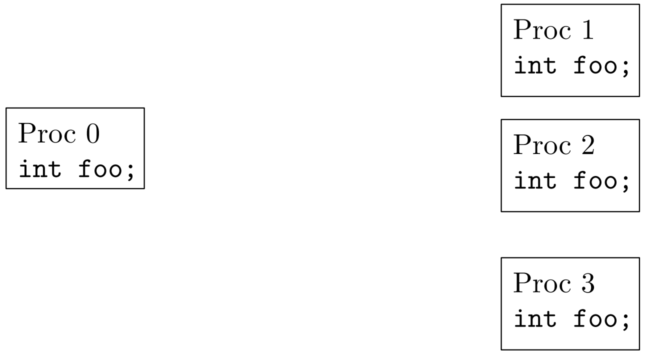
MPI spares the user the detail on how to implement those communication patterns in practice, and in particular which strategy to choose in case of collective communication.

Point to point: send and receive

Messages are *identified* by their **envelope**.
The content is *determined* by the **body**.



Point to point: Send and Receive



Point to point: Send and Receive

Proc 0

```
int foo;
foo=10;
```

Proc 1

```
int foo;
```

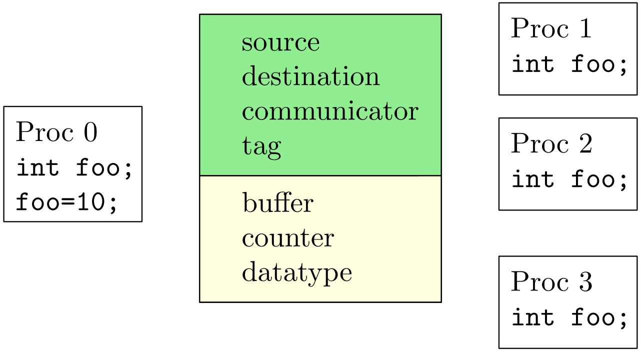
Proc 2

```
int foo;
```

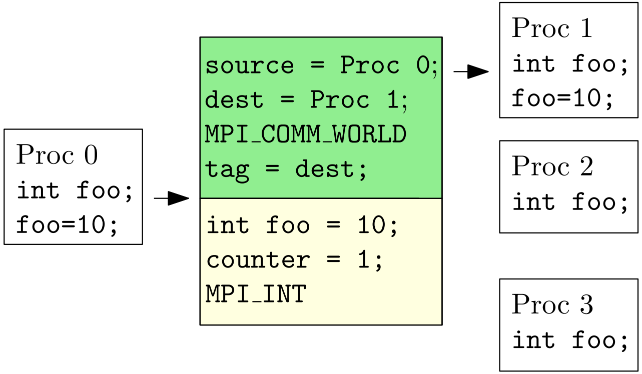
Proc 3

```
int foo;
```

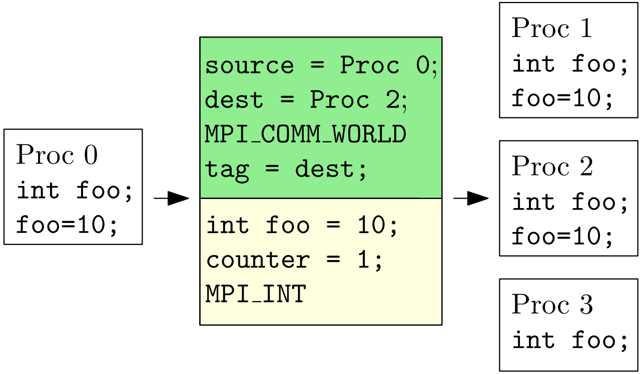
Point to point: Send and Receive



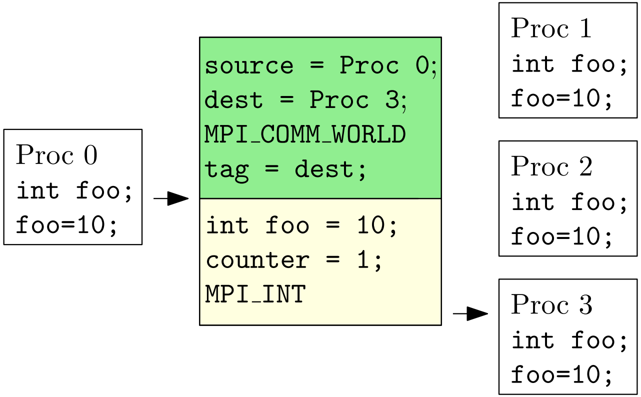
Point to point: Send and Receive



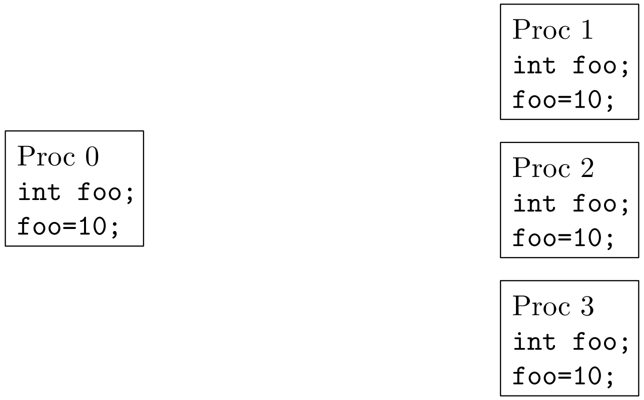
Point to point: Send and Receive



Point to point: Send and Receive



Point to point: Send and Receive



Example: Send and Receive

```
   if ( rank == 0)
   {
      a[0] = 2.0;
      a[1] = 1.0;
      for (int i=0; i < size; i++)
      {
         MPI_Send(a, 2, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      }
   }
   else
   {
      MPI_Recv(a, 2, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }
   std::cout << a[0] << " " << a[1] << std::endl;

   MPI_Finalize();
}
```

Remarks

- `foo` is declared on all the processors.
- The value of `foo` is defined only on processor zero.
- Processor zero communicates the value of `foo` to all the other processors.
- The communication happens with
  - `MPI_Send`
  - `MPI_Recv`
- All the message options (source, tag, dest) need to be consistent.

Syntax review

MPI_Send: Blocking communication that sends a buffer to a processor.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm)
```

• buf	buffer that collects the message
• count	length of the buffer
• datatype	MPI keyword that specify the data type
• dest	label of the target processor
• tag	tag of the message
• comm	communicator

Syntax review

`MPI_Recv`: Blocking communication that receives a buffer from a processor.

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

• `buf`	buffer that collects the message
• `count`	length of the buffer
• `datatype`	MPI keyword that specify the data type
• `source`	label of the source processor
• `tag`	tag of the message
• `comm`	communicator
• `status`	MPI keyword that specifies the status.

MPI Datatype

MPI needs to know the dimensions of the buffer you are sending, depending on the datatype.

MPI_CHAR	MPI_UNSIGNED_CHAR	MPI_FLOAT
MPI_SHORT	MPI_UNSIGNED_SHORT	MPI_DOUBLE
MPI_INT	MPI_UNSIGNED	MPI_LONG_DOUBLE
MPI_LONG	MPI_UNSIGNED_LONG	MPI_BYTE

Remark: data types are limited to only C data types!

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`.

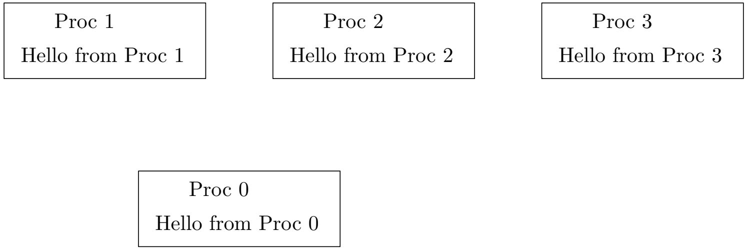
Point to Point Communication

For an MPI communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.
- Receiver's buffer must be large enough.

Example: Sorted Hello World

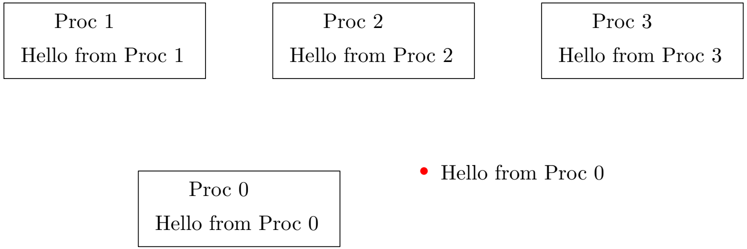
All the processes communicate to a master process that then prints the final result.



Many application have a master node that commands its slave nodes.

Example: Sorted Hello World

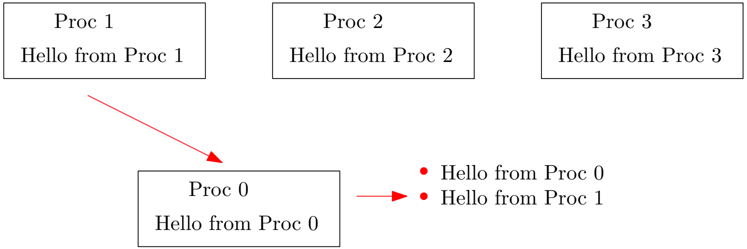
All the processes communicate to a master process that then prints the final result.



Many application have a master node that commands its slave nodes.

Example: Sorted Hello World

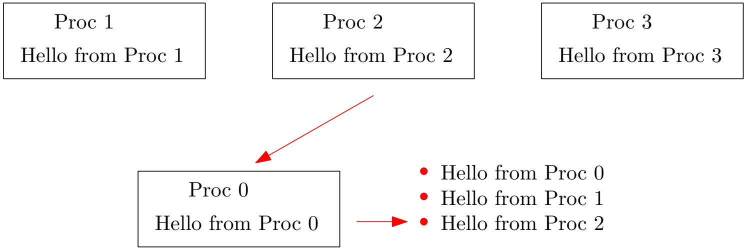
All the processes communicate to a master process that then prints the final result.



Many application have a master node that commands its slave nodes.

Example: Sorted Hello World

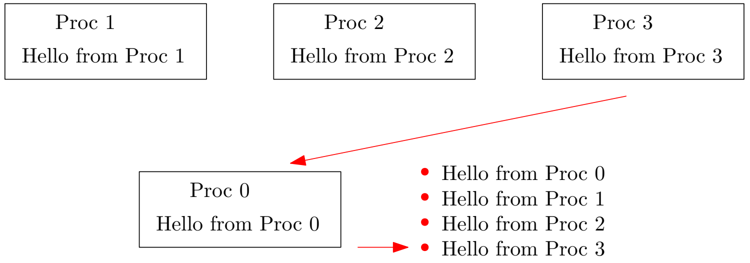
All the processes communicate to a master process that then prints the final result.



Many application have a master node that commands its slave nodes.

Example: Sorted Hello World

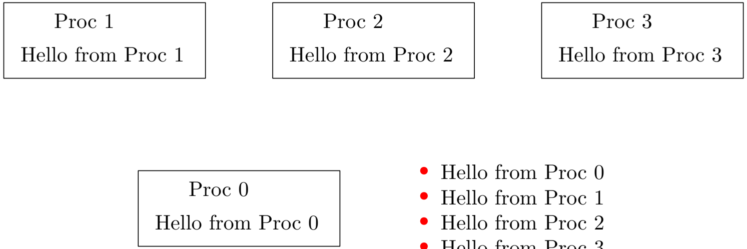
All the processes communicate to a master process that then prints the final result.



Many application have a master node that commands its slave nodes.

Example: Sorted Hello World

All the processes communicate to a master process that then prints the final result.



- Hello from Proc 0
- Hello from Proc 1
- Hello from Proc 2
- Hello from Proc 3

Many application have a master node that commands its slave nodes.

Example: Sorted Hello World

```
char message[100]; // message is a predefined array of chars
if (rank != 0){
   sprintf(message, "Hello from rank %i of %i\n", rank, size); // message
   MPI_Send(message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD); // send message
}
else {
   sprintf(message, "Hello from rank %i of %i\n", rank, size);
   printf("%s", message);

   for (int source=1; source < size; source++)
   {
      MPI_Recv(message, 100, MPI_CHAR, source, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
      printf("%s", message); // print message
   }
}
```

What should **never** happen

```
int foo;
if (rank == 0) {
   foo = 10;
   int tag;
   for (int dest=1; dest < size; dest++) {
      tag = dest;
      if (dest != 2)
      { // CREATES A DEADLOCK!!!
         MPI_Send(&foo, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
      }
   }
}
else {
   int source = 0;
   int tag = rank;
   MPI_Recv(&foo, 1, MPI_INT, source, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Deadlock

`MPI_Recv` from core two waits for a communication from core zero, but that will never happen.
We just created a simple **deadlock**.

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.

Syntax review

`MPI_Isend`: Non-blocking communication that sends a buffer to a processor.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
                 int dest, int tag, MPI_Comm comm, MPI_Request* request)
```

• `buf`	buffer that collects the message
• `count`	length of the buffer
• `datatype`	MPI keyword that specify the data type
• `dest`	label of the destination processor
• `tag`	tag of the message
• `comm`	communicator
• `request`	MPI communicator request.

Syntax review

`MPI_Irecv`: Non blocking communication that receives a buffer from a processor.

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
      int source, int tag, MPI_Comm comm, MPI_Request *request)
```

• `buf`	buffer that collects the message
• `count`	length of the buffer
• `datatype`	MPI keyword that specify the data type
• `source`	label of the source processor
• `tag`	tag of the message
• `comm`	communicator
• `request`	communication request.

Non-blocking point-to-point communication

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_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 length 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(rank == 0){
  v = {1, 2, 3, 4, 5};
  // 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 (rank == 1){
  int amount;
  MPI_Status status;
  MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_INT, &amount);
  v.resize(amount); //make sure v can hold the msg
  MPI_Recv(v.data(), 5, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Collective communication

Communications involving groups of processes are called **collectives**.

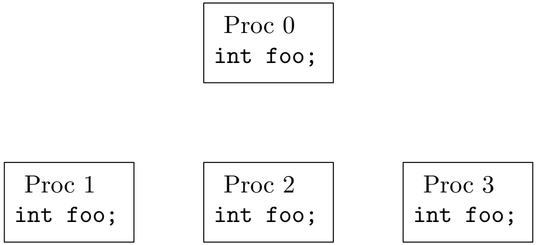
- The calls occur between processes in the communicator and every process must call the collective function
- They do not interfere with point-to-point calls.
- No tags are required.
- Receive buffers must match in size.

Designed to replace loops of point-to-point calls and designed to be more efficient.

When we employ collective calls

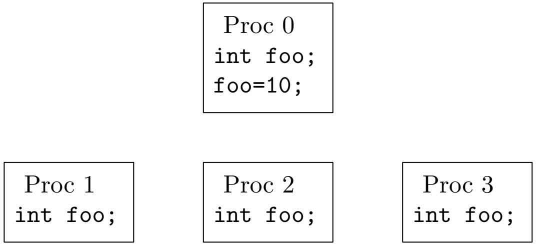
- Reading in data from a file and transferring it other tasks.
  - Simulations options or parameters.
- Synchronizing data amongst all tasks.
  - Simulations output.
- Calculating a value based on the data from all tasks.
  - Postprocessing of you numerical solution.
- Synchronization of tasks.

Broadcast



Remark: `MPI_Bcast` is equivalent to perform loops over `MPI_Send` and `MPI_Recv`.

Broadcast



```
int foo;
foo=10;
```

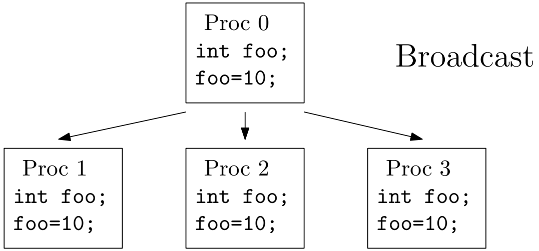
```
int foo;
```

```
int foo;
```

```
int foo;
```

Remark: `MPI_Bcast` is equivalent to perform loops over `MPI_Send` and `MPI_Recv`.

Broadcast



Remark: `MPI_Bcast` is equivalent to perform loops over `MPI_Send` and `MPI_Recv`.

Example: Send and Recv an array

```
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
   int rank, size;
   double a[2];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if ( rank == 0)
   {
      a[0] = 2.0;
      a[1] = 1.0;
      for (int i=0; i < size; i++)
      {
         MPI_Send(a, 2, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      }
   }
   else
   {
      MPI_Recv(a, 2, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }
   std::cout << a[0] << " " << a[1] << std::endl;

   MPI_Finalize();
}
```

Broadcast

```
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
   int rank, size;
   double a[2];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if ( rank == 0) // we populate a
   {
      a[0] = 2.0;
      a[1] = 1.0;
   }

   MPI_Bcast(a, 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   std::cout << a[0] << " " << a[1] << std::endl;

   MPI_Finalize();
}
```

Syntax Review

MPI_Bcast: communicates to everyone.

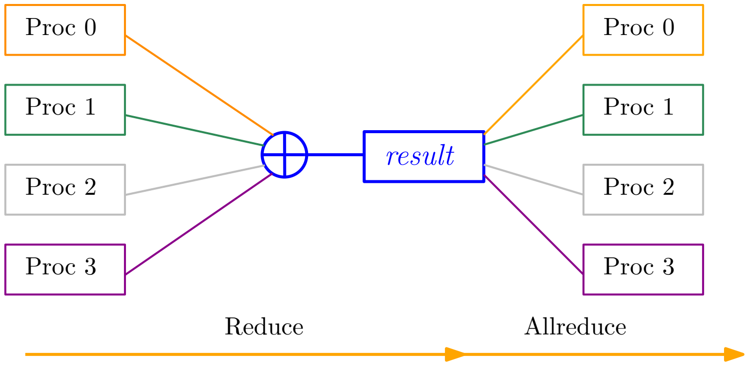
```
int MPI_Bcast(void *buf, int count,
            MPI_Datatype datatype, int root, MPI_Comm comm)
```

• buf	buffer that collects the message
• count	length of the buffer
• datatype	MPI keyword that specify the data type
• root	label of the root processor
• comm	communicator

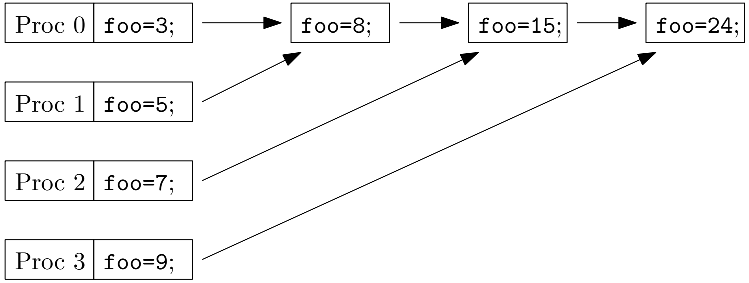
Reduce

A reduction takes values from different processors and generates a single value.

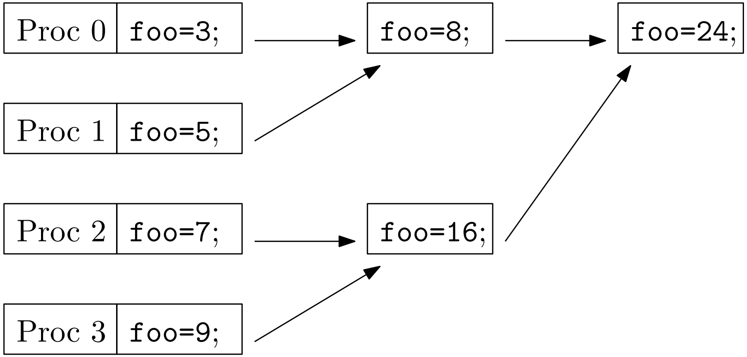
- Collect data from different processors
- Reduce to a single value via some operations



Reduce operations



Reduce operations are optimized



Example: Reduce

```
double a[2], red[2];
red[0] = 0.0;
red[1] = 0.0;
a[0] = 1.0; // everyone knows everything
a[1] = 2.0;

int root = 2;
MPI_Reduce(&a, &red, 2, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);

std::cout << "["<< rank << "] " << a[0] << " " << a[1] << std::endl;
std::cout << "["<< rank << "] " << red[0] << " " << red[1] << std::endl;
```

Syntax Review

MPI_Reduce: applies a certain operation to data coming from all the processors.

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
           MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

• sendbuf	buffer where the operation is applied
• recvbuf	buffer where the result is stored
• count	length of the buffer
• datatype	MPI keyword that specify the data type of the buffer
• op	MPI keyword that specify the operation to apply
• root	rank that will receive the result
• comm	communicator

Example: Allreduce

```
double a[2], red[2];
red[0] = 0.0;
red[1] = 0.0;
a[0] = 1.0; // everyone knows everything
a[1] = 2.0;

MPI_Allreduce(&a, &red, 2, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

std::cout << "["<< rank << "]" << a[0] << " " << a[1] << std::endl;
std::cout << "["<< rank << "]" << red[0] << " " << red[1] << std::endl;

MPI_Finalize();
```

Syntax Review

MPI_Allreduce: applies a certain operation to data coming from all the processors. Then communicates the result to all the processors.

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

• sendbuf	buffer where the operation is applied
• recvbuf	buffer where the result is stored
• count	length of the buffer
• datatype	MPI keyword that specify the data type of the buffer
• op	MPI keyword that specify the operation to apply
• comm	communicator

Remark: MPI_Allreduce performs the same operations as a MPI_Reduce followed by a MPI_Bcast.

Operations

MPI Operation	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND

MPI Operation	Function
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

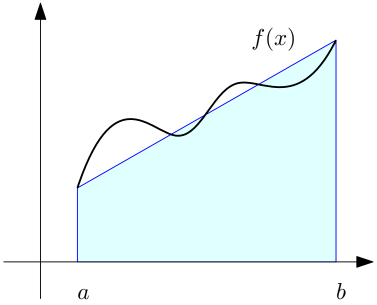
Example: why we need MPI_IN_PLACE?

```
   double a[2];
   a[0] = 1.0; // everyone knows everything
   a[1] = 2.0;

   MPI_Allreduce(MPI_IN_PLACE, &a, 2, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
// MPI_Allreduce(&a, &a, 2, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD); // WRONG

   std::cout << "["<< rank << "] " << a[0] << " " << a[1] << std::endl;
```

Syntax Review

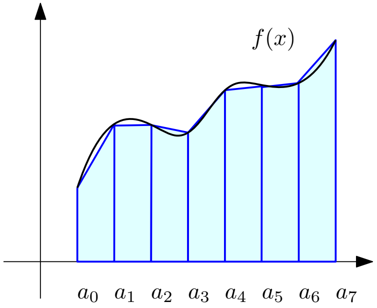
`MPI_IN_PLACE`: keyword employed *only* on the sender when sender and receiver coincide.
If `MPI_IN_PLACE` is not used, a run time error happens due to memory aliasing.

Trapezoidal Rule



$$I = \int_a^b f(x)\mathrm{d}x \approx \frac{h}{2}[f(a) + f(b)]$$

Composite Trapezoidal Rule

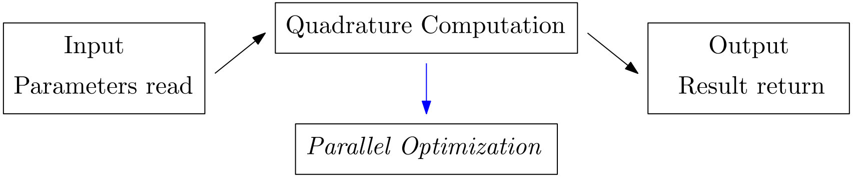


$$I = \int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{N-1} \frac{h}{2}[f(a_i) + f(a_{i+1})]$$

Composite Trapezoidal Rule: sequential

```
h = (b - a) / n; // discretization size

double ai, bi; // extremes of composite integration

for (int i=0; i < n; i++)
{
   ai = a + i*h; // ai
   bi = a + (i+1)*h; // bi
   sum = sum + ( f( ai ) + f( bi ));
}

sum = sum * h/2; // quadrature output
```

Composite Trapezoidal Rule: parallel idea



Composite Trapezoidal Rule: parallel input

```
if (rank == 0)
{
   std::cout << "Insert extrema a: " << std::endl;
   std::cin >> a;
   std::cout << "Insert extrema b: " << std::endl;
   std::cin >> b;
   std::cout << "Insert number of intervals: " << std::endl;
   std::cin >> n; // to simplify, n * np
}

MPI_Bcast(&a,1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&b,1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&n,1, MPI_INT, 0, MPI_COMM_WORLD);
```

Composite Trapezoidal Rule: quadrature update

```
double h;
h = (b - a) / (n*size); // discretization size

double ai, bi; // extremes of composite integration

for (int i= rank*n; i < (rank+1)*n; i++)
{
   ai = a + i*h; // ai
   bi = a + (i+1)*h; // bi
   sum = sum + ( f( ai ) + f( bi ));
}

sum = sum * h/2; // quadrature output
```

Composite Trapezoidal Rule: quadrature update

```
if (rank == 0) // MPI is weird
{
   MPI_Reduce(MPI_IN_PLACE, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
}
else
{
   MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Composite Trapezoidal Rule: output

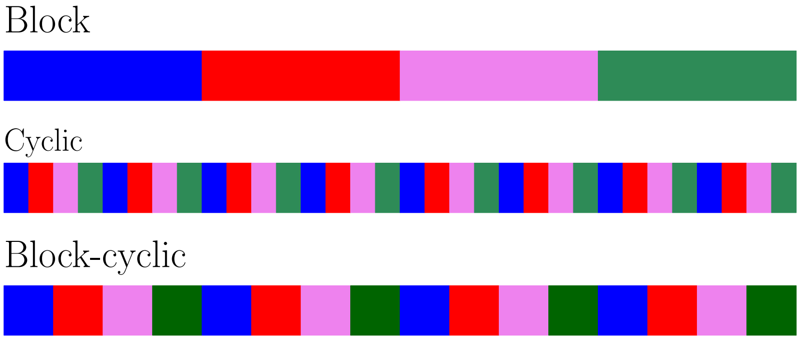
```
  if (rank == 0)
  {
  std::cout << "Integral: " << sum << std::endl;
  std::cout << "Exact: " << (1.0/3*b*b*b + 1.5*b*b + b) -
                        (1.0/3*a*a*a + 1.5*a*a + a) << std::endl;
  }
```

Domain Decomposition

Given a domain with 32 intervals, we want to split it among 4 ranks..

We have different strategies.

Domain Decomposition

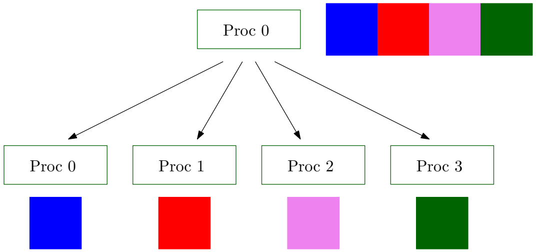


Remark

- Block partitioning is used when the source is available only locally, on one processor.
- Cyclic partitioning is used when the source is already available at all processors.

Scatter

Scatter allows to communicate blocks of memories between the processors.



Example: MPI_Scatter with C array

```
   int recvCount = 2;
   int recvBuffer[2];

   int* sendBuffer = NULL;

   if (rank == 0)
   {
      sendBuffer = (int*) malloc( sizeof(int) * size * 2);
      for (int i=0; i < 2*size; i++){
         sendBuffer[i] = i;
      }
   }

   MPI_Scatter(sendBuffer, recvCount, MPI_INT,
            recvBuffer, recvCount, MPI_INT, 0, MPI_COMM_WORLD);

   std::cout << "rank " << rank << " \nData:" << recvBuffer[0] << " " << recvBuffer[1]
         << std::endl;

   MPI_Finalize();
   return 0;
}
```

Example: MPI_Scatter with C++ vector

```cpp
// omitting the usual stuff...
int recvCount = 2;

std::vector<int> sendBuffer;
std::vector<int> recvBuffer;

if (rank == 0)
{
   for (int i=0; i < 2*size; i++){
      sendBuffer.push_back(i);
   }
}

recvBuffer.resize(recvCount);
MPI_Scatter(sendBuffer.data(), recvCount, MPI_INT,
         recvBuffer.data(), recvCount, MPI_INT, 0, MPI_COMM_WORLD);

std::cout << "rank " << rank << " \nData:" << recvBuffer[0] << " " << recvBuffer[1]
       << std::endl;
// omitting the usual stuff...
```

Important Remark

When employing C++ data structure, you must recover the C data structure.
Compare with:

```
MPI_Scatter(sendBuffer.data(), recvCount, MPI_INT,
         recvBuffer.data(), recvCount, MPI_INT, 0, MPI_COMM_WORLD);
```

Syntax Review

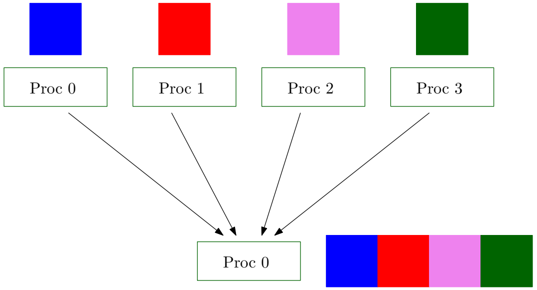
MPI_Scatter: sends data from one process to all other processes.

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype senddatatype, void
    *recvbuf, int recvcount, MPI_Datatype recvdatatype, int root, MPI_Comm comm);
```

• sendbuf	buffer with the sending message
• sendcount	length of the sending buffer
• senddatatype	MPI keyword that specify the sender data type
• recvbuf	buffer that collects the receiving message
• recvcount	dimension of the receiving buffer
• recvdatatype	MPI keyword that specify the receiving data type
• root	buffer that collects the receiving message
• comm	communicator

Gather

Gather allows to collect blocks of memories from all the processors on a single node.



Example: MPI_Gather with C array

```
int sendCount = 2; // "global" variable

// example with array
int sendData[sendCount] = {rank * 2, rank * 2 + 1};

int *recvData = nullptr;

if (rank == 0) {
   // Allocate memory for the gathered data at the root process
   recvData = new int[sendCount * size];
}

MPI_Gather(sendData, sendCount, MPI_INT, recvData, sendCount, MPI_INT, 0,
       MPI_COMM_WORLD);

if (rank == 0) {
   // Display the gathered data at the root process
   std::cout << "Root process [0] gathered data: ";
   for (int i = 0; i < size * sendCount; ++i) {
      std::cout << recvData[i] << " ";
   }
   std::cout << std::endl;

   delete[] recvData;
}
```

Example: MPI_Gather with C++ vector

```
#include <vector>
```

```
   // int sendData[sendCount] = {rank * 2, rank * 2 + 1};
   std::vector<int> sendData;

   for(int i=0; i < sendCount; i++){
      sendData.push_back(rank*sendCount + i);
   }
```

```
   std::vector<int> recvData;

   if (rank == 0) {
      // Allocate memory for the gathered data at the root process
      // recvData = new int[sendCount * size];
      recvData.resize(sendCount * size);
   }
```

```
   MPI_Gather(sendData.data(), sendCount,
                MPI_INT, recvData.data(), sendCount, MPI_INT, 0, MPI_COMM_WORLD);
```

```
      // delete[] recvData;
```

Syntax Review

`MPI_Gather`: collects on the root node the data and stores them in rank order.

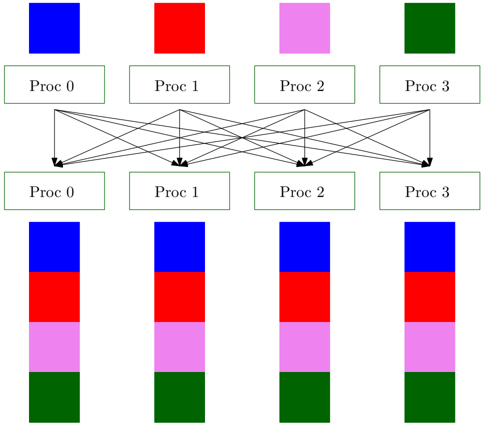
```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype senddatatype, void*
    recvbuf, int recvcount, MPI_Datatype recvdatatype, int root, MPI_Comm comm);
```

• `sendbuf`	buffer that collects the sending message
• `sendcount`	length of the sending buffer
• `senddatatype`	MPI keyword that specify the sender data type
• `recvbuf`	buffer that collects the receiving message
• `recvcount`	dimension of the receiving buffer
• `recvdatatype`	MPI keyword that specify the receiving data type
• `root`	buffer that collects the receiving message
• `comm`	communicator

Remark: `recvbuf` has the dimension of the receiving buffer, not of the global reconstructed data.

Allgather

Allgather allows to collect blocks of memories from all the processors onto all the nodes.



Example: MPI_AllGather

```
int* recvData = nullptr;

recvData = new int[sendCount * size];

MPI_Allgather(sendData, sendCount, MPI_INT,
              recvData, sendCount, MPI_INT,
                   MPI_COMM_WORLD);

std::cout << "Out from processor " << rank <<
     std::endl;
for (int i =0; i < 2*size; i++)
{
    std::cout << recvData[i] << " ";
}
std::cout << std::endl;
```

Syntax Review

`MPI_Allgather`: collects on all the nodes the data and stores them in rank order.

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype senddatatype, void*
    recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm);
```

• `sendbuf`	buffer that collects the sending message
• `sendcount`	length of the sending buffer
• `senddatatype`	MPI keyword that specify the sender data type
• `recvbuf`	buffer that collects the receiving message
• `recvcount`	dimension of the receiving buffer
• `recvdatatype`	MPI keyword that specify the receiving data type
• `comm`	communicator

Remark: `recvbuf` has the dimension of the receiving buffer, not of the global reconstructed data.

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.

Syntax Review

MPI_Scatterv: scatters a buffer in parts to all processes in a communicator.

```
int MPI_Scatterv(const void* sendbuf, const int sendcount, const int displs[],
       MPI_Datatype senddatatype, void* recvbuf,int recvcount,
       MPI_Datatype recvdatatype, int root, MPI_Comm comm);
```

• sendbuf	buffer that collects the sending message
• sendcount	length of the sending buffer
• displs	integer array. Entry *i* specifies the displacement (relative to sendbuf) from which to take the outgoing data to process *i* (integer)
• senddatatype	MPI keyword that specify the sender data type
• recvbuf	buffer that collects the receiving message
• recvcount	dimension of the receiving buffer
• recvdatatype	MPI keyword that specify the receiving data type
• root	rank of sending process
• comm	communicator

Remark: For non-root processes, the send parameters are ignored.

Syntax Review

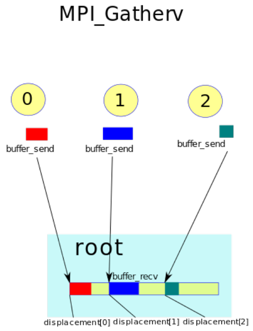
MPI_Gatherv: gathers a buffer in parts to all processes in a communicator.

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
   void *recvbuf, const int recvcounts[], const int displs[],
   MPI_Datatype recvtype, int root, MPI_Comm comm)
```

• sendbuf	buffer that collects the sending message
• sendcount	length of the sending buffer
• displs	integer array. Entry *i* specifies the displacement (relative to sendbuf) from which to take the outgoing data to process *i* (integer)
• senddatatype	MPI keyword that specify the sender data type
• recvbuf	buffer that collects the receiving message
• recvcount	dimension of the receiving buffer
• recvdatatype	MPI keyword that specify the receiving data type
• root	rank of sending process
• comm	communicator

Remark: For non-root processes, the send parameters are ignored.

MPI_Gatherv()


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, displacements;

std::vector<double> v;
for (int i = 0; i < 1000; ++i) {
   v.push_back(0.1*i);
}
int n = v.size();

if(rank ==0) { // root has to prepare some data
  counts_send.resize(size);
  displacements.resize(size,0); // init. by zero

  unsigned int chunk = n/size; // integer division
  unsigned int rest = n%size; // the rest of the division

  for (int i=0; i<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_WORLD);

//each process gets localsize its chunk of v
std::vector<double> local_v(local_size);
MPI_Scatterv(v.data(),counts_send.data(),displacements.data(),MPI_DOUBLE,
         local_v.data(),local_size, MPI_DOUBLE,0,MPI_COMM_WORLD);

std::cout << "Rank " << rank << " received " << local_v[local_size-1]
                      << "\nSize:" << local_size << std::endl;
```

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).

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.

```
std::array<double,2> v;

if ( rank == 0){
 a=0.5, b=0.3, c=0.7;
 v[0]=0.1, v[1]=0.2;
}

if (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; // unpack the returned tuple
}
std::cout << "Rank: " << rank << " a: " << a <<
         " b: " << b << " c: " << c << " v: " << v[0] << " " << v[1] << std::endl;
```

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

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.

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