MPI Communication
Point-to-Point (P2P)

Luca Tornatore - I.N.A.F.

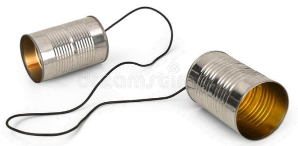
DATA SCIENCE & ARTIFICIAL INTELLIGENCE

SCIENTIFIC & DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

Outline

Intro to MPI

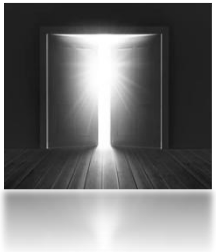


Point-to-Point Communications

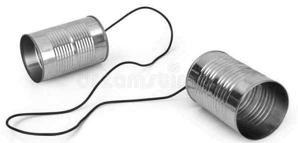
Collective Communications

few JEDI thing

Outline



Intro to MPI



Point-to-Point Communications

Collective Communications

few JEDI thing

The collectives

- The communications in MPI always happen *within* a group of processes in a communicator.

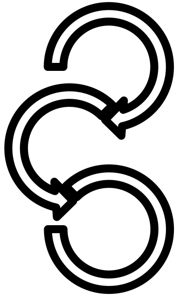
- **Collective communications** are over *all* the processes of the group
    - every collective call **must be performed by all** the processes in the group
    - collective calls are developed with sophisticated, tricky algorithms
    - involving synchronization, collective calls lead to a lost of parallelism, they amount to some serialization

- They may be blocking or non-blocking

The collectives: 3 classes

synchronization

data movement



collective computation

Σ

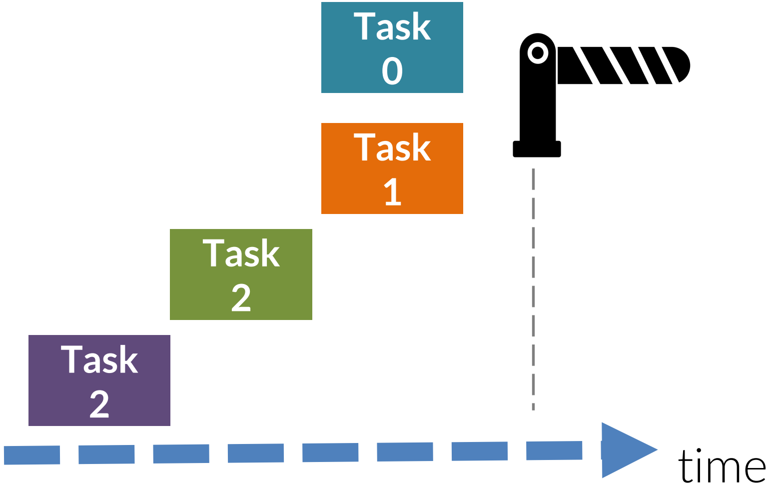
Synchronization

The MPI_Barrier ensures a synchronization among the MPI threads

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

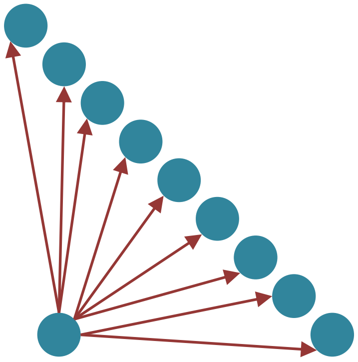
The call completes when all the MPI ranks in the group call the function.
Or, in other words, it blocks until all processes in the group of the communicator `comm` call the function.

Limit the usage as much as possible, since, as all the synchronizations, introduces a serialization.



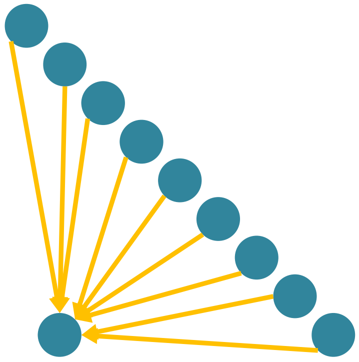
Collective data movement

one-to-many



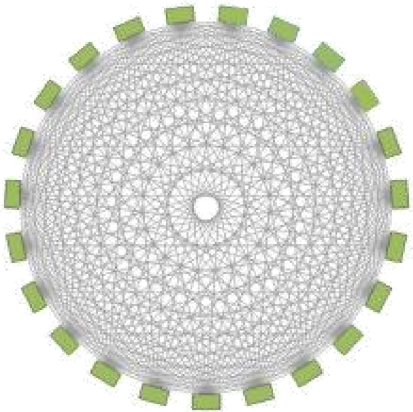
Broadcast
Scatter
Scatterv

many-to-one



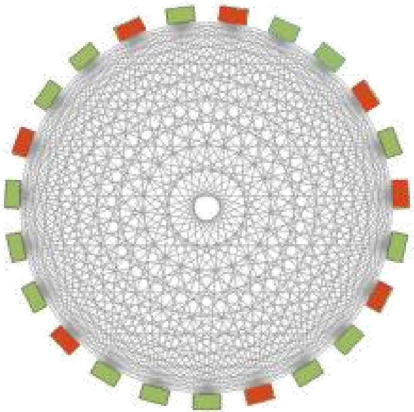
Gather
Gatherv

all-to-all



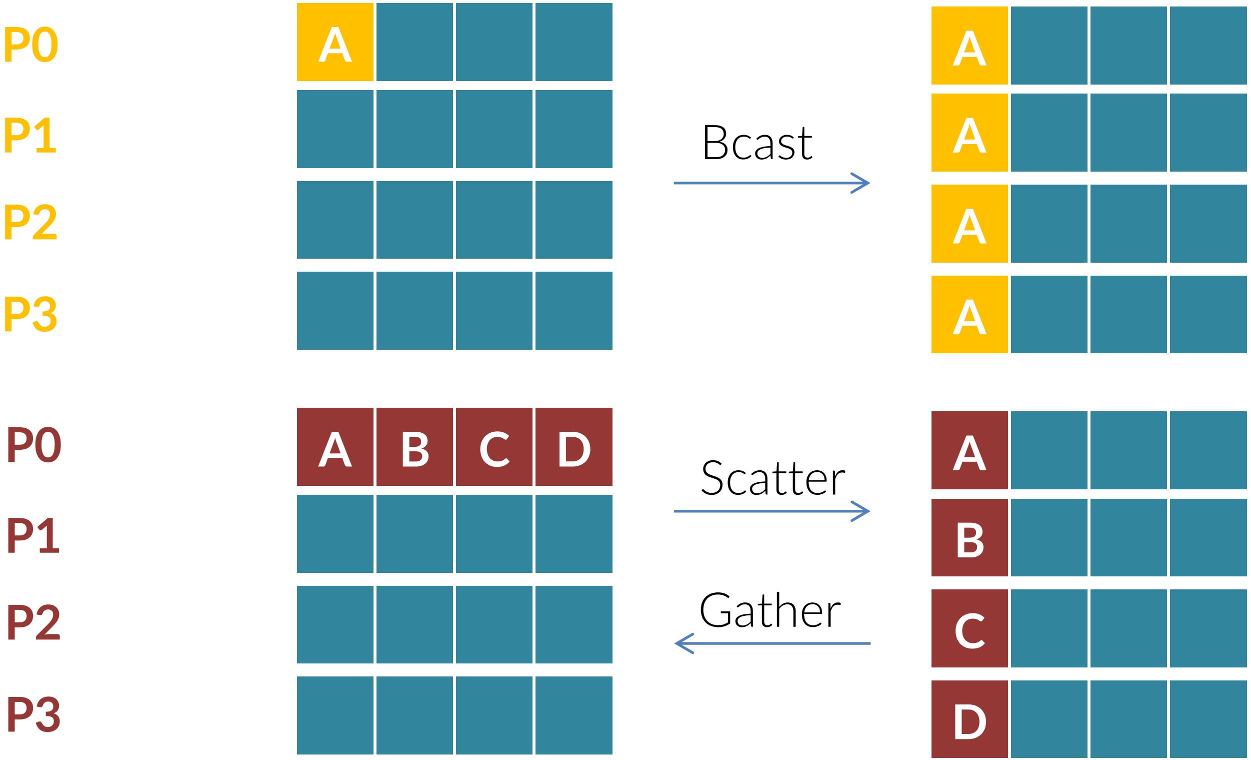
Allgather
Allgatherv
Alltoall

many-to-all



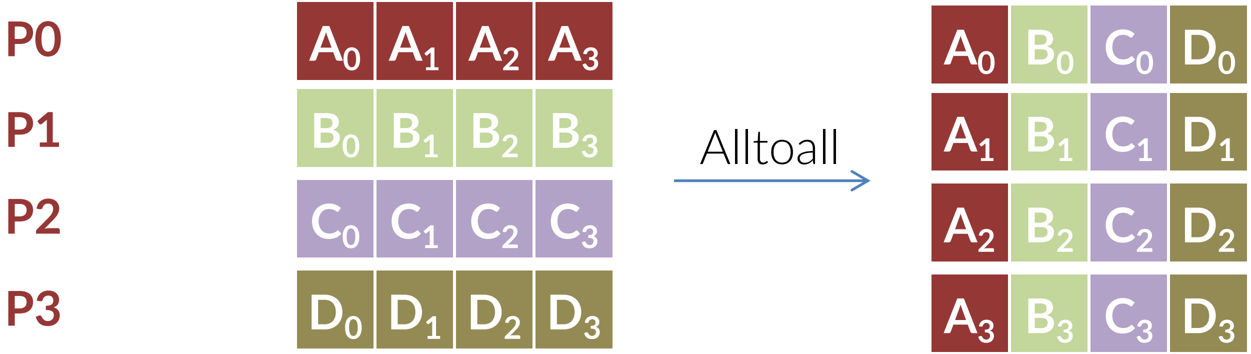
To be crafted
case-by-case

Collective data movement



Collective data movement

Allgather is equivalent to Gather + Bcast
Allgather algorithms can be faster



Alltoall is equivalent to a transpose of the data

V-Collective data movement

The collective that we have seen deal with the same amount of data for each process.

There are several cases in which you may want to deal with a **variable amount of data per process**.

MPI provides the **"v" version** of the routines ("v" stands for "vector") for these cases.
Even if efficient algorithms exist, those are not as efficient as the ordinary fixed-size ones.

note: `MPI_Alltoall -> MPI_Alltoallw`

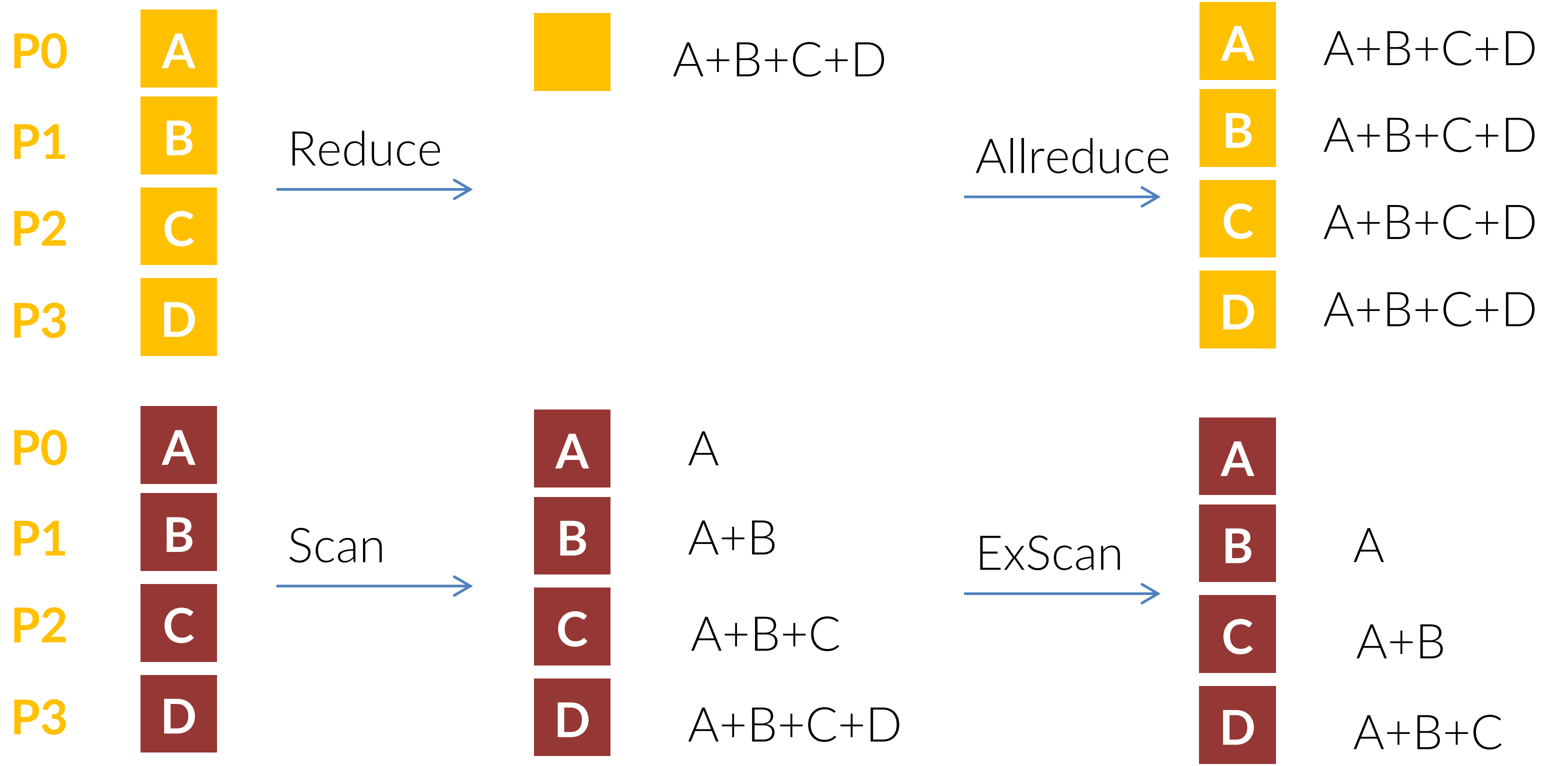
Collective computation

Collective computations combine communications with computation

Reduce	all-to-one, combined with an operation
Scan	the prefix-sum: all prior ranks to all, combined with an op
Reduce_scatter	all-to-all, combined with an operation

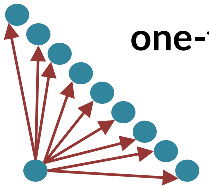
The performed operations can be either the predefined one or a user-defined one

Collective computation



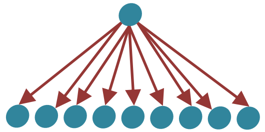
Pre-defined ops in coll. ops.

`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	multiplication
`MPI_SUM`	summation
`MPI_LAND`	logical and
`MPI_LOR`	logical or
`MPI_LXOR`	logical xor
`MPI_BAND`	bitwise and
`MPI_BOR`	bitwise or
`MPI_BXOR`	bitwise xor
`MPI_MAXLOC`	locate the max
`MPI_MINLOC`	locate the min

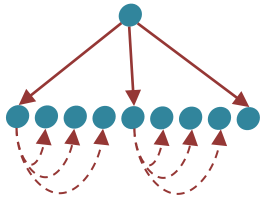


Broadcast

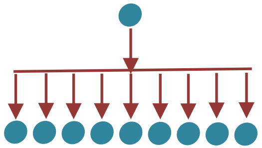
Note: there are many possible algorithms for collective communications.
A Broadcast, for instance, may be implemented as



p2p linear, scales as $O(N)$

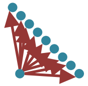


tree, scales as $O(logN)$



hardware based, if available
scales as $O(1)$

Broadcast



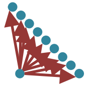
```c
int MPI_Bcast ( void *buffer, int count,
                MPI_Datatype datatype,
                int root, MPI_Comm comm )
```

The broadcast is a one-to-many communication.
In a broadcast, the `root` process sends the same data to all the other processes in a communicator.

Main uses of a broadcast call are

- to send out serial input to all the tasks in a communicator
- to send out input serial data
    - configuration paramenter
    - initial conditions (generally, better to design a parallel I/O pattern)

Broadcast



Broadcasting example

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

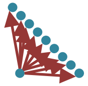
```
int nshots_per_task;

// send around how many random points every task must generate
MPI_Bcast( &nshots_per_task, 1, MPI_INT, 0, myCOMM_WORLD );

int valid_points = 0;
for ( int i = 0; i < nshots_per_task; i++ ) {
    // ... the pi-greek stuff
    valid_points += is_a_valid_point; }

// now we are left with the problem of collecting all the partial results
```

Scatter



Broadcasting example

```
int MPI_Scatter ( void *send_buffer, int s_count,
                  MPI_Datatype s_type,
                  void *recv_buffer, int r_count,
                  int root, MPI_Comm comm )
```

```
if ( myID == sender ) {
  int *data_per_processes = (int*)malloc( Ntasks*Ndata_per_proc*sizeof(int) );
  initialize_data_per_proc( data_per_processes ); }
else
  int *mydata = (int*)malloc( Ndata_per_proc*sizeof(int) );

// distribute data among processes
MPI_Scatter( data_per_processes, Ndata_per_proc, MPI_INT, mydata, Ndata_per_proc, 0,
myCOMM_WORLD );
```

Broadcasting example

```
int MPI_Scatterv ( void *send_buffer, int s_count[],
                   int displ[], MPI_Datatype s_type,
                   void *recv_buffer, int r_count,
                   MPI_Datatype r_type,
                   int root, MPI_Comm comm )
```

```
int offsets[Ntasks] = {0};
if ( myID == sender ) {
  int *data_per_processes = (int*)malloc( Ntasks*Ndata_total*sizeof(int) );
  for ( int i = 0; i < Ntasks; i++) { if ( i != Me ) {
      initialize_data_per_proc( data_per_processes, offsets[i], Ndata_per_proc[i] );
      offsets[i+1] = offsets[i]+Ndata_per_proc[i]; } }
else
  int *mydata = (int*)malloc( myNdata*sizeof(int) );

// distribute data among processes
MPI_Scatterv( data_per_processes, Ndata_per_proc, offsets, MPI_INT,
              mydata, myNdata, 0, myCOMM_WORLD );
```

Reduce example

```
int MPI_Reduce ( void *send_buffer, void *recv_buffer,
                 int count, MPI_Datatype datatype,
                 MPI_Op op, int root, MPI_Comm comm )
```

```
MPI_Bcast( &nshots_per_task, 1, MPI_UNSIGNED_INT, the_root_proc, myCOMM_WORLD );

unsigned int valid_points = 0;
for ( int i = 0; i < nshots_per_task; i++ ) {... valid_points += is_a_valid_point; ... }

// now we are left with the problem of collecting all the partial results
if ( Myrank == the_root_proc ) {
    unsigned int all_valid_points;
    MPI_Reduce ( &valid_point, &all_valid_points, 1, MPI_UNSIGNED_INT, MPI_SUM,
                 the_root_proc, myCOMM_WORLD); }
else
    MPI_Reduce ( &valid_point, 0x0, 1, MPI_UNSIGNED_INT, MPI_SUM,
                 the_root_proc, myCOMM_WORLD);
```

IN-PLACE Reduce

Reduce example

```
int MPI_Reduce ( void *send_buffer, void *recv_buffer,
                 int count, MPI_Datatype datatype,
                 MPI_Op op, int root, MPI_Comm comm )
```

Quite often, the root of a collective computation participates with its own partial result that will not be useful anymore in the future.
For these case instead of having one more variable that contains the result of the operation, MPI offers the special value `MPI_IN_PLACE` to be put in the `send_buffer` field

```
double result;
// everybody calculate its result

if ( I_am_the_root )
   int MPI_Reduce ( MPI_IN_PLACE,  &result, MPI_DOUBLE, MPI_SUM, root, ...);
else
   int MPI_Reduce ( &result, NULL, MPI_DOUBLE, MPI_SUM, root, ...);
```

IN-PLACE Reduce

Reduce example

```
int MPI_Reduce ( void *send_buffer, void *recv_buffer,
                 int count, MPI_Datatype datatype,
                 MPI_Op op, int root, MPI_Comm comm )
```

The `MPI_IN_PLACE` works also for the `All-` versions

```
double result;
// everybody calculate its result

MPI_AllReduce ( MPI_IN_PLACE,  &result, MPI_DOUBLE, MPI_SUM, root, ...);
```

Groups & Communicators

Imagine now that we want to entrust the generation of pseudo-random numbers to a single task, that will send chunks of random numbers to the "workers".

The situation in which a task is preparing data and distributing the work among the other task was called "master/slave". That has always been an awful name, let's call it "**director/orchestra paradigm**".

Then, the final reduce must happen only among the workers, excluding the director task.
That would not be a problem, since it would suffice that the director had a zero-valued variable to participate to the reduce.

However, this is case to illustrate how to create new groups and communicators
(inspired by "Using MPI", by Gropp, Lusk and Skjellum)

Groups & Communicators

When you want to derive a group from an existing communicator (which normally is what you know), as first you need to “extract” the group of tasks associated with the communicator

```
MPI_Group group;
MPI_Comm_group ( communicator, &group );
```

As a second step, you manipulate the group. For example, by selecting/excluding some of the tasks

```
MPI_Group_excl
MPI_Group_incl
MPI_Group_range_incl
MPI_Group_range_excl
MPI_Group_union
```

Groups & Communicators

```
MPI_Group group, new_group;
MPI_Comm_group ( world, &group );

int Nexcluded;
int Ranks_excluded[Nexcluded] = { ... };
MPI_Group_excl( group, Nexclued, Ranks_excluded, &new_group)
```

The new group `new_group` is formed and the `Nexcluded` ranks listed in `Ranks_excluded` are then took off from it (syntax for the other MPI_Group_ calls is very similar and follows the same logic).
At this point, we also need a new communicator:

```
MPI_Comm new_comm;
MPI_Comm_create ( world,  new_group, &new_comm );
MPI_Group_free ( new_group );
MPI_Group_free ( group );
```

Groups & Communicators

```
MPI_Group group, new_group;
MPI_Comm_group ( world, &group );

int Nexcluded;
int Ranks_excluded[Nexcluded] = { ... };
MPI_Group_excl( group, Nexclued, Ranks_excluded, &new_group)
```

The new group `new_group` is formed and the `Nexcluded` ranks listed in `Ranks_excluded` are then took off from it (syntax for the other MPI_Group_ calls is very similar and follows the same logic).
At this point, we also need a new communicator:

```
MPI_Comm new_comm;
MPI_Comm_create ( world,  new_gro
MPI_Group_free ( new_group );
MPI_Group_free ( group );
```

A communicator holds an internal reference to the group, we do not need the groups anymore

Groups & Communicators

Alternatively, it is also possible to operate on the original communicator, sudviding it in two sections

```
MPI_Comm new_comm;
MPI_Comm_split ( world, (Myrank==the_root), Myrank, &new_comm );
```

then you'll need to know what are the ranks in the new communicator

```
MPI_Comm_size ( new_comm, &new_comm_size );
MPI_Comm_rank ( new_comm, &Myrank_new );
```

The root task will get `MPI_INVALID` as Myrank_new and new_comm_size, and `MPI_COMM_NULL` as new_comm.

Groups & Communicators

Alternatively, it is also possible to operate on the original communicator, sudviding it in two sections

```
MPI_Comm new_comm;
MPI_Comm_split ( world, (Myrank==the_root), Myrank, &new_comm );
```

This non-negative integer value is named “the color”, and is the value upon which the split in multiple communicators happens.

```
MPI_Comm_split ( world, (Myrank%3), Myrank, &new_comm );
```

In this case, there will be 3 groups, i.e. those of tasks calling the function with `color={0, 1, 2}`.
Each group will belong to a different `new_comm` communicator.
I.e. there will be 4 communicators: the original one and 3 new.

If a rank uses `color=MPI_UNDEFINED`, then it will not belong to any new communicator (its `new_comm` will have the value `MPI_COMM_NULL`)

Groups & Communicators

Alternatively, it is also possible to operate on the original communicator, sudviding it in two sections

```
MPI_Comm new_comm;
MPI_Comm_split ( world, color, Myrank, &new_comm );
```

The thirs parameter is named “the key” and determines the value of the new rank in the group associated with the new communicator.
Usually, one left it to the rank of the calling process in the original communicator.

Exercises

1) Re-write the π code with a director/orchestra paradigm.

The director should send, upon request, a bunch of random points to a worker, which in turn checks for how many points fall within r=1 form the origin.
All the workers update the collective estimate for π.
The game ends when the relative change of the estimate is less than α, a command-line given parameter

1) non-blocking data processing

Re-write the `non-blocking.c` so that

- `the_sender` has only a limited room to produce the data: it must wait that his buffer is empty before producing new one. Or, it can produce only an amount of data that fits in the available memory
- `the_receiver` has a unique data buffer, instead of two;
  it can receive only as many data as it coud host in the available fraction of the buffer at the moment of receiving

HINT: it may be simpler if you use two threads, one for the calculations and one for the communications

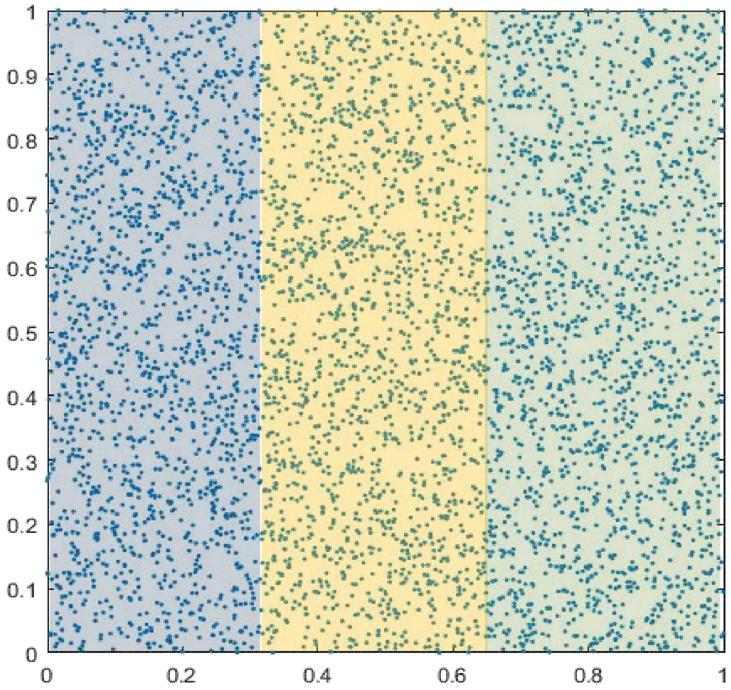
Exercises

3) Domain decomposition

Have a distribution of points in *n*D (let's start with *n*=2 for the sake of simplicity).
Suppose that you want to distribute your points among your MPI tasks by one of the coordinates, say the *x*.

Write an MPI code that allows such a domain decomposition.

`sendrecv.c`



that’s all, have fun