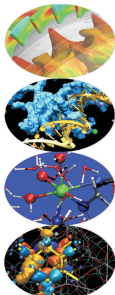


# HPC Scientific programming: tools and techniques

P. Lanucara V. Ruggiero  
F. Salvatore

CINECA Roma - SCAI Department

Roma, 9-11 April 2013





## 9 aprile 2013

9.30-10.30 Architetture

10.30-13.00 La cache ed il sistema di memoria + esercitazioni

14.00-15.00 Pipeline + esercitazioni

15.00-17.00 Profilers + esercitazioni

## 10 aprile 2013

9.30-13.00 Compilatori+esercitazioni

14.00-15.30 Librerie + esercitazioni

15.00-17.00 Floating-point +esercitazioni

## 11 aprile 2013

9.30-11.00 Makefile + esercitazioni

11.00-13.00 Debugging+esercitazioni

14.00-17.00 Debugging+esercitazioni



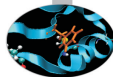
Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers



## Prodotto matrice-matrice (misurato in secondi)

Precisione	singola	doppia
Loop incorretto	7500	7300
senza ottimizzazione	206	246
con ottimizzazione (-fast)	84	181
codice ottimizzato	23	44
Libreria ACML (seriale)	6.7	13.2
Libreria ACML (2 threads)	3.3	6.7
Libreria ACML (4 threads)	1.7	3.5
Libreria ACML (8 threads)	0.9	1.8
Pgi accelerator	3	5
CUBLAS	1.6	3.2



- ▶ Completare il loop principale del codice e verificare le prestazioni ottenute
- ▶ Usando il Fortran e/o il C
  - ▶ Che prestazioni sono state ottenute?
  - ▶ C'è differenza tra Fortran e C?
  - ▶ Cambiano le prestazioni cambiando compilatore?
  - ▶ E le opzioni di compilazione?
  - ▶ Cambiano le prestazioni se cambio l'ordine dei loop?
  - ▶ Posso riscrivere il loop in maniera efficiente?



- ▶ Fortran o C
- ▶ Prodotto riga per colonna  $C_{i,j} = A_{i,k} B_{k,j}$
- ▶ Temporizzazione:
  - ▶ Fortran: `date_and_time` (> 0.001")
  - ▶ C: `clock` (>0.05")
- ▶ Per matrici quadrate di dimensione  $n$ 
  - ▶ Memoria richiesta (doppia precisione)  $\approx (3 * n * n) * 8$
  - ▶ Operazioni totali  $\approx 2 * n * n * n$ 
    - ▶ Bisogna accedere a  $n$  elementi delle due matrici origine per ogni elemento della matrice destinazione
    - ▶  $n$  prodotti ed  $n$  somme per ogni elemento della matrice destinazione
  - ▶ Flops totali  $= 2 * n^3 / \text{tempo}$
- ▶ Verificare sempre i risultati :-)



- ▶ **Stimare il numero di operazioni necessarie per l'esecuzione  $N_{Flop}$** 
  - ▶ 1 FLOP equivale ad un'operazione di somma o moltiplicazione floating-point
  - ▶ Operazioni più complicate (divisione, radice quadrata, funzioni trigonometriche) vengono eseguite in modo più complesso e più lento
- ▶ **Stimare il tempo necessario per l'esecuzione  $T_{es}$**
- ▶ Le prestazioni sono misurate contando il numero di operazioni floating-point eseguite per unità di tempo:

$$Perf = \frac{N_{Flop}}{T_{es}}$$

- ▶ l'unità di misura minima è 1 Floating-pointing Operation per secondo ( FLOPS)
- ▶ Normalmente si usano i multipli:
  - ▶ 1 MFLOPS=  $10^6$  FLOPS
  - ▶ 1 GFLOPS=  $10^9$  FLOPS
  - ▶ 1 TFLOPS=  $10^{12}$  FLOPS



```
cp /afs/icaspur.it/project/open/space/codici.tar /scratch
cd /scratch
tar xvf codici.tar
```

- ▶ **Struttura**
  - ▶ src/eser\_?/fortran
  - ▶ src/eser\_?/c





- Andare nella directory `src/eser_1`
- Completare il loop principale (prodotto righe per colonne) per il codice in Fortran(`mm.f90`) o il codice in C(`mm.c`)
- Misurare i tempi per eseguire il prodotto matrice-matrice per  $N=1024$

Linguaggio	tempo	Mflops
Fortran		
C		



gauss/8.0

gcc/4.5.2

gcc/4.6.0

grace/5.1.22

intel/12.0

maple/10

matlab/r2010b

matlab/r2011a

nvidia/4.2

pgi/11.8

pgi/12.5

tecplot/360

totalview/8.4.1-7



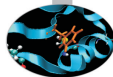
- ▶ Per compilare
  - ▶ make
- ▶ Per pulire
  - ▶ make clean
- ▶ Per cambiare opzioni
  - ▶ make "FC=ifort"
  - ▶ make "CC=icc"
  - ▶ make "OPT=fast"
- ▶ Per compilare in singola precisione
  - ▶ make "FC=ifort -DSINGLEPRECISION"
- ▶ Per compilare in doppia precisione
  - ▶ make "FC=ifort"

# cat /proc/cpuinfo



```

processor           : 0
vendor_id           : GenuineIntel
cpu family          : 6
model               : 37
model name          : Intel(R) Core(TM) i3 CPU           M 330    @ 2.13GHz
stepping            : 2
cpu MHz             : 933.000
cache size          : 3072 KB
physical id         : 0
siblings            : 4
core id             : 0
cpu cores           : 2
apicid              : 0
initial apicid      : 0
fpu                 : yes
fpu_exception       : yes
cpuid level         : 11
wp                  : yes
wp                  : yes
flags               : fpu vme de pse tsc msr pae mce cx8
bogomips            : 4256.27
clflush size        : 64
cache_alignment     : 64
address sizes       : 36 bits physical, 48 bits virtual
...
  
```



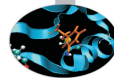
```

Architecture:                x86_64
CPU op-mode(s):              32-bit, 64-bit
Byte Order:                  Little Endian
CPU(s):                       4
On-line CPU(s) list:         0-3
Thread(s) per core:          2
Core(s) per socket:          2
CPU socket(s):               1
NUMA node(s):                1
Vendor ID:                   GenuineIntel
CPU family:                   6
Model:                        37
Stepping:                     2
CPU MHz:                      933.000
BogoMIPS:                     4255.78
Virtualization:              VT-x
L1d cache:                   32K
L1i cache:                   32K
L2 cache:                    256K
L3 cache:                    3072K
NUMA node0 CPU(s):           0-3
  
```

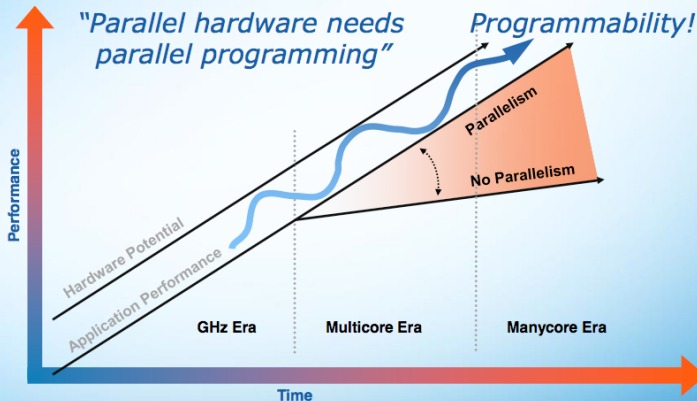


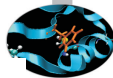
Qualche considerazione sui risultati ottenuti?

# Qual è il margine di manovra?



## Motivation: Performance





Problem

Solution  
method

Algorithms

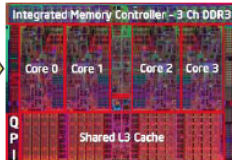
Programming

Source code

Compiling

Compiled and optimized code

Execution



Result

Output

```
#include <stdio.h>
#define N 1000
main(int argc, char** argv) {
    int A[N], B[N], C[N];
    int i;

    for (i=0; i<N; i++) {
        B[i] = i;
        C[i] = N-i;
    }

    /* Add B and C */
    for (i=0; i<N; i++) {
        A[i] = B[i]+C[i];
    }
}
```

```
.globl main
.type main,@function
main:
    pushl %ebp
    movl %ebp,%esp
    subl $12004,%esp
    pushl %edi
    pushl %esi
    nopl
    movl $0,-12004(%ebp)

.L1:
    cmpl $999,-12004(%ebp)
    jle .L4
    jmp .L3
    .align 4

.L2:
    movl -12004(%ebp),%eax
    movl %eax,%edx
    leal 0(%edx,4),%eax
    leal -4000(%ebp),%edx
    movl -12004(%ebp),%ecx
    movl %ecx,%ebx
    leal 0(%ebx,4),%ecx
    leal -4000(%ebp),%ebx
    movl -12004(%ebp),%eax
    movl %eax,%edi
    leal 0(%edi,4),%edi
    leal -12000(%ebp),%edx
    movl (%edx,%edi),%ecx
    imull (%eax,%edi),%ecx
    movl %ecx,(%eax,%edx)

.L4:
    incl -12004(%ebp)
    jmp .L2
    .align 4

.L3:
    leal -12016(%ebp),%esp
    popl %ebx

.L5:
    .size main,.L5f-main
    .ident "GCC: (GNU) 3.8.1"
```

Hierarchical code: Plummer model

nbody	dtime	eps	thetas	usequad	dtime	tstop
1024	0.03126	0.0250	1.00	false	0.2500	2.0000
tnow	TeU	T/U	mtot	nbgw	ncgw	cpuTime
0.000	-0.2627	-0.4943	203185	84	114	0.00
cm pos	0.0000	-0.0000	0.0000			
cm vel	-0.0000	0.0000	0.0000			
am vec	0.0037	0.0136	-0.0222			
tnow	TeU	T/U	mtot	nbgw	ncgw	cpuTime
0.031	-0.2627	-0.4940	203260	81	116	0.01
cm pos	0.0000	-0.0000	0.0000			
cm vel	0.0000	-0.0000	0.0000			
am vec	0.0037	0.0136	-0.0222			

Time is: 9.6 seconds





- ▶ Fondamentale è la scelta dell'algoritmo
  - ▶ algoritmo efficiente → buone prestazioni
  - ▶ algoritmo inefficiente → cattive prestazioni
- ▶ Se l'algoritmo non è efficiente non ha senso tutto il discorso sulle prestazioni
- ▶ Regola d'oro
  - ▶ Curare quanto possibile la scelta dell'algoritmo prima della codifica, altrimenti c'è la possibilità di dover riscrivere!!!



Introduzione

**Architetture**

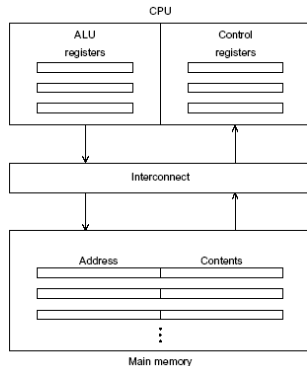
La cache e il sistema di memoria

Pipeline

Profilers



- ▶ **Central processing unit (CPU)**
  - ▶ Unità Logica Aritmetica (esegue le istruzioni)
  - ▶ Unità di controllo
  - ▶ Registri (memoria veloce)
- ▶ **Interconnessione CPU RAM (Bus)**
- ▶ **Random Access Memory (RAM)**
  - ▶ Indirizzo per accedere alla locazione di memoria
  - ▶ Contenuto della locazione (istruzione, dato)



# Architettura di Von Neuman



- ▶ I dati sono trasferiti dalla memoria alla CPU (fetch o read)
- ▶ I dati sono trasferiti dalla CPU alla memoria (written to memory o stored)
- ▶ La separazione di CPU e memoria è conosciuta come la «von Neumann bottleneck» perché è il bus di interconnessione che determina a che velocità si può accedere ai dati e alle istruzioni.
- ▶ Le moderne CPU sono in grado di eseguire istruzioni almeno cento volte più velocemente rispetto al tempo richiesto per recuperare (fetch) i dati nella RAM



Il «von Neumann bottleneck» è stato affrontato seguendo tre percorsi paralleli:

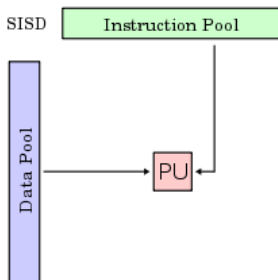
- ▶ **Caching**  
Memorie molto veloci presenti sul chip del processore.  
Esistono cache di primo, secondo e terzo livello.
- ▶ **Virtual memory**  
Sistema sviluppato per fare in modo che la RAM funzioni come una cache per lo storage di grosse moli di dati.
- ▶ **Instruction level parallelism**  
Tecnica utilizzata per avere più unità funzionali nella CPU che eseguono istruzioni in parallelo (pipelining ,multiple issue)



- ▶ Racchiude le architetture dei computer in base alla molteplicità dell'hardware usato per manipolare lo stream di istruzioni e dati
  - ▶ **SISD**:single instruction, single data. Corrisponde alla classica architettura di von Neumann, sistema scalare monoprocesso.
  - ▶ **SIMD**:single instruction, multiple data. Architetture vettoriali, processori vettoriali, GPU.
  - ▶ **MISD**:multiple instruction, single data. Non esistono soluzioni hardware che sfruttino questa architettura.
  - ▶ **MIMD**:multiple instruction, multiple data. Più processori/cores interpretano istruzioni diverse e operano su dati diversi.
- ▶ Le moderne soluzioni di calcolo sono date da una combinazione delle categorie previste da Flynn.



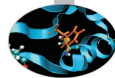
- Il classico sistema di von Neumann. Calcolatori con una sola unità esecutiva ed una sola memoria. Il singolo processore obbedisce ad un singolo flusso di istruzioni (programma sequenziale) ed esegue queste istruzioni ogni volta su un singolo flusso di dati.
- I limiti di prestazione di questa architettura vengono ovviati aumentando il bus dati ed i livelli di memoria ed introducendo un parallelismo attraverso le tecniche di pipelining e multiple issue.





- ▶ La stessa istruzione viene eseguita in parallelo su dati differenti.
  - ▶ modello computazionale generalmente sincrono
- ▶ Processori vettoriali
  - ▶ molte ALU
  - ▶ registri vettoriali
  - ▶ Unità Load/Store vettoriali
  - ▶ Istruzioni vettoriali
  - ▶ Memoria interleaved
  - ▶ OpenMP, MPI
- ▶ Graphical Processing Unit
  - ▶ GPU completamente programmabili
  - ▶ molte ALU
  - ▶ molte unità Load/Store
  - ▶ molte SFU
  - ▶ migliaia di threads lavorano in parallelo
  - ▶ CUDA





- ▶ Molteplici streams di istruzioni eseguiti simultaneamente su molteplici streams di dati
  - ▶ modello computazionale asincrono
- ▶ Cluster
  - ▶ molti nodi di calcolo (centinaia/migliaia)
  - ▶ più processori multicore per nodo
  - ▶ RAM condivisa sul nodo
  - ▶ RAM distribuita fra i nodi
  - ▶ livelli di memoria gerarchici
  - ▶ OpenMP, MPI, MPI+OpenMP



## IBM-BlueGene /Q

Architecture: 10 BGQ Frame with 2 MidPlanes each

Front-end Nodes OS: Red-Hat EL 6.2

Compute Node Kernel: lightweight Linux-like kernel

Processor Type: IBM PowerA2, 16 cores, 1.6 GHz

Computing Nodes: 10.240

Computing Cores: 163.840

RAM: 16GB / node

Internal Network: Network interface

with 11 links ->5D Torus

Disk Space: more than 2PB of scratch space

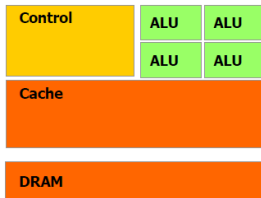
Peak Performance: 2.1 PFlop/s



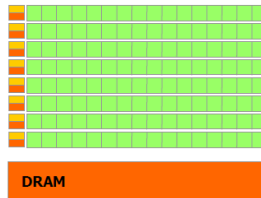
- ▶ Soluzioni ibride CPU multi-core + GPU many-core:
  - ▶ ogni nodo di calcolo è dotato di processori multicore e schede grafiche con processori dedicati per il GPU computing
  - ▶ notevole potenza di calcolo teorica sul singolo nodo
  - ▶ ulteriore strato di memoria dato dalla memoria delle GPU
  - ▶ OpenMP, MPI, CUDA e soluzioni ibride MPI+OpenMP, MPI+CUDA, OpenMP+CUDA, OpenMP+MPI+CUDA



- ▶ Le CPU sono processori general purpose in grado di risolvere qualsiasi algoritmo
  - ▶ threads in grado di gestire qualsiasi operazione ma pesanti, al massimo 1 thread per core computazionale.
- ▶ Le GPU sono processori specializzati per problemi che possono essere classificati come «intense data-parallel computations»
  - ▶ controllo di flusso molto semplice (control unit ridotta)
  - ▶ molti threads leggeri che lavorano in parallelo



**CPU**

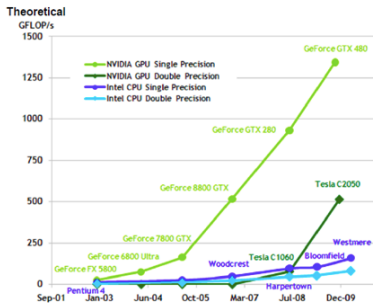


**GPU**

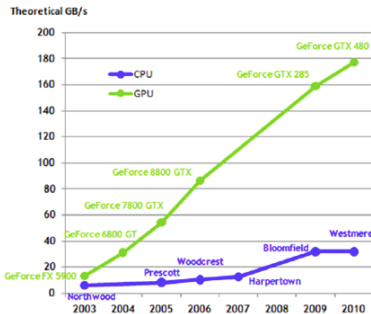
# Multicore vs Manycore



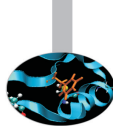
- Una nuova direzione di sviluppo per l'architettura dei microprocessori:
  - incrementare la potenza di calcolo complessiva tramite l'aumento del numero di unità di elaborazione piuttosto che della loro potenza
  - la potenza di calcolo e la larghezza di banda delle GPU ha sorpassato quella delle CPU di un fattore 10.



Numero di operazioni in virgola mobile  
al secondo per la CPU e la GPU



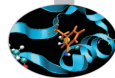
Larghezza di banda della memoria



- ▶ Coprocessore Intel Xeon Phi
  - ▶ Basato su architettura Intel Many Integrated Core (MIC)
  - ▶ 60 core/1,053 GHz/240 thread
  - ▶ 8 GB di memoria e larghezza di banda di 320 GB/s
  - ▶ 1 TFLOPS di prestazioni di picco a doppia precisione
  - ▶ Istruzioni SIMD a 512 bit
  - ▶ Approcci tradizionali come MPI, OpenMP



- ▶ La velocità con la quale è possibile trasferire i dati tra la memoria e il processore
- ▶ Si misura in numero di bytes che si possono trasferire al secondo (Mb/s, Gb/s, etc..)
- ▶  $A = B * C$ 
  - ▶ leggere dalla memoria il dato B
  - ▶ leggere dalla memoria il dato C
  - ▶ calcolare il prodotto  $B * C$
  - ▶ salvare il risultato in memoria, nella posizione della variabile A
- ▶ 1 operazione floating-point  $\rightarrow$  3 accessi in memoria

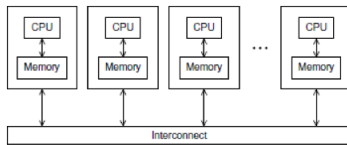
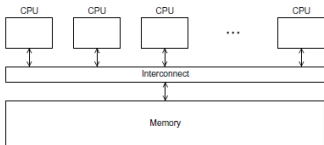


- ▶ Benchmark per la misura della bandwidth da e per la CPU
- ▶ Misura il tempo per
  - ▶ Copia  $a \rightarrow c$  (copy)
  - ▶ Copia  $a*b \rightarrow c$  (scale)
  - ▶ Somma  $a+b \rightarrow c$  (add)
  - ▶ Somma  $a+b*c \rightarrow d$  (triad)
- ▶ Misura della massima bandwidth
- ▶ <http://www.cs.virginia.edu/stream/ref.html>

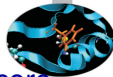




- ▶ Le architetture MIMD classiche e quelle miste CPU GPU sono suddivise in due categorie
  - ▶ Sistemi a memoria condivisa dove ogni singolo core ha accesso a tutta la memoria
  - ▶ Sistemi a memoria distribuita dove ogni processore ha la sua memoria privata e comunica con gli altri tramite scambio di messaggi.

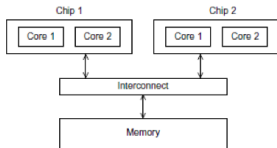


- ▶ I moderni sistemi multicore hanno memoria condivisa sul nodo e distribuita fra i nodi.

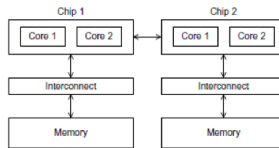


Nelle architetture a memoria condivisa con processori multicore vi sono generalmente due tipologie di accesso alla memoria principale

- **Uniform Memory Access** dove tutti i cores sono direttamente collegati con la stessa priorità alla memoria principale tramite il sistema di interconnessione
- **Non Uniform Memory Access** dove ogni processore multicore può avere accesso privilegiato ad un blocco di memoria e accesso secondario agli altri blocchi.



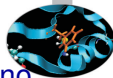
UMA



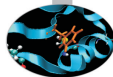
NUMA



- ▶ Problemi principali:
  - ▶ se un processo o thread viene trasferito da una CPU ad un'altra, va perso tutto il lavoro speso per usare bene la cache
  - ▶ macchine UMA: processi "memory intensive" su più CPU contendono per il bus, rallentandosi a vicenda
  - ▶ macchine NUMA: codice eseguito da una CPU con i dati nella memoria di un'altra portano a rallentamenti
- ▶ Soluzioni:
  - ▶ binding di processi o thread alle CPU
  - ▶ memory affinity
  - ▶ su AIX, variabile di ambiente `MEMORY_AFFINITY`
  - ▶ su kernel che lo supportano: `numactl`



- ▶ Tutti i sistemi caratterizzati da elevate potenze di calcolo sono composti da diversi nodi, a memoria condivisa sul singolo nodo e distribuita fra i nodi
  - ▶ i nodi sono collegati fra di loro da topologie di interconnessione più o meno complesse e costose
- ▶ Le reti di interconnessione commerciali maggiormente utilizzate sono
  - ▶ Gigabit Ethernet : la più diffusa, basso costo, basse prestazioni
  - ▶ Infiniband : molto diffusa, elevate prestazioni, costo elevato (50% del costo di un cluster)
  - ▶ Myrinet : sempre meno diffusa dopo l'avvento di infiniband, vi sono comunque ancora sistemi HPC molto importanti che la utilizzano
- ▶ Reti di interconnessione di nicchia
  - ▶ Quadrics
  - ▶ Cray



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	<b>Titan</b> - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
2	DOE/NNSA/LLNL United States	<b>Sequoia</b> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	16324.8	20132.7	7890
3	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
9	CINECA Italy	<b>Fermi</b> - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	163840	1725.5	2097.2	822



Introduzione

Architetture

La cache e il sistema di memoria

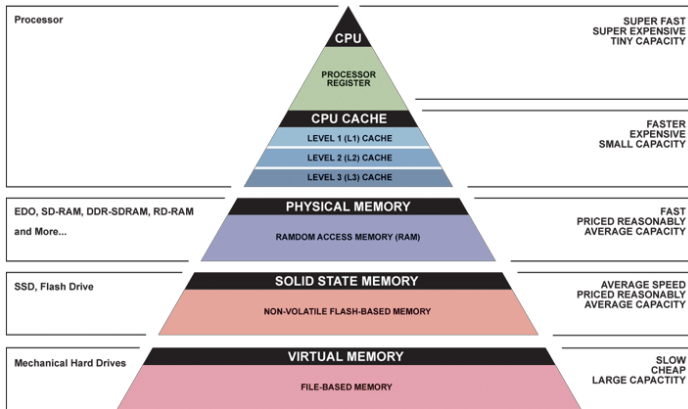
Pipeline

Profilers



- ▶ Capacità di calcolo delle CPU  $2\times$  ogni 18 mesi
- ▶ Velocità di accesso alla RAM  $2\times$  ogni 120 mesi
- ▶ Inutile ridurre numero e costo delle operazioni se i dati non arrivano dalla memoria
- ▶ Soluzione: memorie intermedie veloci
- ▶ Il sistema di memoria è una struttura profondamente gerarchica
- ▶ La gerarchia è trasparente all'applicazione, i suoi effetti no

# La gerarchia di memoria



▲ Simplified Computer Memory Hierarchy  
 Illustration: Ryan J. Leng



Perché questa gerarchia?

La Cache





Perché questa gerarchia?

Non servono tutti i dati disponibili subito



Perché questa gerarchia?

Non servono tutti i dati disponibili subito

La soluzione?



Perché questa gerarchia?

Non servono tutti i dati disponibili subito

La soluzione?

- ▶ La cache è composta di uno (o più livelli) di memoria intermedia, abbastanza veloce ma piccola (kB ÷ MB)
- ▶ Principio fondamentale: si lavora sempre su un sottoinsieme ristretto dei dati
  - ▶ dati che servono → nella memoria ad accesso veloce
  - ▶ dati che (per ora) non servono → nei livelli più lenti
- ▶ Regola del pollice:
  - ▶ il 10% del codice impiega il 90% del tempo
- ▶ Limitazioni
  - ▶ accesso casuale senza riutilizzo
  - ▶ non è mai abbastanza grande ...
  - ▶ più è veloce, più scalda e ... costa → gerarchia di livelli intermedi.



- ▶ La CPU accede al (più alto) livello di cache:
- ▶ Il controllore della cache determina se l'elemento richiesto è effettivamente presente in cache:
  - ▶ **Si**: trasferimento fra cache e CPU
  - ▶ **No**: Carica il nuovo dato in cache; se la cache è piena, applica la politica di rimpiazzamento per caricare il nuovo dato al posto di uno di quelli esistenti
- ▶ Lo spostamento di dati tra memoria principale e cache non avviene per parole singole ma per **blocchi** denominati **linee di cache**
- ▶ **blocco** = minima quantità d'informazione trasferibile fra due livelli di memoria (fra due livelli di cache, o fra RAM e cache)



- ▶ Località spaziale dei dati
  - ▶ vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).



- ▶ Località spaziale dei dati
  - ▶ vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - ▶ Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.



- ▶ Località spaziale dei dati
  - ▶ vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - ▶ Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.
  
- ▶ Località temporale dei dati
  - ▶ vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)





- ▶ Località spaziale dei dati
  - ▶ vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - ▶ Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.
  
- ▶ Località temporale dei dati
  - ▶ vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)
  - ▶ viene sfruttata decidendo di rimpiazzare il blocco utilizzato meno recentemente.

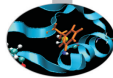
# Rimpiazzamento: principi di località



- ▶ Località spaziale dei dati
  - ▶ vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - ▶ Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.
- ▶ Località temporale dei dati
  - ▶ vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)
  - ▶ viene sfruttata decidendo di rimpiazzare il blocco utilizzato meno recentemente.

Dato richiesto dalla CPU viene mantenuto in cache insieme a celle di memoria contigue il più a lungo possibile.

# Cache: qualche definizione



- ▶ **Hit**: l'elemento richiesto dalla CPU è presente in cache
- ▶ **Miss**: l'elemento richiesto dalla CPU non è presente in cache
- ▶ **Hit rate**: frazione degli accessi a memoria ricompensati da uno hit (cifra di merito per le prestazioni della cache)
- ▶ **Miss rate**: frazione degli accessi a memoria cui risponde un miss (miss rate =  $1 - \text{hit rate}$ )
- ▶ **Hit time**: tempo di accesso alla cache in caso di successo (include il tempo per determinare se l'accesso si conclude con hit o miss)
- ▶ **Miss penalty**: tempo necessario per sostituire un blocco in cache con un altro blocco dalla memoria di livello inferiore (si usa un valore medio)
- ▶ **Miss time**: = miss penalty + hit time, tempo necessario per ottenere l'elemento richiesto in caso di miss.

# Cache: qualche stima quantitativa



Livello	costo di accesso
L1	1 ciclo di clock
L2	7 cicli di clock
RAM	36 cicli di clock

- ▶ 100 accessi con 100% cache hit:  $\rightarrow t=100$
- ▶ 100 accessi con 5% cache miss in L1:  $\rightarrow t=130$
- ▶ 100 accessi con 10% cache miss:in L1  $\rightarrow t=160$
- ▶ 100 accessi con 10% cache miss:in L2  $\rightarrow t=450$
- ▶ 100 accessi con 100% cache miss:in L2  $\rightarrow t=3600$

# Cache miss in tutti i livelli



1. cerco due dati, A e B
2. cerco A nella cache di primo livello (L1)  $O(1)$  cicli
3. cerco A nella cache di secondo livello (L2)  $O(10)$  cicli
4. copio A dalla RAM alla L2 alla L1 ai registri  $O(10)$  cicli
5. cerco B nella cache di primo livello (L1)  $O(1)$  cicli
6. cerco B nella cache di secondo livello (L2)  $O(10)$  cicli
7. copio B dalla RAM alla L2 alla L1 ai registri  $O(10)$  cicli
8. eseguo l'operazione richiesta  
 $O(100)$  cicli di overhead!!!

# Cache hit in tutti i livelli



- ▶ cerco i due dati, A e B
- ▶ cerco A nella cache di primo livello(L1) O(1) cicli
- ▶ cerco B nella cache di primo livello(L1) O(1) cicli
- ▶ eseguo l'operazione richiesta

O(1) cicli di overhead



- ▶ **Dynamic RAM (DRAM) memoria centrale**
  - ▶ Una cella di memoria è composta da 1 transistor
  - ▶ Economica
  - ▶ Ha bisogno di essere "ricaricata"
    - ▶ I dati non sono accessibili nella fase di ricarica
  
- ▶ **Static RAM (SRAM) memoria cache**
  - ▶ Una cella di memoria è composta da 6-7 transistor
  - ▶ Costosa
  - ▶ Non ha bisogno di "ricarica"
    - ▶ I dati sono sempre accessibili



```
float sum = 0.0f;  
for (i = 0; i < n; i++)  
    sum = sum + x[i]*y[i];
```

- ▶ Ad ogni iterazione viene eseguita una somma ed una moltiplicazione floating-point
- ▶ Il numero di operazioni è  $2 \times n$



# Tempo di esecuzione $T_{es}$



- ▶  $T_{es} = N_{flop} * t_{flop}$
- ▶  $N_{flop} \rightarrow \text{Algoritmo}$

# Tempo di esecuzione $T_{es}$



- ▶  $T_{es} = N_{flop} * t_{flop}$
- ▶  $N_{flop} \rightarrow$  Algoritmo
- ▶  $t_{flop} \rightarrow$  Hardware

# Tempo di esecuzione $T_{es}$



- ▶  $T_{es} = N_{flop} * t_{flop}$
- ▶  $N_{flop} \rightarrow$  Algoritmo
- ▶  $t_{flop} \rightarrow$  Hardware
- ▶ Considera solamente il tempo di esecuzione in memoria
- ▶ Trascuro qualcosa?

# Tempo di esecuzione $T_{es}$



- ▶  $T_{es} = N_{flop} * t_{flop}$
- ▶  $N_{flop} \rightarrow$  Algoritmo
- ▶  $t_{flop} \rightarrow$  Hardware
- ▶ Considera solamente il tempo di esecuzione in memoria
- ▶ Trascuro qualcosa?
- ▶  $t_{mem}$  il tempo di accesso in memoria.



- ▶  $T_{es} = N_{flop} * t_{flop} + N_{mem} * t_{mem}$
- ▶  $t_{mem} \rightarrow$  Hardware
- ▶ Qual è l'impatto di  $N_{mem}$  sulle performance?



- ▶  $Perf = \frac{N_{Flop}}{T_{es}}$
- ▶ per  $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- ▶ per  $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- ▶ Fattore di decadimento delle performance
- ▶  $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- ▶ Come posso avvicinarmi alle prestazioni di picco della macchina?



- ▶  $Perf = \frac{N_{Flop}}{T_{es}}$
- ▶ per  $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- ▶ per  $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- ▶ Fattore di decadimento delle performance
- ▶  $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- ▶ Come posso avvicinarmi alle prestazioni di picco della macchina?
- ▶ **Riducendo gli accessi alla memoria.**



- ▶ Prodotto di matrici in doppia precisione 512X512
- ▶ MFlops misurati su louis (Intel(R) Xeon(R) CPU X5660 2.80GHz)
- ▶ compilatore gfortran con ottimizzazione -O0

Ordine indici	Fortran	C
i,j,k	109	128
i,k,j	90	177
j,k,i	173	96
j,i,k	110	127
k,j,i	172	96
k,i,j	90	177

L'ordine di accesso più efficiente dipende dalla disposizione dei dati in memoria e non dall'astrazione operata dal linguaggio.

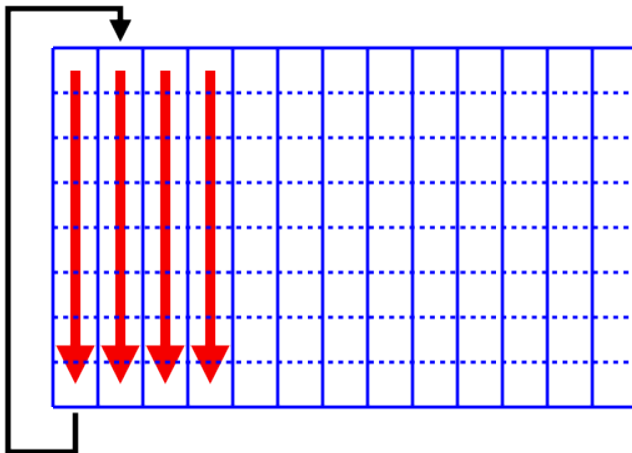




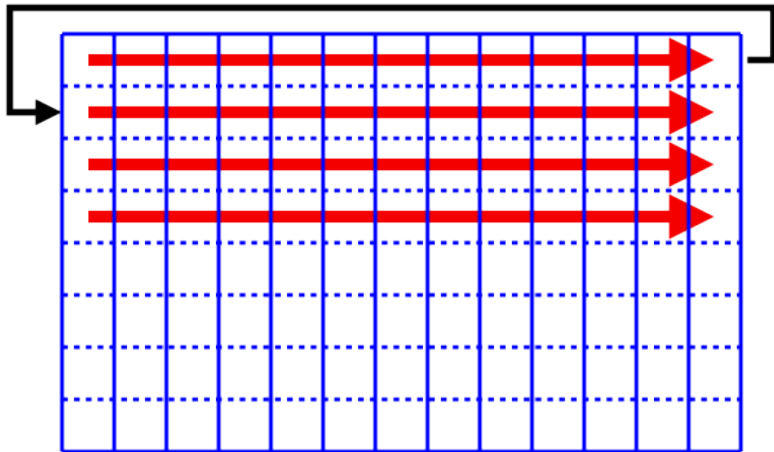
- ▶ Memoria → sequenza lineare di locazioni elementari
- ▶ Matrice  $A$ , elemento  $a_{ij}$ :  $i$  indice di riga,  $j$  indice di colonna
- ▶ Le matrici sono rappresentate con array
- ▶ Come sono memorizzati gli elementi di un array?
- ▶ **C**: in successione seguendo l'ultimo indice, poi il precedente ...  
 $a[1][1] \ a[1][2] \ a[1][3] \ a[1][4] \ \dots$   
 $a[1][n] \ a[2][1] \ \dots \ a[n][n]$
- ▶ **Fortran**: in successione seguendo il primo indice, poi il secondo ...  
 $a(1,1) \ a(2,1) \ a(3,1) \ a(4,1) \ \dots$   
 $a(n,1) \ a(1,2) \ \dots \ a(n,n)$



- ▶ È la distanza tra due dati successivamente acceduti
  - ▶  $\text{stride}=1 \rightarrow$  sfrutto la località spaziale
  - ▶  $\text{stride} \gg 1 \rightarrow$  non sfrutto la località spaziale
- ▶ Regola d'oro
  - ▶ Accedere sempre, se possibile, a stride unitario



# Ordine di memorizzazione:C





► Calcolare il prodotto matrice-vettore:

- Fortran:  $d(i) = a(i) + b(i,j)*c(j)$
- C:  $d[i] = a[i] + b[i][j]*c[j];$

► Fortran

- **do j=1,n**
  - do i=1,n
  - $d(i) = a(i) + b(i,j)*c(j)$
  - end do
- end do

► C

- **for(i=0;i<n,i++1)**
  - for(j=0;j<n,j++1)
  - $d[i] = a[i] + b[i][j]*c[j];$



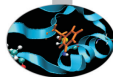
- Prodotto matrice-matrice 1024X1024
- tempi misurati in secondi
- yoda (Intel(R) Xeon(R) CPU E5-2620 2.00GHz)
- gnu versione 4.7.2

Ordine	Fortran	C
i,k,j	23.25	12.07
j,k,i	12.41	22.89



## Soluzione sistema triangolare

- ▶  $Lx = b$
- ▶ Dove:
  - ▶  $L$  è una matrice  $n \times n$  triangolare inferiore
  - ▶  $x$  è un vettore di  $n$  incognite
  - ▶  $b$  è un vettore di  $n$  termini noti
- ▶ Questo sistema è risolvibile tramite:
  - ▶ forward substitution
  - ▶ partizionamento della matrice



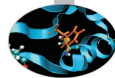
## Soluzione sistema triangolare

- ▶  $Lx = b$
- ▶ Dove:
  - ▶  $L$  è una matrice  $n \times n$  triangolare inferiore
  - ▶  $x$  è un vettore di  $n$  incognite
  - ▶  $b$  è un vettore di  $n$  termini noti
- ▶ Questo sistema è risolvibile tramite:
  - ▶ forward substitution
  - ▶ partizionamento della matrice

Qual è più veloce?

Perché?





Soluzione:

...

do i = 1, n

do j = 1, i-1

$b(i) = b(i) - L(i,j) b(j)$

enddo

$b(i) = b(i)/L(i,i)$

enddo

...



Soluzione:

```
...  
do i = 1, n  
    do j = 1, i-1  
        b(i) = b(i) - L(i,j) b(j)  
    enddo  
    b(i) = b(i)/L(i,i)  
enddo  
...
```

```
[vruggiel@fen07 TRI]$ ./a.out
```

```
Calcolo sistema  $L * y = b$   
time for solution      8.0586
```

# Partizionamento della matrice



Soluzione:

```
...  
do j = 1, n  
  b(j) = b(j)/L(j,j)  
  do i = j+1,n  
    b(i) = b(i) - L(i,j)*b(j)  
  enddo  
enddo  
...
```

# Partizionamento della matrice



Soluzione:

```
...  
do j = 1, n  
  b(j) = b(j)/L(j,j)  
  do i = j+1,n  
    b(i) = b(i) - L(i,j)*b(j)  
  enddo  
enddo  
...
```

```
[vruggiel@fen07 TRI]$ ./a.out
```

```
Calcolo sistema  $L * y = b$   
time for solution    2.5586
```



- ▶ Forward substitution  
do  $i = 1, n$   
  do  $j = 1, i-1$   
     $b(i) = b(i) - L(i,j) b(j)$   
  enddo  
   $b(i) = b(i)/L(i,i)$   
enddo
- ▶ Partizionamento della matrice  
do  $j = 1, n$   
   $b(j) = b(j)/L(j,j)$   
  do  $i = j+1, n$   
     $b(i) = b(i) - L(i,j)*b(j)$   
  enddo  
enddo



- ▶ Forward substitution
  - do  $i = 1, n$ 
    - do  $j = 1, i-1$ 
      - $b(i) = b(i) - L(i,j) b(j)$
    - enddo
    - $b(i) = b(i)/L(i,i)$
  - enddo
- ▶ Partizionamento della matrice
  - do  $j = 1, n$ 
    - $b(j) = b(j)/L(j,j)$
    - do  $i = j+1, n$ 
      - $b(i) = b(i) - L(i,j)*b(j)$
    - enddo
  - enddo
- ▶ Stesso numero di operazioni, ma tempi molto differenti (più di un fattore 3)
- ▶ Perché?

# Vediamo se il concetto è chiaro ...

Questa matrice:

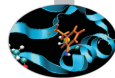
A	D	G	L
B	E	H	M
C	F	I	N

In C è memorizzata:

A	D	G	L	B	E	H	M	C	F	I	N
---	---	---	---	---	---	---	---	---	---	---	---

In Fortran è memorizzata:

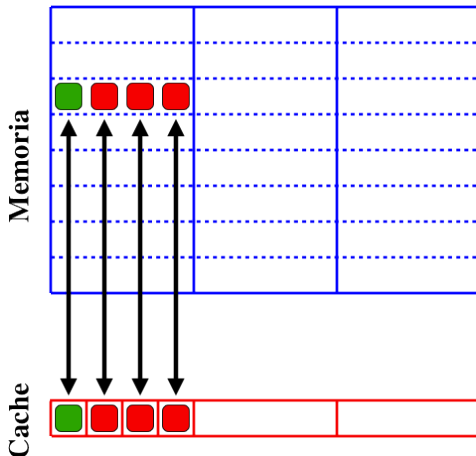
A	B	C	D	E	F	G	H	I	L	M	N
---	---	---	---	---	---	---	---	---	---	---	---



# Località spaziale: righe di cache



- La cache è organizzata in blocchi (righe)
- La memoria è suddivisa in blocchi grandi quanto una riga
- Richiedendo un dato si copia in cache il blocco che lo contiene



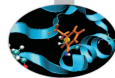




- ▶ Prodotto matrice-matrice in doppia precisione
- ▶ Versioni alternative, differenti chiamate della libreria BLAS
- ▶ Prestazioni in MFlops su Intel(R) Xeon(R) CPU X5660 2.80GHz

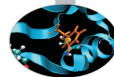
Dimensioni	1 DGEMM	$N$ DGEMV	$N^2$ DDOT
500	5820	3400	217
1000	8420	5330	227
2000	12150	2960	136
3000	12160	2930	186

Stesso numero di operazioni, l'uso della cache cambia!!!



- ▶ I registri sono locazioni di memoria interne alla CPU
- ▶ poche (tipicamente  $< 128$ ), ma con latenza nulla
- ▶ Tutte le operazioni delle unità di calcolo:
  - ▶ prendono i loro operandi dai registri
  - ▶ riportano i risultati in registri
- ▶ i trasferimenti memoria  $\leftrightarrow$  registri sono fasi separate
- ▶ il compilatore utilizza i registri:
  - ▶ per valori intermedi durante il calcolo delle espressioni
  - ▶ espressioni troppo complesse o corpi di loop troppo lunghi forzano lo "spilling" di registri in memoria.
  - ▶ per tenere "a portata di CPU" i valori acceduti di frequente
  - ▶ ma solo per variabili scalari, non per elementi di array

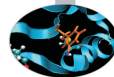
# Elementi di vettori ...



```

do 3000 z=1,nz
  k3=beta(z)
  do 3000 y=1,ny
    k2=eta(y)
    do 3000 x=1,nx/2
      hr(x,y,z,1)=hr(x,y,z,1)*norm
      hi(x,y,z,1)=hi(x,y,z,1)*norm
      hr(x,y,z,2)=hr(x,y,z,2)*norm
      hi(x,y,z,2)=hi(x,y,z,2)*norm
      hr(x,y,z,3)=hr(x,y,z,3)*norm
      hi(x,y,z,3)=hi(x,y,z,3)*norm
      .....
      k1=alfa(x,1)
      k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
      k_quad=1./k_quad
      sr=k1*hr(x,y,z,1)+k2*hr(x,y,z,2)+k3*hr(x,y,z,3)
      si=k1*hi(x,y,z,1)+k2*hi(x,y,z,2)+k3*hi(x,y,z,3)
      hr(x,y,z,1)=hr(x,y,z,1)-sr*k1*k_quad
      hr(x,y,z,2)=hr(x,y,z,2)-sr*k2*k_quad
      hr(x,y,z,3)=hr(x,y,z,3)-sr*k3*k_quad
      hi(x,y,z,1)=hi(x,y,z,1)-si*k1*k_quad
      hi(x,y,z,2)=hi(x,y,z,2)-si*k2*k_quad
      hi(x,y,z,3)=hi(x,y,z,3)-si*k3*k_quad
      k_quad_cfr=0.
3000  continue
  
```

# scalari di appoggio

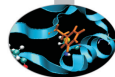


```

do 3000 z=1,nz
  k3=beta(z)
  do 3000 y=1,ny
    k2=eta(y)
    do 3000 x=1,nx/2
      br1=hr(x,y,z,1)*norm
      bil=hi(x,y,z,1)*norm
      br2=hr(x,y,z,2)*norm
      bi2=hi(x,y,z,2)*norm
      br3=hr(x,y,z,3)*norm
      bi3=hi(x,y,z,3)*norm
      .....
      k1=alfa(x,1)
      k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
      k_quad=1./k_quad
      sr=k1*br1+k2*br2+k3*br3
      si=k1*bil+k2*bi2+k3*bi3
      hr(x,y,z,1)=br1-sr*k1*k_quad
      hr(x,y,z,2)=br2-sr*k2*k_quad
      hr(x,y,z,3)=br3-sr*k3*k_quad
      hi(x,y,z,1)=bil-si*k1*k_quad
      hi(x,y,z,2)=bi2-si*k2*k_quad
      hi(x,y,z,3)=bi3-si*k3*k_quad
      k_quad_cfr=0.
    enddo
  enddo
enddo
3000 Continue

```

# scalari di appoggio (durata -25%)



```

do 3000 z=1,nz
  k3=beta(z)
  do 3000 y=1,ny
    k2=eta(y)
    do 3000 x=1,nx/2
      br1=hr(x,y,z,1)*norm
      bil=hi(x,y,z,1)*norm
      br2=hr(x,y,z,2)*norm
      bi2=hi(x,y,z,2)*norm
      br3=hr(x,y,z,3)*norm
      bi3=hi(x,y,z,3)*norm
      .....
      k1=alfa(x,1)
      k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
      k_quad=1./k_quad
      sr=k1*br1+k2*br2+k3*br3
      si=k1*bil+k2*bi2+k3*bi3
      hr(x,y,z,1)=br1-sr*k1*k_quad
      hr(x,y,z,2)=br2-sr*k2*k_quad
      hr(x,y,z,3)=br3-sr*k3*k_quad
      hi(x,y,z,1)=bil-si*k1*k_quad
      hi(x,y,z,2)=bi2-si*k2*k_quad
      hi(x,y,z,3)=bi3-si*k3*k_quad
      k_quad_cfr=0.
3000  Continue

```



- ▶ Trasposizione di matrice  
do  $j = 1, n$   
  do  $i = 1, n$   
     $a(i,j) = b(j,i)$   
  end do  
end do
- ▶ Qual è l'ordine del loop con stride minimo?
- ▶ Per dati all'interno della cache non c'è dipendenza dallo stride
  - ▶ se dividessi l'operazione in blocchi abbastanza piccoli da entrare in cache?
  - ▶ posso bilanciare tra località spaziale e temporale.



- ▶ I dati elaborati in blocchi di dimensione adeguata alla cache
- ▶ All'interno di ogni blocco c'è il riutilizzo delle righe caricate
- ▶ Lo può fare il compilatore, se il loop è semplice, ma a livelli di ottimizzazione elevati
- ▶ Esempio della tecnica: trasposizione della matrice

```
do jj = 1, n , step
  do ii = 1, n, step
    do j= jj,jj+step-1,1
      do i=ii,ii+step-1,1
        a(i,j)=b(j,i)
      end do
    end do
  end do
end do
```

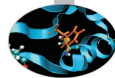
# Cache: capacity miss e trashing



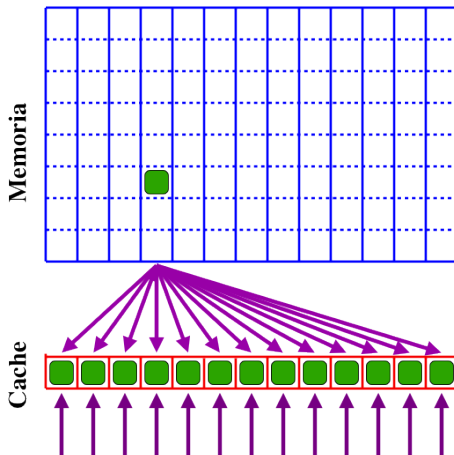
- ▶ La cache può soffrire di capacity miss:
  - ▶ si utilizza un insieme ristretto di righe (reduced effective cache size)
  - ▶ si riduce la velocità di elaborazione
- ▶ La cache può soffrire di trashing:
  - ▶ per caricare nuovi dati si getta via una riga prima che sia stata completamente utilizzata
  - ▶ è più lento di non avere cache
- ▶ Capita quando più flussi di dati/istruzioni insistono sulle stesse righe di cache
- ▶ Dipende dal mapping memoria  $\leftrightarrow$  cache
  - ▶ fully associative cache
  - ▶ direct mapped cache
  - ▶ N-way set associative cache



# Fully associative cache



- Ogni blocco di memoria può essere mappato in una riga qualsiasi di cache

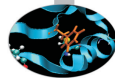


# Fully associative cache

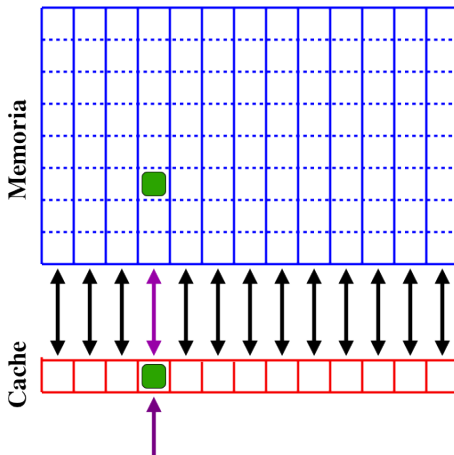


- ▶ **Pro:**
  - ▶ sfruttamento completo della cache
  - ▶ relativamente "insensibile" ai pattern di accesso alla memoria
- ▶ **Contro:**
  - ▶ strutture circuitali molto complesse per identificare molto rapidamente un hit
  - ▶ algoritmo di sostituzione oneroso Least Recently Used (LRU) o limitatamente efficiente First In First Out (FIFO)
  - ▶ costosa e di dimensioni limitate

# Direct mapped cache



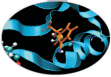
- Ogni blocco di memoria può essere mappato in una sola riga di cache (congruenza lineare)



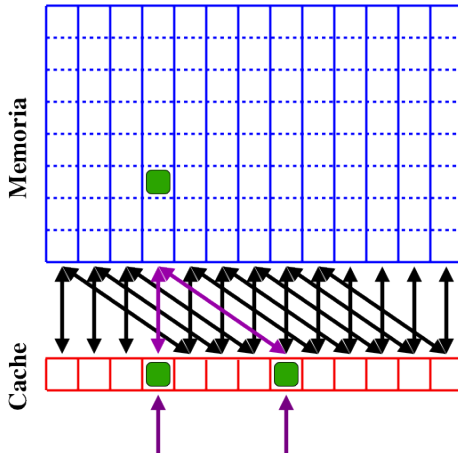


- ▶ **Pro:**
  - ▶ identificazione di un hit facilissima (alcuni bit dell'indirizzo identificano la riga da controllare)
  - ▶ algoritmo di sostituzione banale
  - ▶ cache di dimensione "arbitraria"
- ▶ **Contro:**
  - ▶ molto "sensibile" ai pattern di accesso alla memoria
  - ▶ soggetta a capacity miss
  - ▶ soggetta a cache trashing

# N-way set associative cache



- Ogni blocco di memoria può essere mappato in una qualsiasi riga tra N possibili righe di cache



# N-way set associative cache



## ► Pro:

- è un punto di bilanciamento intermedio
  - $N=1 \rightarrow$  direct mapped
  - $N =$  numero di righe di cache  $\rightarrow$  fully associative
- consente di scegliere il bilanciamento tra complessità circuitale e prestazioni (i.e. costo del sistema e difficoltà di programmazione)
- consente di realizzare cache di dimensione "soddisfacente"

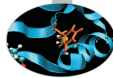
## ► Contro:

- molto "sensibile" ai pattern di accesso alla memoria
- parzialmente soggetta a capacity miss
- parzialmente soggetta a cache trashing

# Cache: la situazione tipica



- ▶ Cache L1:  $4 \div 8$  way set associative
- ▶ Cache L2÷3:  $2 \div 4$  way set associative o direct mapped
- ▶ Capacity miss e trashing vanno affrontati
  - ▶ le tecniche sono le stesse
  - ▶ controllo della disposizione dei dati in memoria
  - ▶ controllo delle sequenze di accessi in memoria
- ▶ La cache L1 lavora su indirizzi virtuali
  - ▶ pieno controllo da parte del programmatore
- ▶ le cache L2÷3 lavorano su indirizzi fisici
  - ▶ le prestazioni dipendono dalla memoria fisica allocata
  - ▶ le prestazioni possono variare da esecuzione a esecuzione
  - ▶ si controllano a livello di sistema operativo



- ▶ Problemi di accesso ai dati in memoria
- ▶ Provoca la sostituzione di una riga di cache il cui contenuto verrà richiesto subito dopo
- ▶ Si presenta quando due o più flussi di dati insistono su un insieme ristretto di righe di cache
- ▶ NON aumenta il numero di load e store
- ▶ Aumenta il numero di transazioni sul bus di memoria
- ▶ In genere si presenta per flussi il cui stride relativo è una potenza di 2



# No trashing: $C(i) = A(i) + B(i)$



## ► Iterazione $i=1$

1. Cerco  $A(1)$  nella cache di primo livello (L1) → **cache miss**
2. Recupero  $A(1)$  nella memoria RAM
3. Copio da  $A(1)$  a  $A(8)$  nella L1
4. Copio  $A(1)$  in un registro
5. Cerco  $B(1)$  nella cache di primo livello (L1) → **cache miss**
6. Recupero  $B(1)$  nella memoria RAM
7. Copio da  $B(1)$  a  $B(8)$  nella L1
8. Copio  $B(1)$  in un registro
9. Eseguo somma

## ► Iterazione $i=2$

1. Cerco  $A(2)$  nella cache di primo livello (L1) → **cache hit**
2. Copio  $A(2)$  in un registro
3. Cerco  $B(2)$  nella cache di primo livello (L1) → **cache hit**
4. Copio  $B(2)$  in un registro
5. Eseguo somma

## ► Iterazione $i=3$

# Trashing: $C(i) = A(i) + B(i)$



## ► Iterazione $i=1$

1. Cerco  $A(1)$  nella cache di primo livello (L1) → **cache miss**
2. Recupero  $A(1)$  nella memoria RAM
3. Copio da  $A(1)$  a  $A(8)$  nella L1
4. Copio  $A(1)$  in un registro
5. Cerco  $B(1)$  nella cache di primo livello (L1) → **cache miss**
6. Recupero  $B(1)$  nella memoria RAM
7. **Scarico la riga di cache che contiene  $A(1)$ - $A(8)$**
8. Copio da  $B(1)$  a  $B(8)$  nella L1
9. Copio  $B(1)$  in un registro
10. Eseguo somma

# Trashing: $C(i) = A(i) + B(i)$



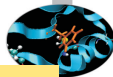
## ► Iterazione $i=2$

1. Cerco  $A(2)$  nella cache di primo livello (L1) → **cache miss**
2. Recupero  $A(2)$  nella memoria RAM
3. **Scarico la riga di cache che contiene  $B(1)$ - $B(8)$**
4. Copio da  $A(1)$  a  $A(8)$  nella L1
5. Copio  $A(2)$  in un registro
6. Cerco  $B(2)$  nella cache di primo livello (L1) → **cache miss**
7. Recupero  $B(2)$  nella memoria RAM
8. **Scarico la riga di cache che contiene  $A(1)$ - $A(8)$**
9. Copio da  $B(1)$  a  $B(8)$  nella L1
10. Copio  $B(2)$  in un registro
11. Eseguo somma

## ► Iterazione $i=3$

## Come identificarlo?

- Effetto variabile in funzione della dimensione del data set

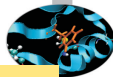


```
...  
integer , parameter    :: offset=..  
integer , parameter    :: N1=6400  
integer , parameter    :: N=N1+offset  
...  
real (8)              :: x (N,N) , y (N,N) , z (N,N)  
...  
do j=1,N1  
  do i=1,N1  
    z (i, j)=x (i, j)+y (i, j)  
  end do  
end do  
...
```

offset	tempo
0	0.361
3	0.250
400	0.252
403	0.253

La soluzione é il padding.

# Come identificarlo?



## ► Effetto variabile in funzione della dimensione del data set

```

...
integer , parameter    :: offset=..
integer , parameter    :: N1=6400
integer , parameter    :: N=N1+offset

....
real (8)              :: x (N,N) , y (N,N) , z (N,N)
...
do j=1,N1
  do i=1,N1
    z (i, j)=x (i, j)+y (i, j)
  end do
end do
...

```

offset	tempo
0	0.361
3	0.250
400	0.252
403	0.253

La soluzione é il padding.



- ▶ Raddoppiano le transazioni sul bus
- ▶ Su alcune architetture:
  - ▶ causano errori a runtime
  - ▶ sono emulati in software
- ▶ Sono un problema
  - ▶ con tipi dati strutturati( TYPE e struct)
  - ▶ con le variabili locali alla routine
  - ▶ con i common
- ▶ Soluzioni
  - ▶ ordinare le variabili per dimensione decrescente
  - ▶ opzioni di compilazione (quando disponibili ...)
  - ▶ common diversi/separati
  - ▶ inserimento di variabili "dummy" nei common



```
parameter (nd=1000000)
real*8 a(1:nd), b(1:nd)
integer c
common /data1/ a,c,b
....
do j = 1, 300
  do i = 1, nd
    somma1 = somma1 + (a(i)-b(i))
  enddo
enddo
```

Diverse performance per:

```
common /data1/ a,c,b
common /data1/ b,c,a
common /data1/ a,b,c
```

Dipende dall'architettura e dal compilatore che in genere segnala e cerca di sanare con opzione di align common

# Come accertare qual è il problema?



- ▶ Tutti i processori hanno contatori hardware di eventi
- ▶ Introdotti dai progettisti per CPU ad alti clock
  - ▶ indispensabili per debuggare i processori
  - ▶ utili per misurare le prestazioni
  - ▶ fondamentali per capire comportamenti anomali
- ▶ Ogni architettura misura eventi diversi
- ▶ Sono ovviamente proprietari
  - ▶ IBM:HPCT
  - ▶ INTEL:Vtune
- ▶ Esistono strumenti di misura multiplatforma
  - ▶ Valgrind,Oprofile
  - ▶ PAPI
  - ▶ Likwid
  - ▶ ...



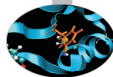
# La cache è una memoria



- ▶ Mantiene il suo stato finché un cache-miss non ne causa la modifica
- ▶ È uno stato nascosto al programmatore:
  - ▶ non influenza la semantica del codice (ossia i risultati)
  - ▶ influenza le prestazioni
- ▶ La stessa routine chiamata in due contesti diversi del codice può avere prestazioni del tutto diverse a seconda dello stato che "trova" nella cache
- ▶ La modularizzazione del codice tende a farci ignorare questo problema
- ▶ Può essere necessario inquadrare il problema in un contesto più ampio della singola routine



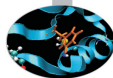
- ▶ Software Open Source utile per il Debugging/Profiling di programmi Linux, di cui non richiede i sorgenti (black-box analysis), ed è composto da diversi tool:
  - ▶ Memcheck (detect memory leaks, ...)
  - ▶ Cachegrind (cache profiler)
  - ▶ Callgrind (callgraph)
  - ▶ Massif (heap profiler)
  - ▶ Etc.
- ▶ <http://valgrind.org>



```
valgrind --tool=cachegrind <nome_eseguibile>
```

- ▶ Simula come il vostro programma interagisce con la gerarchia di cache
  - ▶ due cache indipendenti di primo livello (L1)
    - ▶ per istruzioni (I1)
    - ▶ per dati (D1)
  - ▶ una cache di ultimo livello, L2 o L3(LL)
- ▶ Riporta diverse statistiche
  - ▶ I cache reads (I: numero di istruzioni eseguite), I1 cache read misses(I1mr),LL cache instruction read misses (ILmr)
  - ▶ D cache reads, Dr,D1mr,DLLmr
  - ▶ D cache writes, Dw,D1mw,DLLmw
- ▶ Riporta (opzionale) il numero di branch e quelli mispredicted

# Cachegrind:esempio I



```

==14924== I    refs:          7,562,066,817
==14924== I1   misses:           2,288
==14924== LLi  misses:           1,913
==14924== I1   miss rate:           0.00%
==14924== LLi  miss rate:           0.00%
==14924==
==14924== D    refs:          2,027,086,734 (1,752,826,448 rd + 274,260,286 wr)
==14924== D1   misses:          16,946,127 ( 16,846,652 rd +    99,475 wr)
==14924== LLd  misses:           101,362 (    2,116 rd +    99,246 wr)
==14924== D1   miss rate:           0.8% (    0.9% +    0.0% )
==14924== LLd  miss rate:           0.0% (    0.0% +    0.0% )
==14924==
==14924== LL refs:          16,948,415 ( 16,848,940 rd +    99,475 wr)
==14924== LL misses:           103,275 (    4,029 rd +    99,246 wr)
==14924== LL miss rate:           0.0% (    0.0% +    0.0% )

```



```

==15572== I   refs:          7,562,066,871
==15572== I1  misses:           2,288
==15572== LLi misses:          1,913
==15572== I1  miss rate:           0.00%
==15572== LLi miss rate:          0.00%
==15572==
==15572== D   refs:          2,027,086,744 (1,752,826,453 rd + 274,260,291 wr)
==15572== D1  misses:          151,360,463 ( 151,260,988 rd +    99,475 wr)
==15572== LLd misses:           101,362 (    2,116 rd +    99,246 wr)
==15572== D1  miss rate:           7.4% (    8.6% +    0.0% )
==15572== LLd miss rate:           0.0% (    0.0% +    0.0% )
==15572==
==15572== LL refs:          151,362,751 ( 151,263,276 rd +    99,475 wr)
==15572== LL misses:           103,275 (    4,029 rd +    99,246 wr)
==15572== LL miss rate:           0.0% (    0.0% +    0.0% )
  
```

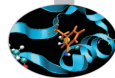


- ▶ Cachegrind genera automaticamente un file `cachegrind.out.<pid>`
- ▶ Oltre alle precedenti informazioni vengono generate anche statistiche funzione per funzione

```
cg_annotate cachegrind.out.<pid>
```



- ▶ `—l1=<size>,<associativity>,<line size>`
- ▶ `—D1=<size>,<associativity>,<line size>`
- ▶ `—LL=<size>,<associativity>,<line size>`
- ▶ `—cache-sim=no|yes [yes]`
- ▶ `—branch-sim=no|yes [no]`
- ▶ `—cachegrind-out-file=<file>`



- ▶ Performance Application Programming Interface
- ▶ Interfaccia standard per accedere agli hardware counters
- ▶ Disponibili 2 interfacce (sia per C che per fortran):
  - ▶ High-level interface: semplice
  - ▶ Low-level interface: piú complicata ma piú programmabile
- ▶ Pro:
  - ▶ Non occorrono i permessi di root per le misure
  - ▶ Nomi standard per gli eventi monitorati (se disponibili nella macchina) con la high-level interface
  - ▶ Monitorazione agevole di specifiche sezioni di codice
- ▶ Contro:
  - ▶ É necessario "instrumentare" il codice a mano
  - ▶ É necessario avere un kernel che renda disponibili gli eventi da monitorare





Sono associati ad hardware counters che dipendono dalla macchina

Esempio:

*PAPI\_TOT\_CYC*: cicli totali

*PAPI\_TOT\_INS*: istruzioni completate

*PAPI\_FP\_INS*: istruzioni floating-point

*PAPI\_L1\_DCA*: accessi in cache dati L1

*PAPI\_L1\_DCM*: misses in cache dati L1

*PAPI\_SR\_INS*: istruzioni di store

*PAPI\_BR\_MSP*: istruzioni di branch mispredicted



- Le chiamate alla libreria di alto livello sono piuttosto intuitive.
- É possibile monitorare piú eventi in contemporanea.
- In Fortran:

```

#include "fpapi_test.h"
integer events(2), retval    ;    integer*8 values(2)
.....
events(1) = PAPI_FP_INS      ;    events(2) = PAPI_L1_DCM
.....
call PAPIf_start_counters(events, 2, retval)
call PAPIf_read_counters(values, 2, retval)  ! Clear values
< sezione di codice da monitorare >
call PAPIf_stop_counters(values, 2, retval)
print*, 'Floating point instructions', values(1)
print*, 'L1 Data Cache Misses: ', values(2)
.....
  
```



- ▶ É possibile gestire eventuali errori analizzando una variabile di ritorno delle subroutine
- ▶ É possibile effettuare query per stabilire se un hardware counter é disponibile o meno, quanti eventi é possibile monitorare, etc...
- ▶ É molto consigliabile chiamare una subroutine dummy dopo la parte monitorata per assicurarsi che l'ottimizzazione non abbia anticipato operazioni...

# Esercitazione: sommario



- ▶ Prodotto matrice matrice:ordine dei loop
- ▶ Prodotto matrice matrice:blocking
- ▶ Prodotto matrice matrice:blocking e padding
- ▶ Misura delle prestazioni delle cache



- Andare su *eser\_2* (fortran/mm.f90 o c/mm.c )
- Misurare i tempi per **N=512** al variare dell'ordine del loop
- Usare fortran e/o c
- Usare compilatori gnu senza ottimizzazioni(**-O0**)

Indici	Tempi	C
i,j,k		
i,k,j		
j,k,i		
j,i,k		
k,i,j,k		
k,j,i,k		



```

1  ...
2  integer, parameter :: n=1024  ! size of the matrix
3  integer, parameter :: step=4
4  integer, parameter :: npad=0
5  ...
6  real(my_kind) a(1:n+npad,1:n) ! matrix
7  real(my_kind) b(1:n+npad,1:n) ! matrix
8  real(my_kind) c(1:n+npad,1:n) ! matrix (destination)
9
10     do jj = 1, n, step
11       do kk = 1, n, step
12         do ii = 1, n, step
13           do j = jj, jj+step-1
14             do k = kk, kk+step-1
15               do i = ii, ii+step-1
16                 c(i,j) = c(i,j) + a(i,k)*b(k,j)
17               enddo
18             ...

```



```

1  #define nn (1024)
2  #define step (4)
3  #define npad (0)
4
5  double a[nn+npad][nn+npad];          /** matrici**/
6  double b[nn+npad][nn+npad];
7  double c[nn+npad][nn+npad];
8  ...
9    for (ii = 0; ii < nn; ii= ii+step)
10      for (kk = 0; kk < nn; kk = kk+step)
11        for (jj = 0; jj < nn; jj = jj+step)
12          for ( i = ii; i < ii+step; i++ )
13            for ( k = kk; k < kk+step; k++ )
14              for ( j = jj; j < jj+step; j++ )
15                c[i][j] = c[i][j] + a[i][k]*b[k][j];
16  ...
  
```



- Andate su `eser_3` (fortran/mm.f90 o c/mm.c )
- Misurare i tempi al variare della dimensione del blocking per matrici con **N=1024**
- Usare Fortran e/o c
- Usare i compilatori gnu con livello di ottimizzazione **-O3**

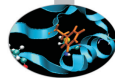
Step	Fortran	C
4		
8		
16		
32		
64		
128		
256		





- Andate su `eser_3` (fortran/mm.f90 o c/mm.c )
- Misurare i tempi al variare della dimensione del blocking per matrici con **N=1024** e **npad=9**
- Usare Fortran e/o c
- Usare i compilatori gnu con livello di ottimizzazione **-O3**

Step	Fortran	C
4		
8		
16		
32		
64		
128		
256		



- ▶ valgrind
  - ▶ Utilizzare il tool cachegrind per "scoprire" come variano le prestazioni mmisurate modificando l'ordine dei loop

- ▶ papi

- ▶ Eseguire i seguenti comandi

```
cp /afs/icaspur.it/project/open/space/papi-5.1.0.2.tar /scratch
cd /scratch
tar xvf papi-5.1.0.2.tar
```

- ▶ Configurare e compilare papi
  - ▶ Familiarizzare con i test
  - ▶ Nella directory src/eser\_papi/fortran o src/eser\_papi/c compilare ed eseguire il prodotto matrice-matrice
  - ▶ Modificare a piacere l'ordine del loop e gli eventi papi da misurare.



Introduzione

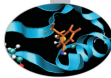
Architetture

La cache e il sistema di memoria

Pipeline

Profilers

# CPU:parallelismo interno?



- ▶ Le CPU sono internamente parallele
  - ▶ pipelining
  - ▶ esecuzione superscalare
  - ▶ unità SIMD MMX, SSE, SSE2, SSE3, SSE4, AVX
- ▶ Per ottenere performance paragonabili con quelle sbandierate dal produttore:
  - ▶ fornire istruzioni in quantità
  - ▶ fornire gli operandi delle istruzioni



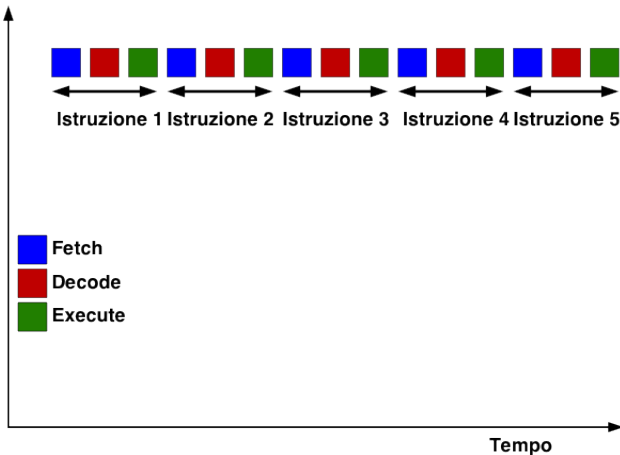
- ▶ Pipeline=tubazione, catena di montaggio
- ▶ Un'operazione è divisa in più passi indipendenti(stage) e differenti passi di differenti operazioni vengono eseguiti **contemporaneamente**
- ▶ Parallelismo sulle diverse fasi delle operazioni
- ▶ I processori sfruttano intensivamente il pipelining per aumentare la capacità di elaborazione



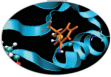
- ▶ **fetch** (prendere, prelevare) reperisce l'istruzione dalla memoria e viene incrementato il valore del Program Counter in modo da puntare all'istruzione successiva
- ▶ **decode** (o decodifica) l'istruzione viene interpretata
- ▶ **execute** invia segnali che rappresentano opportuni comandi per l'esecuzione.



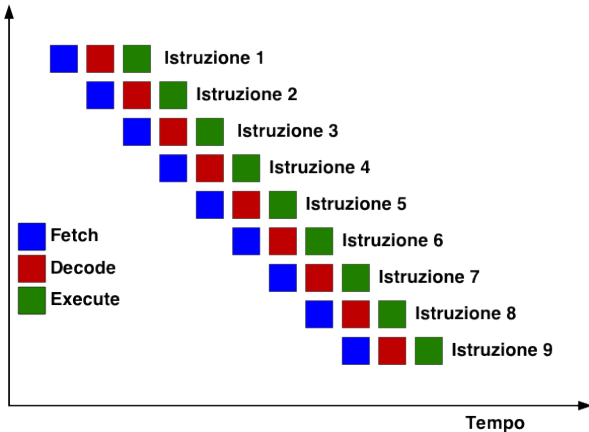
- Un'istruzione completata ogni tre cicli



# Unità di calcolo pipelined

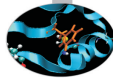


- Un risultato per ciclo a pipeline piena
- Per riempirla 3 istruzioni indipendenti, con gli operandi
- All'estremo opposto, un risultato ogni 3 cicli a pipeline "vuota"

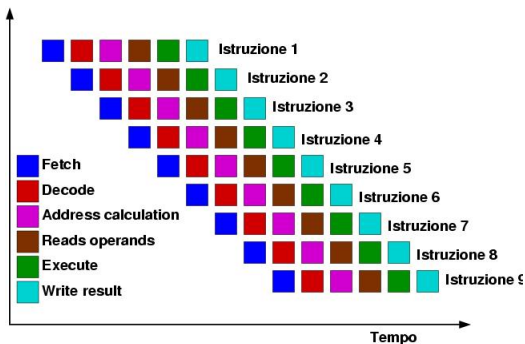




# Unità di calcolo superpipelined



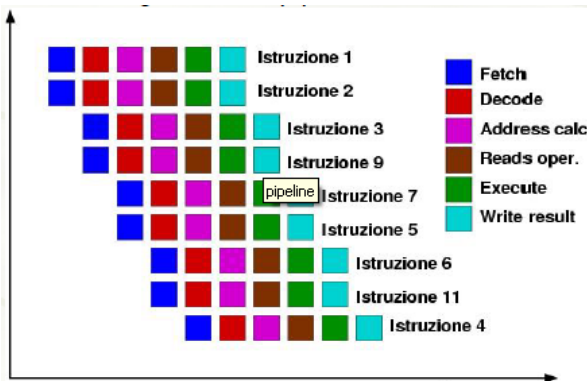
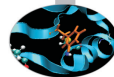
- ▶ Un risultato per ciclo a pipeline piena
- ▶ Per riempirla 6 istruzioni indipendenti, con gli operandi
- ▶ All'estremo opposto, un risultato ogni 6 cicli a pipeline "vuota"
- ▶ É possibile dimezzare il ciclo di clock (ossia raddoppiare la frequenza)





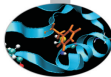
- ▶ Riordina dinamicamente le istruzioni
  - ▶ anticipa istruzioni i cui operandi sono già disponibili
  - ▶ postpone istruzioni i cui operandi non sono ancora disponibili
  - ▶ riordina letture e scritture in memoria
  - ▶ il tutto dipendentemente dalle unità funzionali libere
- ▶ Si appoggia intensivamente su:
  - ▶ renaming dei registri (registri fisici vs. architetturali)
  - ▶ branch prediction
  - ▶ combinazione di read e write multiple in memoria
- ▶ È essenziale per ottenere prestazioni sulle CPU di oggi
- ▶ Non è sufficiente da sola, il codice deve rendere esplicite le possibilità di riordinamento

# Esecuzione out of order





- ▶ Le CPU contengono più unità indipendenti
  - ▶ differenziazione funzionale
  - ▶ replicazione funzionale
- ▶ Operazioni indipendenti sono eseguite in contemporanea
  - ▶ operazioni su interi
  - ▶ operazioni su floating point
  - ▶ calcolo di salti
  - ▶ accessi in memoria
- ▶ Parallelismo sulle istruzioni
- ▶ Mascheramento delle latenze
- ▶ I processori sfruttano intensivamente la superscalarità per aumentare la capacità di elaborazione a parità di clock



- ▶ I problemi principali sono:
  - ▶ come rimuovere la dipendenza tra le istruzioni?
  - ▶ come fornire abbastanza istruzioni indipendenti?
  - ▶ come fare in presenza di salti condizionali (if e loop)?
  - ▶ come fornire tutti i dati necessari?
- ▶ Chi deve modificare il codice?
  - ▶ la CPU? → sì per quel che può, OOO e branch prediction
  - ▶ il compilatore? → sì per quel che può, se lo evince dal codice
  - ▶ l'utente? → sì, nei casi più complessi
- ▶ Tecniche
  - ▶ loop unrolling → srotolo il loop
  - ▶ loop merging → fondo più loop insieme
  - ▶ loop splitting → decompongo loop complessi
  - ▶ inlining di funzioni → evito interruzioni di flusso di istruzioni



- ▶ Prodotto matrice-matrice
  - do j = 1, n
    - do k = 1, n
      - do i = 1, n
        - $c(i,j) = c(i,j) + a(i,k)*b(k,j)$
- ▶ 2 istruzioni utili per iterazione
- ▶ Le 2 istruzioni sono dipendenti tra loro
- ▶ Dominato da:
  - ▶ incremento degli indici di loop
  - ▶ salti condizionali impliciti alla fine del loop
  - ▶ calcolo degli indirizzi



- ▶ Prodotto matrice-matrice con unrolling del loop esterno
  - do j= 1, n, 2
  - do k= 1, n
  - do i= 1, n
  - $c(i,j+0) = a(i,k)*b(k,j+0)+c(i,j+0)$
  - $c(i,j+1) = a(i,k)*b(k,j+1)+c(i,j+1)$
  - end do
  - end do
  - end do
- ▶ Srotolando le iterazioni di un loop
  - ▶ si formano più flussi (stream) indipendenti di dati
  - ▶ due coppie di istruzioni indipendenti per iterazione
  - ▶ un valore è riutilizzato
  - ▶ l'indirizzo di  $b(k,j+1)$  si calcola banalmente da quello di  $b(k,j+0)$
  - ▶ idem per c
  - ▶ il peso di indici e salti di loop è dimezzato

# Come "frenare" il compilatore



- ▶ Il compilatore sa fare automaticamente l'unrolling
- ▶ Ma non lo applica se lo ritiene pericoloso!
- ▶ Esempio: aliasing

```
void accumulate(int n, double *a, double *s)
```

```
    int i;
```

```
    for(i=0; i , n; ++i)
```

```
        a[i] += s[i]
```

- ▶ Il compilatore non fa unrolling, nel timore di un possibile aliasing di a e s in chiamate tipo:  
accumulate(10, b+1, b); /\* che succede con unrolling?\*/



# Come "convincere" il compilatore



- Dichiarando che non vi sarà aliasing:  

```
void accumulate(int n,  
               double * restrict a, double * restrict s)  
    int i;  
    for(i=0; i , n; ++i)  
        a[i] += s[i]
```
- Il compilatore ora potrà fare unrolling fiducioso che il programmatore non verrà meno alla parola data
- Ma chi lo facesse verrebbe punito con risultati errati



- Dipendenze tra le iterazioni successive impediscono l'unrolling:  
do i = 2,n

$a(i) = a(i-1) + 1.0$

end do

do i = 2,n

$b(i) = b(i-1) * 2.0$

end do

- Un unico loop ha più istruzioni indipendenti per iterazione:  
do i = 2,n

$a(i) = a(i-1) + 1.0$

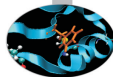
$b(i) = b(i-1) * 2.0$

end do

A volte il compilatore riesce a farlo da solo.



- ▶ Il corpo del loop può essere troppo grande
  - ▶ molte istruzioni e variabili di appoggio temporaneo
  - ▶ poche istruzioni ma espressioni lunghe e complicate
- ▶ In questi casi il compilatore può:
  - ▶ rinunciare
  - ▶ tentare l'unrolling e peggiorare le cose
- ▶ Register spilling
  - ▶ le variabili temporanee vengono allocate in registri della CPU
  - ▶ se sono troppe, si rende necessario memorizzare il contenuto
  - ▶ quando il valore serve, va ricaricato in memoria (lento)
  - ▶ in attesa del valore, si crea una bolla nella pipeline
- ▶ Dividendo il loop in due o più loop separati si può guadagnare velocità



- ▶ I salti condizionali (loop, if) interrompono il flusso regolare delle pipeline finché il valore della condizione non è nota
- ▶ Esecuzione speculativa
  - ▶ la Branch Prediction Unit accumula statistica sulle più recenti istruzioni di salto condizionato, e predice cosa avverrà la prossima volta
  - ▶ predizione sbagliata → ripristinare lo stato precedente al salto
    - ▶ vantaggiosa per pattern ben definiti (loop, if di gestione errori. . .)
    - ▶ inutile quando la condizione è casuale
- ▶ Esecuzione predicativa degli if-then-else (Itanium)
  - ▶ sia il then che l'else vengono eseguiti
  - ▶ quando la condizione è nota, si scelgono gli effetti dell'uno o dell'altro
    - ▶ il tempo di esecuzione è costante, le pipeline meglio utilizzate
    - ▶ onerosa per then e else "corposi"



- ▶ Le chiamate a funzione costano perché la CPU deve saltare ad eseguire un'altra porzione di codice e bisogna passare i parametri alla subroutine
- ▶ Si crea una bolla nella pipeline



- ▶ Prodotto matrice matrice:unrolling
- ▶ Prodotto matrice matrice:unrolling e padding



```

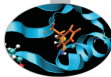
1  ...
2  integer, parameter :: n=1024 ! size of the matrix
3  integer, parameter :: step=4
4  integer, parameter :: npad=0
5  ...
6  real(my_kind) a(1:n+npad,1:n) ! matrix
7  real(my_kind) b(1:n+npad,1:n) ! matrix
8  real(my_kind) c(1:n+npad,1:n) ! matrix (destination)
9
10  do j = 1, n, 2
11    do k = 1, n
12      do i = 1, n
13        c(i,j+0) = c(i,j+0) + a(i,k)*b(k,j+0)
14        c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
15      enddo
16    enddo
17  enddo
18  ...
  
```



```
1  #define nn (1024)
2  #define step (4)
3  #define npad (0)
4
5  double a[nn+npad][nn+npad];          /** matrici**/
6  double b[nn+npad][nn+npad];
7  double c[nn+npad][nn+npad];
8  ...
9  for (i = 0; i < nn; i+=2)
10     for (k = 0; k < nn; k++)
11         for (j = 0; j < nn; j++) {
12             c[i+0][j] = c[i+0][j] + a[i+0][k]*b[k][j];
13             c[i+1][j] = c[i+1][j] + a[i+1][k]*b[k][j];
14         }
15     ...
```



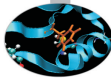
# Unrolling loop esterno



- Andate su `eser_4` (fortran/mm.f90 o c/mm.c )
- Misurare i tempi per matrici con **N=1024** al variare dell'unrolling del loop esterno
- Usare Fortran e/o c
- Usare i compilatori gnu con livello di ottimizzazione **-O3**

Unrolling	Fortran	C
2		
4		
8		
16		

# Unrolling loop esterno e padding



- Andate su `eser_4` (fortran/mm.f90 o c/mm.c )
- Misurare i tempi per matrici con **N=1024** al variare dell'unrolling del loop esterno con padding **npad=9**
- Usare Fortran e/o c
- Usare i compilatori gnu con livello di ottimizzazione **-O3**

Unrolling	Fortran	C
2		
4		
8		
16		



- ▶ Qual é la massima prestazione ottenibile facendo uso di:
  - ▶ blocking
  - ▶ unrolling loop esterno
  - ▶ padding
  - ▶ ... quant'altro
- ▶ per  $N=2048$ ?



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

- Motivazioni

- time

- top

- gprof

- Papi

- Scalasca

- Consigli finali



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

**Motivazioni**

time

top

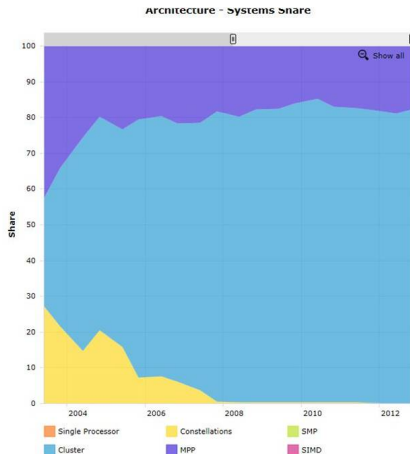
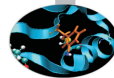
gprof

Papi

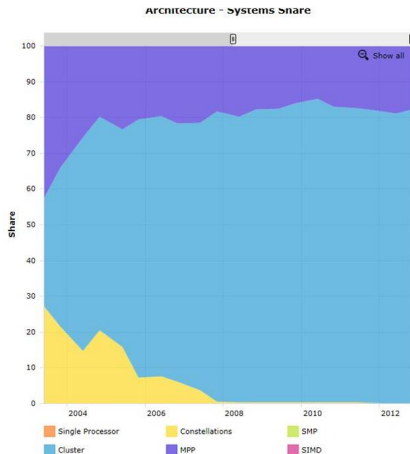
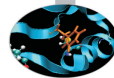
Scalasca

Consigli finali

# Il trend architetturale (Top500 list)

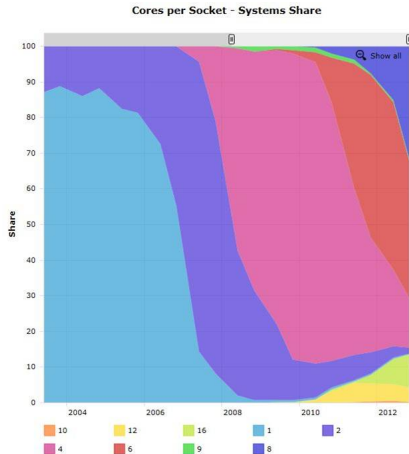
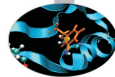


# Il trend architetturale (Top500 list)



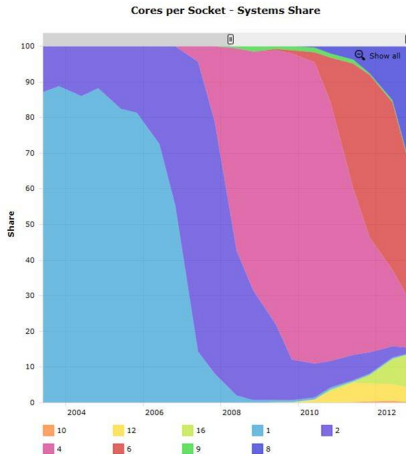
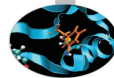
I cluster dominano il mercato dell'High Performance Computing

# Il trend dei processori





# Il trend dei processori



sempre piu processori "multicore" per "socket"

# Perché misurare le prestazioni?



- Architetture sempre più complesse:

# Perché misurare le prestazioni?



- ▶ Architetture sempre più complesse:
  - ▶ "bandwidth" verso la memoria ridotta

# Perché misurare le prestazioni?



- ▶ Architetture sempre più complesse:
  - ▶ "bandwidth" verso la memoria ridotta
  - ▶ quantità di memoria per "core" ridotta

# Perché misurare le prestazioni?



- ▶ Architetture sempre più complesse:
  - ▶ "bandwidth" verso la memoria ridotta
  - ▶ quantità di memoria per "core" ridotta
  - ▶ gerarchia di memoria sempre più complessa

# Perché misurare le prestazioni?



- ▶ Architetture sempre più complesse:
  - ▶ "bandwidth" verso la memoria ridotta
  - ▶ quantità di memoria per "core" ridotta
  - ▶ gerarchia di memoria sempre più complessa
- ▶ La programmazione su queste macchine non è semplice

# Perché misurare le prestazioni?



- ▶ Architetture sempre più complesse:
  - ▶ "bandwidth" verso la memoria ridotta
  - ▶ quantità di memoria per "core" ridotta
  - ▶ gerarchia di memoria sempre più complessa
- ▶ La programmazione su queste macchine non è semplice
- ▶ "Spremere" le prestazioni dei codici di calcolo non è semplice

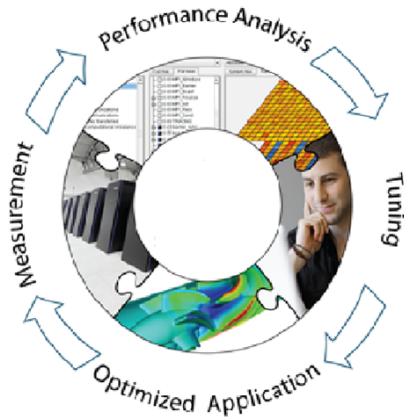
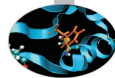
# Perché misurare le prestazioni?



- ▶ Architetture sempre più complesse:
  - ▶ "bandwidth" verso la memoria ridotta
  - ▶ quantità di memoria per "core" ridotta
  - ▶ gerarchia di memoria sempre più complessa
- ▶ La programmazione su queste macchine non è semplice
- ▶ "Spremere" le prestazioni dei codici di calcolo non è semplice
- ▶ Risulta pertanto fondamentale conoscere ed utilizzare strumenti di "Profiling" di supporto ad una successiva ottimizzazione, parallelizzazione, etc della nostra applicazione



# Il ciclo vita delle prestazioni





- ▶ Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- ▶ Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:



- ▶ Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- ▶ Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - ▶ trovare quelle parti dove viene speso gran parte del tempo



- ▶ Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- ▶ Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - ▶ trovare quelle parti dove viene speso gran parte del tempo
  - ▶ trovare il "grafo" e le dipendenze delle varie parti del codice



- ▶ Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- ▶ Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - ▶ trovare quelle parti dove viene speso gran parte del tempo
  - ▶ trovare il "grafo" e le dipendenze delle varie parti del codice
  - ▶ trovare i punti "critici" e i "colli-di-bottiglia" del nostro codice



- ▶ Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- ▶ Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - ▶ trovare quelle parti dove viene speso gran parte del tempo
  - ▶ trovare il "grafo" e le dipendenze delle varie parti del codice
  - ▶ trovare i punti "critici" e i "colli-di-bottiglia" del nostro codice
- ▶ Non è sempre semplice (specie in codici scientifici o di comunità) fare una stima a- priori dei punti di cui sopra.



- ▶ Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- ▶ Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - ▶ trovare quelle parti dove viene speso gran parte del tempo
  - ▶ trovare il "grafo" e le dipendenze delle varie parti del codice
  - ▶ trovare i punti "critici" e i "colli-di-bottiglia" del nostro codice
- ▶ Non è sempre semplice (specie in codici scientifici o di comunità) fare una stima a- priori dei punti di cui sopra.
- ▶ L'idea è dunque quella di iniziare da un "Profiling" della nostra applicazione che essenzialmente consiste nel monitoraggio del nostro codice nel momento stesso in cui viene eseguito.



- Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:





- ▶ Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - ▶ semplicità o meno di utilizzo



- ▶ Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - ▶ semplicità o meno di utilizzo
  - ▶ proprietari o di pubblico dominio



- ▶ Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - ▶ semplicità o meno di utilizzo
  - ▶ proprietari o di pubblico dominio
  - ▶ intrusivi o no



- ▶ Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - ▶ semplicità o meno di utilizzo
  - ▶ proprietari o di pubblico dominio
  - ▶ intrusivi o no
  - ▶ etc, etc



- ▶ Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - ▶ semplicità o meno di utilizzo
  - ▶ proprietari o di pubblico dominio
  - ▶ intrusivi o no
  - ▶ etc, etc
- ▶ Partiamo dai più semplici per arrivare a quelli più complessi, con l'idea che tutte le informazioni che raccogliamo possano essere utilizzate complessivamente per migliorare le prestazioni della nostra applicazione.



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

Motivazioni

**time**

top

gprof

Papi

Scalasca

Consigli finali



- Presente in tutte le architetture *Unix /Linux*.



- Presente in tutte le architetture *Unix /Linux*.
- Fornisce il tempo totale di esecuzione di un programma ed altre utili informazioni.





- ▶ 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**).



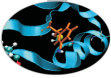
- ▶ Presente in tutte le architetture *Unix /Linux*.
- ▶ Fornisce il tempo totale di esecuzione di un programma ed altre utili informazioni.
- ▶ Non ha bisogno che il programma sia compilato con particolari opzioni di compilazione (**assolutamente non intrusivo**).
- ▶ **time** <nome\_eseguibile>



- ▶ 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**).
- ▶ **time <nome\_eseguibile>**

un tipico output:

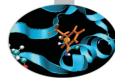
```
[lanucara@louis ~/CORSO2013]$ /usr/bin/time ./a.out<realloc.in
9.29user 6.19system 0:15.52elapsed 99%CPU (0avgtext+0avgdata 18753424maxresident)k
0inputs+0outputs (0major+78809minor)pagefaults 0swaps
```



9.29u

1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.

time: output



9.29u 6.19s

1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione del programma.



9.29u 6.19s 0:15.52

1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione del programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").



9.29u 6.19s 0:15.52 99%

1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione del programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").
4. La percentuale di CPU impiegata nel processo.



9.29u 6.19s 0:15.52 99% 0avgtext+0avgdata  
18753424maxresident)k

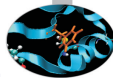
1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione del programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").
4. La percentuale di CPU impiegata nel processo.
5. parametri relativi all' area dati (complessiva) del processo eseguibile (in Kbytes).





9.29u 6.19s 0:15.52 99% 0avgtext+0avgdata  
18753424maxresident)k 0inputs+0outputs

1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione del programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").
4. La percentuale di CPU impiegata nel processo.
5. parametri relativi all' area dati (complessiva) del processo eseguibile (in Kbytes).
6. parametri relativi all'input/output (interi).



9.29u 6.19s 0:15.52 99% 0avgtext+0avgdata  
18753424maxresident)k 0inputs+0outputs 0major+78809minor

1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione del programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").
4. La percentuale di CPU impiegata nel processo.
5. parametri relativi all' area dati (complessiva) del processo eseguibile (in Kbytes).
6. parametri relativi all'input/output (interi).
7. L'uso di page-faults (interi).



- L'uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:



- ▶ L'uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - ▶ (Lo *"user" time* è confrontabile con il *"sys" time*)



- ▶ 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%)



- ▶ 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)*



- ▶ 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'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)





- ▶ 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 più 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.



- Per questo codice *System time*  $\sim$  *User time*.



- ▶ Per questo codice *System time*  $\sim$  *User time*.
- ▶ è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.



- ▶ Per questo codice *System time*  $\sim$  *User time*.
- ▶ è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.  
il programma "alloca" e "dealloca" nel corso dell'esecuzione una serie di matrici: questa cosa è **altamente sconsigliata**.





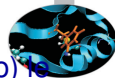
- Per questo codice  $System\ time \sim User\ time$ .
- è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.  
il programma "alloca" e "dealloca" nel corso dell'esecuzione una serie di matrici: questa cosa è **altamente sconsigliata**.
- $System\ time + User\ time \sim Elapsed\ time$



- ▶ Per questo codice  $System\ time \sim User\ time$ .
- ▶ è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.  
il programma "alloca" e "dealloca" nel corso dell'esecuzione una serie di matrici: questa cosa è **altamente sconsigliata**.
- ▶  $System\ time + User\ time \sim Elapsed\ time$   
non era presente alcun altro processo a contendere l'uso delle risorse.

## time: Analisi output

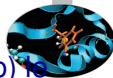
Cambiando la struttura del programma (eliminando le allocazioni e deallocazioni durante l'esecuzione dello stesso) le cose migliorano drasticamente:



```
[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
```

## time: Analisi output

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

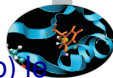


```
[lanucara@louis ~/CORSO2013]$ /usr/bin/time ./a.out<realloc.in
2.28user 0.38system 0:02.67elapsed 99%CPU (0avgtext+0avgdata 9378352maxresident)k
0inputs+0outputs (0major+3153minor)pagefaults 0swaps
```

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

## time: Analisi output

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

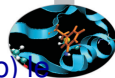


```
[lanucara@louis ~/CORSO2013]$ /usr/bin/time ./a.out<realloc.in
2.28user 0.38system 0:02.67elapsed 99%CPU (0avgtext+0avgdata 9378352maxresident)k
0inputs+0outputs (0major+3153minor)pagefaults 0swaps
```

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

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

## time: Analisi output



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*.

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

Il problema è che risulta difficile, se non impossibile, estrarre qualcosa di interessante da simulazioni "grand-challenge". Vediamo un esempio di output di una versione operativa del codice di meteorologia COSMO, un'ora di simulazione su 48 processori ("cores") di PLX:

```
12973.38user 1915.82system 20:55.80elapsed 1185%CPU (0avgtext+0avgdata 2597648maxresident)k
19608inputs+10649880outputs (147major+223489935minor)pagefaults 0swaps
```



Il run relativo a COSMO ci da un'informazione interessante  
(1185 %CPU):



Il run relativo a COSMO ci da un'informazione interessante (*1185 %CPU*):

- La percentuale di utilizzo della CPU è molto maggiore del 100% (la cosa non dovrebbe sorprendervi, dato che stiamo utilizzando 48 cores di PLX).





Il run relativo a COSMO ci da un'informazione interessante (*1185 %CPU*):

- La percentuale di utilizzo della CPU è molto maggiore del 100% (la cosa non dovrebbe sorprendervi, dato che stiamo utilizzando 48 cores di PLX).
- Per questo codice parallelo dunque, tralasciando il *System time*, lo *User time* risulta pari allo *elapsed time* moltiplicato per un fattore che dipende dalla percentuale di utilizzo della CPU.



Il run relativo a COSMO ci da un'informazione interessante (1185 %CPU):

- La percentuale di utilizzo della CPU è molto maggiore del 100% (la cosa non dovrebbe sorprendervi, dato che stiamo utilizzando 48 cores di PLX).
- Per questo codice parallelo dunque, tralasciando il *System time*, lo *User time* risulta pari allo *elapsed time* moltiplicato per un fattore che dipende dalla percentuale di utilizzo della CPU.
- complessivamente questo fattore è molto al di sotto del numero di processori utilizzati



Il run relativo a COSMO ci da un'informazione interessante (*1185 %CPU*):

- ▶ La percentuale di utilizzo della CPU è molto maggiore del 100% (la cosa non dovrebbe sorprendervi, dato che stiamo utilizzando 48 cores di PLX).
- ▶ Per questo codice parallelo dunque, tralasciando il *System time*, lo *User time* risulta pari allo *elapsed time* moltiplicato per un fattore che dipende dalla percentuale di utilizzo della CPU.
- ▶ complessivamente questo fattore è molto al di sotto del numero di processori utilizzati
- ▶ L'efficienza del codice parallelo non è dunque buona.



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

Motivazioni

time

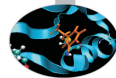
**top**

gprof

Papi

Scalasca

Consigli finali



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.



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 (o insieme di macchine) su cui stiamo in esecuzione e se altri utenti stanno contendendo le nostre stesse risorse (cores, I/O, rete, etc).



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 (o insieme di macchine) 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 - 14:57:46 up 19 days, 23:19, 38 users,  load average: 4.38, 1.68, 0.73
Tasks: 449 total,   3 running, 442 sleeping,   3 stopped,   1 zombie
Cpu(s): 39.3%us,  0.9%sy,  0.0%ni, 59.7%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:   24725848k total, 11623572k used, 13102276k free,  124732k buffers
Swap: 15999960k total,   96420k used, 15903540k free,  8921564k cached
  
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21524	lanucara	20	0	2407m	1.5g	4880	R	860.9	6.3	0:26.85	mm_mkl
21450	fferre	20	0	115m	6752	1640	R	99.0	0.0	0:27.21	parseBlastout.p
21485	lanucara	20	0	17400	1572	976	R	0.7	0.0	0:00.04	top
416	root	20	0	0	0	0	S	0.3	0.0	14:55.00	rpciod/0
424	root	20	0	0	0	0	S	0.3	0.0	0:27.90	rpciod/8
442	root	15	-5	0	0	0	S	0.3	0.0	2:59.49	kslowd001
450	root	20	0	0	0	0	S	0.3	0.0	22:58.02	nfsiod
8430	paoletti	20	0	114m	2116	1040	S	0.3	0.0	0:01.43	sshd
9522	nobody	20	0	167m	13m	1020	S	0.3	0.1	14:54.15	gmond
20338	tbiagini	20	0	114m	1920	872	S	0.3	0.0	0:00.04	sshd
26365	lanucara	20	0	149m	3384	2088	S	0.3	0.0	0:01.82	xterm
26395	lanucara	20	0	17396	1568	972	S	0.3	0.0	0:29.53	top
1	root	20	0	21444	1112	932	S	0.0	0.0	0:05.37	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.45	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:08.27	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:05.73	ksoftirqd/0





Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

**Profilers**

Motivazioni

time

top

**gprof**

Papi

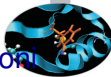
Scalasca

Consigli finali

# gprof: caratteristiche principali

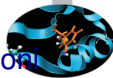


## gprof: caratteristiche principali



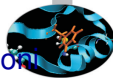
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.

## gprof: caratteristiche principali



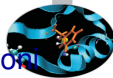
- ▶ time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).

## gprof: caratteristiche principali



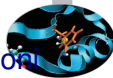
- ▶ time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- ▶ in prima battuta può servire uno strumento che ci dia delle informazioni maggiormente connesse con il nostro codice e che sia "portabile" attraverso piattaforme differenti.

## gprof: caratteristiche principali



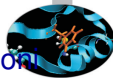
- ▶ **time** è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- ▶ in prima battuta può servire uno strumento che ci dia delle informazioni maggiormente connesse con il nostro codice e che sia "portabile" attraverso piattaforme differenti.
- ▶ **gprof**, che fa parte del pacchetto GNU, soddisfa ovviamente il requisito di portabilità.
- ▶ Caratteristiche principali:

## gprof: caratteristiche principali



- ▶ **time** è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- ▶ in prima battuta può servire uno strumento che ci dia delle informazioni maggiormente connesse con il nostro codice e che sia "portabile" attraverso piattaforme differenti.
- ▶ **gprof**, che fa parte del pacchetto GNU, soddisfa ovviamente il requisito di portabilità.
- ▶ Caratteristiche principali:
  - ▶ limitatamente intrusivo

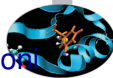
## gprof: caratteristiche principali



- ▶ **time** è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- ▶ in prima battuta può servire uno strumento che ci dia delle informazioni maggiormente connesse con il nostro codice e che sia "portabile" attraverso piattaforme differenti.
- ▶ **gprof**, che fa parte del pacchetto GNU, soddisfa ovviamente il requisito di portabilità.
- ▶ Caratteristiche principali:
  - ▶ limitatamente intrusivo
  - ▶ fornisce informazioni a livello di "subroutine" e/o funzioni

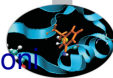


## gprof: caratteristiche principali



- ▶ **time** è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- ▶ in prima battuta può servire uno strumento che ci dia delle informazioni maggiormente connesse con il nostro codice e che sia "portabile" attraverso piattaforme differenti.
- ▶ **gprof**, che fa parte del pacchetto GNU, soddisfa ovviamente il requisito di portabilità.
- ▶ Caratteristiche principali:
  - ▶ limitatamente intrusivo
  - ▶ fornisce informazioni a livello di "subroutine" e/o funzioni
  - ▶ fornisce informazioni sul "grafo" e delle dipendenze della nostra applicazione

# gprof: caratteristiche principali



- ▶ **time** è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- ▶ chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- ▶ in prima battuta può servire uno strumento che ci dia delle informazioni maggiormente connesse con il nostro codice e che sia "portabile" attraverso piattaforme differenti.
- ▶ **gprof**, che fa parte del pacchetto GNU, soddisfa ovviamente il requisito di portabilità.
- ▶ Caratteristiche principali:
  - ▶ limitatamente intrusivo
  - ▶ fornisce informazioni a livello di "subroutine" e/o funzioni
  - ▶ fornisce informazioni sul "grafo" e delle dipendenze della nostra applicazione
  - ▶ basato sia sul concetto di "Sampling" che di "Instrumentation"

# Gprof "Sampling"



# Gprof "Sampling"



- La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.

# Gprof "Sampling"



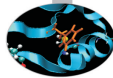
- ▶ La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- ▶ Gprof si basa su un **Time Based Sampling** : ad intervalli di tempo fissati si interroga il "program counter" per individuare a quale punto del codice è arrivata l'esecuzione

# Gprof "Sampling"

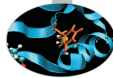


- ▶ La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- ▶ Gprof si basa su un **Time Based Sampling** : ad intervalli di tempo fissati si interroga il "program counter" per individuare a quale punto del codice è arrivata l'esecuzione
- ▶ Tipicamente, il "program counter" viene interrogato un certo numero di volte (supponiamo 100 per fissare le idee) per secondo di "run-time", ma questo numero cambia da macchina a macchina.

# Gprof "Sampling"



- ▶ La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- ▶ Gprof si basa su un **Time Based Sampling** : ad intervalli di tempo fissati si interroga il "program counter" per individuare a quale punto del codice è arrivata l'esecuzione
- ▶ Tipicamente, il "program counter" viene interrogato un certo numero di volte (supponiamo 100 per fissare le idee) per secondo di "run-time", ma questo numero cambia da macchina a macchina.
- ▶ Il "Sampling", essendo di fatto una approssimazione statistica, dipende fortemente dal "sampling period"



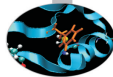
- ▶ La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- ▶ Gprof si basa su un **Time Based Sampling** : ad intervalli di tempo fissati si interroga il "program counter" per individuare a quale punto del codice è arrivata l'esecuzione
- ▶ Tipicamente, il "program counter" viene interrogato un certo numero di volte (supponiamo 100 per fissare le idee) per secondo di "run-time", ma questo numero cambia da macchina a macchina.
- ▶ Il "Sampling", essendo di fatto una approssimazione statistica, dipende fortemente dal "sampling period"
- ▶ Il vantaggio è che essendo meno intrusivo, l'esecuzione non dovrebbe risentire eccessivamente dell'uso del profiling.



# Gprof "Instrumentation"



# Gprof "Instrumentation"



- essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.



- ▶ essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.
- ▶ può risultare, come detto, maggiormente invasivo e pertanto inficiare le prestazioni del nostro eseguibile di partenza.

# Gprof "Instrumentation"



- ▶ essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.
- ▶ può risultare, come detto, maggiormente invasivo e pertanto inficiare le prestazioni del nostro eseguibile di partenza.
- ▶ per quanto riguarda Gprof, questa tecnica è guidata dal compilatore e questo dovrebbe garantire una certa efficienza.

# Gprof "Instrumentation"



- ▶ essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.
- ▶ può risultare, come detto, maggiormente invasivo e pertanto inficiare le prestazioni del nostro eseguibile di partenza.
- ▶ per quanto riguarda Gprof, questa tecnica è guidata dal compilatore e questo dovrebbe garantire una certa efficienza.
- ▶ viene usata da Gprof per tutto quello che concerne le "chiamate a funzioni" all'interno del nostro codice.



- ▶ Per "attivare" Gprof, occorre compilare e linkare il codice (scritto in Fortran, C, C++) con l'opzione -pg
- ▶ Utilizzo  

```
<compiler> -pg programma.f -o nome_eseguibile  
./nome_eseguibile  
gprof nome_eseguibile
```



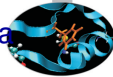
- ▶ Per "attivare" Gprof, occorre compilare e linkare il codice (scritto in Fortran, C, C++) con l'opzione -pg
- ▶ Utilizzo  
`<compiler> -pg programma.f -o nome_eseguibile`  
`./nome_eseguibile`  
`gprof nome_eseguibile`
- ▶ dopo un run andato a buon fine (da non sottovalutare!!!), viene generato il file **gmon.out**



- ▶ Per "attivare" Gprof, occorre compilare e linkare il codice (scritto in Fortran, C, C++) con l'opzione -pg
- ▶ Utilizzo  
`<compiler> -pg programma.f -o nome_eseguibile`  
`./nome_eseguibile`  
`gprof nome_eseguibile`
- ▶ dopo un run andato a buon fine (da non sottovalutare!!!), viene generato il file **gmon.out**
- ▶ Attenzione, i "vecchi" files gmon.out sono sovrascritti in esecuzioni successive.



# gprof: Flat profile



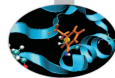
- **Flat profile:** mostra il tempo totale che il programma impiega nell'eseguire ogni subroutine/funzione, che viene ordinata rispetto alla percentuale del tempo totale speso.
- Vediamolo per un semplice programma C:

```

#include <stdio.h>
int a(void) {
    int i=0,g=0;
    while(i++<100000)
    {
        g+=i;
    }
    return g;
}
int b(void) {
    int i=0,g=0;
    while(i++<400000)
    {
        g+=i;
    }
    return g;
}
int main(int argc, char** argv)
{
    int iterations;

    if(argc != 2)
    {
        printf("Usage %s <No of Iterations>\n", argv[0]);
        exit(-1);
    }

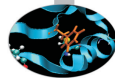
```



## ► Flat profile: continua....

```
else
    iterations = atoi(argv[1]);
printf("No of iterations = %d\n", iterations);

while (iterations--)
{
    a();
    b();
}
```

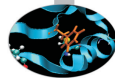


- **Flat profile:** continua....

```
else
    iterations = atoi(argv[1]);
printf("No of iterations = %d\n", iterations);

while (iterations--)
{
    a();
    b();
}
```

- compiliamo e linkiamo il sorgente C con Gprof e analizziamo il Flat profile.



- **Flat profile:** continua....

```
else
    iterations = atoi(argv[1]);
printf("No of iterations = %d\n", iterations);

while (iterations--)
{
    a();
    b();
}
```

- compiliamo e linkiamo il sorgente C con Gprof e analizziamo il Flat profile.
- ci aspettiamo che la routine **b()** pesi circa 4 volte la routine **a()**:



```

/usr/bin/time ./Main\ example.exe 10000
No of iterations = 10000
3.22user 0.00system 0:03.23elapsed 99%CPU (0avgtext+0avgdata 1760maxresident)k
0inputs+0outputs (0major+131minor)pagefaults 0swaps
  
```

```

gcc -O Main\ example.c -o Main\ example_gprof.exe -pg
[lanucara@louis ~]$ /usr/bin/time ./Main\ example_gprof.exe 10000
No of iterations = 10000
3.33user 0.00system 0:03.34elapsed 99%CPU (0avgtext+0avgdata 2064maxresident)k
0inputs+8outputs (0major+150minor)pagefaults 0swaps
  
```

```
gprof ./Main\ example_gprof.exe > Main\ example.gprof
```

Flat profile:

Each sample counts as 0.01 seconds.

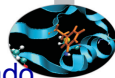
%	cumulative	self	calls	self	total	
time	seconds	seconds		us/call	us/call	name
81.43	2.73	2.73	10000	272.78	272.78	b
19.60	3.38	0.66	10000	65.67	65.67	a



1. Il tempo percentuale (rispetto al totale) impiegato dalla subroutine.
2. Il tempo cumulativo impiegato nella subroutine.
3. Il tempo in secondi impiegato nella subroutine.
4. Il numero delle volte in cui la subroutine viene chiamata.
5. Il tempo medio di ogni singola chiamata della subroutine (in msec).
6. Il tempo medio totale per chiamata (include anche il tempo impiegato da tutte le subroutine "figlie") in msec.
7. Il nome della subroutine.

Per questo esempio non sono presenti subroutine "figlie" per cui "self" e "total" coincidono.

## gprof: flat profile



Complichiamo leggermente il codice precedente introducendo una semplice function:

```
int cinsideb(int d) {  
    {  
    }  
    return d;  
}
```

che collochiamo all'interno della routine **b()** al posto del calcolo di g:

```
int b(void) {  
    int i=0,g=0;  
    while (i++<4000000)  
    {  
        g+=cinsideb(i);  
    }  
    return g;  
}
```

Mandiamo in esecuzione il nuovo eseguibile abilitando Gprof e analizziamo il nuovo Flat profile



## Nuovo Flat profile:

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
44.72	3.28	3.28	10000	327.78	604.55	b
37.76	6.05	2.77	4000000000	0.00	0.00	cinsideb
18.53	7.40	1.36	10000	135.86	135.86	a

## Commenti:





## Nuovo Flaf profile:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
44.72	3.28	3.28	10000	327.78	604.55	b
37.76	6.05	2.77	4000000000	0.00	0.00	cinsideb
18.53	7.40	1.36	10000	135.86	135.86	a

## Commenti:

- ▶ insieme le routines **b()** e **cinsideb()** "pesano" per l'80% del totale.



## Nuovo Flaf profile:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
44.72	3.28	3.28	10000	327.78	604.55	b
37.76	6.05	2.77	4000000000	0.00	0.00	cinsideb
18.53	7.40	1.36	10000	135.86	135.86	a

## Commenti:

- ▶ insieme le routines **b()** e **cinsideb()** "pesano" per l'80% del totale.
- ▶ questa volta il contributo della function "figlia" **cinsideb()** di **b()** va ad accrescere il tempo "total" di **b()**



## Nuovo Flat profile:

Flat profile:

Each sample counts as 0.01 seconds.

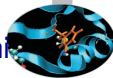
% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
44.72	3.28	3.28	10000	327.78	604.55	b
37.76	6.05	2.77	4000000000	0.00	0.00	cinsideb
18.53	7.40	1.36	10000	135.86	135.86	a

## Commenti:

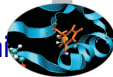
- ▶ insieme le routines **b()** e **cinsideb()** "pesano" per l'80% del totale.
- ▶ questa volta il contributo della function "figlia" **cinsideb()** di **b()** va ad accrescere il tempo "total" di **b()**
- ▶ Per questo codice Gprof risulta molto piu intrusivo dato che la function **cinsideb()** viene chiamata un numero enorme di volte.

## gprof: call tree profile

Mostra il tempo che il programma impiega nell'eseguire ogni subroutine/funzione e quelle da esse chiamate.



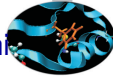
## gprof: call tree profile



Mostra il tempo che il programma impiega nell'eseguire ogni subroutine/funzione e quelle da esse chiamate.

Le subroutine/funzioni sono ordinate in base al tempo totale speso in esse ed in quelle chiamate.

# gprof: call tree profile



Mostra il tempo che il programma impiega nell'eseguire ogni subroutine/funzione e quelle da esse chiamate.

Le subroutine/funzioni sono ordinate in base al tempo totale speso in esse ed in quelle chiamate.

Vediamo il call tree profile sull'ultima versione del codice:

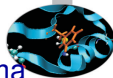
Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.14% of 7.40 seconds

index	% time	self	children	called	name	
[1]	100.0	0.00	7.40		main [1]	<spontaneous>
			3.28	2.77	10000/10000	b [2]
			1.36	0.00	10000/10000	a [4]
-----						
			3.28	2.77	10000/10000	main [1]
[2]	81.7	3.28	2.77	10000	b [2]	
			2.77	0.00	4000000000/4000000000	cinsideb [3]
-----						
			2.77	0.00	4000000000/4000000000	b [2]
[3]	37.4	2.77	0.00	4000000000	cinsideb [3]	
-----						
			1.36	0.00	10000/10000	main [1]
[4]	18.3	1.36	0.00	10000	a [4]	
-----						
...						



1. Un indice che definisce univocamente ogni elemento del "Call graph" precedente.
2. Il peso percentuale della subroutine e delle sue "figlie" rispetto al totale.
3. Il tempo totale impiegato nella subroutine.
4. Il tempo totale impiegato nelle "figlie" della subroutine.
5. Il numero delle volte che la subroutine è stata chiamata nella sezione del "Call graph" rispetto al numero delle volte che viene chiamata nell'intero codice.
6. Il nome della subroutine.



Consideriamo a titolo esemplificativo un semplice programma che realizza il classico prodotto matrice-matrice in due modi:

1. linka una chiamate alle librerie MKL di sistema (ottimizzate per l'architettura e parallele)
2. oppure utilizza una libreria costruita compilando e linkando le BLAS sulla macchina target

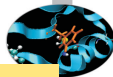
Vediamo il risultato, in termini di "Flat Profile", ottenuto con il profiling delle due versioni del codice:  
versione con le MKL:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
71.43	0.10	0.10				for_simd_random_number
14.29	0.12	0.02	1	20.00	20.00	MAIN__
14.29	0.14	0.02				__intel_memset
0.00	0.14	0.00	4	0.00	0.00	timing_module_mp_timing_





## versione con le BLAS:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
97.76	6.10	6.10				dgemm_
1.60	6.20	0.10				for_simd_random_number
0.32	6.22	0.02	1	20.00	20.00	MAIN__
0.32	6.24	0.02				__intel_memset
0.00	6.24	0.00	4	0.00	0.00	timing_module_mp_timing_

Commenti:

- ▶ Per la versione MKL Gprof non ci fornisce alcuna informazione utile, se non chiamate a funzioni di libreria ausiliarie.
- ▶ La versione BLAS correttamente riporta la chiamata alla funzione *dgemm\_* che è responsabile della quasi totalità del tempo.
- ▶ Fortunatamente possiamo ritenere che le funzioni di libreria come le MKL siano già ottimizzate e pertanto l'uso di Gprof è assolutamente superfluo.

# gprof: altre limitazioni



## gprof: altre limitazioni



- Gprof ha "granularità abbastanza grande. Per codici complessi non è semplice capire, anche partendo da una data routine/function, dove mettere le mani per migliorare le prestazioni del codice.

## gprof: altre limitazioni



- ▶ Gprof ha "granularità abbastanza grande. Per codici complessi non è semplice capire, anche partendo da una data routine/function, dove mettere le mani per migliorare le prestazioni del codice.
- ▶ Gprof può risultare molto intrusivo. Verificare, sperimentalmente, che l'esecuzione del codice senza Gprof sia "confrontabile" con quella con Gprof.

## gprof: altre limitazioni



- ▶ Gprof ha "granularità abbastanza grande. Per codici complessi non è semplice capire, anche partendo da una data routine/function, dove mettere le mani per migliorare le prestazioni del codice.
- ▶ Gprof può risultare molto intrusivo. Verificare, sperimentalmente, che l'esecuzione del codice senza Gprof sia "confrontabile" con quella con Gprof.
- ▶ Tutti i tempi che sono confrontabili con il "sampling period" non dovrebbero essere tenuti in considerazione.



- ▶ Profiling di un codice seriale per la risoluzione di un'Equazione alle Derivate Parziali.
  - ▶ Utilizzare il tool Gprof per effettuare un Profiling del codice al variare della taglia del problema.
  - ▶ Eseguire i seguenti comandi:

```
cp /gpfs/scratch/userinternal/planucar/CORSO2013/gprof.tar.gz .  
tar xvfz gprof.tar.gz  
cd GPROF/JACOBI
```

- ▶ leggere il file README
- ▶ eseguire i test
- ▶ commentare i risultati del Profiling

# Funzioni per misurare il tempo



# Funzioni per misurare il tempo



- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.



# Funzioni per misurare il tempo



- ▶ dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- ▶ è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.

# Funzioni per misurare il tempo



- ▶ dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- ▶ è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.
- ▶ ovviamente la "portabilità e la generalità possono lasciare a desiderare (specie per codici complessi o "terze parti").  
Qualche esempio:

# Funzioni per misurare il tempo



- ▶ dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- ▶ è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.
- ▶ ovviamente la "portabilità e la generalità possono lasciare a desiderare (specie per codici complessi o "terze parti").  
Qualche esempio:
  - ▶ `etime()`, `dtime()` (Fortran 77)

# Funzioni per misurare il tempo



- ▶ dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- ▶ è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.
- ▶ ovviamente la "portabilità e la generalità possono lasciare a desiderare (specie per codici complessi o "terze parti").  
Qualche esempio:

- ▶ `etime(),dtime()` (Fortran 77)
- ▶ `cputime(),system_clock(), date_and_time()` (Fortran 90)

# Funzioni per misurare il tempo



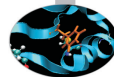
- ▶ dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- ▶ è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.
- ▶ ovviamente la "portabilità e la generalità possono lasciare a desiderare (specie per codici complessi o "terze parti").  
Qualche esempio:

- ▶ `etime(),dtime()` (Fortran 77)
- ▶ `cputime(),system_clock(), date_and_time()` (Fortran 90)
- ▶ `clock()` (C/C++)
- ▶ ...



```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
clock_t time1, time2;
double dub_time;
int main(){
    int i,j,k,nn=1000;
    double c[nn][nn],a[nn][nn],b[nn][nn];
    ...
    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);
    ...
    return 0;
}
```

# Un esempio Fortran



```

real(8)::a(1000,1000),b(1000,1000),c(1000,1000)
real(8) ::t1,t2
integer :: time_array(8)
a=0;b=0;c=0;n=1000
...
...
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
...
...
end
    
```



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

**Profilers**

Motivazioni

time

top

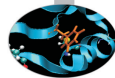
gprof

**Papi**

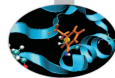
Scalasca

Consigli finali

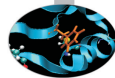




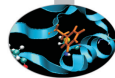
- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.



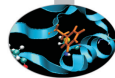
- ▶ Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- ▶ Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.



- ▶ Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- ▶ Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.
- ▶ PAPI (Performance Application Programming Interface) nasce esattamente con questo scopo. Caratteristiche principali:



- ▶ Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- ▶ Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.
- ▶ PAPI (Performance Application Programming Interface) nasce esattamente con questo scopo. Caratteristiche principali:
  - ▶ portabilità sulla maggior parte delle architetture Linux, Windows, etc inclusi acceleratori (NVIDIA, Intel MIC, etc)



- ▶ Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- ▶ Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.
- ▶ PAPI (Performance Application Programming Interface) nasce esattamente con questo scopo. Caratteristiche principali:
  - ▶ portabilità sulla maggior parte delle architetture Linux, Windows, etc inclusi acceleratori (NVIDIA, Intel MIC, etc)
  - ▶ si basa sull'utilizzo dei cosiddetti *Hardware Counters*: un insieme di registri "special-purpose" che misurano determinati eventi durante l'esecuzione del nostro programma.



- PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:



- ▶ PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - ▶ *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)



- ▶ PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - ▶ *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)
  - ▶ *Low level interface*, che fornisce informazioni specifiche dell'hardware a disposizione per indagini maggiormente sofisticate. Molto più complessa da usare.





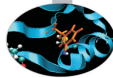
- ▶ PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - ▶ *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)
  - ▶ *Low level interface*, che fornisce informazioni specifiche dell'hardware a disposizione per indagini maggiormente sofisticate. Molto più complessa da usare.
- ▶ Occorre verificare il numero di *Hardware Counters* disponibili sulla macchina. Questo numero ci fornisce la misura del numero di eventi che possono essere monitorati in contemporanea.



- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate
  - ▶ PAPI\_FP\_INS - istruzioni floating-point



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate
  - ▶ PAPI\_FP\_INS - istruzioni floating-point
  - ▶ PAPI\_L1\_DCA - accessi in cache L1



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate
  - ▶ PAPI\_FP\_INS - istruzioni floating-point
  - ▶ PAPI\_L1\_DCA - accessi in cache L1
  - ▶ PAPI\_L1\_DCM - cache misses L1





- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate
  - ▶ PAPI\_FP\_INS - istruzioni floating-point
  - ▶ PAPI\_L1\_DCA - accessi in cache L1
  - ▶ PAPI\_L1\_DCM - cache misses L1
  - ▶ PAPI\_SR\_INS - istruzioni di store



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate
  - ▶ PAPI\_FP\_INS - istruzioni floating-point
  - ▶ PAPI\_L1\_DCA - accessi in cache L1
  - ▶ PAPI\_L1\_DCM - cache misses L1
  - ▶ PAPI\_SR\_INS - istruzioni di store
  - ▶ PAPI\_TLB\_DM - TLB misses



- ▶ Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- ▶ PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - ▶ PAPI\_TOT\_CYC - numero di cicli totali
  - ▶ PAPI\_TOT\_INS - numero di istruzioni completate
  - ▶ PAPI\_FP\_INS - istruzioni floating-point
  - ▶ PAPI\_L1\_DCA - accessi in cache L1
  - ▶ PAPI\_L1\_DCM - cache misses L1
  - ▶ PAPI\_SR\_INS - istruzioni di store
  - ▶ PAPI\_TLB\_DM - TLB misses
  - ▶ PAPI\_BR\_MSP - conditional branch mispredicted



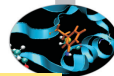
Le chiamate alla libreria di alto livello sono piuttosto intuitive. Sebbene PAPI sia scritto in C è possibile chiamare le funzioni di libreria anche da codici Fortran.

un esempio in Fortran:

```

#include "fpapi_test.h"
... ; integer events(2), retval ; integer*8 values(2)
... ;
events(1) = PAPI_FP_INS ; events(2) = PAPI_L1_DCM
...
    call PAPIf_start_counters(events, 2, retval)
    call PAPIf_read_counters(values, 2, retval) ! Clear values
        [sezione di codice da monitorare]
    call PAPIfstop_counters(values, 2, retval)
    print*, 'Floating point instructions: ', values(1)
    print*, ' L1 Data Cache Misses: ', values(2)
...
  
```

# Papi: interfaccia C e Fortran



## un esempio in C:

```
%\begin{lstlisting}
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"

#define NUM_EVENTS 2
#define THRESHOLD 10000
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: \n",
retval, __FILE__, __LINE__); exit(retval); }
...
/* stupid codes to be monitored */
void computation_mult()
{
    ....
}

int main()
{
    int Events[2] = {PAPI_TOT_INS, PAPI_TOT_CYC};
    long long values[NUM_EVENTS];
    ...
    if ( (retval = PAPI_start_counters(Events, NUM_EVENTS)) != PAPI_OK)
        ERROR_RETURN(retval);
    printf("\nCounter Started: \n");
    computation_add();
    ...
    if ( (retval=PAPI_read_counters(values, NUM_EVENTS)) != PAPI_OK)
        ERROR_RETURN(retval);
    printf("Read successfully\n");
    printf("The total instructions executed for addition are %lld \n",values[0]);
    printf("The total cycles used are %lld \n", values[1] );
    ...
}
```

# PAPI: tutte le funzioni di alto livello



- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:

# PAPI: tutte le funzioni di alto livello



- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili

# PAPI: tutte le funzioni di alto livello



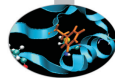
- ▶ Un piccolo insieme di routines che servono a "strumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate





- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate
  - ▶ PAPI\_flops - floating point operation rate

# PAPI: tutte le funzioni di alto livello



- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate
  - ▶ PAPI\_flops - floating point operation rate
  - ▶ PAPI\_ipc - instructions per cycle e time

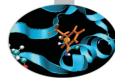


- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate
  - ▶ PAPI\_flops - floating point operation rate
  - ▶ PAPI\_ipc - instructions per cycle e time
  - ▶ PAPI\_accum\_counters

# PAPI: tutte le funzioni di alto livello



- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate
  - ▶ PAPI\_flops - floating point operation rate
  - ▶ PAPI\_ipc - instructions per cycle e time
  - ▶ PAPI\_accum\_counters
  - ▶ PAPI\_read\_counters - read and reset counters



- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate
  - ▶ PAPI\_flops - floating point operation rate
  - ▶ PAPI\_ipc - instructions per cycle e time
  - ▶ PAPI\_accum\_counters
  - ▶ PAPI\_read\_counters - read and reset counters
  - ▶ PAPI\_start\_counters - start counting hw events



- ▶ Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - ▶ PAPI\_num\_counters - ritorna il numero di hw counters disponibili
  - ▶ PAPI\_flips - floating point instruction rate
  - ▶ PAPI\_flops - floating point operation rate
  - ▶ PAPI\_ipc - instructions per cycle e time
  - ▶ PAPI\_accum\_counters
  - ▶ PAPI\_read\_counters - read and reset counters
  - ▶ PAPI\_start\_counters - start counting hw events
  - ▶ PAPI\_stop\_counters - stop counters return current counts



- ▶ Profiling con PAPI del codice seriale che utilizza chiamate alle BLAS per alcune operazioni fondamentali di "linear algebra".
  - ▶ Eseguire i seguenti comandi:

```
cp /afs/icaspur.it/project/open/space/papiprof.tar.gz .  
tar -xvfz papiprof.tar.gz  
cd PAPI
```

- ▶ leggere il file README
- ▶ eseguire i test
- ▶ commentare i risultati del Profiling con PAPI
- ▶ se rimane del tempo instrumentare il codice per valutare le prestazioni degli altri nuclei computazionali presenti nel test e commentare i risultati.



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

Motivazioni

time

top

gprof

Papi

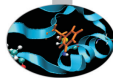
**Scalasca**

Consigli finali



# Scalasca: caratteristiche

- Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.



# Scalasca: caratteristiche



- ▶ Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- ▶ In realtà nasce come il successore di un altro tool di successo (KOJAK)
- ▶ È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).

# Scalasca: caratteristiche



- ▶ Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- ▶ In realtà nasce come il successore di un altro tool di successo (KOJAK)
- ▶ È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).
- ▶ Utilizzabile sulla maggior parte dei sistemi High Performance Computing (HPC) con decine di migliaia di "cores"....

# Scalasca: caratteristiche



- ▶ Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- ▶ In realtà nasce come il successore di un altro tool di successo (KOJAK)
- ▶ È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).
- ▶ Utilizzabile sulla maggior parte dei sistemi High Performance Computing (HPC) con decine di migliaia di "cores"....
- ▶ ...ma anche su architetture parallele "medium-size"
- ▶ È un prodotto "open-source", continuamente aggiornato e mantenuto da Juelich.
- ▶ Il sito: [www.scalasca.org](http://www.scalasca.org)



- Supporta applicazioni scritte in Fortran, C e C++.

# Scalasca: caratteristiche



- Supporta applicazioni scritte in Fortran, C e C++.
- In pratica, Scalasca consente due tipi di analisi:

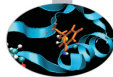


- ▶ Supporta applicazioni scritte in Fortran, C e C++.
- ▶ In pratica, Scalasca consente due tipi di analisi:
  - ▶ una modalità "**summary**" che consente di ottenere informazioni aggregate per la nostra applicazione (ma sempre dettagliate a livello di singola istruzione)
  - ▶ pause una modalità "**tracing**" che è "process-local" e che consente di raccogliere molte più informazioni ma che può risultare particolarmente onerosa dal punto di vista dell'uso di risorse di "storage"



- ▶ Supporta applicazioni scritte in Fortran, C e C++.
- ▶ In pratica, Scalasca consente due tipi di analisi:
  - ▶ una modalità **"summary"** che consente di ottenere informazioni aggregate per la nostra applicazione (ma sempre dettagliate a livello di singola istruzione)
  - ▶ pause una modalità **"tracing"** che è "process-local" e che consente di raccogliere molte più informazioni ma che può risultare particolarmente onerosa dal punto di vista dell'uso di risorse di "storage"
- ▶ Dopo l'esecuzione dell'eseguibile (strumentato) Scalasca è in grado di caricare i files in memoria ed analizzarli in parallelo usando lo stesso numero di "cores" della nostra applicazione.





L'intero processo avviene in tre passi:



L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):



L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):

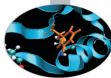
```
ifort -openmp [altre_opzioni]
```

```
<codice_sorgente>
```



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):

`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`

`<codice_sorgente>`

`mpif90 [opzioni] <codice_sorgente>`



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`



L'intero processo avviene in tre passi:

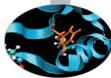
- **Compilazione** (il codice sorgente viene "instrumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`
- **Esecuzione:**



L'intero processo avviene in tre passi:

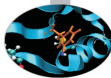
- **Compilazione** (il codice sorgente viene "strumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`
- **Esecuzione:**  
`<codice_eseguibile>`





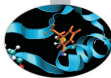
L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "strumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`
- **Esecuzione:**  
`scalasca -analyze [opzioni_scalasca] <codice_eseguibile>`



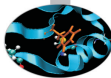
L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`
- **Esecuzione:**  
`scalasca -analyze [opzioni_scalasca] <codice_eseguibile>`  
`mpirun [opzioni] <codice_eseguibile>`



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`
- **Esecuzione:**  
`scalasca -analyze [opzioni_scalasca] <codice_eseguibile>`  
`scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>`



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):

```
scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]
```

```
<codice_sorgente>
```

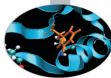
```
scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
```

- **Esecuzione:**

```
scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
```

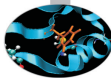
```
scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>
```

Viene creata una directory **epik\_[caratteristiche]**



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):  
`scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]`  
`<codice_sorgente>`  
`scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>`
- **Esecuzione:**  
`scalasca -analyze [opzioni_scalasca] <codice_eseguibile>`  
`scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>`  
Viene creata una directory `epik_[caratteristiche]`
- **Analisi risultati:**



L'intero processo avviene in tre passi:

- **Compilazione** (il codice sorgente viene "instrumentato"):

```
scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni]
```

```
<codice_sorgente>
```

```
scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
```

- **Esecuzione:**

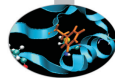
```
scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
```

```
scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>
```

Viene creata una directory **epik\_[caratteristiche]**

- **Analisi risultati:**

```
scalasca -examine [opzioni_scalasca] epik_[caratteristiche]
```



- Codice sismologia elementi finiti (fem.F).

# Scalasca: un esempio



- ▶ Codice sismologia elementi finiti (fem.F).
- ▶ Parallelizzato utilizzando il tool OpenMP.

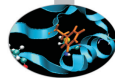


# Scalasca: un esempio



- ▶ Codice sismologia elementi finiti (fem.F).
- ▶ Parallelizzato utilizzando il tool OpenMP.
- ▶ Eseguito su 8 processori (parallelismo "moderato") su singolo nodo

# Scalasca: un esempio



- ▶ Codice sismologia elementi finiti (fem.F).
- ▶ Parallelizzato utilizzando il tool OpenMP.
- ▶ Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- ▶ Compilatore intel (comando ifort -O3 -free -openmp...)

# Scalasca: un esempio



- ▶ Codice sismologia elementi finiti (fem.F).
- ▶ Parallelizzato utilizzando il tool OpenMP.
- ▶ Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- ▶ Compilatore intel (comando ifort -O3 -free -openmp...)
- ▶ Alcuni numeri:



- ▶ Codice sismologia elementi finiti (fem.F).
- ▶ Parallelizzato utilizzando il tool OpenMP.
- ▶ Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- ▶ Compilatore intel (comando ifort -O3 -free -openmp...)
- ▶ Alcuni numeri:
  - ▶ Numeri dei nodi della griglia 2060198.



- ▶ Codice sismologia elementi finiti (fem.F).
- ▶ Parallelizzato utilizzando il tool OpenMP.
- ▶ Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- ▶ Compilatore intel (comando ifort -O3 -free -openmp...)
- ▶ Alcuni numeri:
  - ▶ Numeri dei nodi della griglia 2060198.
  - ▶ Numeri degli elementi non nulli della matrice 57638104.

# Scalasca: compilazione ed esecuzione



```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp *.F
```



```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp *.F
```

```
[ruggiero@neo258 EXE]$ scalasca -analyze ./fem.x
```

# Scalasca: compilazione ed esecuzione



```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp *.F
```

```
[ruggiero@neo258 EXE]$ scalasca -analyze ./fem.x
```

```
S=C=A=N: Scalasca 1.2.2 runtime summarization
S=C=A=N: Abort: measurement blocked by existing archive ./epik_fem_Ox8_sum
[ruggiero@neo258 EXE]$ rm -r epik_fem_Ox8_sum/
[ruggiero@neo258 EXE]$ scalasca -analyze ./fem.x
S=C=A=N: Scalasca 1.2.2 runtime summarization
S=C=A=N: ./epik_fem_Ox8_sum experiment archive
S=C=A=N: Thu Jan  7 16:03:56 2010: Collect start
./fem.x
[.0]EPIK: Activated ./epik_fem_Ox8_sum [NO TRACE]
   tri  25.6178519725800          seconds
S=C=A=N: Thu Jan  7 16:04:23 2010: Collect done (status=130) 27s
Abort: incomplete experiment ./epik_fem_Ox8_sum
.....
```



# Scalasca: analisi dei risultati



```
[ruggiero@neo258 EXE]$ scalasca -examine -s ./epik_fem_Ox8_sum/
```



```
[ruggiero@neo258 EXE]$ scalasca -examine -s ./epik_fem_Ox8_sum/
```

```
cube3_score ./epik_fem_Ox8_sum/epitome.cube
Reading epik_fem_Ox8_sum/epitome.cube... done.
.....
Estimated aggregate size of event trace (total_tbc): 1102182744 bytes
Estimated size of largest process trace (max_tbc): 1090211280 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
or reduce requirements using file listing names of USR regions to be filtered.)
```

flt	type	max_tbc	time	%	region
	ANY	1090211280	6142.73	100.00	(summary) ALL
	OMP	1684464	4324.02	70.39	(summary) OMP
	COM	17184	385.86	6.28	(summary) COM
	USR	1088536224	1432.86	23.33	(summary) USR

# Scalasca output: breve descrizione



## 1. Per tipologia:

# Scalasca output: breve descrizione



## 1. Per tipologia:

- ▶ **ANY**: tutte le routines che compongono il programma

# Scalasca output: breve descrizione



## 1. Per tipologia:

- ▶ **ANY**: tutte le routines che compongono il programma
- ▶ **OMP**: quelle che contengono costrutti di parallelizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).

# Scalasca output: breve descrizione



## 1. Per tipologia:

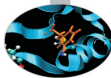
- ▶ **ANY**: tutte le routines che compongono il programma
- ▶ **OMP**: quelle che contengono costrutti di parallelizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
- ▶ **COM**: tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.



# 1. Per tipologia:

- ▶ **ANY**: tutte le routines che compongono il programma
- ▶ **OMP**: quelle che contengono costrutti di parallelizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
- ▶ **COM**: tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
- ▶ **USR**: quelle che sono coinvolte in operazioni puramente locali al processo.

# Scalasca output: breve descrizione



## 1. Per tipologia:

- ▶ **ANY**: tutte le routines che compongono il programma
- ▶ **OMP**: quelle che contengono costrutti di parallelizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
- ▶ **COM**: tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
- ▶ **USR**: quelle che sono coinvolte in operazioni puramente locali al processo.

## 2. La massima capacità del trace-buffer richiesta (misurata in in bytes).



# Scalasca output: breve descrizione



## 1. Per tipologia:

- ▶ **ANY**: tutte le routines che compongono il programma
- ▶ **OMP**: quelle che contengono costrutti di parallelizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
- ▶ **COM**: tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
- ▶ **USR**: quelle che sono coinvolte in operazioni puramente locali al processo.

2. La massima capacità del trace-buffer richiesta (misurata in in bytes).

3. Il tempo impiegato (in secondi) per l'esecuzione di quella parte di codice.

# Scalasca output: breve descrizione

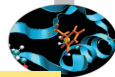


## 1. Per tipologia:

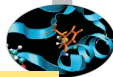
- ▶ **ANY**: tutte le routines che compongono il programma
- ▶ **OMP**: quelle che contengono costrutti di parallelizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
- ▶ **COM**: tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
- ▶ **USR**: quelle che sono coinvolte in operazioni puramente locali al processo.

2. La massima capacità del trace-buffer richiesta (misurata in in bytes).
3. Il tempo impiegato (in secondi) per l'esecuzione di quella parte di codice.
4. La percentuale del tempo impiegato, rispetto a quello totale, per la sua esecuzione.

# Scalasca output: maggior dettaglio



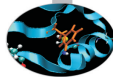
```
[ruggiero@neo258 EXE]$ cube3_score -r epik_fem_Ox8_sum/summary.cube.gz
```



```
[ruggiero@neo258 EXE]$ cube3_score -r epik_fem_Ox8_sum/summary.cube.gz
```

```
...
  USR      889293768      68.51      1.12 expand_
  USR      98889504      9.96      0.16 ordinamento_
  USR      98747208      12.60      0.21 elem2d_
  USR      730224      0.05      0.00 elem_ij_      OMP      349200      0.48
0.01 !$omp do @solutore_parallelo.F:157
  OMP      349200      0.34      0.01 !$omp ibarrier @solutore_parallelo.F:163
  USR      171840      0.03      0.00 dwalltime00_
  USR      171840      0.05      0.00 dwalltime00
  USR      142224      0.01      0.00 abc03_bis_
  USR      142224      0.01      0.00 abc03_ter_
  USR      85896      1.39      0.02 printtime_
  USR      85896      0.02      0.00 inittime_
  OMP      51480      19.53      0.32 !$omp ibarrier @fem.F:2554
  OMP      51480      196.73      3.20 !$omp do @fem.F:2548
  OMP      51480      88.14      1.43 !$omp ibarrier @fem.F:2540
  OMP      51480      1555.91      25.33 !$omp do @fem.F:2532
  OMP      34320      18.10      0.29 !$omp ibarrier @fem.F:2742
  OMP      31460      0.27      0.00 !$omp parallel @fem.F:2511
  OMP      31460      0.17      0.00 !$omp parallel @fem.F:2725
  OMP      31460      0.41      0.01 !$omp parallel @fem.F:2589
  OMP      31460      0.38      0.01 !$omp parallel @solutore_parallelo.F:40
...
....
```

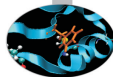
# Aggiunta di "strumentazione" manuale



```

        program fem
        implicit none
#include "epik_user.inc"
...
...
...
EPIK_USER_REG(r_write, "<<write>>")
        real*8, allocatable :: csi(:), eta(:)
...
...
EPIK_USER_START(r_write)
        do i=1,13
            write(i+5000,*) t,csi(2*p(i)-1),eta(2*p(i)-1)
            write(i+6000,*) t,csi(2*p(i)),eta(2*p(i))
        end do
EPIK_USER_END(r_write)
...
...
...
        end
    
```

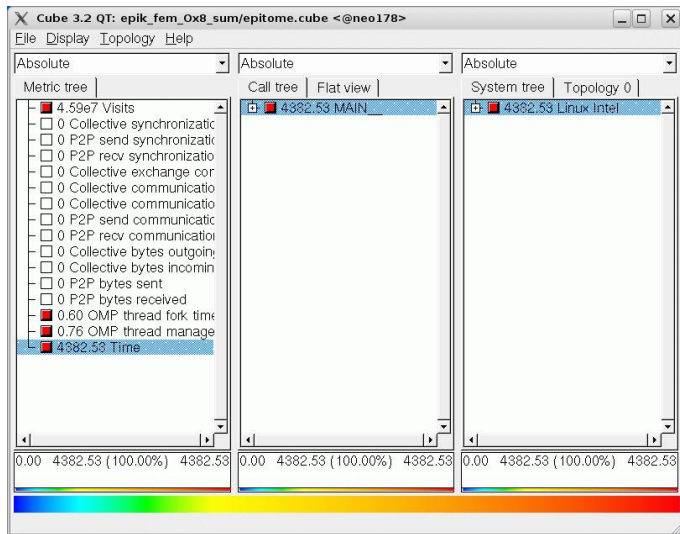
# Scalasca output: maggior dettaglio - 2



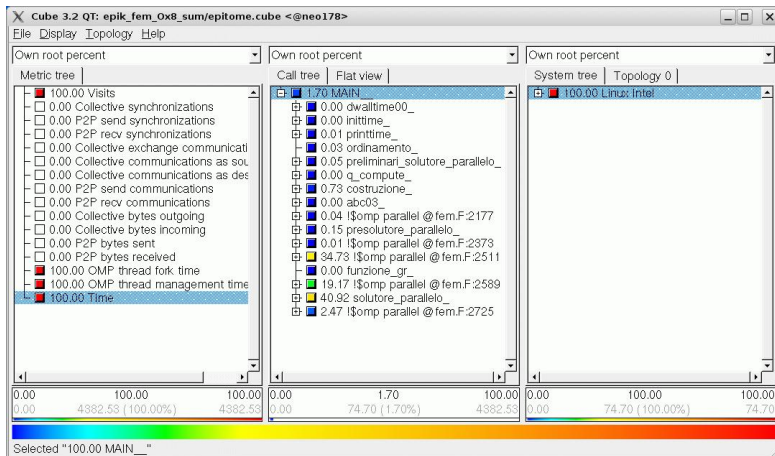
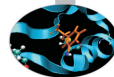
```

OMP      23280      0.21      0.00 !$omp ibarrier @solutore_parallelo.F:86
OMP      23280      0.04      0.00 !$omp single @solutore_parallelo.F:84
OMP      23280     53.29      0.87 !$omp do @solutore_parallelo.F:89
OMP      23280      4.37      0.07 !$omp ibarrier @solutore_parallelo.F:99
OMP      23280     899.44     14.64 !$omp do @solutore_parallelo.F:104
USR      23280     931.42     15.16 sol_
OMP      23280     106.11      1.73 !$omp ibarrier @solutore_parallelo.F:133
OMP      23280     41.25      0.67 !$omp do @solutore_parallelo.F:142
OMP      23280      2.67      0.04 !$omp ibarrier @solutore_parallelo.F:150
OMP      23280      0.05      0.00 !$omp single @solutore_parallelo.F:172
OMP      23280      0.73      0.01 !$omp ibarrier @solutore_parallelo.F:174
USR      17160      3.89      0.06 <<sint>>
OMP      17160      1.43      0.02 !$omp do @solutore_parallelo.F:42
OMP      17160     64.71      1.05 !$omp do @solutore_parallelo.F:47
OMP      17160      7.68      0.12 !$omp ibarrier @solutore_parallelo.F:55
OMP      17160     36.40      0.59 !$omp do @solutore_parallelo.F:56
OMP      17160      6.45      0.11 !$omp ibarrier @solutore_parallelo.F:66
OMP      17160     15.30      0.25 !$omp do @solutore_parallelo.F:67
USR      17160      3.08      0.05 <<write>>
OMP      17160      3.18      0.05 !$omp ibarrier @solutore_parallelo.F:81
OMP      17160     95.11      1.55 !$omp do @fem.F:2726
OMP      17160      0.03      0.00 !$omp ibarrier @solutore_parallelo.F:182
OMP      17160      0.84      0.01 !$omp ibarrier @solutore_parallelo.F:180
OMP      17160      0.01      0.00 !$omp single @solutore_parallelo.F:178

```

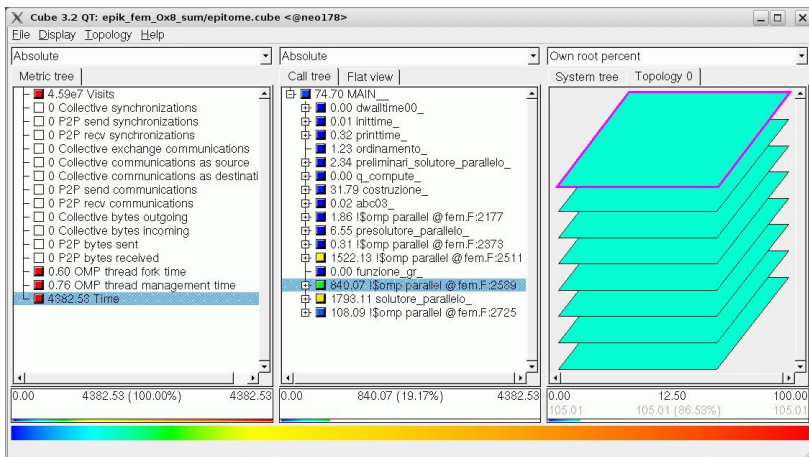
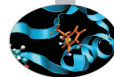


# Scalasca: output grafico

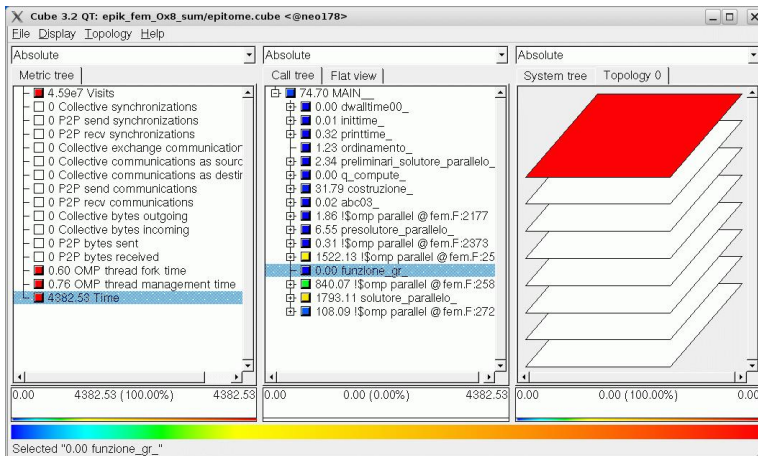
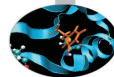




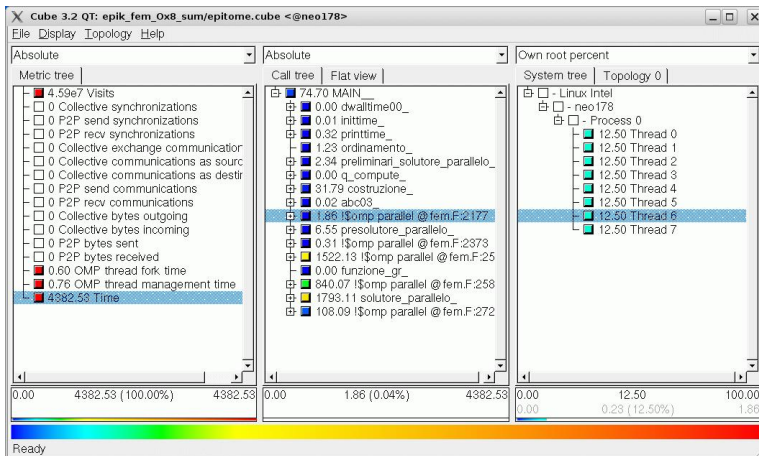
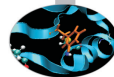
# Scalasca: output grafico



# Scalasca: output grafico



# Scalasca: output grafico





Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

Motivazioni

time

top

gprof

Papi

Scalasca

**Consigli finali**

# Profiling...





- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":





- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":
  - ▶ usare più casi test cercando di attivare tutte le parti del codice



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":
  - ▶ usare più casi test cercando di attivare tutte le parti del codice
  - ▶ scegliere casi test il più possibile realistici



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":
  - ▶ 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



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":
  - ▶ 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



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":
  - ▶ 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 una analisi iniziale)



- ▶ la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ▶ ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- ▶ alcune considerazioni "pratiche":
  - ▶ 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 una analisi iniziale)
  - ▶ usare, se possibile, architetture differenti.