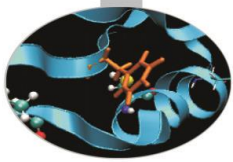
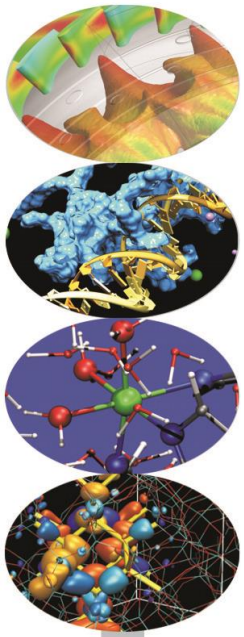
CINECA SCAI
SuperComputing Applications and Innovation



OpenMP
- *exercises* -

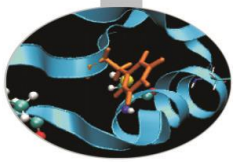


Introduction to Parallel Computing with MPI and OpenMP

P.Dagna

May 2015

CINECA



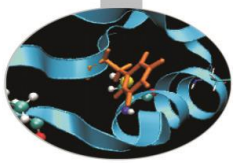
Hello world! (Fortran)

As a beginning activity let's compile and run the *Hello* program, either in C or in Fortran.

The most important lines in Fortran code are emphasized:

```
PROGRAM HELLO
   IMPLICIT NONE
   INTEGER :: ID, NTHREADS
   INTEGER :: OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS

!$OMP PARALLEL &
!$OMP PRIVATE(ID,NTHREADS)
      PRINT*,"Hello world!"
!$OMP END PARALLEL
   STOP
END PROGRAM HELLO
```

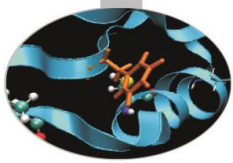


Hello world! (C/C++)

Again, the most important lines in C code are emphasized:

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
#include <stdlib.h>

main (int argc, char *argv[])
{
#pragma omp parallel
{
        fprintf(stdout," Hello world!\n");
}
   return(0);
}
```



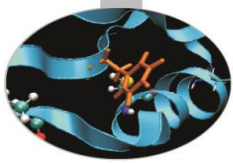
Hello world! (output)

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

```
Hello world!
```

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

```
Hello world!
Hello world!
Hello world!
Hello world!
```



Compiling notes

To compile programs that make use of OpenMP directives:

```
gfortran/gcc/g++ -fopenmp -o <executable> <file 1>
… <file n>
```

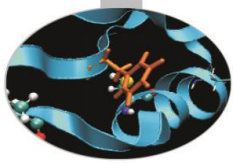
Where: `<file n>` - program source files

`<executable>` - executable file

To start parallel execution:

```
export OMP_NUM_THREADS=<number_of_threads>
./<executable>
```

E1 – exercise

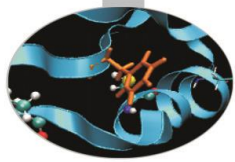


By making use of the proper runtime functions try to add to the former examples instructions for writing number of activated threads and thread ids.

The output generated by the program should look like:

```
Hello world from:
Thread id: 0; total number: 4
Thread id: 2; total number: 4
Thread id: 3; total number: 4
Thread id: 1; total number: 4
```

E2 – example – Pi by quadrature

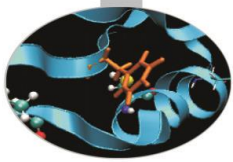


It is known that the mathematical constant π can be approximated by computing the following formula:

$$\pi = 4 \int_0^1 \frac{1}{1+x*x} \, dx$$

The value of the above integral can be approximated by numerical integration, i.e. by computing the mean value of the function f(x) = $\frac{1}{1+x*x}$ in a number of points and multiplying per the x range. This can be easily done in parallel by dividing the [0,1] range into a number of intervals.

E2 – example – Pi by quadrature

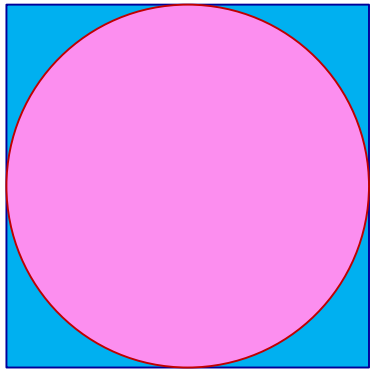


Thus the program may be sketched this way:

- (if my_rank == 0) get number of intervals for quadrature
- Iterate for computing function value in the centre of each interval
- Sum up function values
- Divide by interval range and multiply by 4

Source code: *Pi_integral*

E3 – exercise – Montecarlo Pi

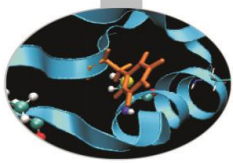


The value of the constant π can be approximated also by recalling that, given A_c the area of the circle and A_s the area of the circumscribing square:

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

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.

E3 – exercise – Montecarlo Pi



Therefore the program may be written this way:

Decide how many points have to be generated

Generate random points in a squared region

Calculate how many points fall in the inscribed circle

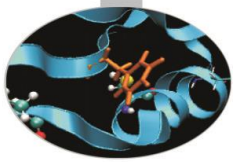
Sum up number of points in the square

Sum up number of points in the circle

Divide the two numbers

Source code: *Pi_area*

E4 – example – Mandelbrot set



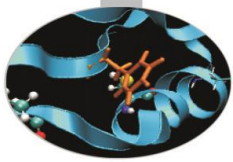
In 1979 Benoît Mandelbrot, who was working at Thomas J. Watson Research Center of IBM, was studying what would have been later known as Mandelbrot set. This mathematical object may be easily studied only by means of numerical computing, with the added support of computer graphics.

Defining the Mandelbrot set is quite easy:

Given the transformation $z \rightarrow z+z_0^2$ in the complex plane, iterate 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.

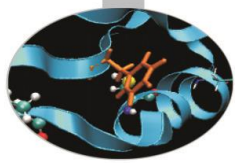
E4 - example – Mandelbrot set



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. In fact 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:

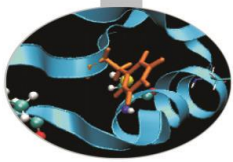
Row 1 – P3

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

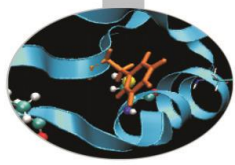
Compute iterations for every point in the square

Produce image

Parallel computation may be implemented by domain decomposition or loop distribution.

Source code: Mandel

E5 – exercise – Matrix multiply



Matrix row-column multiply is an example of program that can be easily parallelized.

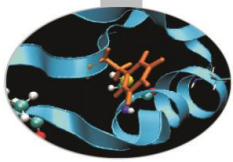
Given the matrices A(L,M), B(M,N), C(L,N) try writing a OpenMP program that computes C = A x B

The program could be written this way:

Decide matrix sizes

Parallel computation by distributing loops iterations

Source code: *MatrMult*

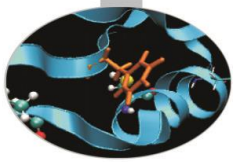


E6 – example – Life game

John Conway's LIFE game has been described since 1970 on Scientific American. It consists in a very large checkerboard where there is a initial configuration of marked (or alive) cells. At each iteration per each cell the number F of the alive cells (taken among the 8 adjacent ones) is counted and the cell is marked alive or not according to the following rules:

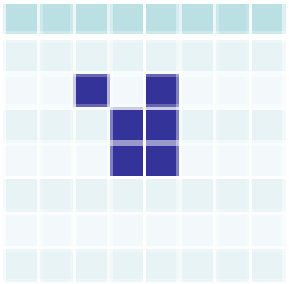
The cell survives if	2 <= F <= 3
The cell dies if	4 <= F or F <= 1
The cell gets alive if	F = 3

The game rules are very simple but it is very difficult to predict the population evolution.

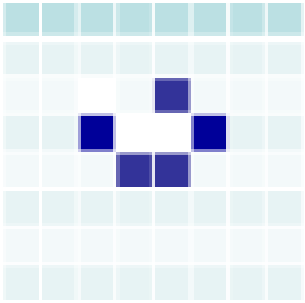


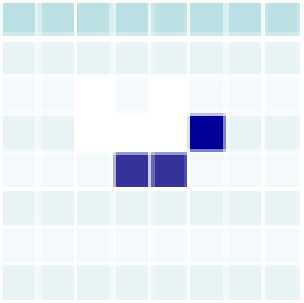
E6 – example – Life game

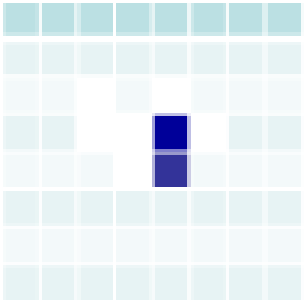
As an example given a very simple initial configuration:

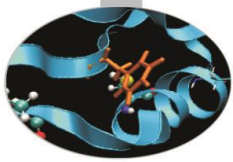


The evolution at first steps is:









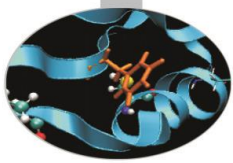
E6 - example – Life game

Programming difficulties for implementing a LIFE game are closed to issues encountered for programming PDE solvers with regular meshes.

- A sequential program may be written in the following way:
- Decide board sizes (squared for simplicity) and number of iterations
- Allocate matrix A(:,:) for current state
- Allocate matrix B(:,:) for next state
- Choose an initial configuration
- Iterate:

    store next state in matrix B by applying rules on matrix A

    swap matrices



E6 - example – Life game

Issues for parallel version:

- Decide board decomposition: divide board in disjoined portions
- Distribute portions (with boundaries) to threads
- Iterate:

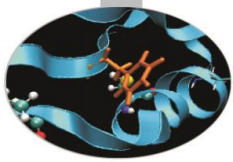
    store next state in matrix B by applying rules on matrix A

    swap matrices

Source code: *LifeGame*

Reference: *http://www.bitstorm.org/gameoflife/*

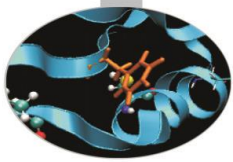
E7 – exercise – Heat equation



- The distribution of heat over time is described by the so called heat equation:
  - df/dt = alpha*(d2f/dx2 + d2f/dy2) for a function f(x,y,t).
- This formula may be discretized in a regular grid G(:,:) by computing the new value G1(x,y) in a point (x,y) at each time step as:
  - G1(x,y) = G(x,y) + Ax * (G(x+1,y) + G(x-1,y) - 2.0 * G(x,y) + Ay * (G(x,y+1) + G(x,y-1) - 2.0 * G(x,y))
- For each point in the grid the next value depends on the values of the four up and down, left and right adjacent points.

- Source code : heat_2D_ser

E8 – exercise – Fibonacci



- The Fibonacci Sequence is the series of numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- The next number is found by adding up the two numbers before it.
- The source code “fibonacci.c” or “fibonacci.f90” compute the Fibonacci sequence in a serial way using a recursive function.
- Try to parallelize the code using OpenMP directives