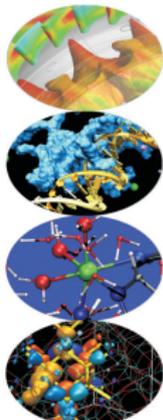


# Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)

Roma, Aprile 2015



## Profilers

### Introduzione

### gprof

## Profilers

### Introduzione

### gprof



La misura e la stima delle prestazioni di un programma permette di:

- ▶ Verificare lo sfruttamento delle risorse hardware/software;
- ▶ Pianificare le simulazione da fare;
- ▶ Evidenziare eventuali problemi di prestazione e/o velocità.
- ▶ Ridurre il tempo totale dalle realizzazione del codice all'effettiva produzione dati.

La misura delle prestazioni però:

- ▶ Fornisce indicazioni di buono o cattivo funzionamento, ma non le cause;
- ▶ Fornisce indicazione a livello di implementazione, ma non a livello di algoritmo.

- ▶ Presente in tutte le architetture *Unix /Linux*.
- ▶ Fornisce il tempo totale di esecuzione di un programma ed altre utili informazioni.
- ▶ Non ha bisogno che il programma sia compilato con particolari opzioni di compilazione (**assolutamente non intrusivo**).
- ▶ Sintassi: **time** <nome\_eseguibile>

un tipico output:

```
[~@louis ~]$ /usr/bin/time ./a.out
9.29user 6.19system 0:15.52elapsed 99%CPU
(0avgtext+0avgdata 18753424maxresident)k
0inputs+0outputs (0major+78809minor)pagefaults 0swaps
```

```
9.29user 6.19system 0:15.52elapsed 99%CPU  
(0avgtext+0avgdata 18753424maxresident)k  
0inputs+0outputs (0major+78809minor)pagefaults 0swaps
```

1. (*User time*) Il tempo di CPU impiegato dal processo
2. (*System time*) Il tempo di CPU impiegato dal processo in chiamate di sistema
3. (*Elapsed time*) Il tempo totale effettivamente impiegato
4. La percentuale di CPU utilizzata dal processo.
5. Memoria usata dal processo
6. I/O
7. Page-faults
8. Numero di volte che il processo è stato scambiato in memoria

- ▶ L'uso di `time` su questo eseguibile ci ha dato alcune informazioni interessanti:
  - ▶ (Lo "user" time è confrontabile con il "sys" time)
  - ▶ (La percentuale di utilizzo della CPU è quasi del 100%)
  - ▶ (Non è presente I/O)
  - ▶ (Non vi sono quasi per nulla "page-faults")
  - ▶ (L'area dati (massima) durante l'esecuzione è di circa 18Gbytes)
  - ▶ **in realtà questo numero deve essere diviso per 4!**
- ▶ è un ben noto "bug" della versione "standard" del comando `time` (GNU). È dovuto al fatto che "time" converte erroneamente da "pages" a Kbytes anche se il dato è già in "Kbytes".
- ▶ Il valore corretto per il nostro eseguibile è dunque circa 4 Gbytes.

- ▶ se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto piu grande (circa 8 milioni). Cosa sta succedendo?

- ▶ se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto più grande (circa 8 milioni). Cosa sta succedendo?
- ▶ Un "page-fault" è un segnale generato dalla CPU, con conseguente reazione del sistema operativo, quando un programma tenta di accedere ad una pagina di memoria virtuale non presente, al momento, in memoria RAM.

- ▶ se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto più grande (circa 8 milioni). Cosa sta succedendo?
- ▶ Un "page-fault" è un segnale generato dalla CPU, con conseguente reazione del sistema operativo, quando un programma tenta di accedere ad una pagina di memoria virtuale non presente, al momento, in memoria RAM.
- ▶ La risposta del sistema operativo consiste nel caricare in memoria la pagina richiesta, facendo spazio spostando su disco altre parti non immediatamente necessarie.

- ▶ se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto più grande (circa 8 milioni). Cosa sta succedendo?
- ▶ Un "page-fault" è un segnale generato dalla CPU, con conseguente reazione del sistema operativo, quando un programma tenta di accedere ad una pagina di memoria virtuale non presente, al momento, in memoria RAM.
- ▶ La risposta del sistema operativo consiste nel caricare in memoria la pagina richiesta, facendo spazio spostando su disco altre parti non immediatamente necessarie.
- ▶ Operazione che richiede un gran dispendio di risorse e che rallenta l'esecuzione del nostro eseguibile.

Cambiando la struttura del programma (eliminando le allocazioni e deallocazioni durante l'esecuzione dello stesso) le cose migliorano drasticamente:

```
[lanucara@louis ~/CORSO2013]$ /usr/bin/time ./a.out<realloc.in
2.28user 0.38system 0:02.67elapsed 99%CPU (0avgtext+0avgdata 9378352maxresident)k
0inputs+0outputs (0major+3153minor)pagefaults 0swaps
```

Cambiando la struttura del programma (eliminando le allocazioni e deallocazioni durante l'esecuzione dello stesso) le cose migliorano drasticamente:

```
[lanucara@louis ~/CORSO2013]$ /usr/bin/time ./a.out<realloc.in
2.28user 0.38system 0:02.67elapsed 99%CPU (0avgtext+0avgdata 9378352maxresident)k
0inputs+0outputs (0major+3153minor)pagefaults 0swaps
```

e ora correttamente *System time* << *User time*.

Cambiando la struttura del programma (eliminando le allocazioni e deallocazioni durante l'esecuzione dello stesso) le cose migliorano drasticamente:

```
[lanucara@louis ~/CORSO2013]$ /usr/bin/time ./a.out<realloc.in
2.28user 0.38system 0:02.67elapsed 99%CPU (0avgtext+0avgdata 9378352maxresident)k
0inputs+0outputs (0major+3153minor)pagefaults 0swaps
```

e ora correttamente *System time*  $\ll$  *User time*.

time è uno strumento che ci fornisce informazioni utili in modo non intrusivo.

- ▶ L'informazione che ritorna dal comando `time` è utile ma non descrive dinamicamente (nel tempo) e con una buona approssimazione il "comportamento" della nostra applicazione.
- ▶ Inoltre, `time` non ci fornisce alcuna informazione dello stato della macchina su cui stiamo in esecuzione e se altri utenti stanno contendendo le nostre stesse risorse (cores, I/O, rete, etc).
- ▶ `Top` è un semplice comando Unix che ci fornisce queste e altre informazioni.
- ▶ Sintassi: `top [options...]`

```
top - 09:42:01 up 16 min,  6 users,  load average: 0.16, 0.23, 0.24
Tasks: 144 total,   3 running, 141 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.6%us,  2.1%sy,  0.0%ni, 95.2%id,  0.0%wa,  0.2%hi,  0.0%si,  0.0%st
Mem:   2071324k total,   678944k used, 1392380k free,   45808k buffers
Swap: 1052216k total,    0k used, 1052216k free,   324492k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
5778	root	20	0	69592	34m	11m	S	3	1.7	0:28.16	Xorg
6978	ruggiero	20	0	110m	24m	13m	R	2	1.2	0:04.04	gnome-terminal
7125	ruggiero	20	0	112m	43m	27m	R	1	2.1	0:08.62	ld-linux.so.2
1	root	20	0	3056	1900	576	S	0	0.1	0:01.42	init

```
...  
PR  -- Priority  
      The priority of the task.  
NI  -- Nice value  
      The nice value of the task.  A negative nice value means higher  
      priority, whereas a positive nice value means lower priority.  
VIRT -- Virtual Image (kb)  
      The total amount of virtual memory used by the task.  
...
```

- ▶ Funzione del **fortran90** che fornisce il tempo di CPU impiegato (vedi anche **data\_and\_time**, **system\_clock**)
- ▶ È ovviamente intrusiva
- ▶ È utile per il profiling di singoli blocchi (e.g. cicli do...)
- ▶ Non fidarsi troppo sotto il centesimo di secondo

```
real t1, t2
.....
CALL CPU_TIME(t1)
    istruzione 1
    istruzione 2
.....
CALL CPU_TIME(t2)
tempo = t2-t1
write, tempo, 's'
```

- ▶ Funzione **C** che fornisce il tempo
- ▶ È ovviamente intrusiva
- ▶ È utile per il profiling di singoli blocchi (cicli for...)
- ▶ includere la libreria `<time.h>`
- ▶ Non fidarsi sotto il millesimo di secondo

```
#include <time.h>
....
time1 = clock();
for (j = 0; j < (nn); j++) {....}
time2 = clock();
dub_time=(time2-time1)/(double) CLOCKS_PER_SEC;
```

- ▶ Funzione `MATLAB` che fornisce il tempo di CPU impiegato
- ▶ È ovviamente intrusiva
- ▶ È utile per il profiling di singoli blocchi (cicli do...)
- ▶ Non fidarsi sotto il millesimo di secondo

```
tic
```

```
k = 1:1:n;
```

```
    j = 1:1:n;
```

```
        i = 1:1:n;
```

```
            q1(i,j) = q1(i,j) + a(i,k)*b(k,j);
```

```
toc
```

```
tic
```

```
q2=a*b;
```

```
toc
```

- ▶ La funzione `gettimeofday` del C fornisce il tempo dalla mezzanotte in microsecondi
- ▶ Si può costruire una funzione di timing
- ▶ È multiplatforma...
- ▶ Risoluzione dell'ordine del microsecondo
- ▶ Attenzione alle misure a cavallo di mezzanotte.

Sintassi:

```
time1 = dwalltime00()  
do k=1,nd  
    . . . . .  
enddo  
time2 = dwalltime00()
```

## Definizione funzione: dwalltime.c

```
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>

double dwalltime00()
{
    double sec;
    struct timeval tv;
    gettimeofday(&tv, 0);
    sec = tv.tv_sec + tv.tv_usec/1000000.0;
    return sec;
}
```

- ▶ Il profiling permette di:
  - ▶ conoscere il tempo (assoluto e relativo) impiegato in ogni funzione o blocco computazionale
  - ▶ individuare i blocchi computazionali più pesanti
  - ▶ risalire a tutta la gerarchia delle chiamate
  - ▶ individuare funzioni poco efficienti e modificarle
- ▶ Strumenti:
  - ▶ strumentazione manuale tramite funzioni di timing
  - ▶ gprof (Unix/Linux)
  - ▶ Scalasca

- ▶ Profiling a livello di funzioni
  - ▶ descrive il flusso di esecuzione del codice
  - ▶ limitatamente intrusivo
  - ▶ le prestazioni misurate sono “di solito” realistiche
- ▶ Profiling a livello di singoli statement
  - ▶ descrive ogni singolo statement eseguito
  - ▶ molto intrusivo
  - ▶ le prestazioni misurate non sono indicative
  - ▶ permette di identificare costrutti eseguiti più del necessario

- ▶ Il profiling si basa sul campionamento (sampling) delle informazioni
  - ▶ Time Based Sampling (TBS): ad intervalli di tempo fissati viene individuato quale punto del codice è in esecuzione
    - ▶ bisogna instrumentare/ricompilare il codice
    - ▶ possibili errori di campionamento
  - ▶ Call Based Sampling: si contano le volte che si entra in una procedura ed il tempo speso al suo interno
    - ▶ bisogna instrumentare il codice
    - ▶ non si tiene conto degli argomenti passati alle procedure
  - ▶ Event Based Sampling (EBS): si contano gli eventi tramite appositi contatori hardware
    - ▶ non c'è necessità di instrumentare/ricompilare il codice
    - ▶ forte dipendenza dall'architettura usata

## Profilers

- Introduzione
- gprof**

- ▶ Compilando con l'opzione `-pg` si inseriscono nell'eseguibile chiamate ad una libreria (sia **F**ortran che **C**)
- ▶ Vengono contate le chiamate ed il tempo impiegato in ogni funzione (attenzione.....)
- ▶ Vengono registrate le dipendenze tra le varie funzioni
- ▶ Dopo l'esecuzione viene prodotto un file (*gmon.out*) che va elaborato con il comando `gprof`
- ▶ È una misura intrusiva: l'eseguibile prodotto può essere sensibilmente diverso
- ▶ Dà informazioni solo a livello di subroutine;
- ▶ Linkare staticamente (quando possibile).

```
ruggiero@laptop:~$ gcc -c -pg nome_sorgenteC.c
ruggiero@laptop:~$ gfortran -c -pg nome_sorgenteF.f
ruggiero@laptop:~$ gfortran -pg nome_sorgenteC.o \
                    nome_sorgenteF.o -o nome_eseguibile
ruggiero@laptop:~$ nome_eseguibile
ruggiero@laptop:~$ gprof nome_eseguibile >output
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
25.49	14.95	14.95	502002000	0.00	0.00	fcvx_
24.98	29.60	14.65	502002000	0.00	0.00	fcvy_
20.61	41.69	12.09	251001000	0.00	0.00	ccsl_
17.17	51.76	10.07	251001000	0.00	0.00	interpbl_
5.23	54.83	3.07	1000	0.00	0.04	sl_
2.20	56.12	1.29	1	1.29	8.77	maxfx_
2.17	57.39	1.27	1	1.27	58.65	MAIN__
2.13	58.64	1.25	1	1.25	8.57	maxfy_
0.02	58.65	0.01	1	0.01	0.01	u0_
...						

```
cumulative a running sum of the number of seconds accounted
seconds   for by this function and those listed above it.

self      the number of seconds accounted for by this
seconds   function alone. This is the major sort for this
          listing.

calls     the number of times this function was invoked, if
          this function is profiled, else blank.

self      the average number of milliseconds spent in this
ms/call   function per call, if this function is profiled,
          else blank.

total     the average number of milliseconds spent in this
ms/call   function and its descendents per call, if this
          function is profiled, else blank.
```

granularity: each sample hit covers 4 byte(s) for 0.02% of 58.65 seconds

```

index % time      self  children   called   name
-----
[1]    100.0      1.27   57.38     1/1      main [2]
      100.0      1.27   57.38     1        MAIN__ [1]
      3.07   36.96    1000/1000  sl_ [3]
      0.00   17.34     1/1      cfl_ [4]
      0.00    0.01     1/1      input_ [11]
      0.00    0.00    1000/1001  cc_ [13]
      0.00    0.00    1000/1000  output_ [14]
      0.00    0.00       2/2      dwelltime00_ [15]
-----
[2]    100.0      0.00   58.65
      1.27   57.38     1/1      main [2]
      1.27   57.38     1/1      MAIN__ [1]
-----
[3]    68.3       3.07   36.96    1000/1000  MAIN__ [1]
      3.07   36.96     1000      sl_ [3]
      12.09  0.00  251001000/251001000  ccs1_ [7]
      10.07  0.00  251001000/251001000  interpbl_ [8]
      7.47   0.00  251001000/502002000  fcvx_ [5]
      7.33   0.00  251001000/502002000  fcvy_ [6]
-----
[4]    29.6       0.00   17.34     1/1      MAIN__ [1]
      0.00   17.34     1/1      cfl_ [4]
      1.29   7.47     1/1      maxfx_ [9]
      1.25   7.33     1/1      maxfy_ [10]
      0.00   0.00     1/1      maxdhx_ [16]
      0.00   0.00     1/1      maxdhy_ [17]
-----
.....
    
```

```
For the line with the index number
% time This is the percentage of the 'total' time that was spent in
      this function and its children.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
         function by its children

called This is the number of times the function was called.

For the function's parents (that above the function)
self    the amount of time spent in the function itself when it
        was called from the parent

children the amount of time spent in the children of the function
        when it was called from the parent

called This is the number of times this parent called the
        function '/' the total number of times the function
        was called.

For the function's children (that below the function)
self    the amount of time spent in the children when it was called from the function

children the amount of time spent in the children of the children when it was called
        from the function

called This is the number of times the function called this child '/'
        the total number of times the child was called.
```

- ▶ **Attenzioni alle librerie matematiche:**
  - ▶ Possono non essere instrumentate;
  - ▶ Può essere risolto linkando staticamente;
- ▶ **Attenzione agli argomenti delle procedure:**
  - ▶ Si misurano le prestazioni complessive;
- ▶ Usare casi test significativi
- ▶ Tende ad essere intrusiva quando si usano molte funzioni brevi
  - ▶ verificare se nel profiling c'è la funzione `__count?`
  - ▶ verificare se il tempo riportato è uguale a quello totale
- ▶ Sui compilatori per Intel e AMD:
  - ▶ presente su **GNU, Intel, Portland**

```
<ruiggiero@borneo ~/TEMP/ORIG>xlf -O3 -pg separazioneICAO.f
<ruiggiero@borneo ~/TEMP/ORIG>time a.out
192.860u 0.050s 3:13.06 99.9% 51+2378k 0+0io 7pf+0w
<ruiggiero@borneo ~/TEMP/ORIG>xlf -O3 separazioneICAO.f
<ruiggiero@borneo ~/TEMP/ORIG>time a.out
118.370u 0.010s 1:58.36 100.0% 43+239k 0+0io 0pf+0w

<ruiggiero@borneo ~/TEMP/ORIG>xlf -O5 -pg separazioneICAO.f
<ruiggiero@borneo ~/TEMP/ORIG>time a.out
157.120u 0.040s 2:37.20 99.9% 48+2370k 0+0io 1pf+0w
<ruiggiero@borneo ~/TEMP/ORIG>xlf -O5 separazioneICAO.f
<ruiggiero@borneo ~/TEMP/ORIG>time a.out
90.340u 0.010s 1:30.35 100.0% 39+239k 0+0io 0pf+0w
```

# SCAIprof: e le librerie matematiche?

## (linkare statico)



Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
68.13	121.61	121.61				sin
8.44	136.67	15.06	502002000	0.00	0.00	fcvx_
6.97	149.11	12.44	502002000	0.00	0.00	fcvy_
6.23	160.24	11.13	251001000	0.00	0.00	ccsl_
5.46	169.98	9.74	251001000	0.00	0.00	interpbl_
1.71	173.04	3.06	1000	0.00	0.04	sl_
0.82	174.50	1.46	1	1.46	7.68	maxfy_
0.70	175.75	1.25	1	1.25	8.78	maxfx_
0.69	176.99	1.24	1	1.24	55.45	MAIN_
0.12	177.21	0.22				vfprintf
0.10	177.38	0.17				__printf_fp
0.08	177.52	0.14				__mpn_mul_1
0.07	177.64	0.12				fd_alloc_w_at
0.06	177.75	0.11				_IO_default_xsputn
0.05	177.84	0.09				mempcpy
0.04	177.91	0.07				__write_nocancel
0.03	177.97	0.06				__vsnprintf_chk
0.03	178.02	0.05	1001	0.00	0.00	cc_
0.02	178.06	0.04				__strtol_l_internal
0.02	178.10	0.04				mempcpy
0.02	178.13	0.03				_IO_str_init_static_internal
0.02	178.16	0.03				_gfortran_transfer_real
0.02	178.19	0.03				_gfortrani_next_format
0.02	178.22	0.03				formatted_transfer
0.02	178.25	0.03				formatted_transfer_scalar
0.02	178.28	0.03				output_float

- ▶ `gprof` fornisce anche un profiling per linea
  - ▶ utile per vedere quali linee vengono maggiormente accedute
  - ▶ utile per capire quali istruzioni sono più lente delle altre
  - ▶ attenzione è intrusivo
- ▶ Utilizzo:
  - ▶ compilare eseguibile con l'opzione `-g`
  - ▶ invocare `gprof` con le opzioni:
    - `-l` `---` per il profiling line by line
    - `-l -A -x` `---` per il listing del codice

## Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
69.00	123.21	123.21				sin
6.28	134.43	11.21				fcvx (FCVX.F90:28@805e8ba)
5.29	143.88	9.46				fcvy (FCVY.F90:28@805ca92)
3.02	149.27	5.39				ccsl (CCSL.F90:29@805ea53)
2.09	153.00	3.73				interpbl (INTERBL.F90:18@804d578)
1.37	155.45	2.45				interpbl (INTERBL.F90:18@804d5a7)
1.16	157.52	2.07				interpbl (INTERBL.F90:12@804d4fe)
1.15	159.57	2.05				ccsl (CCSL.F90:30@805ea92)
1.05	161.45	1.88	502002000	3.75	3.75	fcvx_ (FCVX.F90:8@805e420)
0.73	162.76	1.31				ccsl (CCSL.F90:29@805ea8d)

## Call graph

granularity: each sample hit covers 4 byte(s) for 0.01% of 178.57 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	69.0	123.21	0.00		sin [1]
		0.94	0.00	251001000/502002000	sl (SL.F90:26@804cae5) [10]
		0.94	0.00	251001000/502002000	maxfx (MAXFX.F90:17@8057ac6) [11]
[9]	1.1	1.88	0.00	502002000	fcvx_ (FCVX.F90:8@805e420) [9]
		0.28	0.00	251001000/502002000	sl (SL.F90:27@804cb56) [19]
		0.28	0.00	251001000/502002000	maxfy (MAXFY.F90:17@804d876) [17]
[21]	0.3	0.56	0.00	502002000	fcvy_ (FCVY.F90:8@805ca50) [21]
.....					

```
502002000 ->      FUNCTION FCVX(a,b)
                  USE DIM2,ONLY:ss,cx,l,dt,pi
                  IMPLICIT NONE
                  REAL::FCVX,a,b
502002000 ->      FCVX=cx*(SIN(pi*a)**2 * SIN(2*pi*b))*SIN(pi*l*dt)
                  END FUNCTION FCVX
```

#### Top 10 Lines:

Line	Count
8	502002000

#### Execution Summary:

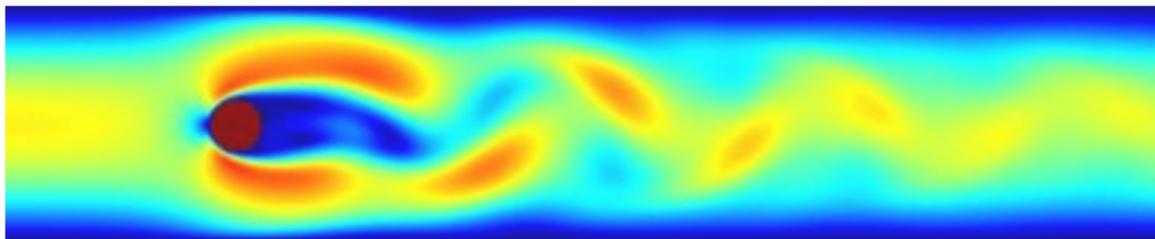
2	Executable lines in this file
2	Lines executed
100.00	Percent of the file executed
502002000	Total number of line executions
251001000.00	Average executions per line

- ▶ Il tool GNU **gcov** fornisce un profiling per linea
  - ▶ utile per vedere quali linee vengono maggiormente accedute
  - ▶ utile per capire quali istruzioni sono più lente delle altre
  - ▶ attenzione è intrusivo
- ▶ Utilizzo:
  - ▶ compilare con l'opzione  
`-fprofile-arcs -ftest-coverage`
  - ▶ eseguire il programma
  - ▶ invocare:  
`gcov codice\_sorgente`
- ▶ Le informazioni saranno raccolte nei file **\*.gcov**

- ▶ Benchmarking e Profiling di un codice cinetico (Lattice Boltzmann)
  - ▶ Scaricabile gratuitamente da <http://wiki.palabos.org/numerics:codes> nelle versioni C, C++, Fortran90, Matlab, etc.

`eser_gprof`

- ▶ Flusso attraverso un cilindro



- Come scala il codice al variare delle dimensioni?

Dimensioni	250*50	500*100	1000*200

- Quali sono le quattro subroutine più impegnative?

Subroutine	Tempo	%	Num. chiamate

- ▶ usare più casi test cercando di attivare tutte le parti del codice
- ▶ scegliere casi test il più possibile "realistici"
- ▶ usare casi test caratterizzati da diverse "dimensioni" del problema
- ▶ attenzione alla fase di input/output
- ▶ usare più strumenti di Profiling (magari raffinando l'analisi iniziale)
- ▶ usare, se possibile, architetture differenti.