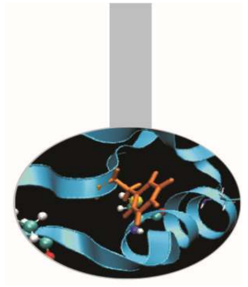
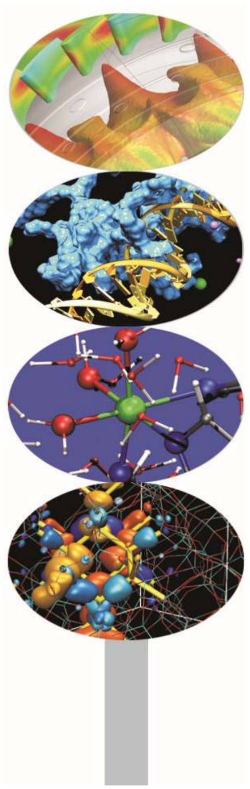
SuperComputing Applications and Innovation



Advanced MPI
- exercises -

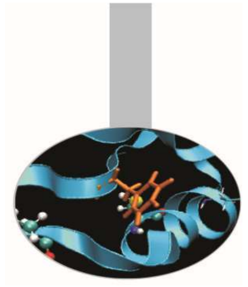


Introduction to Parallel Computing with MPI and OpenMP

M.Cremonesi

Segrate, November 2016

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

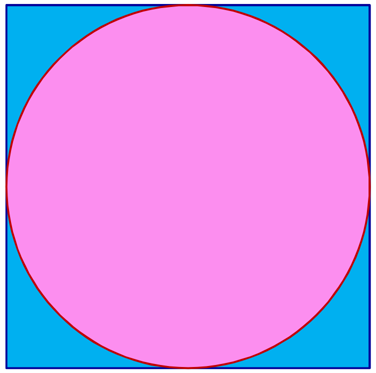
To start parallel execution on one node only:

```
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>
```

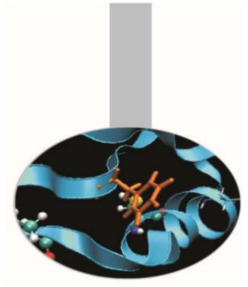
E1 – exercise – Montecarlo Pi



The value of the constant π can be approximated also by recalling that, given A_c the area of the circle and A_s the area of the circumscribing square:

$$\pi = 4\frac{A_c}{A_s}$$

Thus π could be approximated in a MonteCarlo style by counting the number of the random points in a square that are contained in the inscribed circle.

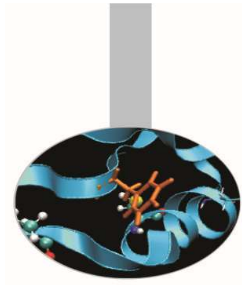


E1 – exercise – Montecarlo Pi

Therefore the program may be written this way:

- Divide a square in an number of parts (as many as the processes)
- Generate a number of random points in the area of each process
- Calculate how many points fall in the inscribed circle
- Sum up number of points in the square
- Sum up number of points in the circle
- Divide the two numbers

Source code: Pi_area



E2 – exercise – Round pack

Define a data structure composed of:

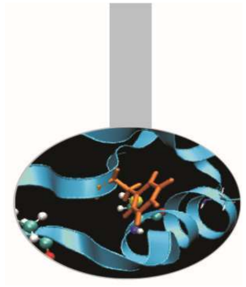
2 x MPI_INTEGER;

3 x MPI_DOUBLE_PRECISION;

9 x MPI_CHARACTER

Use MPI_Pack and MPI_Unpack to pack data in one buffer and send it round-robin wise.

Source code: Round_Pack



E3 – exercise – Round struct

Modify the code of the previous Round_Pack exercise to define a MPI derived data type that matches the data structure composed of:

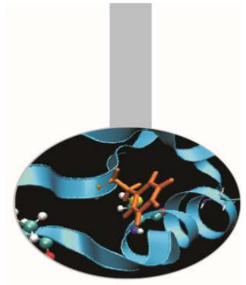
2 x MPI_INTEGER;

3 x MPI_DOUBLE_PRECISION;

9 x MPI_CHARACTER

Source code: Round_Struct

E4 – example – Mandelbrot set

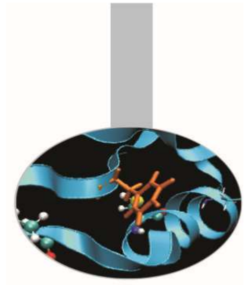


In 1979 Benoît Mandelbrot, who was working at Thomas J. Watson Research Center of IBM, was studying what would have been later known as the *Mandelbrot set*. This mathematical object may be easily studied only by means of numerical computing, with the added support of computer graphics.

Defining the Mandelbrot set is quite easy:

Given the transformation $z \rightarrow z^2$ in the complex plane, iterate it at each point of the circle of radius 2 centred in the origin.

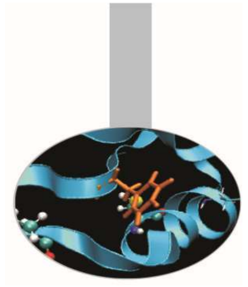
The Mandelbrot set is the set of points that do not diverge outside this circle.



E4 – example – Mandelbrot set

Of course points inside the circle with radius 1 always remain in the set, but there is no simple rules to decide whether the other points do belong to the set or not. The border of the set has fractal properties. Moreover, because of chaos behavior coming from exponent operations, points starting very closed together may diverge considerably.

The example program computes the Mandelbrot set in a given area (inside the radius two circle) and creates an image on the basis of how many iterations are needed to send a point outside the circle. The result is a well known image that can also be used to effectively check the correctness of the program.



E4 – example – Mandelbrot set

The image is generated in PGM or PPM formats because they are very easy to remember and realize.

PGM format:

Row 1 – P2

Row 2 - <rows> <columns>

Row 3 - <Maximum value>

… <point values> …

PPM format:

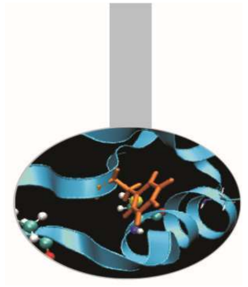
Row 1 – P3

Row 2 - <rows> <columns>

Row 3 - <Maximum value>

… <R G B point values> …

E4 – example – Mandelbrot set



The program could thus be sketched this way:

Define area in complex plane (squared for simplicity)

Define image size (squared for simplicity)

Define maximum iterations per point

Broadcast data to all processes

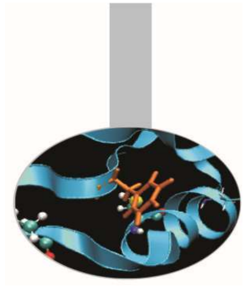
Parallel computation by domain decomposition

Gather results

Produce image

Source code: *Mand*

Reference: *http://mrob.com/pub/muency.html*

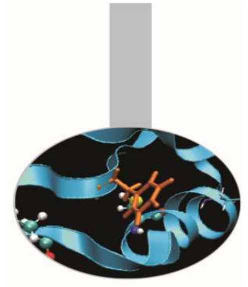


E5 – example – Cartesian

Create a 2-dimensional cartesian grid topology to communicate between processes. In each task a variable is initialized with the local rank of the cartesian communicator. The exercise is in three steps:

1) Compare the local rank with the global MPI_COMM_WORLD rank. Are they the same number?

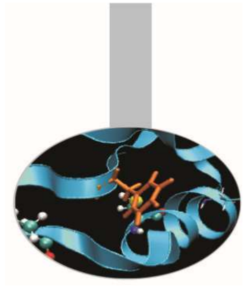
2) Calculate on each task the average between its local rank and the local rank from each of its neighbours (north, east, south, west). Notice that in order to do this the cartesian communicator has to be periodic (the bottom rank is neighbour of the top)



E5 – example – Cartesian

3) Calculate the average of the local ranks on each row and column. Create a family of sub-cartesian communicators to allow the communications between rows and columns

Source code: *Cartesian*

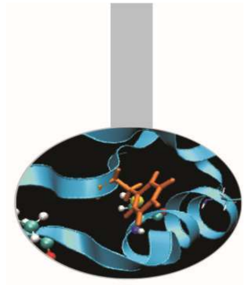


E6 – example – Diagonal

Each task initializes a square N x N matrix (N is the total number of the tasks) with 0s, except for the diagonal elements of the matrix that are initialized with the task's rank number.

Each task sends to rank 0 an array containing all the elements of its diagonal. Task 0 overwrites the array sent by process i on the i-th row (column if Fortran) of its local matrix. At the end, task 0 prints its final matrix, on which each element should be the number of its row (or column).

Source code: *Diagonal*



E7 – example – Mpi2io

Write a code that writes and reads a binary file in parallel according to the following steps:

I) First process writes integers 0-9 from the beginning of the file, the second process writes integer 10-19 from the position 10 in the file and so on.

II) Re-open the file. Each process reads the data just written. Check that the reading has been performed correctly.

NOTE: to see the binary file in ASCII use the command:

```
od -i <binary_file>
```

Source code: *Mpi2io*