High-Performance Mathematics

Programming in a distributed setting: MPI II

Progetto Speciale per la Didattica 2023/24

Fabio Durastante (L3)

April 24, 2024

Table of Contents

▶ Setup

▶ Point-to-Point Communications
- Deadlock
- Nonblocking communications
- Sendreceive
- Things left out

▶ A first scientific computation

▶ References

Starting again

To start programming and running our codes again, let’s start again by **recovering the environment**.

Starting again

To start programming and running our codes again, let's start again by **recovering the environment**.

first we need to connect/use an `MPI` available machine, fire up a terminal and write

```
ssh n.cognomeXX@a3-dottY.cs.dm.unipi.it
```

Starting again

1 Setup

To start programming and running our codes again, let's start again by **recovering the environment**.

- first we need to connect/use an `MPI` available machine, fire up a terminal and write

```
ssh n.cognomeXX@a3-dottY.cs.dm.unipi.it
```

- navigate to the local version of you git repository and check that everything is **up-to-date**:

```
cd Documents/my/repository/folder
git pull
```

Starting again

To start programming and running our codes again, let's start again by **recovering the environment**.

- first we need to connect/use an `MPI` available machine, fire up a terminal and write

```
ssh n.cognomeXX@a3-dottY.cs.dm.unipi.it
```

- navigate to the local version of you git repository and check that everything is **up-to-date**:

```
cd Documents/my/repository/folder
git pull
```

- use the **environment modules** to **load** the **compiler** and the **MPI implementation**:

```
module load gcc/12.2.0 openmpi/5.0.2-cuda-12.3.0-gcc-12.2.0
```

Starting again

- We can **check if everything works** by compiling and executing one of the test program from last time, *e.g.*,

```
mpicc hamlet.c - o hamlet
mpirun -np 6 ./hamlet
```

Starting again

- We can **check if everything works** by compiling and executing one of the test program from last time, *e.g.*,

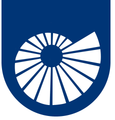
```
mpicc hamlet.c - o hamlet
mpirun -np 6 ./hamlet
```

If everything went well, we should read something similar to:

```
Hello world! I'm process 3 of 6
Hello world! I'm process 2 of 6
Hello world! I'm process 0 of 6
Hello world! I'm process 1 of 6
Hello world! I'm process 4 of 6
Hello world! I'm process 5 of 6
```

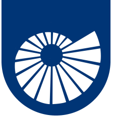
What did we see?

- `MPI_Init(&argc,&argv);` and `MPI_Finalize();` to **initialize the MPI execution environment** and **terminate MPI execution environment**,



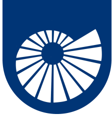
What did we see?

- `MPI_Init(&argc,&argv);` and `MPI_Finalize();` to **initialize the MPI execution environment** and **terminate MPI execution environment**,
- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );` to discover our **rank**, i.e., our **process number**,



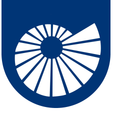
What did we see?

- `MPI_Init(&argc,&argv);` and `MPI_Finalize();` to **initialize the MPI execution environment** and **terminate MPI execution environment**,
- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );` to discover our **rank**, i.e., our **process number**,
- `MPI_Comm_size( MPI_COMM_WORLD, &size );` to discover the **total number of processes**.



What did we see?

- `MPI_Init(&argc,&argv);` and `MPI_Finalize();` to **initialize the MPI execution environment** and **terminate MPI execution environment**,
- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );` to discover our **rank**, i.e., our **process number**,
- `MPI_Comm_size( MPI_COMM_WORLD, &size );` to discover the **total number of processes**.
- `MPI_COMM_WORLD` the default communicator.



What did we see?

- `MPI_Init(&argc,&argv);` and `MPI_Finalize();` to **initialize the MPI execution environment** and **terminate MPI execution environment**,
- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );` to discover our **rank**, i.e., our **process number**,
- `MPI_Comm_size( MPI_COMM_WORLD, &size );` to discover the **total number of processes**.
- `MPI_COMM_WORLD` the default communicator.

Communicator

A **Communicator object** connects a group of processes in one MPI session. There can be more than one communicator in an MPI session, each of them gives each contained process an independent identifier and arranges its contained processes in an ordered topology.

Table of Contents

▶ Setup

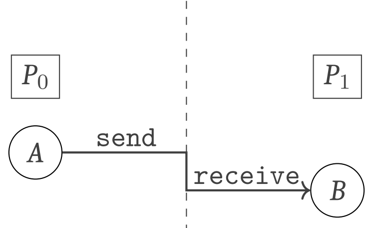
▶ Point-to-Point Communications
- Deadlock
- Nonblocking communications
- Sendreceive
- Things left out

▶ A first scientific computation

▶ References

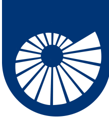
Sending and Receiving Messages

We have seen that each process within a *communicator* is identified by its *rank*, how can we exchange data between two processes?



We need to posses several information to have a meaningful message

- Who is sending the data?
- To whom the data is sent?
- What type of data are we sending?
- How does the receiver can identify it?



The blocking send and receive


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

`void *message` points to the message content itself, it can be a simple scalar or a group of data,

`int count` specifies the number of data elements of which the message is composed,

`MPI_Datatype datatype` indicates the data type of the elements that make up the message,

`int dest` the rank of the destination process,

`int tag` the user-defined tag field,

`MPI_Comm comm` the communicator in which the source and destination processes reside and for which their respective ranks are defined.

The blocking send and receive


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

`void *message` points to the message content itself, it can be a simple scalar or a group of data,

`int count` specifies the number of data elements of which the message is composed,

`MPI_Datatype datatype` indicates the data type of the elements that make up the message,

`int source` the rank of the source process,

`int tag` the user-defined tag field,

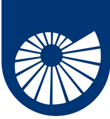
`MPI_Comm comm` the communicator in which the source and destination processes reside,

`MPI_Status *status` is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`.

Basic MPI Data Types

Of the previous slides inputs the only ones that is specific to MPI is the `MPI_Datatype`:

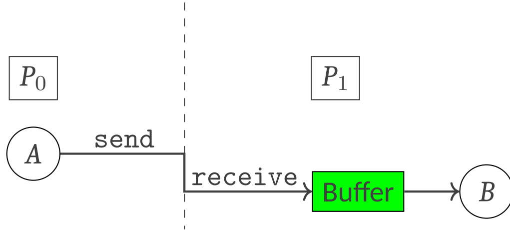
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int



Why “blocking” send and receive?

For the `MPI_Send` to be *locally* blocking means that it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer: it is a *non local* operation.

Note: The message might be copied directly into the matching receive buffer (as in the first figure), or it might be copied into a temporary system buffer.



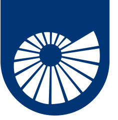
Why “blocking” send and receive?

For the `MPI_Send` to be *locally* blocking means that it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer: it is a *non local* operation.

The `MPI_Receive`, on the other hand returns **only** after the receive buffer contains the newly received message. A receive can’t complete before the matching send has completed, but, of course, it can complete only after the matching send has started.

A simple send/receive example


```
#include "mpi.h"
#include <string.h>
#include <stdio.h>
int main( int argc, char **argv){
char message[20]; int myrank; MPI_Status status;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
if (myrank == 0){  /* code for process zero */
strcpy(message,"Hello, there");
MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1){ /* code for process one */
MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
printf("received :%s:\n", message);
}
MPI_Finalize();
return 0; }
```



A simple send/receive example

We can compile our code by simply adding to our Makefile

```
easysendrecv: easysendrecv.c
$(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
```

then, we type `make`, and we run our program with

```
mpirun -np 2 easysendrecv
```

getting as answer

```
received :Hello, there:
```

So, what have we done?

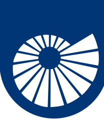
A simple send/receive example

```
MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
```

Process 0 sends the content of the `char` array `message[20]`, whose size is `strlen(message)+1` size of `char` (`MPI_CHAR`) to processor 1 with tag 99 on the communicator `MPI_COMM_WORLD`.

```
MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
```

on the other side process 1, receives into the buffer `message[20]` an array with size 20 size of `MPI_CHAR`, from process 0 with tag 99 on the same communicator `MPI_COMM_WORLD`.



A simple send/receive example : programmer smash!

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?
- What happens if we have a mismatch in the ranks of the sending and receiving processes?
- What happens if we use the wrong message size?
- What happens if we have a mismatch in the type?

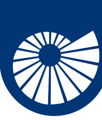
A simple send/receive example : programmer smash!

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?
- What happens if we use the wrong message size?
- What happens if we have a mismatch in the type?



A simple send/receive example : programmer smash!

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

A: The process stays there hanging trying to match messages that will never come...

- What happens if we use the wrong message size?

- What happens if we have a mismatch in the type?

A simple send/receive example : programmer smash!

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

A: The process stays there hanging trying to match messages that will never come...

- What happens if we use the wrong message size?

A: If the size of the arriving message is longer than the expected we get an error of `MPI_ERR_TRUNCATE: message truncated`, note that there are combinations of wrong sizes for which things still works

- What happens if we have a mismatch in the type?

A simple send/receive example : programmer smash!

It is a good exercise to try and mess things up, so let us see some damaging suggestions:

- What happens if we have a mismatch in the tags?

A: The process stays there hanging waiting for a message with a tag that will never come...

- What happens if we have a mismatch in the ranks of the sending and receiving processes?

A: The process stays there hanging trying to match messages that will never come...

- What happens if we use the wrong message size?

A: If the size of the arriving message is longer than the expected we get an error of `MPI_ERR_TRUNCATE: message truncated`, note that there are combinations of wrong sizes for which things still works

- What happens if we have a mismatch in the type?

A: There are combinations of instances in which things seems to work, but the code is erroneous, and the behavior is not deterministic.

Checkpointing to git

Exercise

It's a **good exercise** at this point to try updating your git repository with the new file and the updated `Makefile`.

Do you remember?

`git status`, `git add`, `git commit -m "..."`, and then `git push`.

Checkpointing to git

Exercise

It's a **good exercise** at this point to try updating your git repository with the new file and the updated `Makefile`.

Do you remember?

`git status`, `git add`, `git commit -m "..."`, and then `git push`.

Exercise

A good idea as a home exercise is to try updating the `README` file as well.
Inside you can use *Markdown* to format the text: www.markdownguide.org.

Dealing with more than one send and receive

We have two processes that exchange data: `MPI_Comm_rank(comm, &myrank);`

- Solution 1:

```
if (myrank == 0){
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
}else if(myrank == 1){
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
}
```

Dealing with more than one send and receive

We have two processes that exchange data: `MPI_Comm_rank(comm, &myrank);`

- Solution 1:

```
if (myrank == 0){
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
}else if(myrank == 1){
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
}
```

- Solution 2:

```
if (myrank == 0){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
}else if(myrank == 1){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

Dealing with more than one send and receive

We have two processes that exchange data: `MPI_Comm_rank(comm, &myrank);`

- Solution 2:

```
if (myrank == 0){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
}else if(myrank == 1){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

- Solution 3:

```
if (myrank == 0){
MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, status);
}else if(myrank == 1){
MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, status);
MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

Dealing with more than one send and receive

In the case of Solution 1:

```
MPI_Comm_rank(comm, &myrank);
if (myrank == 0){
MPI_Send(...);
MPI_Recv(...);
}else if(myrank == 1){
MPI_Send(...);
MPI_Recv(...);
}
```

- The call `MPI_Send` is blocking, therefore the message sent by each process has to be copied out before the send operation returns and the receive operation starts.
- For the call to complete successfully, it is then necessary that at least one of the two messages sent be buffered, otherwise ...
- a deadlock situation occurs: both processes are blocked since there is no buffer space available!

Dealing with more than one send and receive

In the case of Solution 1:

```
MPI_Comm_rank(comm, &myrank);
if (myrank == 0){
MPI_Send(...);
MPI_Recv(...);
}else if(myrank == 1){
MPI_Send(...);
MPI_Recv(...);
}
```

- The call `MPI_Send` is blocking, therefore the message sent by each process has to be copied out before the send operation returns and the receive operation starts.
- For the call to complete successfully, it is then necessary that at least one of the two messages sent be buffered, otherwise ...
- a deadlock situation occurs: both processes are blocked since there is no buffer space available!

Here what happens to your program when you encounter Deadlock

Dealing with more than one send and receive

In the case of Solution 2:

```
MPI_Comm_rank(comm, &myrank);
if (myrank == 0){
MPI_Recv(...);
MPI_Send(...);
}else if(myrank == 1){
MPI_Recv(...);
MPI_Send(...);
}
```

- The receive operation of process 0 must complete before its send. It can complete only if the matching send of processor 1 is executed.
- The receive operation of process 1 must complete before its send. It can complete only if the matching send of processor 0 is executed.
- This program will always deadlock.

Dealing with more than one send and receive

Here what happens to your program when you encounter Deadlock

In the case of Solution 2:

```
MPI_Comm_rank(comm, &myrank);
if (myrank == 0){
MPI_Recv(...);
MPI_Send(...);
}else if(myrank == 1){
MPI_Recv(...);
MPI_Send(...);
}
```

- The receive operation of process 0 must complete before its send. It can complete only if the matching send of processor 1 is executed.
- The receive operation of process 1 must complete before its send. It can complete only if the matching send of processor 0 is executed.
- This program will always deadlock.

In the case of Solution 3:

```
MPI_Comm_rank(comm, &myrank);
if (myrank == 0){
MPI_Send(...);
MPI_Recv(...);
}else if(myrank == 1){
MPI_Recv(...);
MPI_Send(...);
}
```

- This program will succeed even if no buffer space for data is available.

Dealing with more than one send and receive

This way you can beat Deadlock!

In the case of Solution 3:

```
MPI_Comm_rank(comm, &myrank);
if (myrank == 0){
MPI_Send(...);
MPI_Recv(...);
}else if(myrank == 1){
MPI_Recv(...);
MPI_Send(...);
}
```

- This program will succeed even if no buffer space for data is available.

Nonblocking communications

As we have seen the use of **blocking communications** ensures that

- the send and receive buffers used in the `MPI_Send` and `MPI_Recv` arguments are safe to use or reuse after the function call,
- but it also means that unless there is a simultaneously matching send for each receive, the code will deadlock.

Nonblocking communications

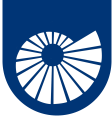
There exists a version of the point-to-point communication that **returns immediately** from the function call before confirming that the send or the receive has completed, these are the **nonblocking send** and **receive** functions.

- To verify that the data has been copied out of the send buffer a separate call is needed,
- To verify that the data has been received into the receive buffer a separate call is needed,

Nonblocking communications

There exists a version of the point-to-point communication that **returns immediately** from the function call before confirming that the send or the receive has completed, these are the **nonblocking send** and **receive** functions.

- To verify that the data has been copied out of the send buffer a separate call is needed,
- To verify that the data has been received into the receive buffer a separate call is needed,
- The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.
- The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.



Nonblocking comms: MPI_Isend and MPI_Irecv

The two nonblocking point-to-point communication call are then

```
int MPI_Isend(void *message, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *send_request);

int MPI_Irecv(void *message, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *recv_request);
```

- The `MPI_Request` variables substitute the `MPI_Status` and store information about the status of the pending communication operation.
- The way of saying when this communications must be completed is by using the

  ```
  int MPI_Wait(MPI_Request *request, MPI_Status *status)
  ```

  when is called, the nonblocking request originating from `MPI_Isend` or `MPI_Irecv` is provided as an argument.

Nonblocking communications: an example


```
int main(int argc, char **argv) {
int a, b, size, rank, tag = 0;
MPI_Status status;
MPI_Request send_request, recv_request;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
a = 314159;
MPI_Isend(&a, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &send_request);
MPI_Irecv (&b, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &recv_request);
MPI_Wait(&send_request, &status);
MPI_Wait(&recv_request, &status);
printf ("Process %d received value %d\n", rank, b);
}
```


Continue on the next slide

Nonblocking communications: an example

continued

Continued from previous slide

```c
else {
a = 667;
MPI_Isend (&a, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &send_request);
MPI_Irecv (&b, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &recv_request);
MPI_Wait(&send_request, &status);
MPI_Wait(&recv_request, &status);
printf ("Process %d received value %d\n", rank, b);
}
MPI_Finalize();
return 0;
}
```

A simple send/receive example

We can compile our code by simply adding to our `Makefile`

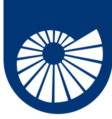
```
nonblockingsendrecv: nonblockingsendrecv.c
  $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
```

then, we type `make`, and we run our program with

```
mpirun -np 2 nonblockingsendrecv
```

getting as answer

```
Process 0 received value 667
Process 1 received value 314159
```



A simple send/receive example

We can compile our code by simply adding to our `Makefile`

```
nonblockingsendrecv: nonblockingsendrecv.c
  $(MPICC) $(CFLAGS) $(LDFLAGS) $? $(LDLIBS) -o $@
```

then, we type `make`, and we run our program with

```
mpirun -np 2 nonblockingsendrecv
```

getting as answer

```
Process 0 received value 667
Process 1 received value 314159
```

Another useful instruction for the case of nonblocking communication is represented by

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

A call to `MPI_TEST` returns `flag = true` if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation.

Send-Receive

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process.

- Source and destination are possibly the same,
- Send-receive operation is very useful for executing a shift operation across a chain of processes,
- A message sent by a send-receive operation can be received by a regular receive operation

```
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);
```

Send-Receive-Replace

A slight variant of the `MPI_Sendrecv` operation is represented by the `MPI_Sendrecv_replace` operation

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
```

as the name suggests, the same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

Clearly, if you confront its arguments with the one of the `MPI_Sendrecv`, the arguments `void *recvbuf, int recvcount` are absent.

Things left out

We are leaving out some variants of the point-to-point communication:

- Both for blocking and nonblocking communications we have left out the **synchronous** and **ready** mode,
- For nonblocking communications we have also the **buffered** variants,
- Instead of waiting/testing for a single communication at the time we could wait for the completion of some, or all the operations in a list. There are specific routines for achieving this.

You can read about this on the manual:

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, High Performance Computing Center Stuttgart (HLRS).

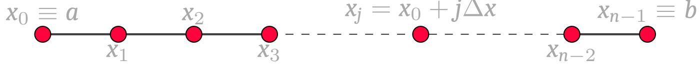
Table of Contents

- Setup
- Point-to-Point Communications
  - Deadlock
  - Nonblocking communications
  - Sendreceive
  - Things left out
- A first scientific computation
- References

The 1st derivative of a function with finite differences

Given a function $f(x) : [a, b] \to \mathbb{R}$ we want to approximate $f'(x)$ on a (uniform) grid on the $[a, b]$ interval by using a finite difference scheme in parallel.

- Given an integer $n \in \mathbb{N}$ we can subdivide the interval $[a, b]$ into intervals of length $\Delta x = {}^{(b-a)}/_{n-1}$ with grid points $\{x_j\}_{j=0}^{n} = \{x_j = a + j\Delta x\}_{j=0}^{n-1}$:



,

- and consider the values $\{f_j\}_{j=0}^{n-1} = \{f(x_j)\}_{j=0}^{n-1}$
- We can approximate the values of $f'(x_j)$, for $j = 1, \ldots, n-2$, by using only the values of f at the knots $\{f_j\}_{j=0}^{n-1}$

The 1st derivative of a function with finite differences

- The first derivative of f at $x = x_j$ can be expressed by using knots for $j' > j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_{j+1} - f_j}{\Delta x} \approx \frac{f_{j+1} - f_j}{\Delta x} \triangleq D_+ f_j,$$

The 1st derivative of a function with finite differences

- The first derivative of f at $x = x_j$ can be expressed by using knots for $j' > j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_{j+1} - f_j}{\Delta x} \approx \frac{f_{j+1} - f_j}{\Delta x} \triangleq D_+ f_j,$$

- or equivalently by using knots for $j' < j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_j - f_{j-1}}{\Delta x} \approx \frac{f_j - f_{j-1}}{\Delta x} \triangleq D_- f_j,$$

The 1st derivative of a function with finite differences

- The first derivative of f at $x = x_j$ can be expressed by using knots for $j' > j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_{j+1} - f_j}{\Delta x} \approx \frac{f_{j+1} - f_j}{\Delta x} \triangleq D_+ f_j,$$

x_{j-1} x_j x_{j+1}

- or equivalently by using knots for $j' < j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_j - f_{j-1}}{\Delta x} \approx \frac{f_j - f_{j-1}}{\Delta x} \triangleq D_- f_j,$$

x_{j-1} x_j x_{j+1}

- at last we can consider the arithmetic mean of previous two:

$$f'(x_j) \approx D_0 f_j \triangleq \frac{1}{2}(D_- f_j + D_+ f_j) = \frac{f_{j+1} - f_{j-1}}{2\Delta x},$$

x_{j-1} x_j x_{j+1}

Writing the sequential algorithm

The sequential algorithms needs to break the approximation process into three parts

1. evaluate the derivative $f'(x_i)$ for $i = 1, \ldots, n-2$,
2. evaluate the derivative at the left–hand side $f'(x_0)$,
3. evaluate the derivative at the right–hand side $f'(x_{n-1})$.

To have the same *order of approximation* at each point of the grid we need to use a one–sided formula for the steps 2. and 3., specifically

$$f'(x_0) \approx \frac{-3f_0 + 4f_1 - f_2}{2\Delta x}, \quad f'(x_{n-1}) \approx \frac{3f_{n-1} - 4f_{n-2} + f_{n-3}}{2\Delta x}$$

Writing the sequential algorithm


```
void firstderiv1D_vec(int n, double dx, double *f, double *fx){
 double scale;
 scale = 1.0/(2.0*dx);
 for (int i = 1; i < n-1; i++){
   fx[i] = (f[i+1] - f[i-1])*scale;
 }
 fx[0] = (-3.0*f[0] + 4.0*f[1] - f[2])*scale;
 fx[n-1] = (3.0*f[n-1] - 4.0*f[n-2] + f[n-3])*scale;
 return;
}
```

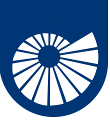
The function takes as input

- the number of grid points is n,
- the amplitude of such intervals Δx,
- the array containing the evaluation of f (intent: input),
- the array that will contain the value of the derivative (intent: output)

Writing the parallel algorithm

To implement the sequential differencing functions in parallel with MPI, we have to perform several steps

1. partition our domain $[a, b]$ among the processors,
2. each processor computes the FD for all the points contained on that processor

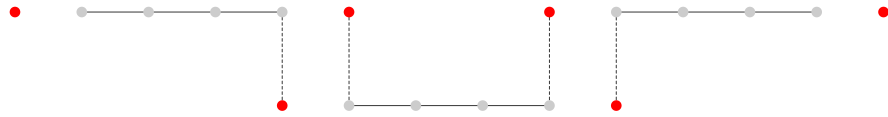


Writing the parallel algorithm

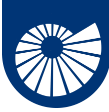
To implement the sequential differencing functions in parallel with MPI, we have to perform several steps

1. partition our domain $[a, b]$ among the processors,
2. each processor computes the FD for all the points contained on that processor

To actually perform the second step, we observe that the end-points on each subdomain needs information not contained on the processor, but that resides on a different one, we **need to communicate boundary data**!



Red dots are ***halo* data**, the one we need to communicate, gray dots are owned data.



Writing the parallel algorithm

The prototype of the function we want to write can be, in this case,

```
void firstderiv1Dp_vec(int n, double dx, double *f, double *fx,
int mynode, int totalnodes)
```

where

- `int n` is the number of points per process,
- `double dx` the amplitude of each interval,
- `double *f, double *fx` the local portions with the values of $f(x)$ (input) and $f'(x)$ (output),
- `int mynode` the rank of the current process,
- `int totalnodes` the size of the communicator

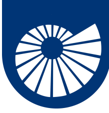
We declare then the variables

```
double scale = 1.0/(2.0*dx);
double mpitemp;
MPI_Status status;
```

Then we can treat the case in which we are at the beginning or at the end of the global interval

```
if(mynode == 0){
 fx[0] = (-3.0*f[0] + 4.0*f[1] - f[2])*scale;
}
if(mynode == (totalnodes-1)){
 fx[n-1] = (3.0*f[n-1] - 4.0*f[n-2] + f[n-3])*scale;
}
```

this approximate the derivative at the first and last point of the global interval.



Writing the parallel algorithm

Then we can treat the case in which we are at the beginning or at the end of the global interval

```
if(mynode == 0){
 fx[0] = (-3.0*f[0] + 4.0*f[1] - f[2])*scale;
}
if(mynode == (totalnodes-1)){
 fx[n-1] = (3.0*f[n-1] - 4.0*f[n-2] + f[n-3])*scale;
}
```

this approximate the derivative at the first and last point of the global interval.
Then, we can compute the inner part (the gray points) of the local interval by doing:

```
for(int i=1;i<n-1;i++){
 fx[i] = (f[i+1]-f[i-1])*scale;
}
```

The other case we need to treat is again the particular case in which we are in the first, or in the last interval. In both cases we have **only one communication** to perform

```
if(mynode == 0){
 mpitemp = f[n-1];
 MPI_Send();
 MPI_Recv();
 fx[n-1] = (mpitemp - f[n-2])*scale;
}
else if(mynode == (totalnodes-1)){
 MPI_Recv();
 fx[0] = (f[1]-mpitemp)*scale;
 mpitemp = f[0];
 MPI_Send();
}
```

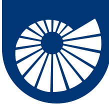
Writing the parallel algorithm

The other case we need to treat is again the particular case in which we are in the first, or in the last interval. In both cases we have **only one communication** to perform

```
if(mynode == 0){
 mpitemp = f[n-1];
 MPI_Send(&mpitemp,1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
 MPI_Recv(&mpitemp,1,MPI_DOUBLE,1,1,MPI_COMM_WORLD,&status);
 fx[n-1] = (mpitemp - f[n-2])*scale;
}
else if(mynode == (totalnodes-1)){
 MPI_Recv(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD,
 &status);
 fx[0] = (f[1]-mpitemp)*scale;
 mpitemp = f[0];
 MPI_Send(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD);
}
```

Finally, the only remaining case is the one in which we need to communicate both the extremes of the interval

```
else{
 MPI_Recv();
 fx[0] = (f[1]-mpitemp)*scale;
 mpitemp = f[0];
 MPI_Send();
 mpitemp = f[n-1];
 MPI_Send();
 MPI_Recv();
 fx[n-1] = (mpitemp-f[n-2])*scale;
}
```



Writing the parallel algorithm

Finally, the only remaining case is the one in which we need to communicate both the extremes of the interval

```
else{
 MPI_Recv(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD,
 &status);
 fx[0] = (f[1]-mpitemp)*scale;
 mpitemp = f[0];
 MPI_Send(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD);
 mpitemp = f[n-1];
 MPI_Send(&mpitemp,1,MPI_DOUBLE,mynode+1,1,MPI_COMM_WORLD);
 MPI_Recv(&mpitemp,1,MPI_DOUBLE,mynode+1,1,MPI_COMM_WORLD,
 &status);
 fx[n-1] = (mpitemp-f[n-2])*scale;
}
```

And the routine is complete!

Writing the parallel algorithm

A simple (and not very useful) principal program for this routine can be written by first initializing the parallel environment, and discovering who we are.

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &mynode );
MPI_Comm_size( MPI_COMM_WORLD, &totalnodes );
```

Then we build the local values of the f function

```
globala = 0; globalb = 1;
a = globala + ((double) mynode)*(globalb - globala)/( (double) totalnodes);
b = globala + ((double) mynode+1)*(globalb - globala)/( (double) totalnodes);
f  = (double *) malloc(sizeof(double)*(n));
fx = (double *) malloc(sizeof(double)*(n));
dx = (b-a)/((double) n);
for( int i = 0; i < n; i++){ f[i] = fun(a+((double) i)*dx); }
```

Finally we invoke our parallel computation

```
firstderiv1Dp_vec( n, dx, f, fx, mynode, totalnodes);
```

Writing the parallel algorithm

To check if what we have done makes sens we evaluate the error in the $\|\cdot\|_2$ norm on the grid, i.e., $\sqrt{\Delta x}\|\mathbf{f}' - \mathbf{fx}\|_2$ on every process

```
error = 0.0;
for(int i = 0; i < n; i++){
 error += pow( fx[i]-funprime(a+((b-a)*((double) i))/((double) n)),2.0);
}
error = sqrt(dx*error);
printf("Node %d ||f' - fx||_2 = %e\n",mynode,error);
```

Then we clear the memory and close the parallel environment

```
free(f);
free(fx);
MPI_Finalize();
```

Table of Contents

- Setup
- Point-to-Point Communications
  - Deadlock
  - Nonblocking communications
  - Sendreceive
  - Things left out
- A first scientific computation
- References

References

There are more books, notes, tutorials, online courses and oral tradition on scientific and parallel computing than we would have time to read and listen in a life. Pretty much everything that contains the words Parallel Programming and Scientific Computing is good...

I suggest here the book

[1] Rouson, D., Xia, J., & Xu, X. (2011). Scientific software design: the object-oriented way. Cambridge University Press.

that discusses general aspect of scientific computing (not perfectly related to parallel computing), and to have on your bedside

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0. `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`, High Performance Computing Center Stuttgart (HLRS).