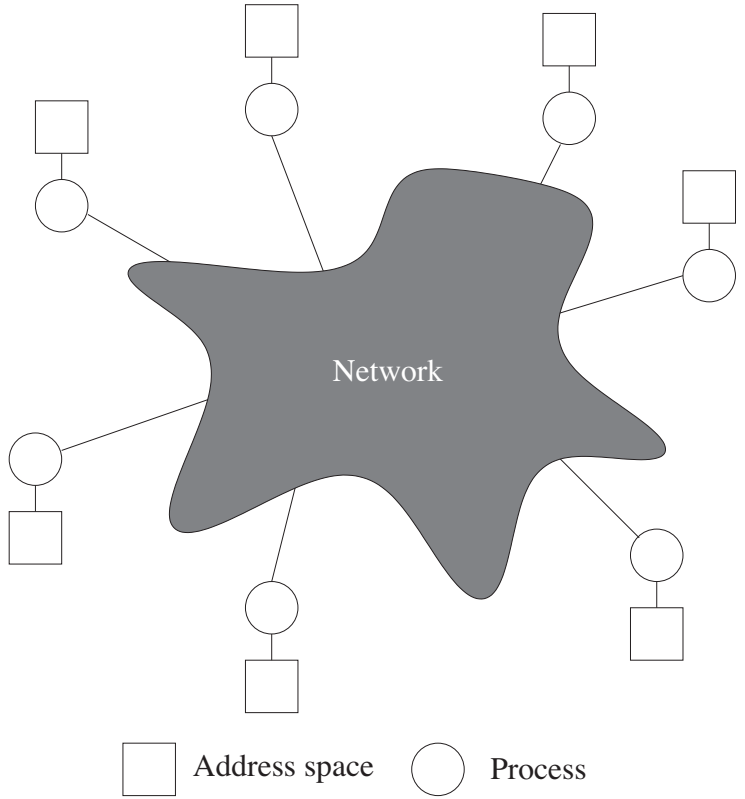
Obstacles

- Inter-communication networks (switches) are still slow compared with intra-processor speeds (although significant advances have been made recently).
- Compilers that automatically parallelize sequential algorithms remain very limited in their capabilities.
- Trade-off between expressivity, portability, and efficiency.

The Message-Passing Model

universidade de aveiro

- Processes have only local memory.
- Processes are able to communicate with each other by sending and receiving messages.
- Communication operations between two process (i.e., transfer data from local memory of one process to local memory of another) must be performed by both process.



Advantages of the Message-Passing Model (particularly MPI)

universidade de aveiro

- Universality
  - Works well with fast and slow communication networks (from parallel supercomputers to workstation networks or dedicated PC clusters).
  - Whenever the hardware supplies shared-memory, the message-passing can use it to speed up data transfer among processes.
  - GPUs can be used with the MPI.
- Expressivity
  - Is a complete model to express parallel algorithms.
  - Provides high control over local data.
  - It is well suited for self-scheduling algorithms, and to deal with imbalances in process speed found in heterogeneous networks.

Advantages of the Message-Passing Model (particularly MPI)

universidade de aveiro

- Ease of debugging
  - Debugging parallel programs remains a challenge.
  - Compartmentalization of memory in MPI makes it easier to find the wrong reads and writes.

- Performance:
  - Memory (and cache) management is key to extract maximum performance from modern CPUs.
  - MPI provides a way for the programmer explicitly associate specific data to processes, which allows both compilers and cache-management hardware to function fully.

What is MPI?

- MPI is a library, not a language. It specifies names, calling sequences and results of functions (which are called from C or Fortran).
- MPI is a standard, or specification, it is not a particular implementation. In this discipline we will use the implementation *mpich*, although a correct MPI program should run on any MPI implementation without changes.
- MPI addresses the message-passing model of parallel computing described above. That is, a collection of processes (with only local memory) communicating using messages.

MPI basic concepts: a minimal interface

universidade de aveiro

- Each communication requires the cooperation of the processes involved; while one process execute a send operation, the other must execute a receive.
- Minimal set of arguments for the `send` and `receive` functions:
  - *sender*: data to be sent (address and length of message), and identity of destination process.
  - *receiver*: address and length of space in local memory to store received data, variable to be filled with identity of *sender* process (so the receiver can know who sent the message).

MPI basic concepts: a minimal interface

universidade de aveiro

- In practice more features may be useful, or even required, by many applications.
- Matching: a process is able to control the messages it receives by using a `tag` (an integer that specifies the 'type' of message). The `tag` is an argument of both the `send` and `receive` functions. In a `receive` operation, it may also be convenient to specify the identity of the *sender*, as an additional screening parameter.
- The `length` of the message received may not be known beforehand. The `receive` specifies a maximum length for the message but allows shorter messages to arrive. So, the actual length of the message is returned in `actlen`.

MPI basic concepts: a minimal interface

universidade de aveiro

```
send(address, length, destination, tag)

receive(address, length, source, tag, actlen)
```

- Problems:
  - The buffer may not be continuous.
  - Different representations of the same information (integer values, floating-point values, etc) in different machines.

- Solution:
  - Message buffer is defined by a triple `(address, count, datatype)`, where `datatype` can be a user defined datatype (or *derived datatype*) that maps noncontiguous memory addresses.

MPI basic concepts: a minimal interface

```
send(address, length, destination, tag)

receive(address, length, source, tag, actlen)
```

- Another problem:
  - *Tags* are integers chosen arbitrary, but must be used in predefined a coherent way throughout the whole program. Complications arise particularly when using libraries written by others whose *tags* may overlap with ours: context is required for correct *tag* interpretation.
- Solution:
  - *Communicators* are objects that combine the notions of context and group of process. Processes belong to groups and, within a group, are identified by *ranks*. The same process may belong to several groups, and within each group is identified with a different *rank*. Each group has its own *communicator*, which is an argument of all communication operations. The destination or `source` arguments of a `send` or `receive` refers to the *rank* of the process in the group identified by the given *communicator*.

MPI basic send and receive operations (blocking)

universidade de aveiro

```
MPI_Send(address, count, datatype, destination, tag, comm)
```

- `(address, count, datatype)` describes `count` occurrences of items of the form `datatype` starting at `address`,
- `destination` is the *rank* of the destination in the group associated with the *communicator* `comm`,
- `tag` is an integer used for message matching, and
- `comm` is the *communicator,* identifies a group of processes and a communication context.

MPI basic send and receive operations (blocking)

universidade de aveiro

`MPI_Recv(address, maxcount, datatype, source, tag, comm, status)`

- `(address, maxcount, datatype)` are the same as in MPI_Send, although it is allowed for less than `maxcount` occurrences to be received,
- `tag` and `comm` are as in MPI_Send, with the addition that a wildcard, matching any `tag`, is allowed.
- The `source` is the *rank* of the source of the message in the group associated with the *communicator* `comm`, or a wildcard matching any source.
- Finally, `status` holds information about the actual message size, `source`, and `tag`, useful when wildcards have been used.

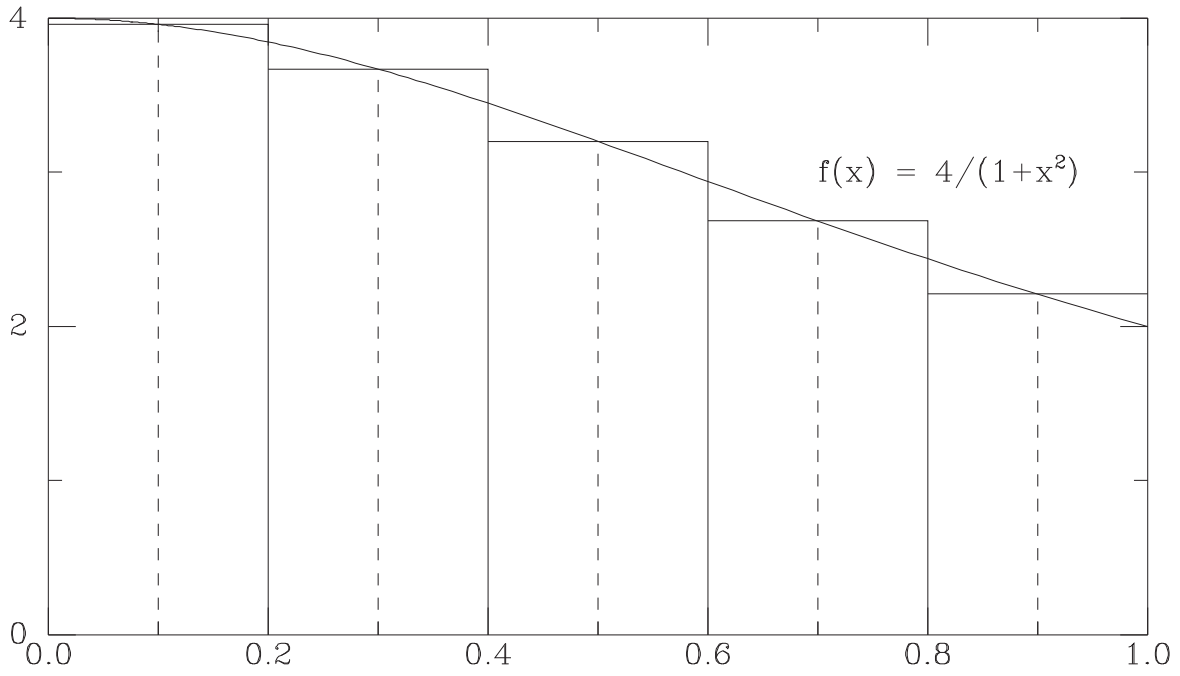
Some other useful features

- Collective operations: collective data movement and collective computations;
- Virtual topologies;
- Communication modes: blocking, non-blocking, synchronous, buffered, ready;
- Debugging and profiling;
- Support for libraries;
- Support for heterogeneous networks;
- Processes vs processors.

A simple parallel program - calculation of an integral

universidade de aveiro

- Goal: obtain π from $\int_0^1 {}^4/_{1+x^2}\, dx = \pi$.
- Calculate and sum the area of `n` rectangles, as in the figure.
- Each process is responsible calculating the contribution of a sub-set of rectangles.



A simple parallel program - calculation of an integral

universidade de aveiro

- Keeping things simple, we will use only collective communication operations:

```
MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD)
MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION,
MPI_SUM, 0, MPI_COMM_WORLD)
```

- Additionally, we are need to initialize the MPI 'environment':

```
MPI_Init(&argc,&argv)
```

A simple parallel program - calculation of an integral

- Each process needs to know the total number of processes and its own identification (*rank*) within the group associated with the default communicator `MPI_COMM_WORLD`:

```
MPI_Comm_size(MPI_COMM_WORLD, &numprocs)
MPI_Comm_rank(MPI_COMM_WORLD, &myid)
```

- Finally, at the end of the program every process must terminate the MPI 'environment':

```
MPI_Finalize()
```

Compiling and running MPI programs

universidade de aveiro

- The *mpich* implementation provides an MPI C compiler: `mpicc`. The syntax is similar to `gcc`, `icc`, etc, and it allows to link any desired C libraries, and other standard C compiler options:

```
mpicc -o prog prog.c -mylib
```

- Run `prog` in 4 parallel processes with the command:

```
mpiexec -n 4 ./prog
```

Point-to-point communications
Standard send (blocking)

universidade de aveiro 

```
MPI_Send(address, count, datatype, destination,
tag, comm)
```

- `(address, count, datatype)` describes `count` occurrences of items of the form `datatype` starting at `address`,
- `destination` is the *rank* of the destination in the group associated with the *communicator* `comm`,
- `tag` is an integer used for message matching, and
- `comm` is the *communicator,* identifies a group of processes and a communication context.

Point-to-point communications
Standard receive (blocking)

```
MPI_Recv(address, maxcount, datatype, source,
tag, comm, status)
```

- `(address, maxcount, datatype)` are the same as in MPI_Send, although it is allowed for less than `maxcount` occurrences to be received,
- `tag` and `comm` are as in MPI_Send, with the addition that a wildcard, matching any `tag`, is allowed.
- The `source` is the *rank* of the source of the message in the group associated with the *communicator* `comm`, or a wildcard matching any source.
- Finally, `status` holds information about the actual message size, `source`, and `tag`, useful when wild cards have been used.

Point-to-point communications
Synchronous send

```
MPI_Ssend(address, count, datatype, destination, tag, comm)
```

`MPI_Ssend` has the same arguments as `MPI_Send`, but only returns when *receiver* process finishes receiving the message.

The point of the synchronous send operations is avoiding the *sender* process to change the values to be sent before the sending actually occurs.

Point-to-point communications
Buffered send

```
MPI_Bsend(address, count, datatype, destination,
tag, comm)
```

`MPI_Bsend` has similar arguments as `MPI_Send` and `MPI_Ssend`, but uses a buffer to store the message while the *receiver* is not ready.

This way, the *sender* process can proceed without the risk of overwriting the message to be sent.

In buffered sends, it is necessary need to allocate enough memory for the buffer, and attach/detach it with:

```
MPI_Buffer_attach(buffer,count);
MPI_Buffer_detach(buffer,count).
```

Timing MPI programs

Timing of parallel programs is especially relevant, since the goal of parallelization is to reduce execution time.

MPI provides a function for timing programs, and sections of programs:

```
MPI_Wtime()
```

Calling `MPI_Wtime()` returns the number of seconds that have passed since some arbitrary point of time in the past, which does not change during the execution of the process.

Elapsed time can be measured with the difference two calls of `MPI_Wtime()`.

Timing MPI programs

The resolution of the output of `MPI_Wtime` is hardware dependent, and can be found by calling

```
MPI_Wtick()
```

Another function that becomes useful for timing programs is

```
MPI_Barrier(comm)
```

This function is a collective operation that does nor let the calling process to continue until all processes in the communicator `comm` have called `MPI_Barrier`.

Manager-worker prototype

- One of the processes, called *manager*, is responsible for coordinating the work of the other processes, called *workers*.
- This kind of algorithm is especially appropriate when:
  - *worker* processes do not have to communicate with each other,
  - and the amount of work to be performed by each *worker* is difficult to predict.
- Communications will be made individually between the manager and each of the workers (point-to-point communications).

A self-scheduling example: Matrix-vector multiplication

- Given a matrix $\hat{A}$ and a vector $\vec{b}$, calculate the vector $\vec{c}$ resulting from the product of $\hat{A}$ by $\vec{b}$:

$$\vec{c} = \hat{A}\vec{b}.$$

- The unit of work to be given out by the *manager* to the *workers* consists of the dot product between a row of matrix $\hat{A}$ by the vector $\vec{b}$, which returns

$$c_i = \sum_j A_{ij} b_i.$$

Self-scheduling matrix-vector multiplication algorithm

universidade de aveiro

Manager Part

- The *manager* begins by broadcasting $\vec{b}$ to all *workers*.
- Initially the *manager* sends a row of $\hat{A}$ to each *worker*, and then starts a loop which will terminate when all of the c_i's have been received.
- In each step of the loop the *manager* receives a c_i from whichever *worker* sends one first, and sends the next task (row of $\hat{A}$) to that *worker*.
- Once all tasks have been handed out to the *workers,* termination messages are sent instead.

Self-scheduling matrix-vector multiplication algorithm

universidade de aveiro

Worker Part

- After each *worker* receives the broadcast of vector $\vec{b}$, it also enters a loop.
- In each step of the loop the *worker*
  i. receives a row of $\hat{A}$,
  ii. calculates the dot product of that row with $\vec{b}$,
  iii. and sends the result back to the *manager*.
- The *worker* exits the loop when the termination message is received from the *manager*.

Self-scheduling matrix-vector multiplication algorithm

universidade de aveiro

The code for this program is divided in three parts: the *manager* and *worker* parts described above, and the part that is common to both *manager* and *workers.*

Common part

- MPI initialization
- Variable declarations and initializations
- Memory allocations
- MPI_Finalize()

Self-scheduling sends and receives

universidade de aveiro 

The distinctive feature of self-scheduling programs is that the *manager* is prepared to receive messages from whichever *worker* sends one first.

So, the receive function called by the *manager* must allow for the message to arrive from any *worker* (`source`) with any `tag`:

```
MPI_Recv(&ans, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

Nevertheless, the *manager* still needs to know who was the `source` of the message (which is also destination of the next task), and the `tag` with which the message was sent (the `tag` is used to tell the *manager* where to store `ans`, ie. `tag` is the matrix row's index).

Self-scheduling sends and receives

universidade de aveiro

Similarly, the receive function of the *worker* must allow for any `tag` (matrix row) of the received message:

```
MPI_Recv(row, ncols, MPI_INT, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
```

(Here the rank of the *manager* is `0`.)

The actual `source` and `tag` of a message received can be retrieved from the `status` parameter as:

```
source = status.MPI_SOURCE
tag = status.MPI_TAG
```

Self-scheduling sends and receives

Beware, the sends must always indicate the destination and `tag` of the message, both for the *manager*

```
MPI_Send(row, ncols, MPI_INT, worker_rank,
row_index, MPI_COMM_WORLD);
```

and for the *workers*

```
MPI_Send(&ans, 1, MPI_INT, 0, row_index,
MPI_COMM_WORLD);
```

Additional notes

- Define/load matrix $\hat{A}$ and vector $\vec{b}$ in the *manager* only.
- Use collective `MPI_Bcast` to pass $\vec{b}$ onto the *workers*:

```
MPI_Bcast(b, ncols, MPI_INT, manager_rank,
MPI_COMM_WORLD);
```

- Termination messages to the *workers* will be sent with a particular value of `tag` different from all possible values of `row_index`, for example `tag_term = nrows+1`:

```
MPI_Send(MPI_BOTTOM, 0, MPI_INT, worker_rank,
tag_term, MPI_COMM_WORLD);
```

- Allow for a number of rows smaller than the number of *workers*.

Recall Jacobi algorithm

universidade de aveiro

- Jacobi method finds solutions of *diagonally dominant* systems of linear equations of the type $\hat{A}\vec{x} = \vec{b}$, using an iterative procedure.
- Decomposing $\hat{A} = \left(\hat{A} - \widehat{D}\right) + \widehat{D}$, and rearranging the matrix eq. we write $\vec{x} = (\hat{I} - \widehat{D}^{-1}\hat{A})\vec{x} + \widehat{D}^{-1}\vec{b}$.
- Computing the right-hand side for arbitrary vector, say $\vec{x}^{(i)}$, results in a vector $\vec{x}^{(i+1)}$ that better approximates the desired solution. In other words, by iterating enough times

$$\vec{x}^{(i+1)} = (\hat{I} - \widehat{D}^{-1}\hat{A})\vec{x}^{(i)} + \widehat{D}^{-1}\vec{b}$$

the vectors $\vec{x}^{(i)}$ converge to solution of the original system.

Recall 2D Poisson equation

universidade de aveiro

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)V(x,y)=f(x,y)$$

Approximate 2nd derivatives with finite differences

$$\frac{\partial^2}{\partial x^2}V(x,y)\approx\frac{1}{h^2}[V(x+h,y)-2V(x,y)+V(x-h,y)],$$

$$\frac{\partial^2}{\partial y^2}V(x,y)\approx\frac{1}{h^2}[V(x,y+h)-2V(x,y)+V(x,y-h)],$$

and write the (Jacobi) iterative discretized Poisson equation as

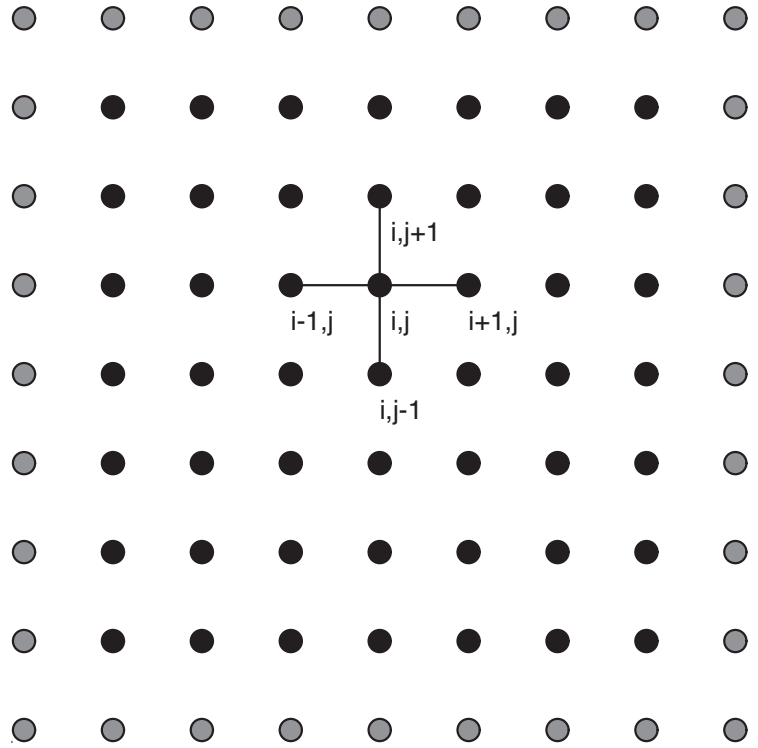
$$V_{i,j}{}^{(i+1)}=\frac{1}{4}\left(V_{i+1,j}{}^{(i)}+V_{i-1,j}{}^{(i)}+V_{i,j+1}{}^{(i)}+V_{i,j-1}{}^{(i)}-h^2f_{i,j}\right).$$

Boundary condition is needed to stop the recursion at domain border.

Recall 2D Poisson equation

universidade de aveiro

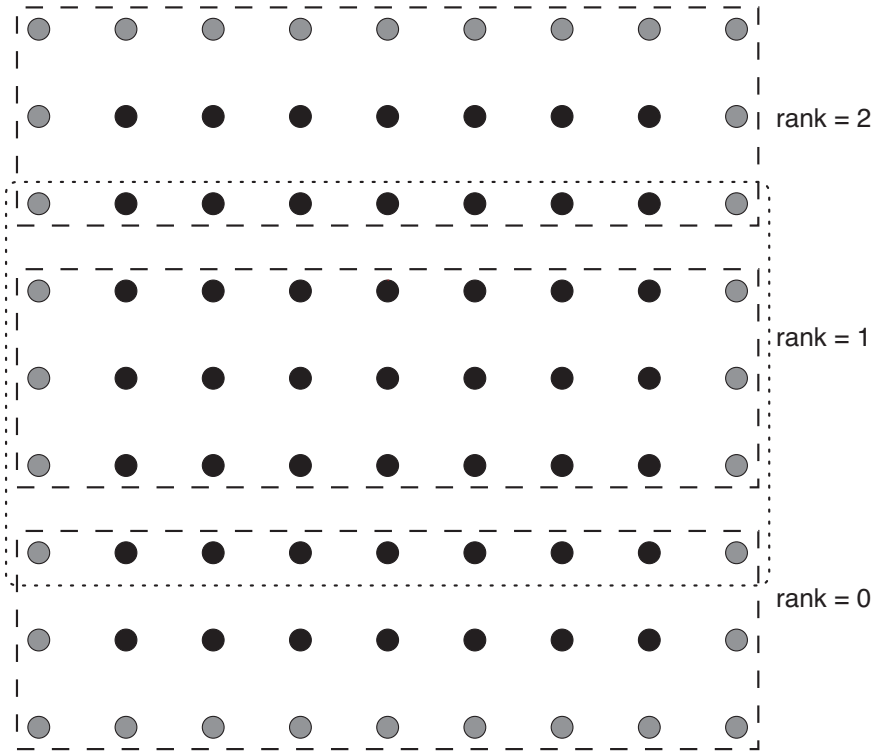
```
integer i, j, n
double precision u(0:n+1,0:n+1), unew(0:n+1,0:n+1)
do j=1, n
   do i=1, n
      unew(i,j) = &
          0.25*(u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j) - &
          h * h * f(i,j))
   enddo
enddo
```



2D Jacobi parallel algorithm
Domain decompositions

universidade de aveiro

```
integer i, j, n
double precision u(0:n+1,s:e), unew(0:n+1,s:e)
do j=s, e
   do i=1, n
      unew(i,j) = &
          0.25*(u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j) - &
          h * h * f(i,j))
   enddo
enddo
```



Virtual Topologies

universidade de aveiro 

Creating a new Cartesian communicator:

```
MPI_Cart_create(old_comm, ndims, dims[],
isperiodic[], reorder, &new_cart_comm)
```

Getting the rank of neighbours (in cartesian decomposition) with whom there will be communications:

```
MPI_Cart_shift(new_cart_comm, direction, displ,
&src, &dest)
```

Avoiding communications deadlock with MPI_Sendrecv

universidade de aveiro

```
MPI_Sendrecv(&sendbuf, sendcount, sendtype,
dest, sendtag, &recvbuf, recvcount, recvtype,
source, recvtag, comm, MPI_STATUS_IGNORE)
```

Collective data movements: MPI_Scatter & MPI_Gather

universidade de aveiro

```
MPI_Scatter(&sendbuf, sendcount, sendtype,
&recvbuf, recvcount, recvtype, root, comm)
```

```
MPI_Gather(&sendbuf, sendcount, sendtype,
&recvbuf, recvcount, recvtype, root, comm)
```

Derived Datatypes

universidade de aveiro 

- MPI supports a wide range of functions creation of custom user-defined datatypes.
- These are typically used to transfer pieces of data that are stored/saved non-contiguously in the local memory, avoiding the need for repeated invocations of send/receive instructions, with advantages for simplicity of code writing and especially for communications efficiency.
- MPI derived datatypes may include different elementary datatypes (`int`, `float`, etc) with arbitrary spacings between them.

Derived Datatypes

In many kinds situations it is useful (and recommendable) to take advantage of these derived datatypes. Consider, as an example relevant to our programs, the communication of a column of a 2D array:
In memory the matrix is stored in row-major order, and elements to be sent and received are scattered with regular spacings of `ncols` between them. To deal with this inconvenience we may:

- Send the elements one by one using a cycle, at the cost of a lot of overhead in communications;
- Copy the elements to a contiguous buffer, and make a regular send, with still some overhead and extra coding work;
- Define a vector-like datatype, including skips of `ncols-1` spaces, and access the desired elements directly in a single operation.

Derived Datatypes

universidade de aveiro 

Suppose process 1 has a matrix `a[nrows][ncols]` of `double float` numbers and we want it to send the second column to process 0, who will store it as the last column of its own matrix `b[nrows][ncols]`. Note that `ncols` may be different in processes 0 and 1.

Each process defines a new vector-like datatype with

```
MPI_Type_vector(nrows, 1, ncols, MPI_DOUBLE, &column);
```

followed by

```
MPI_Type_commit(&column);
```

Derived Datatypes

universidade de aveiro 

To send and receive the column we use any standard communication function and provide the new datatype as an argument. For example, process 1 could call a send as,

```
MPI_Send(a[0][1], 1, column, 0, tag, comm)
```

While process 0 could receive the message with a

```
MPI_Recv(b[0][ncols-1], 1, column, 1, tag, comm,
MPI_STATUS_IGNORE)
```

Parallel reading and writing files

universidade de aveiro 

- MPI contains the notion of *file view*, which provides an easy way of multiple processes to read and write separate parts of the same file.
- Each process has its own *file view* that defines which parts (contiguous or non-contiguous) of the file are visible to the process. A read or write function can only read or write the data that is visible to the process, and skips all other data.
- The *file views* are specified using basic and derived datatypes.
- For example, in the cartesian decomposition of our programs we want each process to write a sub-matrix that is a part of the global matrix to be stored in the file.

Parallel reading and writing files

universidade de aveiro 

Each process defines a datatype that allows access only to its own sub-matrix in the global matrix:

```
MPI_Type_create_subarray(2, gsizes, lsizes,
start_ind, MPI_ORDER_C,MPI_DOUBLE, &filetype);
```

not forgetting to commit

```
MPI_Type_commit(&filetype);
```

- The fist argument is the number of dimensions of the array, we use 2.
- `gsizes` are the sizes of the global array in each dimension.
- `lsizes` are the sizes of the local array in each dimension.
- `start_ind` is the position of the first element of the subarray on the global array.

Parallel reading and writing files

universidade de aveiro 

In our programs the local matrices have extra ghost rows and/or columns that we do not wish to be written by the process, because they belong to neighboring processes and should be written by those neighbors.

This issue can also be fixed with the same function, defining a sub-array datatype that includes all the points of the local matrix that belong to the process, and excludes the ghost points:

```
MPI_Type_create_subarray(2, memsizes, lsizes,
start_ind, MPI_ORDER_C, MPI_DOUBLE, &memtype);
```

Parallel reading and writing files

universidade de aveiro 

At last, we are ready to create and open a file for storing the global matrix

```
MPI_File fp;

MPI_File_open(comm, ”matrix.bin", MPI_MODE_CREATE
| MPI_MODE_WRONLY, MPI_INFO_NULL, &fp);
```

And to change the *file view* of each process using the derived `filetype`

```
MPI_File_set_view(fp, 0, MPI_DOUBLE, filetype,
"native", MPI_INFO_NULL);
```

(The second argument is an offset to the beginning of the file. We use `0` since we created `filetype` with `MPI_Type_create_subarray`).

Parallel reading and writing files

universidade de aveiro 

Now that each process ‘sees’ only its own part of the file, we can use the MPI’s convenient collective I/O operations for each process to write the desired data in the desired part of the file:

```
MPI_File_write_all(fp, myVnew, 1, memtype,
MPI_STATUS_IGNORE);
```

Notice that to this function only the local memory subarray `memtype` is passed explicitly. The subarray `filetype` was already used to define the *file view* when we used `MPI_File_set_view()`, and is held by `fp`.

Lets not forget to close the file:

```
MPI_File_close(&fp);
```

MPI_Send:

Syntax: MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Function: Sends a message from the specified buffer to a process with the given destination rank.

Arguments:

buf: Pointer to the send buffer.

count: Number of elements in the send buffer.

datatype: MPI datatype of the elements being sent.

dest: Rank of the destination process.

tag: Message tag for the communication.

comm: Communicator that defines the group of processes involved in the communication.

MPI_Ssend:

Syntax: MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Function: Similar to MPI_Send, but it uses synchronous send mode, ensuring the receive operation starts before the send call completes.

Arguments: Same as MPI_Send.

MPI_Isend:

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

Function: Initiates a non-blocking send operation, allowing the program to continue execution without waiting for the send to complete.

Arguments:

buf: Pointer to the send buffer.

count: Number of elements in the send buffer.

datatype: MPI datatype of the elements being sent.

dest: Rank of the destination process.

tag: Message tag for the communication.

comm: Communicator that defines the group of processes involved in the communication.

request: Pointer to an MPI_Request object that can be used to query the status of the send operation.

MPI_Issend:

Syntax: MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request* request)

Function: Similar to MPI_Isend, but uses synchronous send mode, ensuring the receive operation starts before the send call completes.

Arguments: Same as MPI_Isend.

MPI_Bsend:

Syntax: MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Function: Performs a buffered send operation, which uses an intermediate buffer to store the message until it is received by the destination process.

Arguments: Same as MPI_Send.

MPI_Recv:

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

Function: Receives a message from a specific source process and stores it in the specified receive buffer.

Arguments:

buf: Pointer to the receive buffer.

count: Maximum number of elements that can be received.

datatype: MPI datatype of the elements being received.

source: Rank of the source process.

tag: Message tag for the communication.

comm: Communicator that defines the group of processes involved in the communication.

status: Pointer to an MPI_Status object that provides information about the received message.

MPI_Sendrecv:

Syntax: MPI_Sendrecv(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)

Function: Combines both sending and receiving into a single call, allowing simultaneous communication between two processes.

Arguments:

sendbuf: Pointer to the send buffer.

sendcount: Number of elements in the send buffer.

sendtype: MPI datatype of the elements being sent.

dest: Rank of the destination process.

sendtag: Message tag for the send operation.

recvbuf: Pointer to the receive buffer.

recvcount: Maximum number of elements that can be received.

recvtype: MPI datatype of the elements being received.

source: Rank of the source process.

recvtag: Message tag for the receive operation.

comm: Communicator that defines the group of processes involved in the communication.

status: Pointer to an MPI_Status object that provides information about the received message.

MPI_Scatter:

Syntax: MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Function: Distributes data from the root process to all processes in the communicator in a scatter-like fashion.

Arguments:

sendbuf: Pointer to the send buffer (used by the root process).

sendcount: Number of elements sent to each process from the send buffer.

sendtype: MPI datatype of the elements being sent.

recvbuf: Pointer to the receive buffer (used by each process).

recvcount: Number of elements received by each process into the receive buffer.

recvtype: MPI datatype of the elements being received.

root: Rank of the root process (the process that scatters the data).

comm: Communicator that defines the group of processes involved in the communication.

MPI_Gather:

Syntax: MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Function: Gathers data from all processes in the communicator to the root process.

Arguments: Same as MPI_Scatter, but with reversed roles for sendbuf and recvbuf.

MPI_Allgather:

Syntax: MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Function: Gathers data from all processes and distributes it to all processes in the communicator, creating a copy of the data on each process.

Arguments: Same as MPI_Gather, but with recvbuf used by all processes.

MPI_Allgatherv:

Syntax: MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int* recvcounts, int* displs, MPI_Datatype recvtype, MPI_Comm comm)

Function: Gathers data from all processes and distributes it to all processes, with varying amounts of data received by each process.

Arguments:

sendbuf: Pointer to the send buffer (used by each process).

sendcount: Number of elements sent by each process from the send buffer.

sendtype: MPI datatype of the elements being sent.

recvbuf: Pointer to the receive buffer (used by each process).

recvcounts: Array specifying the number of elements received from each process.

displs: Array specifying the displacement of the data for each process in the receive buffer.

recvtype: MPI datatype of the elements being received.

comm: Communicator that defines the group of processes involved in the communication.

MPI_Barrier:

Syntax: MPI_Barrier(MPI_Comm comm)

Function: Synchronizes all processes in the communicator, ensuring that no process proceeds past the barrier until all processes have reached it.

Arguments:

comm: Communicator that defines the group of processes involved in the barrier synchronization.

MPI_Bcast:

Syntax: MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Function: Broadcasts data from the root process to all other processes in the communicator.

Arguments:

buffer: Pointer to the send/receive buffer used by each process.

count: Number of elements sent/received by each process.

datatype: MPI datatype of the elements being sent/received.

root: Rank of the root process (the process that broadcasts the data).

comm: Communicator that defines the group of processes involved in the communication.

MPI_Reduce:

Syntax: MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Function: Reduces values from all processes in the communicator to a single result on the root process, using a specified reduction operation.

Arguments:

sendbuf: Pointer to the send buffer (used by each process).

recvbuf: Pointer to the receive buffer (used by the root process).

count: Number of elements sent/received by each process.

datatype: MPI datatype of the elements being sent/received.

op: MPI reduction operation to be applied during the reduction.

root: Rank of the root process (the process that receives the reduced result).

comm: Communicator that defines the group of processes involved in the communication.

MPI_Cart_Shift:

Syntax: MPI_Cart_Shift(MPI_Comm comm, int direction, int displ, int* source, int* dest)

Function: Computes the source and destination ranks for shifting data in a Cartesian topology.

Arguments:

comm: Communicator that defines the Cartesian topology.

direction: Coordinate direction of the shift.

displ: Number of positions to shift.

source: Pointer to the variable that will receive the rank of the source process.

dest: Pointer to the variable that will receive the rank of the destination process.

MPI_Cart_Create:

Syntax: MPI_Cart_Create(MPI_Comm comm_old, int ndims, const int* dims, const int* periods, int reorder, MPI_Comm* comm_cart)

Function: Creates a new communicator with a Cartesian topology based on an existing communicator.

Arguments:

comm_old: The original communicator.

ndims: Number of dimensions in the Cartesian grid.

dims: Array specifying the size of each dimension.

periods: Array specifying whether each dimension is periodic or not.

reorder: Flag indicating whether ranks can be reordered.

comm_cart: Pointer to the new communicator with the Cartesian topology.

MPI_Comm_rank:

Syntax: MPI_Comm_rank(MPI_Comm comm, int* rank)

Function: Retrieves the rank of the calling process in the specified communicator.

Arguments:

comm: The communicator.

rank: Pointer to the variable that will receive the rank of the calling process.

MPI_Comm_size:

Syntax: MPI_Comm_size(MPI_Comm comm, int* size)

Function: Retrieves the size (number of processes) in the specified communicator.

Arguments:

comm: The communicator.

size: Pointer to the variable that will receive the size of the communicator.

MPI_Type_Vector:

Syntax: MPI_Type_Vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype* newtype)

Function: Creates a new datatype representing a strided vector derived from an existing datatype.

Arguments:

count: Number of blocks in the vector.

blocklength: Number of elements in each block.

stride: Distance between the start of each block (in multiples of the old datatype size).

oldtype: The old datatype to be used as the base type.

newtype: Pointer to the variable that will receive the newly created datatype.

MPI_Type_create_subarray:

Syntax: MPI_Type_create_subarray(int ndims, const int* sizes, const int* subsizes, const int* starts, int order, MPI_Datatype oldtype, MPI_Datatype* newtype)

Function: Creates a new datatype representing a subarray derived from an existing datatype.

Arguments:

ndims: Number of dimensions in the full array.

sizes: Array specifying the size of each dimension in the full array.

subsizes: Array specifying the size of each dimension in the subarray.

starts: Array specifying the starting position of the subarray in each dimension of the full array.

order: Ordering of the dimensions (either MPI_ORDER_C or MPI_ORDER_FORTRAN).

oldtype: The old datatype to be used as the base type.

newtype: Pointer to the variable that will receive the newly created datatype.

Computação Paralela – MPI — 2019/2020

Exercício 1
Comunicação ponto-a-ponto bloqueante

Na primeira aula de do módulo de MPI de Computação Paralela analisámos sem profundidade um primeiro programa MPI para ficar com uma ideia geral de como se usam as bibliotecas MPI e de como se compilam e executam os programas.

Curiosamente, não usámos as funções MPI de comunicação ponto-a-ponto, ou seja, aquelas que servem para trocar mensagens entre 2 processos de ordens (*ranks*) especificadas. A subrotina/função padrão MPI para enviar uma mensagem ponto-a-ponto é

```
MPI_Send(address, count, datatype, destination, tag, comm)
```

Dos seis argumentos, apenas o quarto e o quinto não foram discutidos na aula anterior:

- `destination` é a ordem, dentro do comunicador `comm`, do processo a que se destina a mensagem.
- `tag` (etiqueta) é um inteiro que identifica a mensagem e permite ao destinatário distinguir diferentes mensagens provenientes de um mesmo remetente.

Para que o destinatário receba a mensagem, tem que chamar a função

```
MPI_Recv(address, maxcount, datatype, source, tag, comm, status)
```

Na lista de argumentos,

- `maxcount` é o número máximo de elementos do tipo `datatype` que o destinatário aceita receber. É possível que o destinatário não tenha informação prévia sobre o tamanho da mensagem que vem do remetente.
- `source` é a ordem, dentro do comunicador `comm`, do processo que emitiu a mensagem. É possível, em alternativa, usar `MPI_ANY_SOURCE` como argumento e, assim, aceitar uma mensagem de qualquer proveniência.
- `tag` é o inteiro que o destinatário usou para identificar a mensagem. É possível, em alternativa, usar `MPI_ANY_TAG` como argumento e, assim, aceitar uma mensagem com qualquer etiqueta.
- `status` contém informação sobre o tamanho, origem e etiqueta da mensagem efetivamente recebida. Vimos que é possível que qualquer um destes valores possa não ser conhecido à partida. Quando não estamos interessados nesta informação, podemos usar `MPI_STATUS_IGNORE`.

Os vínculos (*bindings*) destas funções em C são:

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

Algoritmo 1: Comunicação ponto-a-ponto

Input: Número de elementos n de um vetor b de inteiros (0 para sair).
Output: Mensagens com informações sobre as comunicações.

```
begin
    inicialização do MPI
    while n ≠ 0 do
        if rank = 0 then                    /* Tarefa exclusiva para o processo 0) */
            leitura de n
        o master faz o broadcast de n, os outros recebem
        se n = 0, todos os processos terminam
        todos os processos definem o vetor b
        esperam uns pelos outros, para começar a contar o tempo no
          mesmo instante
        if rank = 0 then                    /* Tarefa exclusiva para o processo 0) */
            atribui valores ao vetor b
            envia o vetor b ao processo 1
            avisa quando está pronto para continuar
        else if rank = 1 then               /* Tarefas exclusivas para o processo 1) */
            faz-se difícil por 5 segundos
            recebe o vetor b do processo 0
            avisa quando está pronto para continuar
        else                                /* Tarefa exclusiva para os outros processos) */
            não fazem nada
```

O objetivo principal deste exercício é perceber como funciona a comunicação ponto-a-ponto bloqueante. Como vamos observar no fim deste exercício, o problema das funções padrão de comunicação ponto-a-ponto bloqueante é que o seu funcionamento não está completamente definido e depende da implementação. Vamos começar por usar, em vez da função padrão `MPI_Send`, a função

```
MPI_Ssend(address, count, datatype, destination, tag, comm)
```

Esta função define uma comunicação síncrona. Isto quer dizer que a mensagem é enviada diretamente ao destinatário e que o programa do remetente só poderá avançar para a seguinte instrução quando o destinatário tiver confirmado que acabou de receber a mensagem. Note que o destinatário continua a usar a função `MPI_Recv`.

O algoritmo que vamos usar até ao fim do exercício é o Algoritmo 1. Vamos estudar a diferença de resultados quando usamos diferentes modos de comunicação ponto-a-ponto bloqueante. O essencial do algoritmo é o seguinte: o processo de ordem 0 envia ao processo de ordem 1 uma mensagem de tamanho escolhido pelo utilizador, mas o segundo processo decide esperar 5 segundos antes de executar a função que recebe a mensagem.

Compile e execute o Programa 1, que usa comunicação bloqueante síncrona. Irá observar que o processo 0 só irá estar pronto para continuar ao fim de mais de 5 segundos, porque para

além do tempo usado no processo de comunicação, tem que ficar à espera que o destinatário inicie a receção. O programa está pensado para correr apenas em 2 processos: aqueles que usar a mais vão apenas ocupar memória sem necessidade (seria fácil corrigir isto). Note que o Programa 1 introduz pela primeira vez mais uma função MPI. Para podermos analisar os tempos medidos, temos que sicronizar o instante em que o processo 0 começa a enviar a mensagem[1] e o instante em que o processo 1 inicia os seus 5 segundos de ócio. Para isso, usa-se a função

---

`MPI_Barrier(comm)`

Cada processo do comunicador informa os outros quando chega a esta instrução e depois fica à espera. Quando é recebida a notícia de que todos os processos do comunicador estão neste ponto do programa, eles avançam em simultâneo para a seguinte instrução (que poderá ser diferente para cada um).

Ainda não discutimos a questão mais importante: porquê bloquear o avanço do programa do remetente até completar a comunicação? A explicação é simples. Se a execução avançasse, a informação que está na memória correspondente ao bloco a ser enviado poderia ser alterada pelo programa. Quando a comunicação finalmente acontecesse, o destinatário iria receber informação diferente daquela que se pretendia enviar.

Programa 1: Comunicação ponto-a-ponto sincronizada

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int n, myid, numprocs;
  int *bloco;
  double MPI_Wtime(), MPI_Wtick();
  double starttime, endtime;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /* Queremos repetir com vários tamanhos de mensagem. */
  while (1) {

    /* O master lê o nº de inteiros na mensagem, n. */
     usleep(3E5);
    if (myid == 0) {
      printf("Número de inteiros a enviar: (0 p/ sair) ");
      (void)! scanf("%d",&n);
    }
```

[1] Na verdade, ainda tem que escrever os valores do vetor antes, mas não vamos ser preciosistas.

```
26
27      /* Valor de n é distribuido por todos. */
28      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
29      if (n == 0) break; /* Todos terminam.*/
30
31      /* Todos definem um vetor de inteiros.
32       * Não havia necessidade nenhuma de os processos com myid > 1
33       * fazerem isto. Só estamos a usar memória sem necessidade.*/
34      bloco = (int*)malloc(n * sizeof(int));
35      /* Todos começam a contar o tempo no mesmo instante. */
36      MPI_Barrier(MPI_COMM_WORLD);
37      starttime = MPI_Wtime();
38
39      if (myid == 0) {
40        for (int i=0; i < n; ++i) bloco[i]=2*i;
41        MPI_Ssend(bloco, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
42        endtime=MPI_Wtime();
43        printf("Sou o %d. Passaram aproximadamente %f segundos "
44               "desde que comecei a enviar o bloco "
45               "e agora estou pronto para continuar.\n", myid, endtime-starttime);
46      }
47      else if (myid == 1) {
48        usleep(5E6);
49        MPI_Recv(bloco, n, MPI_INT, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
50        printf("Sou o %d. Estive 5 segundos a dormir e acabei agora de "
51               "receber o bloco.\n", myid);
52      }
53      else {
54        /* Não se faz nada. */
55      }
56
57       free(bloco);
58       /* Não quero que o 0 comece a pedir o n antes de o 1 fazer o output. */
59       MPI_Barrier(MPI_COMM_WORLD);
60    }
61    MPI_Finalize();
62    return 0;
63  }
```

Para evitar que possa ser desperdiçado tempo de CPU enquanto se espera que um envio sincrono seja concluído, o processo remetente pode em alternativa criar uma cópia do conteúdo da mensagem e colocá-la num buffer. O programa pode agora continuar, alterando à vontade a versão local do conteúdo em questão, porque é a cópia que está no buffer que vai ser enviada ao destinatário quando este estiver preparado para a receber. Formalmente, continua a tratar-se de uma comunicação ponto-a-ponto bloqueante. A função é a seguinte:

```
MPI_Bsend(address, count, datatype, destination, tag, comm)
```

Será que basta substituir `MPI_Ssend` por `MPI_Bsend` no Programa 1 para passar a usar comunicação com buffer em vez de comunicação síncrona? Vamos experimentar. No linux

não é necessário abrir o ficheiro para fazer a substituição, basta executar o seguinte comando:

```
$ sed -i 's/Ssend/Bsend/g' Ex1Prog1.c
```

Compile e execute o programa. Deverá encontrar um erro, porque o buffer de envio tem tamanho zero. Isto quer dizer que quando um processo quer usar um buffer para enviar mensagens tem que o preparar previamente com a função

---

`MPI_Buffer_attach(buffer, size)`

Cada processo só pode ter associado a si um único buffer. Tem que se ter cuidado na escolha do tamanho do buffer. Tem que ser suficientemente grande para ir guardando as mensagens que ainda não foram enviadas com sucesso, mas não deverá ser demasiado grande, de forma a não ter que reservar muita memória. Lembre-se que pode haver outros processos no mesmo computador a usarem os seus próprios buffers.

Quando o buffer deixar de ser necessário, pode ser desativado com

---

`MPI_Buffer_detach(buffer, size)`

Pode fazer modificações no Programa 2 para confirmar que invocar `MPI_Buffer_detach` não é suficiente para libertar a memória. Compare o Programa 2 com o Programa 1 para ver as modificações que tiveram de ser feitas para usar comunicação ponto-a-ponto com buffer. Compile e execute o Programa 2. Vai verificar que o processo 0 já não vai ter que esperar que o processo 1 receba a mensagem.

Programa 2: Comunicação ponto-a-ponto com buffer

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int n, myid, numprocs;
  int *bloco;
  double MPI_Wtime(), MPI_Wtick();
  double starttime, endtime;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  /* Queremos repetir com vários tamanhos de mensagem. */
  while (1) {
    /* O master lê o nº de inteiros na mensagem, n. */
    usleep(3E5);
    if (myid == 0) {
      printf("Número de inteiros a enviar: (0 p/ sair) ");
```

```
22        (void)! scanf("%d",&n);
23      }
24
25      /* Valor de n é distribuido por todos. */
26      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
27      if (n == 0) break; /* Todos terminam.*/
28
29      /* Todos definem um vetor de inteiros.
30       * Não havia necessidade nenhuma de os processos com myid > 1
31       * fazerem isto. Só estamos a usar memória sem necessidade.*/
32      bloco = (int*)malloc(n * sizeof(int));
33      /* Todos começam a contar o tempo no mesmo instante. */
34      MPI_Barrier(MPI_COMM_WORLD);
35      starttime = MPI_Wtime();
36
37      if (myid == 0) {
38        for (int i=0; i < n; ++i) bloco[i]=2*i;
39        /* É preciso criar o buffer. */
40        int buffer_size = (MPI_BSEND_OVERHEAD + n*sizeof(int));
41        char* buffer = malloc(buffer_size);
42        MPI_Buffer_attach(buffer, buffer_size);
43        MPI_Bsend(bloco, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
44        endtime=MPI_Wtime();
45        printf("Sou o %d. Passaram aproximadamente %f segundos "
46               "desde que comecei a enviar o bloco "
47               "e agora estou pronto para continuar.\n", myid, endtime-starttime);
48        MPI_Buffer_detach(&buffer, &buffer_size);
49        free(buffer);
50      }
51      else if (myid == 1) {
52        usleep(5E6);
53        MPI_Recv(bloco, n, MPI_INT, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
54        printf("Sou o %d. Estive 5 segundos a dormir e acabei agora de "
55               "receber o bloco.\n", myid);
56      }
57      else {
58        /* Não se faz nada. */
59      }
60
61       free(bloco);
62       /* Não quero que o 0 comece a pedir o n antes de o 1 fazer o output. */
63       MPI_Barrier(MPI_COMM_WORLD);
64    }
65    MPI_Finalize();
66    return 0;
67  }
```

Estamos finalmente preparados para analisar o funcionamento da função padrão de comunicação ponto-a-ponto bloqueante. Assumindo que ainda não reverteu o comando que estragou o Programa 1, faça

```
$ sed -i 's/Bsend/Send/g' Ex1Prog1.c
```

Compile e corra o Programa 1. Deverá verificar que para mensagens grandes, o comportamento vai ser o de uma comunicação sincronizada. No entanto, para mensagens pequenas o resultado deverá ser semelhante ao de uma comunicação com buffer. Como o Programa 1 não define nenhum buffer de utilizador, isto significa que a implementação de MPI gere um buffer de sistema que é usado para algumas comunicações ponto-a-ponto padrão. Há um quarto modelo de comunicações ponto-a-ponto bloqueante: pode fazer uma pesquisa se estiver curioso. No próximo exercício vamos estudar a comunicação ponto-a-ponto não bloqueante.

MPI apontamentos

Vasco Costa - 97746

junho 2023

Listing 1: Codigo das aulas.

```
// COMO CORRER O PROGRAMA - 2D
// ----------------------
// COMPILAR: mpicc Parte_3.c -o Parte_3
// CORRER:  mpiexec -n 4 Parte_3
// ----------------------
// Meter os includes das bibliotecas
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// Definicao de constantes para que antes de compilar, o ficheiro de codigo que vai ser compilado,
     substitua todas as instanias pelo que esta a frente
#define TOL 1e-6 // tolerancia
#define ITERMAX 500000 // criterio de paragem
#define NXMAX 500 // para evitar alocacao de memoria excessiva. definir numero maximo de pontos da
     grelha
#define L 1.0 // Tamanho do lado do quadrado, do dominio
// Definir uma funcao qualquer
double f(double x, double y){
    return (x + y);
}
int main(int argc, char *argv[]) {
    // Declarar variaveis locais
    int nprocs; // Numero de processos
    int myid; // Rank de cada processo
    int nx, ny; // Tamanho da matriz para ambas as coordenadas
    // Definir o MPI
    MPI_Init(&argc, &argv); // Inicializar o MPI -> A partir daqui, cada processo vai correr
         "individualmente"
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // Definir o tamanho do comunicador MPI. Aqui fornecemos
         o endereco para onde esta armazenado o valor da variavel n_procs
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); // Definir o rank de cada processo. O rank tem siginifcado
         apenas no contexto do comunicador
    // Isolar o processo para fazer o scanf para eprguntar apenas uma vez e nao em todos os
         processos, o tamanho que utilizador quer para a matriz
    if(myid == 0){ // Isolamos o processo 0
        // Printar para pedir ao utilizador
        printf("Introduza o numero de pontos (max %d, 0 para sair): ", NXMAX);
        // Ir buscar o valor
        scanf(" %d", &nx); // Guardamos em nx. Nota que nx = ny
    }
    // Fazer o Brodcast para que todos os processos tenham a informacao adequirida anteriomente
    // Todos os processo que tem id diferente de 0 recebem. O que tiver igual a 0 envia o nx. Um
         comunica com todos
    MPI_Bcast(&nx, 1, MPI_INT, 0, MPI_COMM_WORLD); // O que queremos enviar/receber, o tamnho, o tipo
         de dados, quem esta a enviar
    // Definir o ny
    ny = nx;
    // Vamos testar o valor nx, introduzido pelo utilizador
    if(nx == 0){ // O utilizador quer sair
        // Vamos finalizar o MPI antes de sair
        MPI_Finalize();
        // Sair
        return 0;
    }
    else if (nx > NXMAX) // Algo invalido
    {
        // Vamos finalizar o MPI antes de sair
        MPI_Finalize();
        // Sair
```

```
    return 1;
}
// Definicao e inicializacao de variaveis para a criacao do comunicador cartesiano
int ndims = 2; // Numero de dimensoes
int dims[2] = {(int)(nprocs / 2), 2}; // NUmero de linhas, numero de colunas
int periodic[2] = {0, 0}; // E uma variavel booleana, so que em C nao ha, entao metemos um int.
     Mas so pode ser 0 ou 1
MPI_Comm comm2D;
// Vamos criar um comunicador cartesiano. NOTA: um comunicador que e criado apartir de outro nao
     pode ter mais processos do que o seu criador, mas pode ter menos
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periodic, 1, &comm2D); // comunicador anterior,
     numero de dimensoes, tamnho do comunicador, periodicidade, reorder, datatype do comunicador
// Definir o novo id, isto e, o novo rank
int newid;
// Vamos ver quais sao os novos ranks, dai temos criado um novo comunicador
MPI_Comm_rank(comm2D, &newid);
// Definir os vizinhos
int nbrbottom, nbrtop, nbrright, nbrleft;
// Cada processo tem de saber o rank dos seus vizinhos, o de cima e o de baixo
MPI_Cart_shift(comm2D, 0, 1, &nbrtop, &nbrbottom); // Fazemos o shift no comm2D para descobrir
     vizinho de cima e baixo
MPI_Cart_shift(comm2D, 1, 1, &nbrleft, &nbrright); // Fazemos o shift no comm2D para descobrir
     vizinho da direita e da esquerda
// Printar para ver/dar debug as direcoes do shift; uma verificacao
printf("newid = %d nbrtop = %d   nbrbottom = %d   nbrleft = %d   nbrright = %d\n", newid, nbrtop,
     nbrbottom, nbrleft, nbrright);
MPI_Barrier(comm2D); // Para organizar os prints. VErifica se todos os processos ja chegaram
     aqui. SE ainda falta algum, esperam. Serve para sincronizar os processos
                  // O primeiro a acabar fica a espera do ultimo
// Declarar variaveis que vao ser utilziadas por todos os processos
int firstrow, nrows;
int firstcol, ncols; // Para a versao 2D
// Identificar qual o processo que nao e incluido no novo comunicador. Este nao vai guardar em
     comm2D poque nao tem rank nem acesso. Entao e devolvido o apotador null que permite
     descobrir qual o processo
if(comm2D == MPI_COMM_NULL){
    // Terminamos o processo
    MPI_Finalize();
    // E terminamos/fechamos o ambeinte MPI neste proesso
    return 0;
}
// Definir isto para nao estar sempre a aceder a um array, so uma questao de efeciencia
int dim0 = dims[0];
// Atualizar o numero de processos
nprocs = dim0 * dims[1];
// O processo 0, porque este existe sempre, e quem cria estes arrays
if(newid == 0){
    // Fazer a distribuicao das linhas e colunas
    int listfirstrow[nprocs];
    int listnrows[nprocs];
    int listfirstcol[nprocs];
    int listncols[nprocs];
    // Numero de linhas por processo, dividimos pelo numero de factias em vez do numero de
         processos
    int nrowsP = (int)(((double)(ny - 2)) / ((double)dim0) + 0.5);
    // Ciclo for para percorrer todos os processos. O i e o rank do processo
    for(int i = 0; i < dim0 - 1; i++){
        // Para cada processo indicamos
        listfirstrow[2 * i] = 1 + i * nrowsP;
        listnrows[2 * i] = nrowsP;
        listfirstrow[2 * i + 1] = 1 + i * nrowsP;
        listnrows[2 * i + 1] = nrowsP;
    }
    // Linhas
    listfirstrow[nprocs - 1] = 1 + (dim0 - 1) * nrowsP;
    listnrows[nprocs - 1] = ny - 2 - (dim0 - 1) * nrowsP;
    listfirstrow[nprocs - 2] = 1 + (dim0 - 1) * nrowsP;
    listnrows[nprocs - 2] = ny - 2 - (dim0 - 1) * nrowsP;
    // Colunas
    for(int i = 0; i < dim0; i++){
        // Coluna dos pares
        listfirstcol[2 * i] = 1; // sao os processo 0, 2, 4, 6, ... Ver esquema no caderno
        listncols[2 * i] = (int)(nx / 2) - 1; // -1 porque a primeira nao conta
        // Coluna dos impares
        listfirstcol[2 * i + 1] = (int)(nx / 2); // Metade dos pontos de uma linha
        listncols[2 * i + 1] = nx - 1 - (int)(nx / 2);
```

```
    }
    // Linhas
    MPI_Scatter(listfirstrow, 1, MPI_INT, &firstrow, 1, MPI_INT, 0, comm2D);
    MPI_Scatter(listnrows, 1, MPI_INT, &nrows, 1, MPI_INT, 0, comm2D);
    // Colunas
    MPI_Scatter(listfirstcol, 1, MPI_INT, &firstcol, 1, MPI_INT, 0, comm2D);
    MPI_Scatter(listncols, 1, MPI_INT, &ncols, 1, MPI_INT, 0, comm2D);
}
// Os restantes recebem o array que foi criado
else{
    // Aqui so se vai receber, entao metemos um apontador NULL (MPI_BOTTOM) que nao precisamos de
        enviar nada. E sempre o 0 que envia. Ele esta aqui para os outros processos saberem que
        nao sao eles que enviam. Eles tem o papel de receber
    MPI_Scatter(MPI_BOTTOM, 1, MPI_INT, &firstrow, 1, MPI_INT, 0, comm2D);
    MPI_Scatter(MPI_BOTTOM, 1, MPI_INT, &nrows, 1, MPI_INT, 0, comm2D);
    MPI_Scatter(MPI_BOTTOM, 1, MPI_INT, &firstcol, 1, MPI_INT, 0, comm2D);
    MPI_Scatter(MPI_BOTTOM, 1, MPI_INT, &ncols, 1, MPI_INT, 0, comm2D);
}
// Verificar se a atribuicao das linhas e colunas foi bem feita. A last row de um, tem de ser
    igual a firstrow do proximo + 1
printf("newid = %d firstrow = %d   lastrow = %d   firstcol = %d     lastcol = %d\n", newid,
    firstrow, firstrow + nrows - 1, firstcol, firstcol + ncols - 1);
MPI_Barrier(comm2D);
// Declarar matrizes com declaracao dinamica
double (*Vnew)[ncols + 2]; // Numero de linhas da matriz Vnem. O +2 e por causa das fantasma
double (*Vold)[ncols + 2];
double (*myf)[ncols + 2];
Vnew = calloc(nrows + 2, sizeof(*Vnew)); // O numero de linhas tem de ser igual ao numero de
    linhas +2, as tais "fantastma". Tamanho de cada elemento
                                    // Para aceder a esta matriz fazemos Vnem[i][j]
                                    // calloc inicializa a memoria alocada a 0. O malloc apenas
                                        aloca a memoria
Vold = calloc(nrows + 2, sizeof(*Vold));
myf = calloc(nrows + 2, sizeof(*myf));
// Definir o h
double h = L / ((double)(nx - 1)); // E a distancia entre pontos consecutivos. E o numero de
    intervalors
// Inicializar a matriz da funcao f, ou seja, meter la cenas
for(int i = 1; i <= nrows; i++){
    for(int j = 1; j <= ncols; j++ ){
        // Preencher myf
        myf[i][j] = f((-L/2.0) + (firstcol + j - 1) * h, (L/2.0) - (firstrow + i - 1) * h); //
            Para 2D
    }
}
// Preencher as condicoes fronteira. Esses valores nao sao alterados mas necessitam de ser
    inicializados
// Para o processo de cima/linha de cima que agora esta no processo 0 e 1
if(newid == 0 || newid == 1){
    // Percorrer elementos da linha de cima
    for(int i = 0; i < (ncols + 2); i++){ // Ou comecar em i = 1 e ter i < ncols
        // Preeenche o valor da condicao fronteira, elemento a elemento
        Vnew[0][i] = 0;
        Vold[0][i] = 0;
    }
}
// Para o processo de baixo/linha de baixo
if(newid == (nprocs - 1) || newid == (nprocs - 2)){
    // Percorrer elementos da linha de baixo
    for(int i = 0; i < (ncols + 2); i++){
        // Preeenche o valor da condicao fronteira, elemento a elemento
        Vnew[nrows + 1][i] = 0;
        Vold[nrows + 1][i] = 0;
    }
}
// Se der 0 e par, se der 1 e impar. Fazemos isto porque as condicoes fronteira sao diferentes
    para os processo par e impar
if(newid % 2 == 0){
    for(int i = 0; i < (nrows + 2); i++){
        // Preeenche o valor da condicao fronteira da esquerda, elemento a elemento
        Vnew[i][0] = 0;
        Vold[i][0] = 0;
    }
}
else{
    for(int i = 0; i < (nrows + 2); i++){
```

```
        // Preeenche o valor da condicao fronteira da direita, elemento a elemento
        Vnew[i][ncols + 1] = 0;
        Vold[i][ncols + 1] = 0;
    }
}
// Criar o data type para as colunas
MPI_Datatype column;
// Definir
MPI_Type_vector((nrows + 2), 1, (ncols + 2), MPI_DOUBLE, &column); // Numero total
     blocos/colunas, numero de elementos, periodicidade, O tipo dos elementos, oonde armazenar
// Dar commit
MPI_Type_commit(&column);
// COmecar contador de EScrita
double tm1 = MPI_Wtime();
// Iterar por cada dominio - Iterar o momento Jacobi
for(int iter = 0; iter < ITERMAX; iter++){
    // Definir um array para somas
    double sums[2];
    // Linhas do subdominio
    for(int i = 1; i <= nrows; i++){
        // Colunas do subdminio
        for(int j = 1; j <= ncols - 1; j++){
            // Preencher a matriz Vnew
            Vnew[i][j] = (Vold[i+1][j] + Vold[i-1][j] + Vold[i][j-1] + Vold[i][j+1] -
                 h*h*myf[i][j]) / 4.0;
            // Guardar a soma no array
            sums[0] += (Vnew[i][j] - Vold[i][j]) * (Vnew[i][j] - Vold[i][j]);
            sums[1] += Vnew[i][j] * Vnew[i][j];
        }
    }
    // Definir uma variavel para guardar todas as somas
    double global_sums[2];
    // Meter as sums no global_sums
    MPI_Allreduce(sums, global_sums, 2, MPI_DOUBLE, MPI_SUM, comm2D);
    // Vamos testar/comparar com a condicao
    if((global_sums[0] / global_sums[1]) < TOL){
        // Ver o tempo de escrita
        if(newid == 0){
            printf("Calculo demorou %f segundos para %d iteracoes\n", MPI_Wtime() - tm1, iter);
            tm1 = MPI_Wtime();
        }
        // Escrever o ficheiro -> NaO VAMOS FAZER ISTO (AFINAL VAMOS)
        // Definir as variaveis necessarias
        int gsizes[2] = {ny, nx}; // Tamanho da matriz global
        int lsizes[2] = {nrows, ncols + 1}; // Tamanho da matriz local. O +1 e a coluna da
             fronteira vertical, uns a direita(rank impar) outros a esquerda(rank par)
        int start_ind[2] = {firstrow, firstcol - 1 + newid % 2}; // indice de comeco
        // Fazer ajustes ao tamanho -> Dois processo de cima
        if(newid == 0 || newid == 1){
            lsizes[0]++;
            start_ind[0]--;

        }
        // Fazer mais ajustes again -> Dois processos de baixo
        if(newid == nprocs - 2 || newid == nprocs - 1){
            lsizes[0]++;
        }
        // Definir um novo datatype para o ficheiro
        MPI_Datatype filetype;
        MPI_Type_create_subarray(2, gsizes, lsizes, start_ind, MPI_ORDER_C, MPI_DOUBLE, &filetype);
        MPI_Type_commit(&filetype);
        // Definir variaveis para a memoria
        int memsizes[2] = {nrows + 2, ncols + 2};
        start_ind[0] = 1;
        start_ind[1] = newid % 2;
        // AJuste aos dois processo de cima
        if(newid == 0 || newid == 1){
            start_ind[0] = 0;
        }
        // Definir um novo datatype para o memoria
        MPI_Datatype memtype;
        MPI_Type_create_subarray(2, memsizes, lsizes, start_ind, MPI_ORDER_C, MPI_DOUBLE,
             &memtype);
        MPI_Type_commit(&memtype);
        // Definir um ficheiro para efetuar escrita
        MPI_File fp;
```

```
            MPI_File_open(comm2D, "resultados_2D.bin", MPI_MODE_CREATE | MPI_MODE_WRONLY,
                 MPI_INFO_NULL, &fp);
            MPI_File_set_view(fp, 0, MPI_DOUBLE, filetype, "native", MPI_INFO_NULL);
            MPI_File_write_all(fp, Vnew, 1, memtype, MPI_STATUS_IGNORE);
            MPI_File_close(&fp);
            // Libertar memoria dos datatypes
            MPI_Type_free(&filetype);
            MPI_Type_free(&memtype);
            // Vamos dar print ao processo que esta em uso
            printf("\nID Processo: %d\n", newid);
            // Vamos dar print ao que o processo fez
            for(int i = 0; i < nrows + 2; i++){
                for(int j = 0; j < (ncols + 2); j++){
                    // Elemento da matriz
                    printf("%f ", Vnew[i][j]);
                }
                // Introduzir uma mundaca de linha
                printf("\n");
            }
            // Terminar depois da escrita
            break;
        }
        // Antes de passar a proxima iteracao temos de trocar as linhas fantasma
        MPI_Sendrecv(Vnew[1], (ncols + 2), MPI_DOUBLE, nbrtop, 0, Vnew[nrows + 1], (ncols + 2),
             MPI_DOUBLE, nbrbottom, 0, comm2D, MPI_STATUS_IGNORE); // Sentido ascendente
        MPI_Sendrecv(Vnew[nrows], (ncols + 2), MPI_DOUBLE, nbrbottom, 1, Vnew[0], (ncols + 2),
             MPI_DOUBLE, nbrtop, 1, comm2D, MPI_STATUS_IGNORE); // Sentido descendente
        MPI_Sendrecv(&(Vnew[0][ncols]), 1, column, nbrright, 2, &(Vnew[0][0]), 1, column, nbrleft, 2,
             comm2D, MPI_STATUS_IGNORE); // Sentido para a direita
        MPI_Sendrecv(&(Vnew[0][1]), 1, column, nbrleft, 3, &(Vnew[0][ncols + 1]), 1, column,
             nbrright, 3, comm2D, MPI_STATUS_IGNORE); // Sentido para a esquerda
        // Atualizar o array, isto e, dizer que o Vold da proxima iteracao e o Vnew que acabamos de
             calcular. Tem de ser feito elemento a elemento: Vold[][] = Vnew[][]
        // Ou seja, copiar o Vnew para o Vold
        for(int i = 0; i < nrows + 2; i++){
            for(int j = 0; j < (ncols + 2); j++){
                Vold[i][j] = Vnew[i][j];
            }
        }
    }
    // Dar free ao data type
    MPI_Type_free(&column);
    // Libertar memoria
    free(Vnew);
    free(Vold);
    free(myf);
    // Finalizar o MPI
    MPI_Finalize();
    return 0;
}
```

Exercicios

1

Num programa MPI, o processo de rank 1 faz operacoes sobre gr, um array quadrado de dimensoes m × m, onde m e um inteiro par. Num dado instante do programa, pretendese que o processo 1 envie para o processo 0 a quarta parte inferior direita do array gr, de dimensoes (m/2) × (m/2). Escreva as linhas de código que preparam e executam o envio, fazendo uso de

a) MPI_Type_vector;

```
int sendCount = m / 2; // Tamanho da parte a ser enviada (m/2 x m/2)
int sendBuf[m][m]; // Array a ser enviado
MPI_Datatype sendType;
// Criar um tipo de dado MPI para a parte a ser enviada
MPI_Type_vector(sendCount, sendCount, m, MPI_INT, &sendType);
MPI_Type_commit(&sendType);
MPI_Send(&sendBuf[m / 2][m / 2], 1, sendType, 0, 0, MPI_COMM_WORLD);
```

b) MPI_Type_create_subarray. Nao tem que escrever a parte do código respeitante a rececao pelo processo 0.

```
int sendCount = 1; // Numero de partes a serem enviadas (neste caso, 1)
```

```c
int sendSizes[2] = {m / 2, m / 2}; // Tamanho da parte a ser enviada (m/2 x m/2)
int sendSubsizes[2] = {m / 2, m / 2}; // Tamanho da parte em relacao ao array completo
int sendStarts[2] = {m / 2, m / 2}; // Posicao inicial da parte em relacao ao array completo
int sendBuf[m][m]; // Array a ser enviado
MPI_Datatype sendType;

// Criar um tipo de dado MPI para a parte a ser enviada
MPI_Type_create_subarray(2, sendSizes, sendSubsizes, sendStarts, MPI_ORDER_C, MPI_INT, &sendType);
MPI_Type_commit(&sendType);
// Enviar a parte inferior direita do array para o processo 0
MPI_Send(&sendBuf[0][0], sendCount, sendType, 0, 0, MPI_COMM_WORLD);
```

2

Num dado programa, o processo de rank 2 tem que enviar parte de a, um array quadrado de numeros inteiros, de dimensões 100×100, para o processo de rank 1, atraves de um unica chamada de uma funcao de comunicacao ponto a ponto do MPI. Escreva as linhas de código que permitem ao processo preparar e realizar o envio quando a parte do array a enviar e a) os primeiros 50 elementos da coluna 6;

```c
MPI_Datatype columnType;
MPI_Type_vector(50, 1, 100, MPI_INT, &columnType);
MPI_Type_commit(&columnType);
MPI_Send(&(array[0][5]), 1, columnType, 1, 0, MPI_COMM_WORLD);
MPI_Recv(columnData, ROWS_PER_PROCESS, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

b) as linhas 0, 2, 4, . . . , 98;

```c
int sendcount = 50; // Numero de linhas a serem enviadas (50 linhas)
int blocklength = 100; // Numero de elementos em cada linha
int stride = 200; // Espaco entre as linhas que serao enviadas (pular uma linha)

// Criar um tipo MPI personalizado usando MPI_Type_vector
MPI_Datatype rowType;
MPI_Type_vector(sendcount, blocklength, stride, MPI_INT, &rowType);
MPI_Type_commit(&rowType);

// Enviar as linhas para o processo de rank 1
MPI_Send(&array[0][0], 1, rowType, 1, 0, MPI_COMM_WORLD);
```

c) as colunas 1, 3, 4, . . . , 99;

```c
int send_count = SIZE / 2; // Numero de colunas a serem enviadas
int send_blocklength = 1; // Tamanho do bloco de dados a ser enviado
int send_stride = 2;      // Intervalo entre as colunas a serem enviadas

MPI_Datatype column_type;
MPI_Type_vector(SIZE, send_count, send_stride, MPI_INT, &column_type);
MPI_Type_commit(&column_type);
MPI_Send(&a[0][0], 1, column_type, 1, 0, MPI_COMM_WORLD);
```

d) todos os elementos da diagonal principal

```c
// Criar o tipo de dado diagonal_type usando MPI_Type_vector
MPI_Type_vector(SIZE, 1, SIZE + 1, MPI_INT, &diagonal_type);
MPI_Type_commit(&diagonal_type);
// Enviar os elementos da diagonal principal para o processo de rank 1
if (rank == 2) {
    MPI_Send(&a[0][0], 1, diagonal_type, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
int received_data[SIZE];
MPI_Recv(&received_data[0], SIZE, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Energia parcial

```c
// Cada processo deve armazenar seu valor de m na posicao correspondente do array local_m
int local_m; // Valor de m no processo atual
int *gathered_m; // Array para armazenar os valores de m de todos os processos
int *sorted_m; // Array para armazenar os valores de m ordenados

// Coleta os valores de m de todos os processos em gathered_m
```

```
MPI_Allgather(&local_m, 1, MPI_INT, gathered_m, 1, MPI_INT, MPI_COMM_WORLD);

// Ordena os valores de m em ordem crescente usando a funcao sortint
sorted_m = sortint(gathered_m);

// Calcula o deslocamento e o numero de elementos para cada processo no array sorted_m
int *recvcounts; // Numero de elementos que cada processo recebera
int *displs; // Deslocamento inicial de cada bloco de elementos

recvcounts = (int*)malloc(size * sizeof(int));
displs = (int*)malloc(size * sizeof(int));

int i;
for (i = 0; i < size; i++) {
    recvcounts[i] = 1; // Cada processo recebera apenas um elemento
    displs[i] = i; // Deslocamento inicial de cada bloco e igual ao rank do processo
}

// Cada processo recebe seu valor correspondente de m no array local_m_sorted
int *local_m_sorted = (int*)malloc(sizeof(int));
MPI_Scatterv(sorted_m, recvcounts, displs, MPI_INT, local_m_sorted, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Agora cada processo tem seu valor de m reatribuido em ordem crescente
```

Exame CP 30/06/20 Parte A

1.

a) O MPI_Ssend obriga a uma comunicação síncrona entre o processo que envia a informação e o que recebe. Isto significa que tanto o processo 0 que envia como o 3 que recebe, tem de esperar que a comunicação conclua antes de avançar para a instrução seguinte, pois a instrução send é blocking quando a comunicação trata muitos dados. se os dados a enviar fossem poucos, então a comunicação seria buffered.
No que diz respeito à informação trocada neste caso, são enviados 2 000 000 números reais presentes no buffer "b". Em cada ciclo a tag é incrementada.

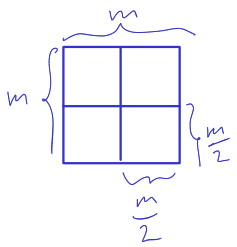
b) MPI_Recv(b, 2000000, MPI_REAL, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

c) MPI_Issend(b, 2000000, MPI_REAL, 3, i, MPI_COMM_WORLD, &request)
O parâmetro extra "request" serve para mais tarde verificar o estado da comunicação ou para esperar pela sua conclusão.

d)
```
MPI_Status status;
MPI_Request request;
if ...
  for ...
    MPI_Issend(b, 2000000, MPI_REAL, 3, i, MPI_COMM_WORLD, &request);
    (...)
    MPI_Wait(&request, &status);
    for ...
(
```
Primeiro criam-se os tipos de dados status e request, depois utiliza-se uma comunicação síncrona non-blocking e assim podem-se executar novas instruções durante o envio de dados até chegar ao ponto onde se iriam fazer alterações nesses dados, então aí chama-se MPI_Wait para confirmar o estado da comunicação e verifica se já terminou, se já tiver terminado pode-se avançar, caso contrário é necessário esperar que ela termine.

e) A vantagem do MPI_Issend sobre o MPI_Ssend, é que pode executar novas instruções ao mesmo tempo que envia dados através da comunicação, desde que se tenha o cuidado de evitar que estas instruções alterem os dados enquanto estiverem a ser enviados.

2.



a) MPI_Type_vector($\frac{m}{2}$, $\frac{m}{2}$, m, MPI_DOUBLE, &stype)

MPI_Type_commit(&stype)

MPI_Send(&gr[$\frac{m}{2}$][$\frac{m}{2}$], 1, stype, 0, 0, MPI_COMM_WORLD)

---

b)
```
gsizes[0] = m; gsizes[1] = m;
lsizes[0] = m/2; lsizes[1] = m/2;
start_indices[0] = m/2; start_indices[1] = m/2;
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices, MPI_ORDER_C,
                         MPI_DOUBLE, &stype)

MPI_Type_commit(&stype)
MPI_Send(&gr, 1, stype, 0, 0, MPI_COMM_WORLD)
```

3.

a)
```
MPI_Comm_rank(comm1, &newid)
MPI_Cart_shift(comm1, 1, 1, &left, &right)
```

---

b) Os processos 0 e 3 têm 10001 colunas das quais são responsáveis por actualizar 9999 pontos, pois excluí-se o ponto da fronteira à esquerda em relação ao processo 0 e o da direita em relação ao processo 3, enquanto que a coluna extra que perfaz as 10001 colunas, serve para guardar o ghost point do vizinho à direita do processo 0 e o do vizinho à esquerda do processo 3.

Os processos 1 e 2 têm 10002 colunas das quais são responsáveis por actualizar 10000 pontos, enquanto que as colunas extra que perfazem as 10002 colunas, servem para guardar os ghost points dos vizinhos à esquerda e direita de cada um dos processos.

---

c) Inicialmente define-se a variável mystart para cada processo, que diz respeito ao índice da primeira coluna que cada processo irá tratar, sendo o mystart = 0 para o processo 0, pois a primeira coluna do seu Tlocal diz respeito à condição fronteira à sua esquerda que irá precisar para fazer cálculos.

O MPI_Scatterv pega na linha inicial considerada nos cálculos e divide-a em quatro partes e distribui uma parte para cada processo no comunicador comm1. Apesar de todos os processos executarem esta instrução, apenas o processo 0 irá efectuar a distribuição propriamente dita, enquanto os restantes processos apenas irão receber dados.

O primeiro MPI_Sendrecv pega no valor mais à direita, excluindo ghost points, de cada array de cada processo e envia-o para o vizinho da direita, e em simultâneo cada processo guarda no seu valor mais à esquerda, excluindo ghost points, o valor que recebe do seu vizinho à esquerda.

O segundo MPI_Sendrecv pega no valor mais à esquerda, excluindo ghost points, de cada processo e envia-o para o seu vizinho da esquerda, e em simultâneo cada processo guarda no seu valor mais à direita, excluindo ghost points, o valor que recebe do seu vizinho à direita.

d)

```
for i ...
    for j ...
        (...)
        {
        MPI_Sendrecv(&Tlocal[i+1][cols-2], 1, MPI_DOUBLE, right, 0, &Tlocal[i+1][0], 1,
                     MPI_DOUBLE, left, 0, comm1, MPI_STATUS_IGNORE)

        MPI_Sendrecv(&Tlocal[i+1][1], 1, MPI_DOUBLE, left, 1, &Tlocal[i+1][cols-1], 1,
                     MPI_DOUBLE, right, 1, comm1, MPI_STATUS_IGNORE)
        }
```

e) `MPI_Gather(&Tlocal[Nt-1][mystart], Nx/4, MPI_DOUBLE, &perfilfinal, Nx/4, MPI_DOUBLE, 0, comm1)`

Para que todos os processos recebessem o array perfilfinal, bastaria usar a função MPI_Allgather em vez de MPI_Gather.

4.
```
// 1000000000 inteiros ocupam tanto espaço como 500000000 doubles
N = 1000000000 / 2;
gsizes = 3 * N;
lsizes = N;
start_index = myid * N;
MPI_Type_Create_Subarray(1, gsizes, lsizes, start_index, MPI_ORDER_C, MPI_DOUBLE, &filetype);
MPI_File_open(MPI_COMM_WORLD, "output.sim", MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_DOUBLE, filetype, "native", MPI_INFO_NULL);
if (myid == 0)
  MPI_File_write_all(fh, &a[0], N, MPI_DOUBLE, MPI_STATUS_IGNORE);
if (myid == 1)
  MPI_File_write_all(fh, &a[N], N, MPI_DOUBLE, MPI_STATUS_IGNORE);
if (myid == 2)
  MPI_File_write_all(fh, &b[0], 2*N, MPI_DOUBLE, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

1.

a)

```
int * bloco;
(...)
if(myid == 0) {
   (...)
   int buffer_size = (MPI_BSEND_OVERHEAD + n * sizeof(int));
   char * buffer = malloc(buffer_size);
   MPI_Bsend(
   MPI_Buffer_attach(&buffer, &buffer_size);
   MPI_Bsend(bloco, n, MPI_Int, 1, 0, MPI_COMM_WORLD);
   (...)
   MPI_Buffer_detach(&buffer, &buffer_size);
   free(buffer);
}
```

b)

O envio buffered bloqueante obriga a que o processo ~~espera~~ que envia os dados, espere até a estes estarem copiados para um buffer que apenas é utilizado para comunicar a informação que nele está contida, sem se efetuarem alterações nos dados. Após este buffer ter todos os dados que se pretendem comunicar, então o processo pode prosseguir e executar novas instruções, desta forma a informação está segura e não é corrompida.
Por outro lado, o envio buffered não bloqueante permite que o processo que envia os dados comece logo a executar novas instruções e ao mesmo tempo que os dados são copiados para o buffer, o que significa que pode dar-se o caso em que estes dados são alterados enquanto ainda ~~se está a~~ estão a ser copiados para o buffer, o que não devia acontecer.
Posto isto, com o buffered ~~[illegible]~~ bloqueante não é preciso ter cuidados extra, no entanto com o buffered não bloqueante é preciso ter cuidados extra. Para resolver ~~[illegible]~~ este problema, bastaria usar a função MPI_Wait juntamente com o parâmetro request.

2.

a) ~~MPI_Type_vector(50,~~

```
MPI_Type_vector(50, 100, 100, MPI_INT, &atype)
MPI_Type_commit(&atype)
MPI_Send(a[0][6], 1, atype, 1, 0, MPI_COMM_WORLD)
```

b)
```
MPI_Type_vector(50, 100, 200, MPI_INT, &stype)
MPI_Type_commit(&stype)
MPI_Send(a[0][0], 1, stype, 1, 1, MPI_COMM_WORLD)
```

c)
```
MPI_Type_vector(5000, 2, 2, MPI_INT, &ctype)
MPI_Type_commit(&ctype)
MPI_Send(a[0][1], 1, ctype, 1, 2, MPI_COMM_WORLD)
```

d)
```
MPI_Type_vector(100, 100, 100, MPI_INT, &dtype)
MPI_Type_commit(&dtype)
MPI_Send(a[0][0], 1, dtype, 1, 3, MPI_COMM_WORLD)
```

4.

a)
```
MPI_Comm_rank(Comm1, &mlwid)
MPI_Cart_shift(Comm1, 0, 1, &top, &bottom)
MPI_Cart_shift(Comm1, 1, 1, &left, &right)
```

b)
```
MPI_Send(top, 1, MPI_INT, left, MPI_ANY_TAG, comm1)
```

c)
Ao inicializar o valor do identificação do vizinho no canto superior direito a MPI_PROC_NULL, está-se a garantir que antes de comunicação, todos os processos vêem o seu vizinho nesse canto como não existente. Assim, garante-se sempre que no limite esse vizinho é visto como não existente.

d)
Quando o comunicador não é periódico, essencialmente significa que não há condições fronteira periódicas relativamente aos processos, ou seja o processo mais em cima não tem como vizinho de cima o processo mais em baixo e vice-versa. Para além disto, o processo mais à direita não tem como vizinho à direita o processo mais à esquerda e vice-versa.
Caso o comunicador fosse periódico, as respostas não seriam as mesmas, pois agora os vizinhos nas diagonais
Caso o comunicador fosse periódico, as respostas seriam iguais pois os vizinhos seriam achados corretamente. ~~Na~~ ~~estrutura~~ ~~a~~ ~~resposta~~ ~~é~~ ~~dada~~ ~~b~~ ~~e~~ ~~diz~~ ~~que~~ ~~é~~ ~~diferente~~ ~~pois~~ ~~agora~~ ~~os~~ ~~vizinhos~~ ~~no~~ ~~canto~~ ~~superior~~ ~~direito~~ ~~já~~ ~~não~~ ~~seriam~~ achados corretamente, pois podia dar-se o caso em que (por exemplo) o vizinho não era uma comunicação

Exemplo: processos → 

o processo 0 via como vizinho no canto superior direito como o 5.