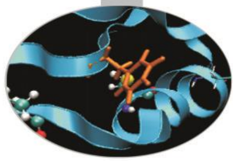
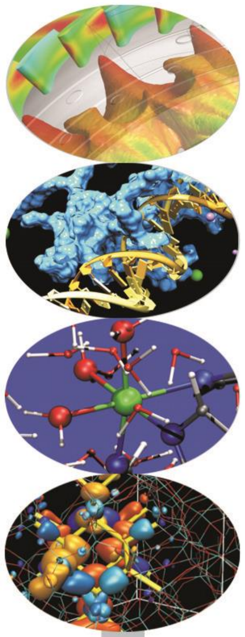
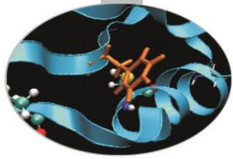
MPI introduction
- *exercises* -



Compiling notes

To compile programs that make use of MPI library:

```
mpif90/mpicc/mpiCC -o <executable> <file 1> <file 2> … <file n>
```

Where: `<file n>` - program source files

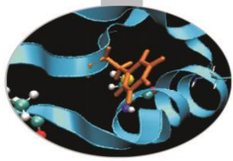
`<executable>` - executable file

To start parallel execution on one node only:

```
mpirun -np <processor_number> <executable> <exe_params>
```

To start parallel execution on many nodes:

```
mpirun -np <processor_number> -machinefile <node_list_file> \
       <executable> <exe_params>
```



Hello world! (Fortran)

As an ice breaking activity try to compile and run the *Hello* program, either in C or in Fortran.

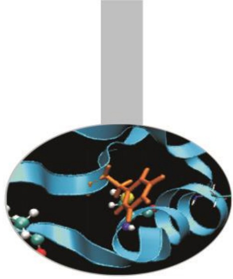
The most important lines in Fortran code are emphasized:

```
PROGRAM HelloWorld
    INCLUDE 'mpif.h'
    INTEGER my_rank, p
    INTEGER source, dest, tag
    INTEGER ierr, status(MPI_STATUS_SIZE)

    CALL MPI_Init(ierr)
    CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
    CALL MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

    WRITE(*,FMT="(A,I4)") "Hello world from process ", my_rank

    CALL MPI_Finalize(ierr)
END PROGRAM HelloWorld
```



Hello world! (C/C++)

The most important lines in C code are emphasized:

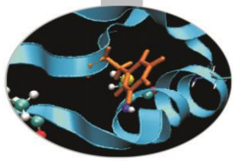
```
#include "mpi.h"

int main( int argc, char *argv[])
{
    int my_rank, numprocs;
    int dest, tag, source;
    MPI_Status status;

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

    printf("Hello world from process %d\n",my_rank);

    MPI_Finalize();
    return 0;
}
```



Hello world! (output)

If the program is executed with one process the output is:

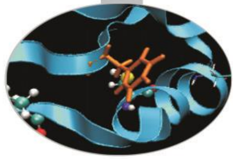
```
Hello world from process 0
```

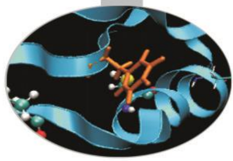
If the program is executed with four processes the output is:

```
Hello world from process 0
Hello world from process 1
Hello world from process 2
Hello world from process 3
```

Send array

Use point to point communication so that processes send an array of floats containing their rank to the next process. Each process will declare two float arrays, A and B, of a fixed dimension (10000). All of the elements of the array A will be initialized with the rank of the process. Then, A and B will be used as the buffers for SEND and RECEIVE, respectively. The program terminates with each process printing out one element of the array B.





Send array (Fortran)

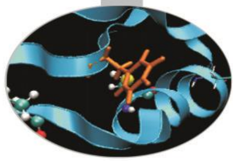
```
PROGRAM SENDARRAY
  USE MPI
  IMPLICIT NONE
  INTEGER IERR, ME, NPROCS, ERRCODE, SNDTO, RCVFROM
  INTEGER STATUS(MPI_STATUS_SIZE)
  INTEGER,PARAMETER :: NDATA = 10000
  REAL              :: A(NDATA)
  REAL              :: B(NDATA)

  CALL MPI_INIT(IERR)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS, IERR)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, ME, IERR)

  !$ INITIALIZE DATA
  A = ME

. . .
```

Send array (Fortran)

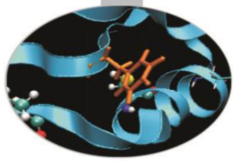


```
. . .
!$ SEND AND RECEIVE DATA
! SNDTO = ??
! RCVFROM = ??
! SENDING/RECEIVING CALLS?

  PRINT *,'I AM PROC ',ME,' AND I HAVE RECEIVED B(1) = ',B(1)
!$ FINALIZE MPI
! FINALIZE CALLS?

END PROGRAM SENDARRAY
```

Send array (C)



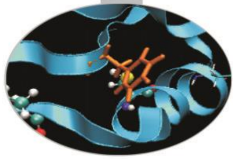
```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#define NDATA 10000
int main(int argc, char* argv[]){
   int me, nprocs, i = 0, sndto, rcvfrom;
   MPI_Status status;
   float a[NDATA];
   float b[NDATA];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &me);
. . .
```

Send array (C)

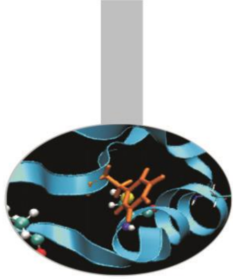
```
. . .
    /* Initialize data */
    for(i=0;i<NDATA;++i)   a[i] = me;
    /* Send and Receive data */
    // sndto = ??; rcvfrom = ??
    // Sending/receiving calls

    printf("\tI am task %d and I have received b(0) = %1.2f \n", me,
b[0]);

// Finalizing calls
    return 0;
}
```



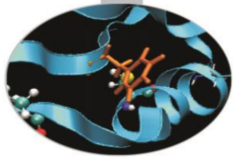
Sendrecv array



The solution of the last exercise could be simplified by using MPI_Sendrecv function:

```
int 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)
```

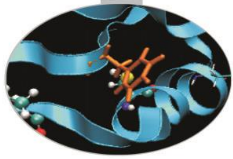
Ghost cells



Distribute a global square NxN matrix over P processors, so that each task has a local portion of it in its memory. Initialize such portion with the task's rank.

Each task sends its first and last columns (if Fortran) or rows (if C) to its neighbours (i.e.: the first column/row to the left processor, the last column/row to the right processor). Note that each task has to actually allocate a larger portion of its submatrix (ghost cells), with two extra columns/rows at the extremities to hold the received values.

Ghost cells (Fortran)



```
program ghost_cells
   use mpi
   implicit none
   integer, parameter :: N=20
           .          .          .
!!         Inizialize MPI


   !  number of columns for each mpi task
!!         Compute local columns (take care of remainder if any)
!!         Allocate and initialize local matrix
```

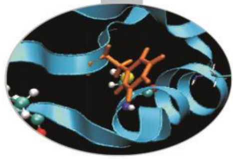
Ghost cells (Fortran)

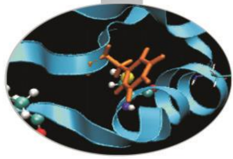
```
!!          Compute id of neighbours (proc_right, proc_left)


!!           send receive of the ghost regions
   deallocate(matrix)


!!          Finalize


  end program
```





Ghost cells (C)

. . .

```
#define N 20
int main(int argc, char* argv[]){

//        Initialize MPI
   /*  numer of rows for each mpi task */
//        Compute local rows (take care of remainder if any)
//        Allocate and initialize local matrix
```

Ghost cells (C)

```
//        Compute id of neighbours (proc_up, proc_down)


//         send receive of the ghost regions




   free(matrix);
//        Finalize
   return 0;
}
```