



INAF HPC course 2024

hands-on session

Jacobi solver
MPI implementation

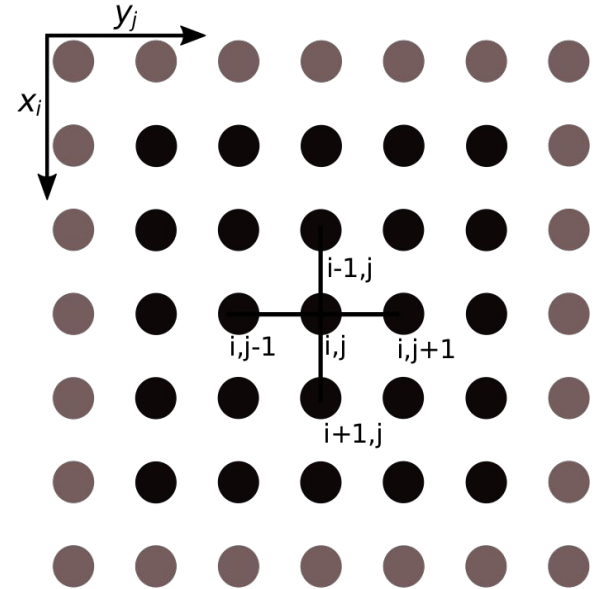
Jacobi's iterative method

The equation of the *Jacobi iteration*,

$$u_{i,j}^{k+1} = 0.25(u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k + u_{i+1,j}^k - h^2 f_{i,j})$$

To parallelize this algorithm using MPI, it is required to parallelize the loops of the iterations distributing the data across the MPI processes.

Several approaches are possible for domain decomposition, defining the application topology or virtual topology.



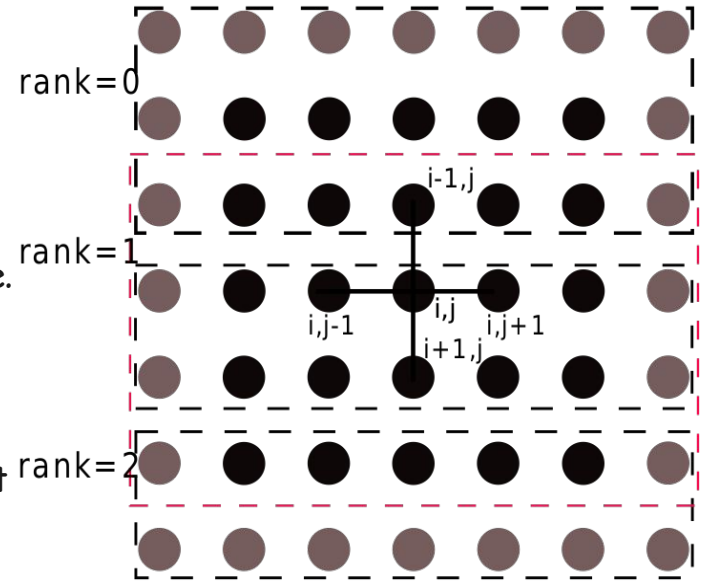
Approximation for 2-D Poisson problem, with $n = 5$. The boundaries of the domain are shown in gray.

Jacobi with 1-D decomposition

The simplest decomposition is shown in Figure 2.13, where the physical domain is sliced into slabs along the vertical direction (i.e. 1-D domain decomposition), while the arrays u , u_{new} , and f are replicated across the MPI processes.

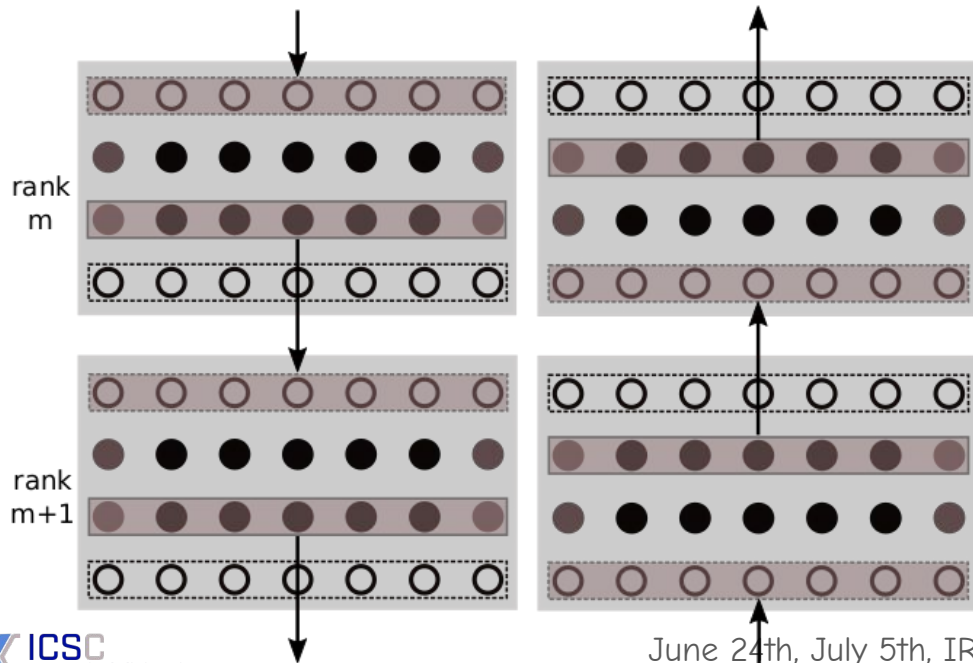
Looking at the Figure, the calculation of the element $u[i][j]$, handled by the MPI process with rank = 1, will require the element $u[i-1][j]$ that belongs to the MPI process with rank = 0.

This will imply that data must be exchanged between neighboring MPI processes, and consequently, the array managed by each MPI process must be expanded to hold data from other processes. The elements of the array that are used for MPI communications are called ghost points.



1-D MPI domain decomposition. Black dashed boxes show the computational domain of each MPI process. Red dashed box shows the computational domain, with ghost points, for the MPI process with rank = 1.

Jacobi with 1-D decomposition



Two-step MPI communication between contiguous processes (i.e. rank = m and rank = $m + 1$).

Data to be communicated is shaded. The mesh points are shown as black circles, while the ghost points are shown as unfilled circles.

- **First stage (on the left):**
the process with rank = m sends data to the process with rank = $m + 1$ and receives data from the process with rank = $m - 1$ (not shown in the sketch);
- **Second stage (on the right):**
the process with rank = m sends data to the process with rank = $m - 1$ (not shown in the sketch) and receives data from the process with rank = $m + 1$.

June 24th, July 5th, IRA Bologna

1-D MPI decomposition - blocking Send and Recv

```
void mpi_exchange_1d(MyData **const buffer,
                    const int    n,
                    const int    nbrtop,
                    const int    nbrbottom,
                    const int    start,
                    const int    end,
                    const MPI_Comm comm1d)
{
    /* The function is called by each MPI rank
       - nbrtop is the MPI process with rank + 1
       - nbrbottom is the MPI process with rank - 1 */

    /****** First communication stage *****/

    /* Perform a blocking send to the top (rank+1) process */
    MPI_Send(&buffer[end][1], n, MPI_MyDatatype, nbrtop, 0,
comm1d);

    /* Perform a blocking receive from the bottom (rank-1) process */
    MPI_Recv(&buffer[start-1][1], n, MPI_MyDatatype,
nbrbottom, 0, comm1d, MPI_STATUS_IGNORE);
    /****** Second communication stage *****/

    /* - Perform a blocking send to the bottom (rank-1) process */
    MPI_Send(&buffer[start][1], n, MPI_MyDatatype, nbrbottom, 1,
comm1d);

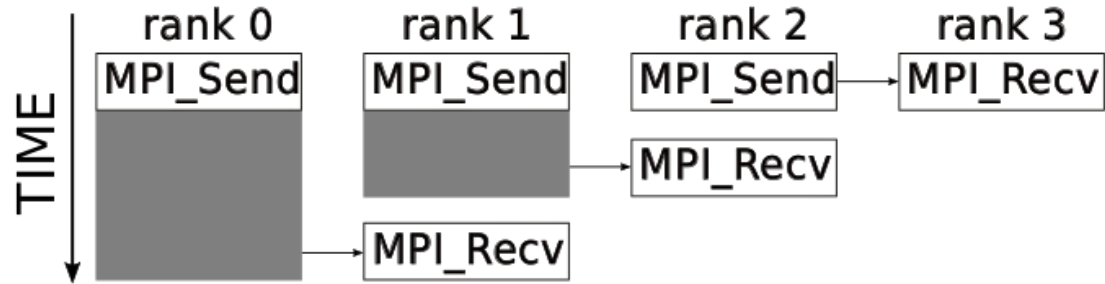
    /* Perform a blocking receive from the top (rank+1) process */
    MPI_Recv(&buffer[end+1][1], n, MPI_MyDatatype,
nbrtop, 1, comm1d, MPI_STATUS_IGNORE);
    /****** */
}
```

Code to manage data exchange in 1-D for ghost points using blocking MPI send and receive routines.

1-D MPI decomposition - blocking Send and Recv

Considering 4 ranks:

- rank = 0:
 - `nrbrtop` = 1;
 - `nrbrbottom` = `MPI_PROC_NULL`;
- rank = 1:
 - `nrbrtop` = 2;
 - `nrbrbottom` = 0;
- rank = 2:
 - `nrbrtop` = 3;
 - `nrbrbottom` = 1;
- rank = 3:
 - `nrbrtop` = `MPI_PROC_NULL`;
 - `nrbrbottom` = 2.



MPI sequential communication due to sends blocking until the matching receive is posted. The shaded area shows the idle time, while the process is waiting until the send communication is allowed to transfer the data to the neighboring process.

1-D MPI decomposition - Send and Recv paired

```
if ((rank % 2) == 0) /* even rank */
{
    /****** First communication stage *****/
    /* Blocking send to the top process */
    MPI_Send(&buffer[end][1], n, MPI_MyDatatype,
    nbrtop, 0, comm1d);

    /* Blocking receive from the bottom process */
    MPI_Recv(&buffer[start-1][1], n, MPI_MyDatatype,
    nbrbottom, 0, comm1d, MPI_STATUS_IGNORE);

    /****** Second communication stage *****/

    ...
} /* even rank */

else /* odd rank */
{
    /****** First communication stage *****/
    /* Blocking receive from the bottom process */
    MPI_Recv(&buffer[start-1][1], n, MPI_MyDatatype,
    nbrbottom, 0, comm1d, MPI_STATUS_IGNORE);

    /* Blocking send to the top process */
    MPI_Send(&buffer[end][1], n, MPI_MyDatatype,
    nbrtop, 0, comm1d);

    /****** Second communication stage *****/

    ...
} /* odd rank */
```

Code to manage data exchange in 1-D for ghost points using paired MPI sends and receives. MPI communication is no longer sequential.

June 24th, July 5th, IRA Bologna

1-D MPI decomposition - combined Sendrecv

```
/* The function is called by each MPI rank      */  
/* - nbrtop is the MPI process with rank + 1    */  
/* - nbrbottom is the MPI process with rank - 1 */  
  
MPI_Sendrecv(&buffer[end][1],      n, MPI_MyDatatype, nbrtop,      0,  
             &buffer[start-1][1], n, MPI_MyDatatype, nbrbottom, 0,  
             comm1d, MPI_STATUS_IGNORE);  
  
MPI_Sendrecv(&buffer[start][1], n, MPI_MyDatatype, nbrbottom, 1,  
             &buffer[end+1][1], n, MPI_MyDatatype, nbrtop,      1,  
             comm1d, MPI_STATUS_IGNORE);
```

Code snippet to manage data exchange in 1-D for ghost points using send-receive MPI routine.

1-D MPI decomposition - nonblocking Send and Recv

```
/* communication request (handle) */
MPI_Request request[4];

/****** Process is ready to receive data *****/
/* nonblocking receive from the bottom process */
MPI_Irecv(&buffer[start-1][1], n, MPI_MyDatatype, nbrbottom, 0, comm1d, &request[0]);
/* nonblocking receive from the top process */
...

/****** Process begins to send data *****/
/* nonblocking send to the top process */
MPI_Isend(&buffer[end][1], n, MPI_MyDatatype, nbrtop, 0, comm1d, &request[2]);
/* nonblocking send to the bottom process */
...
/****** Wait for all given MPI Requests to complete *****/
MPI_Waitall(4, request, MPI_STATUSES_IGNORE);
```

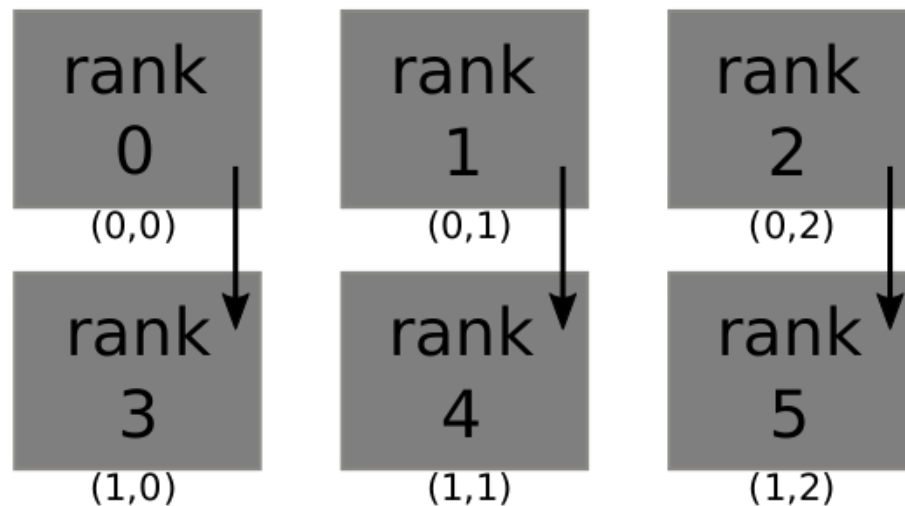
Code snippet to manage data exchange in 1-D for ghost points using nonblocking MPI routine.

June 24th, July 5th, IRA Bologna

Jacobi with 2-D decomposition

Another important virtual topology is the Cartesian topology, which is simply a decomposition in the natural coordinate (e.g. x, y) directions. MPI provides a collection of routines for defining and managing the Cartesian topology, namely:

- `MPI_Cart_create(...)` (it makes a communicator with cartesian topology);
- `MPI_Cart_coords(...)` (it determines process coordinates in cartesian topology given rank in group);
- `MPI_Cart_shift(...)` (it determines the neighboring processes).



A 2-D Cartesian domain decomposition using six MPI processes, also showing a shift by one in the first dimension.

Jacobi parallel 2-D decomposition

```
typedef struct MyGrid
{
    int local_start[2]; /* Local start index in each dimension */
    int local_end[2];   /* Local end index in each dimension */
    int global_start[2]; /* Global start index in each dimension */
    int global_end[2];  /* Global end index in each dimension */
    int local_dim[2];   /* Local domain size (no ghosts) */
} myDomain;

typedef struct Task_2D_Cartesian
{
    int rank; /* Local process rank */
    int nranks; /* Communicator size */
    int coords[NDIM]; /* Cartesian topology coordinate */
    myDomain domain; /* MyGrid structure (defined above) */
    int nbrtop; /* Top neighbor process in 2D topology */
    int nbrbottom; /* Bottom neighbor process in 2D topology */
    int nbrleft; /* Left neighbor process in 2D topology */
    int nbrright; /* Right neighbor process in 2D topology */
    MPI_Comm comm2d; /* Cartesian communicator */
} Task;

/* create the cartesian communicator */
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
                &ThisTask.comm2d);

/* get the comm size */
MPI_Comm_size(ThisTask.comm2d, &ThisTask.nranks);

/* determine the process coords in cartesian topology
given rank in group */
MPI_Cart_coords(ThisTask.comm2d, rank, 2, ThisTask.coords);

/* determines process rank in communicator given cartesian location */
MPI_Cart_rank(ThisTask.comm2d, ThisTask.coords, &ThisTask.rank);

/* get bottom and top neighbors (X direction) */
MPI_Cart_shift(ThisTask.comm2d, 0, 1,
                &ThisTask.nbrbottom, &ThisTask.nbrtop);

/* get left and right neighbors (Y direction) */
MPI_Cart_shift(ThisTask.comm2d, 1, 1,
                &ThisTask.nbrleft, &ThisTask.nbrright);
```

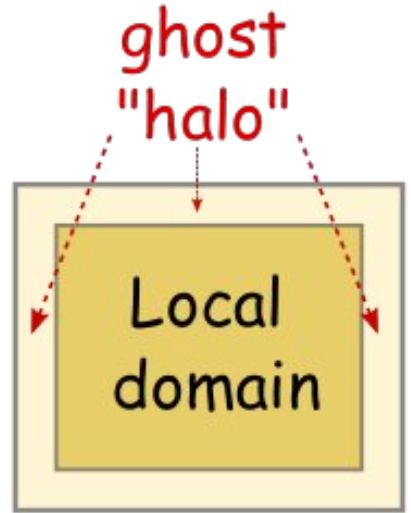
Jacobi with 2-D decomposition

In the data exchange routines seen so far, buffers are contiguous areas in memory, so that, for example, the datatypes `MPI_DOUBLE_PRECISION` coupled with the count of occurrences are sufficient to describe the buffers (i.e rows) to be sent. MPI derived datatypes allow the programmer to specify non-contiguous areas in memory, such as a column of an array stored in a row-major order like in C programming language.

```
MPI_Datatype column;
MPI_Type_vector(column_size, 1, row_size + 2, mpi_old_type, &column);
/* commit the new datatype */
MPI_Type_commit(column);

/* ... perform the communication ... */

/* free the datatype */
MPI_Type_free(column);
```



Local domain are surrounded by a "halo" of ghost cells, namely top and bottom rows, left and right columns.



Computation and communication overlapping

How to arrange the program so that some useful work can be done while processes are performing communications?

Scheme:

- I. begin nonblocking sending/receiving data to/from the other neighboring processes (MPI_Isend/MPI_Irecv or MPI_Isendrecv);
- II. perform the computation with the local data, excluding points that require ghost cells;
- III. check if data have been received from the other processes and finish computing with them.

Computation and communication overlapping

Task I

```
/* 2-D Cartesian domain decomposition is assumed */  
/* 4 nonblocking MPI_Irecv + 4 nonblocking MPI_Isend */  
MPI_Request request[8];  
MPI_Irecv(..., nbrbottom, 0, comm2d, &request[0]);  
MPI_Irecv(..., nbrtop, 1, comm2d, &request[1]);  
MPI_Irecv(..., nbrleft, 2, comm2d, &request[2]);  
MPI_Irecv(..., nbrright, 3, comm2d, &request[3]);  
  
MPI_Isend(..., nbrtop, 0, comm2d, &request[4]);  
MPI_Isend(..., nbrbottom, 1, comm2d, &request[5]);  
MPI_Isend(..., nbrright, 2, comm2d, &request[6]);  
MPI_Isend(..., nbrleft, 3, comm2d, &request[7]);
```

Snippet of code to begin nonblocking send/recv to/from the other processes

June 24th, July 5th, IRA Bologna

Computation and communication overlapping

Task II

```
for (int i=x_start+1 ; i<=x_end-1 ; i++)
{
    for (int j=y_start+1 ; j<=y_end-1 ; j++)
    {
        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));
    } /* loop over j */
} /* loop over i */
```

Core algorithm of the Jacobi iteration assuming the 2-D Cartesian domain decomposition on local data (i.e. ghost points are not required).



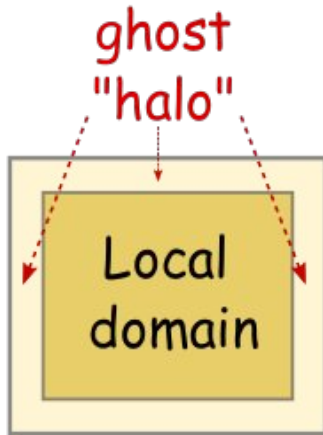
Computation and communication overlapping

Task III

```
/* wait the data on the boundaries */  
MPI_Waitall(8, request, MPI_STATUSES_IGNORE);  
  
JacobiAlgorithm( on_top_row      );  
JacobiAlgorithm( on_bottom_row   );  
JacobiAlgorithm( on_left_column  );  
JacobiAlgorithm( on_right_column );
```

Snippet of code to finalize the calculation on points of the grid that require ghost points.

Writing the solution using MPI: defining the local array type



Local domain are surrounded by a “halo” of ghost cells, namely top and bottom rows, left and right columns.

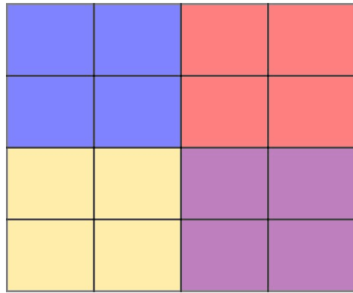
```
/* Every MPI process sets up the subarray */
/* (i.e. its own domain without ghost cells) */

MPI_Datatype subDomain;
/* (local) sub-domain size (i.e. without ghost cells) */
int lsize[2] = {local_dim[X], local_dim[Y]};
/* entire domain (i.e. with ghost cells) */
int domain[2] = {lsize[X] + 2, lsize[Y] + 2};
/* (local) starting coordinates of the subDomain in each dimension */
int lstart[2] = {local_start[X], local_start[Y]};

MPI_Type_create_subarray(2, domain, lsize, lstart, MPI_ORDER_C,
                        MPI_MyDatatype, &subDomain);

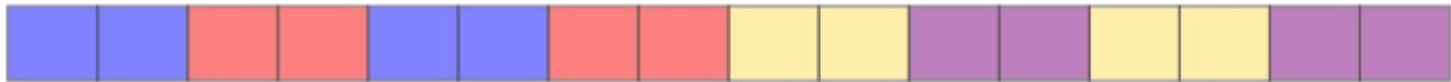
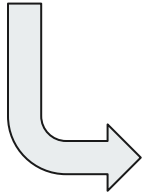
MPI_Type_commit(&subDomain);
```

Writing the solution: defining the file view



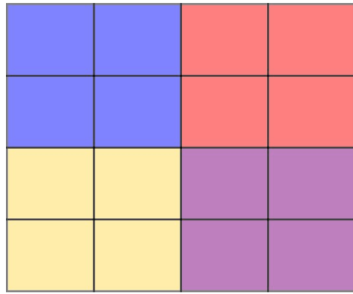
Global domain

```
MPI_Datatype ProcessDomain;  
/* (global) domain size (i.e. actually what we write) */  
int gsize[2] = {NX_GLOB, NY_GLOB};  
/* (global) starting coordinates of the subDomain in each dimension */  
int gstart[2] = {global_start[X] - 1, global_start[Y] - 1};  
  
MPI_Type_create_subarray(2, gsize, lsize, gstart,  
                        MPI_ORDER_C, MPI_MyDatatype,  
                        &ProcessDomain);  
  
MPI_Type_commit(&ProcessDomain);
```

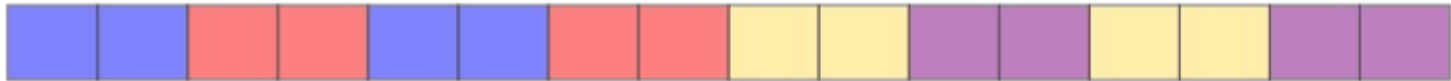
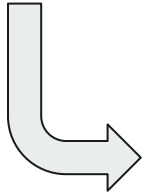


File

Writing the solution: putting all together



Global domain



File

```
MPI_File fh;
MPI_File_open(ThisTask->comm2d, fname, MPI_MODE_CREATE | MPI_MODE_WRONLY,
               MPI_INFO_NULL, &fh);

/* Set the process's view of the data in the file */
MPI_File_set_view(fh, 0, MPI_MyDatatype, ProcessDomain, "native", MPI_INFO_NULL);

/* Writes a file starting at the locations specified by individual file pointers */
/* (blocking collective) */
MPI_File_write_all(fh, buffer, 1, subDomain, MPI_STATUS_IGNORE);

/* Close a file (collective). */
MPI_File_close(&fh);
```