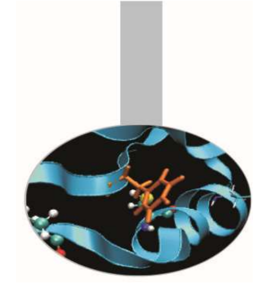
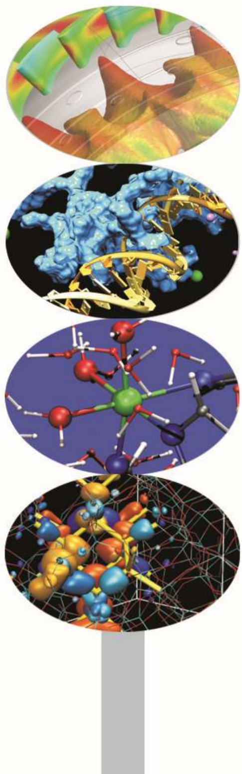
CINECA SCAI
SuperComputing Applications and Innovation



Profiling



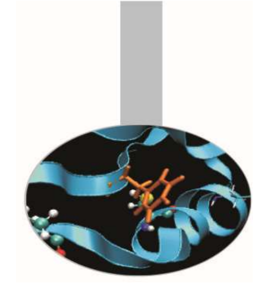
P.Dagna, M.Cremonesi

May 2016

CINECA

SCAI
SuperComputing Applications and Innovation

Introduction



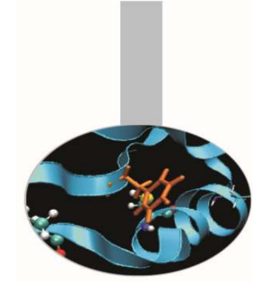
A serial or parallel program is normally composed by a large number of procedures.

To optimize and parallelize a complex code it is fundamental to find out the parts where most of time is spent.

Moreover is very important to understand the graph of computation and the dependencies and correlations between the different sections of the code.

CINECA

SuperComputing Applications and Innovation



Introduction

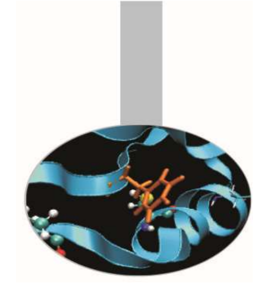
For a good scalability in **parallel programs**, it's necessary to have a good load and communication balancing between processes.

To **discover** the **hotspots** and the **bottlenecks** of a code and find out the **best optimization and parallelization strategy** the programmer can follow two common methods:

- Manual instumentation inserting timing and collecting functions (not so easy)
- Automatic profiling using **profilers** (easier and very powerful)

CINECA

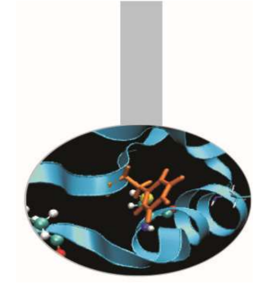
Measuring execution time



Both C/C++ and Fortran programmers are used to instrument the code with timing and printing functions to measure, collect or visualize the time spent in critical or computationally intensive code sections.

- **Fortran77**
  - `etime(),dtime()`
- **Fortran90**
  - `cputime(), system_clock(), date_and_time()`
- **C/C++**
  - `clock()`

CINECA SCAI
SuperComputing Applications and Innovation



Measuring execution time

This kind of measurements are affected by:

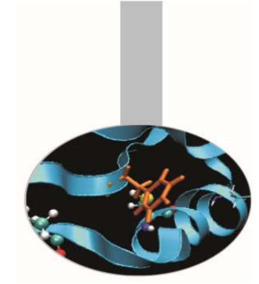
- Intrusivity
- Granularity
- Reliability
- Overhead

Very difficult task for third party complex codes

CINECA

SCAI
SuperComputing Applications and Innovation

Measuring execution time



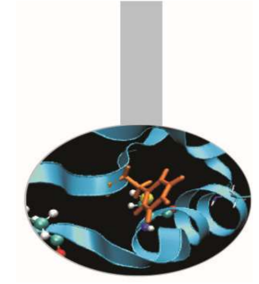
C example:

```
#include <time.h>
clock_t time1, time2;
double dub_time;
…
time1 = clock();
for (i = 0; i < nn; i++)
for (k = 0; k < nn; k++)
for (j = 0; j < nn; j ++)
c[i][j] = c[i][j] + a[i][k]*b[k][j];
time2 = clock();
dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
printf("Time -----------------> %lf \n", dub_time);
```

CINECA

SCAI
SuperComputing Applications and Innovation

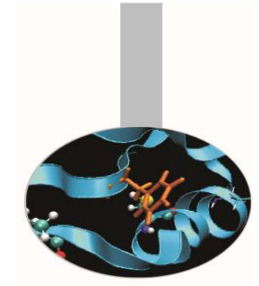
Measuring execution time



Fortran example:

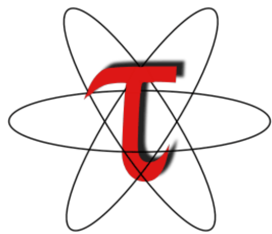
```
real(my_kind), intent(out) :: t
integer :: time_array(8)
…
call date_and_time(values=time_array)
t1 = 3600.*time_array(5) + 60.*time_array(6) + &
   & time_array(7) + time_array(8)/1000.
do j = 1,n
   do k = 1,n
        do i = 1,n
                c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
   enddo
enddo
call date_and_time(values=time_array)
t2 = 3600.*time_array(5) + 60.*time_array(6) + &
   & time_array(7) + time_array(8)/1000.
write(6,*) t2-t1
```

CINECA



Profilers

There are many versions of commercial profilers, developed by manufacturers of compilers and specialized software house. In addition there are **free profilers**, as those resulting from the GNU, TAU or Scalasca project.



Tau Performance System
- University of Oregon

Intel® VTune™ Amplifier

Scalasca
-Research Centre Juelich

The Portland Group PGPROF

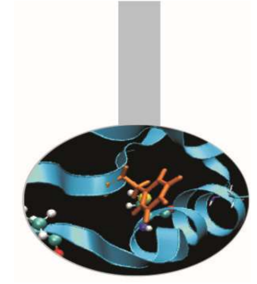
GNU gprof

OPT

PerfSuite
– National Center for Supercomputing Applications

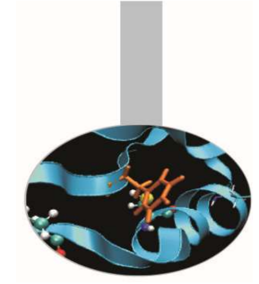
SuperComputing Applications and Innovation

Profilers



Profilers allow the programmer to obtain very useful information on the various parts of a code with basically two levels of profiling:

- **Subroutine/Function level**
- **Construct/instruction/statement level**



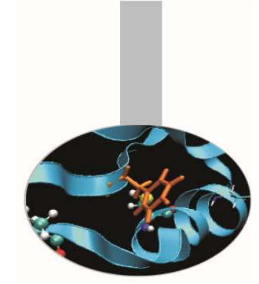
Profilers

- **Subroutine/Function level**
  - Timing at routine/function level, graph of computation flow
  - less intrusive
  - Near realistic execution time

- **Construct/instruction/statement level**
  - capability to profile each instrumented statement
  - more intrusive
  - very accurate timing information
  - longer profiling execution time

SuperComputing Applications and Innovation

GNU Profiler

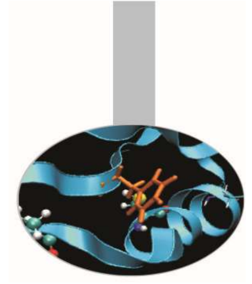


The GNU profiler "gprof" is an open-source tool that allows profiling of serial and parallel codes.

Code is automatically instrumented by the compiler when using the **–pg** flag, during the execution:

- the **number of calls** and the **execution time** of each subroutine is collected
- a call graph containing **dependences between subroutines** is implemented
- a binary file containing above information is generated (**gmon.out**)

CINECA

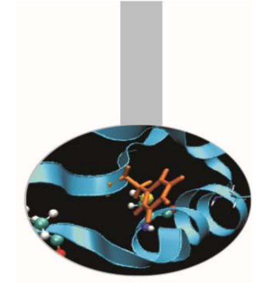


GNU Profiler

Using data contained in the file *gmon.out, gprof* is able to give precise information about:

1. the **number of calls** of each routine
2. the **execution time** of a routine
3. the **execution time** of a routine and all the child routines called by that routine
4. a **call graph profile** containing **timing information and relations** between subroutines

GNU Profiler



GNU profiler how to:

- Recompile source code using compiler profiling flag:

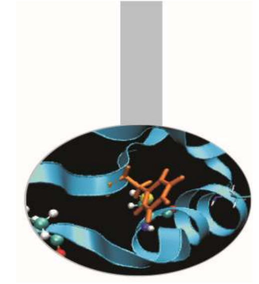
```
gcc/g++ -pg source code
gfortran -pg source code
```

- Run the executable to allow the generation of the files containing profiling information:
  - At the end of the execution in the working directory will be generated a specific file generally named *“gmon.out”* containing all the analytic information for the profiler
- Produce analysis results:

```
gprof executable gmon.out
```

SCAI
SuperComputing Applications and Innovation

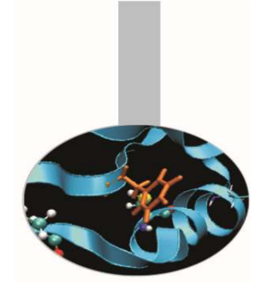
Example



```
#include<stdio.h>

double add3(double x) {
        return x+3; }

double mysum(double *a, int n) {
    double sum=0.0;
    for(int i=0;i<n;i++)
         sum+=a[i]+add3(a[i]);
    return sum; }

double init(double *a,int n) {
    double res;
    for (int i=0;i<n;i++) a[i]=(double)i/(double)1000;
    res=mysum(a,n);
    return res; }

int main(){
    double res,mysum;
    int n=20000;
    double a[n];

    for (int i=0;i<n;i++){
         res=init(a,n);
    }
    printf("Result %f\n",res);
return 0;}
```

CINECA

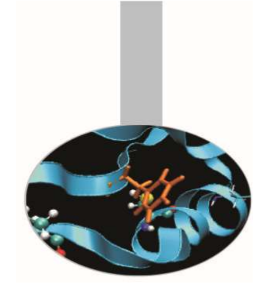
Profiler output



Execute these commands to produce profiler output:

```
gcc  -std=c99 -pg 0601-Gprof_example.c

time ./a.out

gprof a.out
```

Profiler output



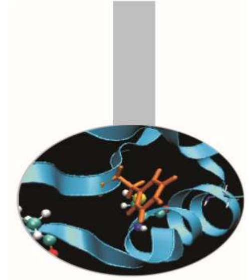
The profiler **gprof** produces two kinds of statistical output: "**flat profile**" and "**call graph profile**".

According to previous example **flat profile** gives the following information:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %    cumulative   self              self     total
 time    seconds   seconds    calls  us/call  us/call  name
 57.37       2.65     2.65    20000   132.52   227.75  init
 33.16       4.18     1.53    20000    76.59    95.23  mysum
  8.07       4.56     0.37 400000000     0.00     0.00   add3
```

SCAI
SuperComputing Applications and Innovation

Flat profile



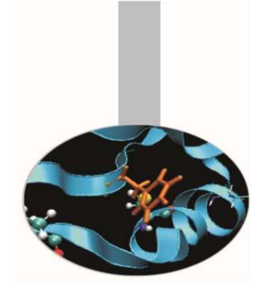
The meaning of the columns displayed in the **flat profile** is:

- **% time**: percentage of the total execution time your program spent in this function
- **cumulative seconds**: cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table
- **self seconds**: number of seconds accounted for by this function alone.
- **calls**: total number of times the function was called
- **self us/calls**: represents the average number of microseconds spent in this function per call
- **total us/call**: represents the average number of microseconds spent in this function and its descendants per call if this function is profiled, else blank
- **name**: name of the function

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Call Graph



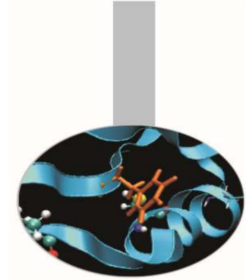
Call Graph Profile: gives more detailed timing and calling sequence information through a dependency call graph.

```
Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.21% of 4.66 seconds

index % time    self  children    called     name
                2.65    1.90   20000/20000        main [2]
[1]     97.8    2.65    1.90   20000          init [1]
                1.53    0.37   20000/20000        mysum [3]
-----------------------------------------------
                                                  <spontaneous>
[2]     97.8    0.00    4.56                  main [2]
                2.65    1.90   20000/20000        init [1]
-----------------------------------------------
                1.53    0.37   20000/20000        init [1]
[3]     40.9    1.53    0.37   20000          mysum [3]
                0.37    0.00 400000000/400000000     add3 [4]
-----------------------------------------------
                0.37    0.00 400000000/400000000     mysum [3]
[4]      8.0    0.37    0.00 400000000          add3 [4]
-----------------------------------------------
```

CINECA

SuperComputing Applications and Innovation

Line level profiling



If necessary it’s possible to profile single lines or blocks of code with the “*gcov*” tool to see:

- lines that are most frequently accessed
- computationally critical statements or regions

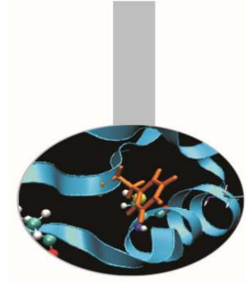
NOTES:

- `gcov` is compatible only with code compiled with GNU compilers
- use low level optimization flags.

CINECA

SuperComputing Applications and Innovation

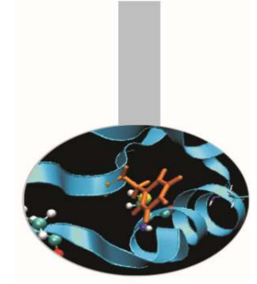
Line level profiling



Line level profiling with gcov requires the following steps

- compile with `-fprofile-arcs -ftest-coverage`
  At the end of compilation files `*.gcno` will be produced
- Run the executable. The execution will produce `*.gcda` files
- Run `gcov`: `gcov [options] sourcefiles`
- At the end of execution a specific file with extension `*.gcov` will be present in the working directory. It contains all the analytic information for the profiler

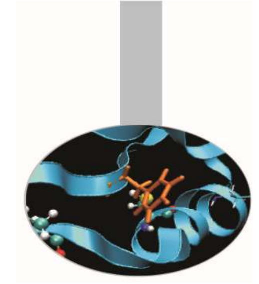
Example



```
#include <stdlib.h>
#include <stdio.h>

int prime (int num);
int main() {
    int i;
    int cnt = 0;
    for (i=2; i <= 1000000; i++)
        if (prime(i)) {
            cnt++;
            if (cnt%9 == 0) {
               printf("%5d\n",i);
               cnt = 0;
            } else
                printf("%5d ", i);
        }
        putchar('\n');
        if (i<2) printf("OK\n");
        return 0; }

int prime (int num) {
   int i;
   for (i=2; i < num; i++)
        if (num %i == 0) return 0;
   return 1; }
```

SCAI
SuperComputing Applications and Innovation

Profiler output

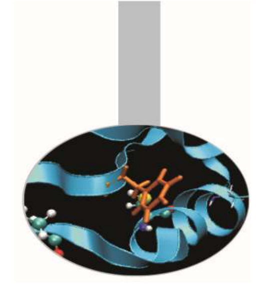


Execute these commands to produce line level profiler output:

```
gcc -std=c99 -fprofile-arcs -ftest-coverage \
      gprof_prime.c -lm
./a.out >& primes.log
gcov gprof_prime.c
more gprof_prime.c.gcov
```

CINECA

Example



Routine level profiling produces the following information:

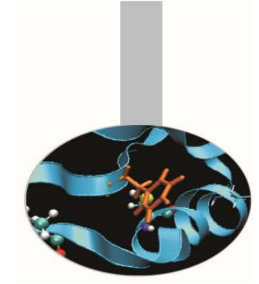
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.99    109.74   109.74   999999   109.74   109.74  prime(int)
```

call-graph output:

```
granularity: each sample hit covers 2 byte(s) for 0.01% of 109.74 seconds

index % time    self  children    called     name

[1]    100.0    0.00  109.74                 main [1]
              109.74    0.00  999999/999999     prime(int) [2]
-----------------------------------------------
              109.74    0.00  999999/999999     main [1]
[2]    100.0  109.74    0.00  999999         prime(int) [2]
```

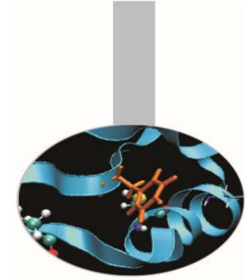
How is time effectively spent in routine `prime`??

Example



```
        -:    1:#include <stdlib.h>
        -:    2:#include <stdio.h>
        -:    3:
        -:    4:int prime (int num);
        -:    5:
        1:    6:int main()
        -:    7: {
        -:    8:        int i;
        1:    9:        int cnt = 0;
  1000000:   10:        for (i=2; i <= 1000000; i++)
   999999:   11:                if (prime(i)) {
    78498:   12:                cnt++;
    78498:   13:                if (cnt%9 == 0) {
     8722:   14:                        printf("%5d\n",i);
     8722:   15:                        cnt = 0;
        -:   16:                }
        -:   17:                else
    69776:   18:                printf("%5d ", i);
        -:   19:                }
        1:   20:        putchar('\n');
        1:   21:        if (i<2)
    #####:   22:                printf("OK\n");
        1:   23:        return 0;
        -:   24: }
        -:   25:
   999999:   26:int prime (int num) {
        -:   27: /* check to see if the number is a prime? */
        -:   28: int i;
37567404990:   29: for (i=2; i < num; i++)
37567326492:   30:        if (num %i == 0) return 0;
    78498:   31: return 1;
        -:   32: }
```

Example



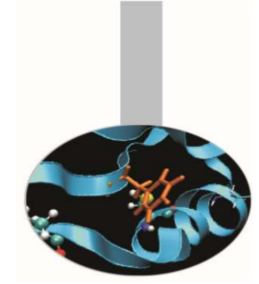
Line level profiling shows that most of time is spent in the `for` loop and in the `if` construct contained in the `prime` function.

- **Let's check for a more efficient algorithm.**

If a number "n" is not a prime, it can be factored into two factors "a" and "b" : n = a*b

If both a and b were greater than the square root of n, a*b would be greater than n.

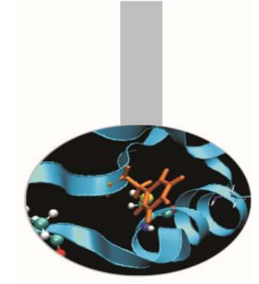
At least one of the factors must be less or equal to the square root of n, and to check if n is prime, we only need to test for factors less than or equal to the square root.



Example

```
int prime (int num) {
/* check to see if the number is a prime? */
  int i;
  for (i=2; i <= faster(num); i++)
  if (num %i == 0)
       return 0;
  return 1;
}

int faster (int num) {
  return (int) sqrt( (float) num);
}
```

SCAI
SuperComputing Applications and Innovation



Example

```
      1:    7:int main(){
      -:    8: int i;
      1:    9: int colcnt = 0;
1000000:   10: for (i=2; i <= 1000000; i++)
 999999:   11: if (prime(i)) {
  78498:   12: colcnt++;
  78498:   13: if (colcnt%9 == 0) {
   8722:   14: printf("%5d\n",i);
   8722:   15: colcnt = 0;
      -:   16: }
      -:   17: else
  69776:   18: printf("%5d ", i);
      -:   19: }
      1:   20: putchar('\n');
      1:   21: return 0;
      -:   22: }
      -:   23:
 999999:   24: int prime (int num) {
      -:   25: int i;
67818902:  26: for (i=2; i <= faster(num); i++)
67740404:  27: if (num %i == 0)
 921501:   28:         return 0;
  78498:   29: return 1;
      -:   30: }
      -:   31:
67818902:  32: int faster (int num)
      -:   33: {
67818902:  34: return (int) sqrt( (float) num);
      -:   35: }
```

Results

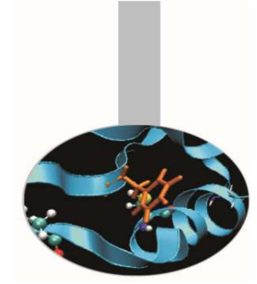
0.96 sec Vs 109.67 sec

10^7 operations VS 10^10 operations

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

gprof execution time impact



- Routine level and above all line level profiling can cause a overhead in execution time:
- Travelling Salesman Problem (TSP):

```
g++ -pg -o tsp_prof tsp.cc
    g++ -o tsp_no_prof tsp.cc
```

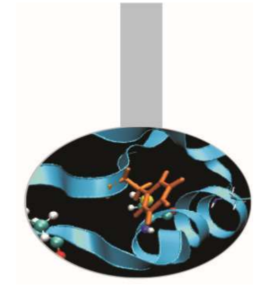
- Execution time

```
time ./TSP.noprof
10.260u 0.000s 0:10.26 100.0%

time ./TSP.prof
15.480u 0.020s 0:15.87 97.6%
```

- **Be careful when you have to choose input dataset and configuration for profiling**

CINECA

SCAI
SuperComputing Applications and Innovation



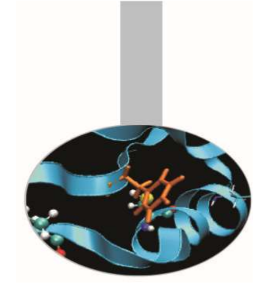
Real case Air Pollution Model

- Model structure and call graph
- Fluid dynamics equations are solved over a 3D grid

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Real case Air Pollution Model



- Profiling with GNU profiler (call graph)

```
index % time    self  children    called     name

 [2]     95.3    0.00 9511.19                  main [2]
                0.00 9511.19       1/1            MAIN__ [1]
-----------------------------------------------
                0.00 9507.46       1/1            MAIN__ [1]
[3]     95.2    0.00 9507.46       1          comp_ [3]
              192.03 9047.81     360/360          opspltae_ [4]
              110.52    0.00     360/360          pmcalcdry_ [31]
               59.29    6.23     119/119          aestim_ [33]
               48.95    8.22     120/120          qgridae_ [35]
               19.46    0.00     958/2398         units_ [36]
```

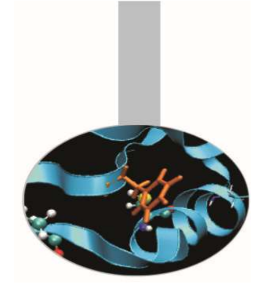
- **5 days of simulation.** Only the computationally intensive routines of the model are shown

- **Dependency call graph of "`opspltae`" routine**

```
index % time    self  children    called     name

              192.03 9047.81     360/360          comp_ [3]
[4]     92.6  192.03 9047.81     360          opspltae_ [4]
               11.71 4346.21 22096800/22096800     chemnew_ [5]
              926.45 2381.89     720/720          horizae_ [10]
              861.92    0.00 8035200/8035200      ztrans_ [15]
               36.54  413.18 22096800/22096800     aero_iso_ [17]
               40.31    0.00 22096800/22096800     phfact_ [39]
               29.26    0.00    1440/2398         units_ [36]
```

CINECA

Real case air pollution model parallelization strategy

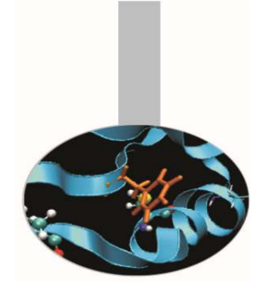


- `Opspltae` is called every time step by "`comp`" and calls `chemnew,horizae,ztrans,aero_iso,phfact` and `units` routines. In these routines is spent 92,6% of simulation time.

- The rest of time is spent for initialization, finalization and I/O operations which are not parallelizable or which parallelization doesn't make sense for.

SCAI
SuperComputing Applications and Innovation

Real case air pollution model parallelization strategy



- Ideal speedup obtainable according to profiler output is:

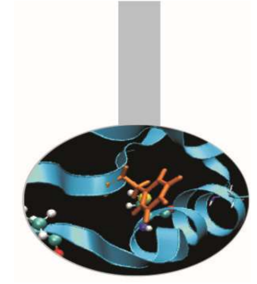
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \quad \Rightarrow \quad S(N) = 14$$

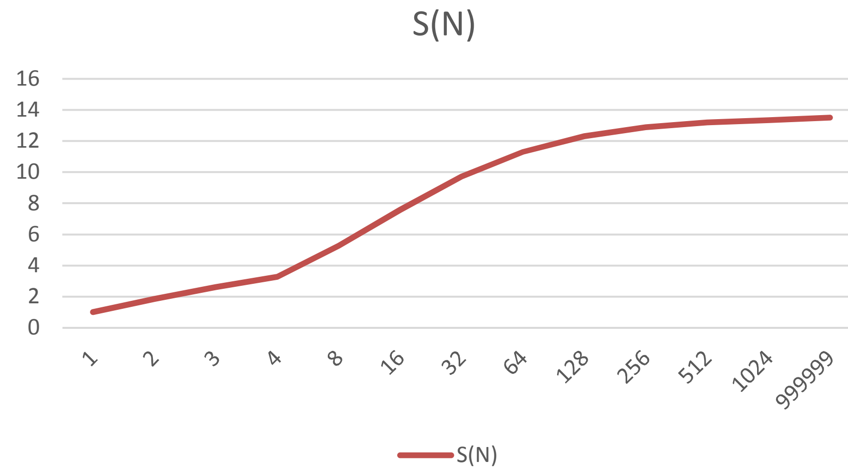
- Results
  - Real speedup : 7.6 ☹ **Why?**

CINECA

SCAI
SuperComputing Applications and Innovation

Real case air pollution model parallelization strategy





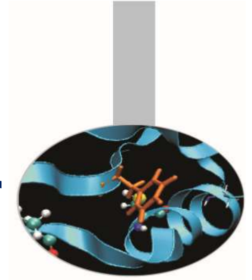
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \quad \Rightarrow \quad S(N) = 14$$

- Results
  - Real speedup : 7.6 ☹

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Parallel codes profiling with gprof



GNU profiler can be used to profile **parallel codes** too but analysis is not straightforward. To profile parallel codes the user must follow these steps:

- Set the environment variable `GMON_OUT_PREFIX:`

```
export GMON_OUT_PREFIX="profile_data_file"
```

- Compile with "–p" flag:

```
mpic++/mpicc/mpif70/mpif90 -p filenames
```

- Run the executable: `mpirun –np number executable`

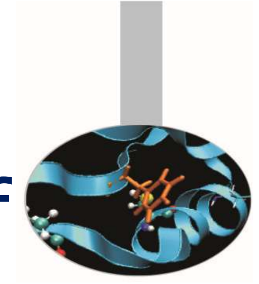
In the working directory at the end of simulation as many `profile_data_file.pid` files will be present as many MPI or OpenMP processes were used.

Each profiling file must be analyzed individually and the results have to be matched together:

```
gprof ./executable profile_data_file.pid
```

CINECA

SuperComputing Applications and Innovation



SCalable performance Analysis of LArge SCale Applications

SCALASCA is a toolset for performance analysis of parallel applications on a large scale

It manages MPI, OpenMP, MPI+OpenMP programs

See an introduction at https://hpc-forge.cineca.it/files/ScuolaCalcoloParallelo_WebDAV/public/anno-2014/23_summer_school/debug_prof.pdf.zip

CINECA