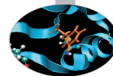
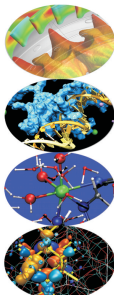
CINECA SCAI
SuperComputing Applications and Innovation



HPC Scientific programming: tools and techniques

F. Affinito

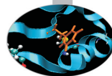
CINECA Roma - SCAI Department



Roma, 4-6 November 2013

CINECA

AGENDA DEL CORSO



4 novembre 2013

- Architetture
- La cache ed il sistema di memoria
- Pipeline
- Profilers

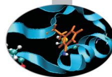
5 novembre 2013

- Compilatori
- Librerie
- Floating-point

6 novembre 2013

- Makefile
- Debugging

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

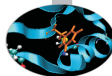
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

Floating Point Computing

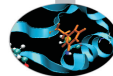
Quanto possono variare le prestazioni?



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

Qual è il margine di manovra?



Motivation: Performance

"Parallel hardware needs parallel programming"

Programmability!

Performance

Hardware Potential

Application Performance

Parallelism

No Parallelism

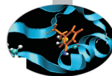
GHz Era

Multicore Era

Manycore Era

Time

Software development



Problem

Solution method

Algorithms

Programming

Source code

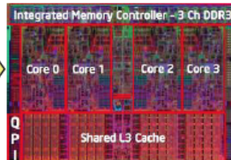
```
#include <stdio.h>
#define SIZE 1000
main(int argc, char** argv) {
  int A[SIZE], B[SIZE], C[SIZE];
  int i;

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

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

Compiling

Compiled and optimized code

```
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $12004,%esp
        pushl %edi
        pushl %esi
        pushl %ebx
        nop
        movl $0,-12004(%ebp)
.L2:
        cmpl $999,-12004(%ebp)
        jle .L6
        jmp .L3
        .align 4
.L6:
        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 -8000(%ebp),%ebx
        movl -12004(%ebp),%esi
        movl %esi,%edi
        leal 0(,%edi,4),%esi
        leal -12000(%ebp),%edi
        movl (%ecx,%ebx),%ecx
        imull (%esi,%edi),%ecx
        movl %ecx,(%eax,%edx)
.L4:
        incl -12004(%ebp)
        jmp .L2
        .align 4
.L1:
        leal -12016(%ebp),%esp
        popl %ebx

.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.8.1"
```

Execution

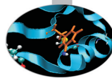


Result

Output

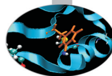
```
Hierarchical code: Plummer model

nbody     dtime      eps     theta   usequad    dtout     tstop
 1024   0.03125   0.0250     1.00     false    0.2500    2.0000

 tnow       T+U      T/U     nttot     nbavg     ncavg   cputime
0.000   -0.2527  -0.4943    203186        84       114      0.00

         cm pos   0.0000   -0.0000    0.0000
         cm vel  -0.0000    0.0000    0.0000
         am vec   0.0057    0.0196   -0.0222

 tnow       T+U      T/U     nttot     nbavg     ncavg   cputime
0.031   -0.2527  -0.4940    202260        81       116      0.01

         cm pos   0.0000   -0.0000    0.0000
         cm vel   0.0000   -0.0000    0.0000
         am vec   0.0057    0.0196   -0.0222

Time is:    3.6 seconds
```

Algoritmi



- Fondamentale è la scelta dell'algoritmo
  - algoritmo efficiente $\rightarrow$ buone prestazioni
  - algoritmo inefficiente $\rightarrow$ cattive prestazioni
- Se l'algortimo 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!!!

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

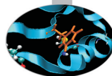
Profilers

Compilatori e ottimizzazione

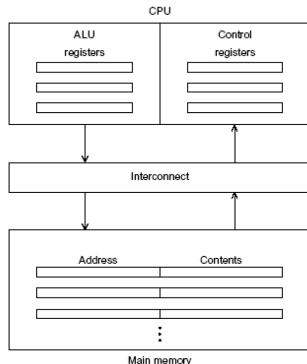
Librerie scientifiche

Floating Point Computing

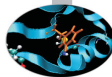
Architettura di Von Neuman



- 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)

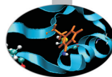


Architettura di Von Neuman



- 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

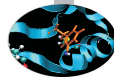
Evoluzione dei sistemi



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

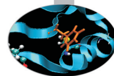
- Caching
  Memorie molto veloci presenti sul chip del processore.
  Esistono cache di primo, secondo e terzo livello.
- Virtual memory
  Sistema sviluppato per fare in modo che la RAM funzioni come una cache per lo storage di grosse moli di dati.
- Instruction level parallelism
  Tecnica utilizzata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

Classificazione di Flynn

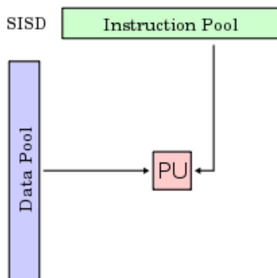


- Racchiude le architetture dei computer in base alla molteplicitá dell'hardware usato per manipolare lo stream di istruzioni e dati
  - SISD:single instruction, single data. Corrisponde alla classica achitettura di von Neumann, sistema scalare monoprocessore.
  - SIMD:single instruction, multiple data. 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.

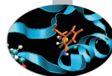
SISD



- Il classico sistema di von Neumann. Calcolatori con una sola unitá 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.

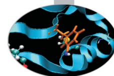


SIMD



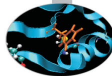
- La stessa istruzione viene eseguita in parallelo su dati differenti.
  - modello computazionale generalmente sincrono
- Processori vettoriali
  - molte ALU
  - registri vettoriali
  - Unitá Load/Store vettoriali
  - Istruzioni vettoriali
  - Memoria interleaved
  - OpenMP, MPI
- Graphical Processing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

MIMD



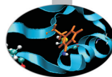
- Molteplici streams di istruzioni eseguiti simultaneamente su molteplici streams di dati
  - modello computazionale asincrono
- Cluster
  - molti nodi di calcolo (centinaia/migliaia)
  - più processori multicore per nodo
  - RAM condivisa sul nodo
  - RAM distribuita fra i nodi
  - livelli di memoria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

MIMD:Fermi Blue Gene/Q



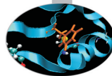
IBM-BlueGene /Q
Architecture: 10 BGQ Frame with 2 MidPlanes each
Front-end Nodes OS: Red-Hat EL 6.2
Compute Node Kernel: lightweight Linux-like kernel
Processor Type: IBM PowerA2, 16 cores, 1.6 GHz
Computing Nodes: 10.240
Computing Cores: 163.840
RAM: 16GB / node
Internal Network: Network interface
with 11 links ->5D Torus
Disk Space: more than 2PB of scratch space
Peak Performance: 2.1 PFlop/s

Cluster CPU GPU

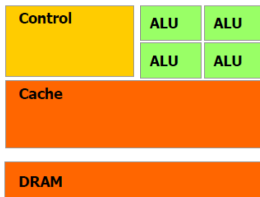


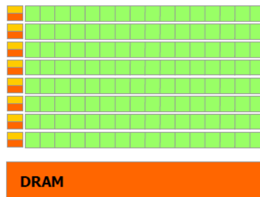
- Soluzioni ibride CPU multi-core + GPU many-core:
  - ogni nodo di calcolo è dotato di processori multicore e schede grafiche con processori dedicati per il GPU computing
  - notevole potenza di calcolo teorica sul singolo nodo
  - ulteriore strato di memoria dato dalla memoria delle GPU
  - OpenMP,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

CPU vs GPU

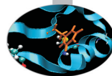


- Le CPU sono processori general purpose in grado di risolvere qualsiasi algoritmo
  - threads in grado di gestire qualsiasi operazione ma pesanti, al massimo 1 thread per core computazionale.
- Le GPU sono processori specializzati per problemi che possono essere classificati come «intense data-parallel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

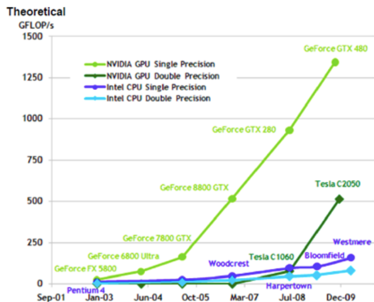




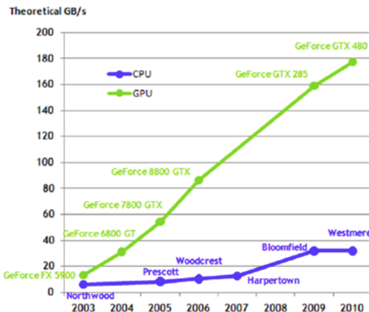
Multicore vs Manycore



- Una nuova direzione di sviluppo per l'architettura dei microprocessori:
  - incrementare la potenza di calcolo complessiva tramite l'aumento del numero di unitá 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.

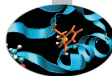


Numero di operazioni in virgola mobile al secondo per la CPU e la GPU



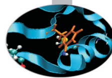
Larghezza di banda della memoria

INTEL MIC



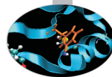
- Coprocessore Intel Xeon Phi
  - Basato su architettura Intel Many Integrated Core (MIC)
  - 60 core/1,053 GHz/240 thread
  - 8 GB di memoria e larghezza di banda di 320 GB/s
  - 1 TFLOPS di prestazioni di picco a doppia precisione
  - Istruzioni SIMD a 512 bit
  - Approcci tradizionali come MPI, OpenMP

Bandwidth



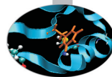
- La velocità con la quale è possibile trasferire i dati tra la memoria e il processore
- Si misura in numero di bytes che si possono traferire al secondo (Mb/s, Gb/s, etc..)
- A = B * C
  - leggere dalla memoria il dato B
  - leggere dalla memoria il dato C
  - calcolare il prodotto B * C
  - salvare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

Stream

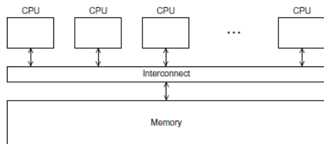


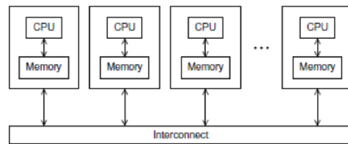
- Benchmark per la misura della bandwidth da e per la CPU
- Misura il tempo per
  - Copia a $\rightarrow$ c (copy)
  - Copia a*b $\rightarrow$ c (scale)
  - Somma a+b $\rightarrow$ c (add)
  - Somma a+b*c $\rightarrow$ d (triad)
- Misura della massima bandwith
- http://www.cs.virginia.edu/stream/ref.html

Condivisa Distribuita



- 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.

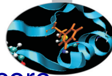




- I moderni sistemi multicore hanno memoria condivisa sul nodo e distribuita fra i nodi.

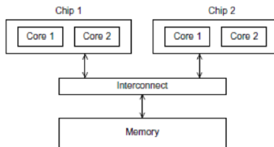
CINECA SCAI SuperComputing Applications and Innovation

Memoria condivisa

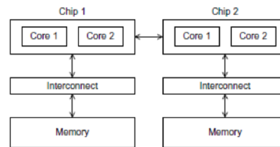


Nelle architetture a memoria condivisa con processori multicore vi sono generalmente due tipologie di accesso alla memoria principale

- 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.



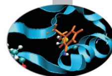
UMA



NUMA

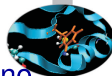
CINECA

UMA NUMA



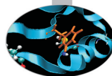
- Problemi principali:
  - se un processo o thread viene trasferito da una CPU ad un'altra, va perso tutto il lavoro speso per usare bene la cache
  - macchine UMA: processi "memory intensive" su più CPU contendono per il bus, rallentandosi a vicenda
  - macchine NUMA: codice eseguito da una CPU con 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

RETI



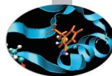
- Tutti i sistemi caratterizzati da elevate potenze di calcolo sono composti da diversi nodi, a memoria condivisa sul singolo nodo e distribuita fra i nodi
  - i nodi sono collegati fra di loro da topologie di interconnessione più o meno complesse e costose
- Le reti di interconnessione commerciali maggiormente 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

www.top500.org



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - 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	**Sequoia** - 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	**Fermi** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	163840	1725.5	2097.2	822

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

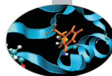
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

Floating Point Computing

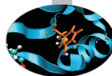
Il sistema di memoria



- Capacità di calcolo delle CPU 2× ogni 18 mesi
- Velocità di accesso alla RAM 2× 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

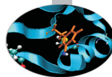
La gerachia di memoria



Processor	CPU PROCESSOR REGISTER	SUPER FAST SUPER EXPENSIVE TINY CAPACITY
	CPU CACHE LEVEL 1 (L1) CACHE LEVEL 2 (L2) CACHE LEVEL 3 (L3) CACHE	FASTER EXPENSIVE SMALL CAPACITY
EDO, SD-RAM, DDR-SDRAM, RD-RAM and More...	PHYSICAL MEMORY RAMDOM ACCESS MEMORY (RAM)	FAST PRICED REASONABLY AVERAGE CAPACITY
SSD, Flash Drive	SOLID STATE MEMORY NON-VOLATILE FLASH-BASED MEMORY	AVERAGE SPEED PRICED REASONABLY AVERAGE CAPACITY
Mechanical Hard Drives	VIRTUAL MEMORY FILE-BASED MEMORY	SLOW CHEAP LARGE CAPACTITY

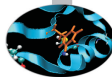
▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

La Cache



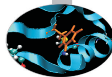
Perché questa gerarchia?

La Cache



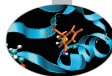
Perché questa gerarchia?
Non servono tutti i dati disponibili subito

La Cache



Perché questa gerarchia?
Non servono tutti i dati disponibili subito
La soluzione?

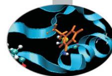
La Cache



Perché questa gerarchia?
Non servono tutti i dati disponibili subito
La soluzione?

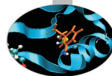
- La cache è composta di uno (o più livelli) di memoria intermedia, abbastanza veloce ma piccola (kB $\div$ MB)
- Principio fondamentale: si lavora sempre su un sottoinsieme ristretto dei dati
  - dati che servono $\rightarrow$ nella memoria ad accesso veloce
  - dati che (per ora) non servono $\rightarrow$ 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 $\rightarrow$ gerachia di livelli intermedi.

La Cache



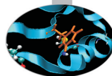
- La CPU accede al (più alto) livello di cache:
- Il controllore della cache determina se l'elemento richiesto è effettivamente presente in cache:
  - Si: trasferimento fra cache e CPU
  - No: Carica il nuovo dato in cache; se la cache è piena, applica la politica di rimpiazzamento per caricare il nuovo 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)

Rimpiazzamento:principi di località



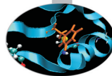
- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).

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.

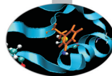
Rimpiazzamento:principi di località



- 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.

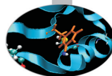
- Località temporale dei dati
  - vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)

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.

Rimpiazzamento:principi di località

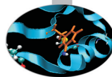


- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - Viene sfruttata leggendo normalmente più dati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

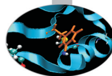
Dato richiesto dalla CPU viene mantenuto in cache insieme a celle di memoria contigue il più a lungo possibile.

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-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.

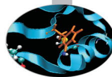
Cache: qualche stima quantitativa



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

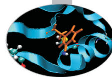


1. cerco due dati, A e B
2. cerco A nella cache di primo livello (L1) O(1) cicli
3. cerco A nella cache di secondo livello (L2) O(10) cicli
4. copio A dalla RAM alla L2 alla L1 ai registri O(10) cicli
5. cerco B nella cache di primo livello (L1) O(1) cicli
6. cerco B nella cache di secondo livello (L2) 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!!!

SCAI SuperComputing Applications and Innovation

Cache hit in tutti i livelli

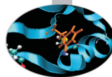


- cerco i due dati, A e B
- cerco A nella cache di primo livello(L1) — O(1) cicli
- cerco B nella cache di primo livello(L1) — O(1) cicli
- eseguo l'operazione richiesta

O(1) cicli di overhead

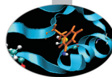
CINECA

SRAM vs. DRAM



- 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

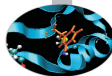
Misura prestazioni:esempio



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

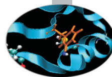
- Ad ogni iterazione viene eseguita una somma ed una moltiplicazione floating-point
- Il numero di operazioni è $2 \times n$

Tempo di esecuzione T_{es}



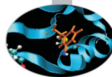
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo

Tempo di esecuzione T_{es}



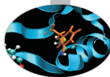
- $T_{es} = N_{flop} * t_{flop}$
- N_{flop} → Algoritmo
- t_{flop} → Hardware

Tempo di esecuzione T_{es}



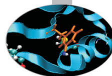
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo
- $t_{flop} \rightarrow$ Hardware
- Considera solamente il tempo di esecuzione in memoria
- Trascuro qualcosa?

Tempo di esecuzione T_{es}



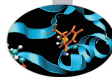
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo
- $t_{flop} \rightarrow$ Hardware
- Considera solamente il tempo di esecuzione in memoria
- Trascuro qualcosa?
- t_{mem} il tempo di accesso in memoria.

Per cui . . .



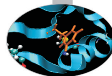
- $T_{es} = N_{flop} * t_{flop} + N_{mem} * t_{mem}$
- $t_{mem} \rightarrow$ Hardware
- Qual è l'impatto di N_{mem} sulle performance?

Impatto di N_{mem}



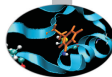
- $Perf = \frac{N_{Flop}}{T_{es}}$
- per $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- per $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- Fattore di decadimento delle performance
- $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- Come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

Impatto di N_{mem}



- $Perf = \frac{N_{Flop}}{T_{es}}$
- 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 accesssi alla memoria.

Località spaziale:ordine di accesso

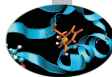


- Prodotto di matrici in doppia precisione 512X512
- MFlops misurati su louis (Intel(R) Xeon(R) CPU X5660 2.80GHz)
- compilatore gfortran con ottimizzazione -O0

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.

Layout degli array in memoria

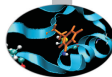


- Memoria $\rightarrow$ sequenza lineare di locazioni elementari
- Matrice A, elemento a_{ij}: i indice di riga, j indice di colonna
- Le matrici sono rappresentate con array

- Come sono memorizzati gli elementi di un array?

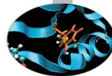
- C: in successione seguendo l'ultimo indice, poi il precedente . . .
  a[1][1] a[1][2] a[1][3] a[1][4] . . .
  a[1][n] a[2][1] . . . a[n][n]
- Fortran: in successione seguendo il primo indice, poi il secondo . . .
  a(1,1) a(2,1) a(3,1) a(4,1) . . .
  a(n,1) a(1,2) . . . a(n,n)

Lo stride

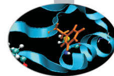


- È la distanza tra due dati successivamente acceduti
  - stride=1 $\rightarrow$ sfrutto la località spaziale
  - stride » 1 $\rightarrow$ non sfrutto la località spaziale
- Regola d'oro
  - Accedere sempre, se possibile,a stride unitario

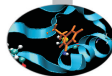
Ordine di memorizzazione:Fortran



Ordine di memorizzazione:C

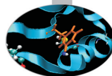


Ordine ottimale di accesso



- 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;i<n,j++1)
          d[i] = a[i] + b [i][j]*c[j];
    ```

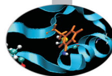
Ordine di accesso



- 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

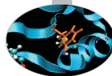
Località spaziale: sistema lineare



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

Località spaziale: sistema lineare

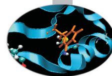


Soluzione sistema triangolare

- $Lx = b$
- 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?

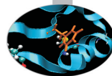
Forward substitution



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
...
```

Forward substitution



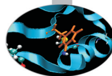
Soluzione:

```
...
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
...
```

```
[vruggie1@fen07 TRI]$ ./a.out
```

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

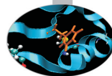
Partizionamento della matrice



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



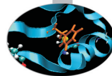
Soluzione:

```
...
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
...
```

```
[vruggie1@fen07 TRI]$ ./a.out
```

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

Cosa è cambiato



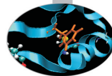
- Forward substitution

```
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
```

Cosa è cambiato



- Forward substitution

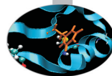
```
do i = 1, n
    do j = 1, i-1
       b(i) = b(i) - L(i,j) b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
```

- 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

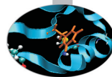
In C è memorizzata:

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

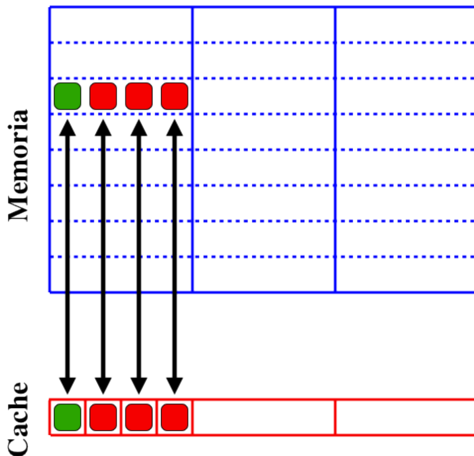
In Fortran è memorizzata:

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

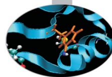
Località spaziale: righe di cache



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



Dimensione e riutilizzo dei dati

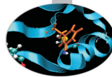


- Prodotto matrice-matrice in doppia precisione
- Versioni alternative, differenti chiamate della libreria BLAS
- Prestazioni in MFlops su Intel(R) Xeon(R) CPU X5660 2.80GHz

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

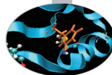
Stesso numero di operazioni, l’uso della cache cambia!!!

I registri



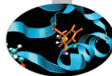
- I registri sono locazioni di memoria interne alla CPU
- poche (tipicamente < 128), ma con latenza nulla
- Tutte le operazioni delle unità di calcolo:
  - prendono i loro operandi dai registri
  - riportano i risultati in registri
- i trasferimenti memoria $\leftrightarrow$ registri sono fasi separate
- 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 . . .



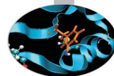
```
do 3000 z=1,nz
         k3=beta(z)
         do 3000 y=1,ny
              k2=eta(y)
              do 3000 x=1,nx/2
                   hr(x,y,z,1)=hr(x,y,z,1)*norm
                   hi(x,y,z,1)=hi(x,y,z,1)*norm
                   hr(x,y,z,2)=hr(x,y,z,2)*norm
                   hi(x,y,z,2)=hi(x,y,z,2)*norm
                   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



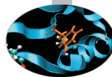
```
do 3000 z=1,nz
         k3=beta(z)
         do 3000 y=1,ny
              k2=eta(y)
              do 3000 x=1,nx/2
                   br1=hr(x,y,z,1)*norm
                   bi1=hi(x,y,z,1)*norm
                   br2=hr(x,y,z,2)*norm
                   bi2=hi(x,y,z,2)*norm
                   br3=hr(x,y,z,3)*norm
                   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*bi1+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)=bi1-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
```

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
                   bi1=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*bi1+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)=bi1-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
```

Località spaziale e temporale

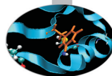


- Trasposizione di matrice

```
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.

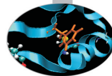
Cache blocking



- I dati elaborati in blocchi di dimensione adeguata alla cache
- All’interno di ogni blocco c’è il riuso 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
```

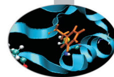
Cache: capacity miss e trashing



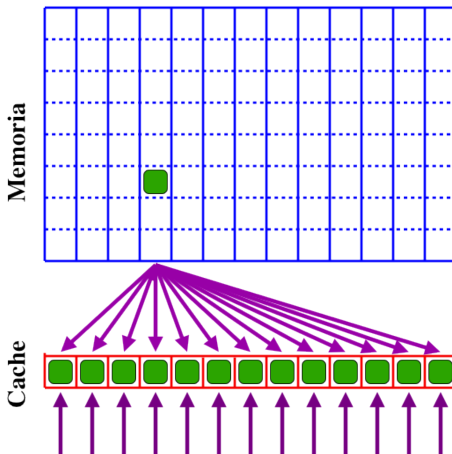
- La cache può soffrire di capacity miss:
  - si utilizza un insieme ristretto di righe (reduced effective cache size)
  - si riduce la velocità di elaborazione
- La cache può soffrire di trashing:
  - per caricare nuovi dati si getta via una riga prima che sia stata completamente utilizzata
  - è più 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

CINECA SCAI SuperComputing Applications and Innovation

Fully associative cache

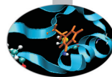


- Ogni blocco di memoria può essere mappato in una riga qualsiasi di cache



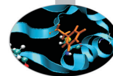
CINECA

Fully associative cache

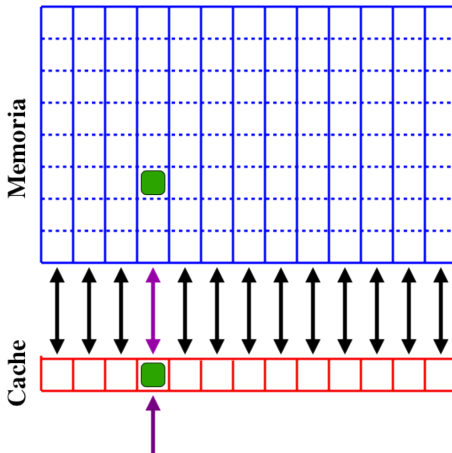


- Pro:
  - sfruttamento completo della cache
  - relativamente "insensibile" ai pattern di accesso alla memoria
- Contro:
  - strutture circuitali molto complesse per identificare molto rapidamente un hit
  - algoritmo di sostituzione oneroso Least Recently Used (LRU) o limitatamente efficiente First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

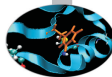
Direct mapped cache



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

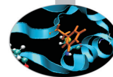


Direct mapped cache

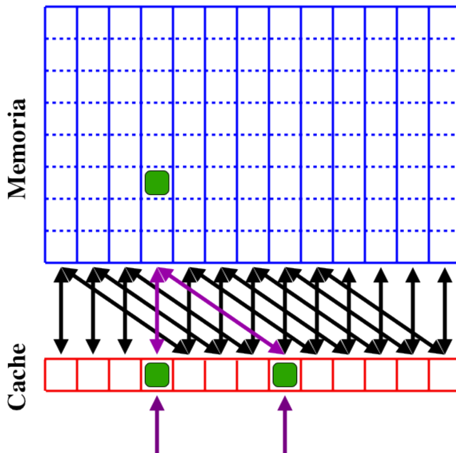


- Pro:
  - identificazione di un hit facilisima (alcuni bit dell’indirizzo identificano la riga da controllare)
  - algoritmo di sostituzione banale
  - cache di dimensione "arbitraria"
- Contro:
  - molto "sensibile" ai pattern di accesso alla memoria
  - soggetta a capacity miss
  - soggetta a cache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

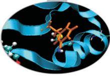
N-way set associative cache



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

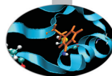


N-way set associative cache



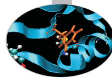
- Pro:
  - è un punto di bilanciamento intermedio
    - N=1 $\rightarrow$ direct mapped
    - N= numero di righe di cache $\rightarrow$ fully associative
  - consente di scegliere il bilanciamento tra complessità circuitale e prestazioni (i.e. costo del sistema e difficoltà di programmazione)
  - consente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



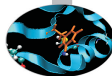
- Cache L1: 4÷8 way set associative
- Cache L2÷3: 2÷4 way set associative o direct mapped
- Capacity miss e trashing vanno affrontati
  - le tecniche sono le stesse
  - controllo della disposizione dei dati in memoria
  - controllo delle sequenze di accessi in memoria
- La cache L1 lavora su indirizzi 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

Cache Trashing



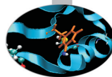
- Problemi di accesso ai dati in memoria
- Provoca la sostituzione di una riga di cache il cui contenuto verrà richiesto subito dopo
- Si presenta quando due o più flussi di dati insistono su un insieme ristretto di righe di cache
- NON aumenta il numero di load e store
- Aumenta il numero di transazioni 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)



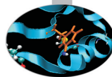
- Iterazione i=1
    1. Cerco A(1) nella cache di primo livello (L1) → cache miss
    2. Recupero A(1) nella memoria RAM
    3. Copio da A(1) a A(8) nella L1
    4. Copio A(1) in un registro
    5. Cerco B(1) nella cache di primo livello (L1) → cache miss
    6. Recupero B(1) nella memoria RAM
    7. 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)



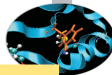
- Iterazione i=1
  1. Cerco A(1) nella cache di primo livello (L1) $\rightarrow$ cache miss
  2. Recupero A(1) nella memoria RAM
  3. Copio da A(1) a A(8) nella L1
  4. Copio A(1) in un registro
  5. Cerco B(1) nella cache di primo livello (L1) $\rightarrow$ cache miss
  6. Recupero B(1) nella memoria 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) $\rightarrow$ 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) $\rightarrow$ 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?



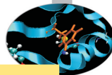
- Effetto variabile in funzione della dimensione del data set

```
...
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?



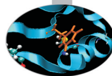
- Effetto variabile in funzione della dimensione del data set

```
...
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

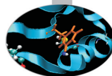
La soluzione é il padding.

Accessi Disallineati



- 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

Accessi disallineati



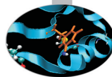
```
     parameter (nd=1000000)
     real*8 a(1:nd), b(1:nd)
     integer c
     common /data1/ a,c,b
....
     do j = 1, 300
        do i = 1, nd
           somma1 = somma1 + (a(i)-b(i))
        enddo
     enddo
```

Diverse performance per:

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

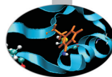
Dipende dall’architettura e dal compilatore che in genere segnala e cerca di sanare con opzione di align common

Come accertare qual è il problema?



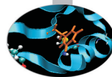
- Tutti i processori hanno contatori hardware di eventi
- Introdotti dai progettisti per CPU ad alti clock
  - indispensabili per debuggare i processori
  - utili per misurare le prestazioni
  - fondamentali per capire comportamenti anomali
- Ogni architettura misura eventi diversi
- Sono ovviamente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

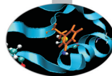
Valgrind



- 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

SCAI SuperComputing Applications and Innovation

Cachegrind

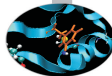


```
valgrind --tool=cachegrind <nome_eseguibile>
```

- 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 (Ir numero di istruzioni eseguite), I1 cache read misses(I1mr),LL cache instrunction read misses (ILmr)
  - D cache reads, Dr,D1mr,DLlmr
  - D cache writes, Dw,D1mw,DLlmw
- Riporta (opzionale) il numero di branch e quelli mispredicted

CINECA

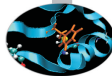
Cachegrind:esempio I



```
==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%  )
```

CINECA SCAI SuperComputing Applications and Innovation

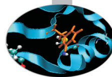
Cachegrind:esempio II



```
==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%  )
```

CINECA

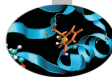
Cachegrind:cg_annotate



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

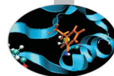
```
cg_annotate cachegrind.out.<pid>
```

Cachegrind:opzioni



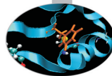
- —I1=<size>,<associativity>,<line size>
- —D1=<size>,<associativity>,<line size>
- —LL=<size>,<associativity>,<line size>
- —cache-sim=no|yes [yes]
- —branch-sim=no|yes [no]
- —cachegrind-out-file=<file>

Papi



- 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

Eventi Papi



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

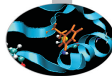
PAPI_L1_DCA: accessi in cache dati L1

PAPI_L1_DCM: misses in cache dati L1

PAPI_SR_INS: istruzioni di store

PAPI_BR_MSP: istruzioni di branch mispredicted

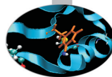
Papi: interfaccia



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

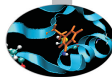
```
#include "fpapi_test.h"
integer events(2), retval   ;   integer*8 values(2)
......
events(1) = PAPI_FP_INS     ;   events(2) = PAPI_L1_DCM
......
call PAPIf_start_counters(events, 2, retval)
call PAPIf_read_counters(values, 2, retval)  ! Clear values
< sezione di codice da monitorare >
call PAPIf_stop_counters(values,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

Papi: interfaccia



- 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...

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

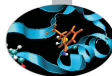
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

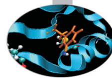
Floating Point Computing

CPU:parallelismo interno?



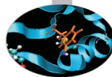
- Le CPU sono internamente parallele
  - pipelining
  - esecuzione superscalare
  - unità SIMD MMX, SSE, SSE2, SSE3, SSE4, AVX
- Per ottenere performance paragonabili con quelle sbandierate dal produttore:
  - fornire istruzioni in quantità
  - fornire gli operandi delle istruzioni

La pipeline



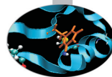
- Pipeline=tubazione, catena di montaggio
- Un’operazione è divisa in più passi indipendenti(stage) e differenti passi di differenti operazioni vengono eseguiti contemporaneamente
- Parallelismo sulle diverse fasi delle operazioni
- I processori sfruttano intensivamente il pipelining per aumentare la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

Ciclo della CPU



- 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.

Unità di calcolo NON pipelined



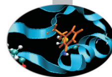
- Un'istruzione completata ogni tre cicli

Istruzione 1 Istruzione 2 Istruzione 3 Istruzione 4 Istruzione 5

Fetch
Decode
Execute

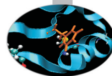
Tempo

Unità di calcolo pipelined

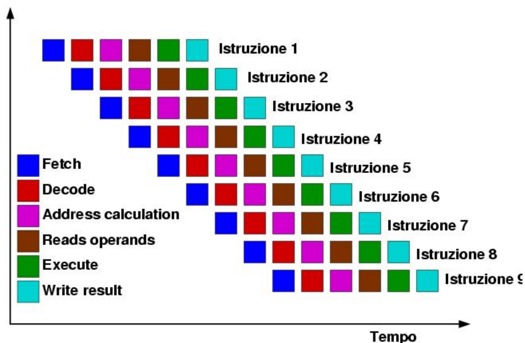


- Un risultato per ciclo a pipeline piena
- Per riempirla 3 istruzioni indipendenti, con gli operandi
- All’estremo opposto, un risultato ogni 3 cicli a pipeline "vuota"

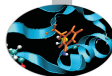
Unità di calcolo superpipelined



- 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)

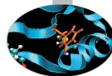


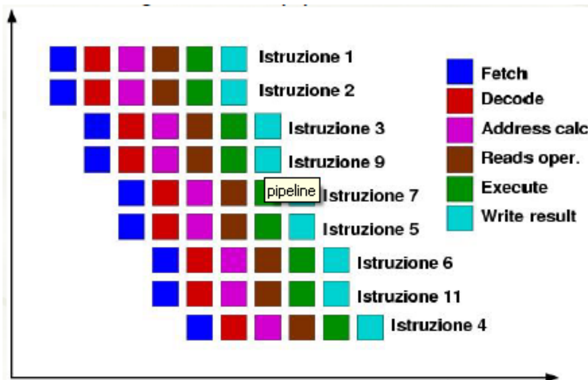
Esecuzione out of order



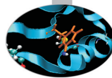
- Riordina dinamicamente le istruzioni
  - anticipa istruzioni i cui operandi sono già disponibili
  - postpone istruzioni i cui operandi non sono ancora disponibili
  - riordina letture e scritture in memoria
  - il tutto dipendentemente dalle unità funzionali libere
- Si appoggia intensivamente su:
  - 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



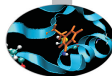


L’esecuzione superscalare



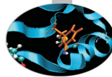
- Le CPU contengono più unità indipendenti
  - differenziazione funzionale
  - replicazione funzionale
- Operazioni indipendenti sono eseguite in contemporanea
  - operazioni su interi
  - operazioni su floating point
  - calcolo di salti
  - accessi in memoria
- Parallelismo sulle istruzioni
- Mascheramento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

Come sfrutto il parallelismo interno?



- 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? $\rightarrow$ sì per quel che può, OOO e branch prediction
  - il compilatore? $\rightarrow$ sì per quel che può, se lo evince dal codice
  - l'utente? $\rightarrow$ sì, nei casi più complessi
- Tecniche
  - loop unrolling $\rightarrow$ srotolo il loop
  - loop merging $\rightarrow$ fondo più loop insieme
  - loop splitting $\rightarrow$ decompongo loop complessi
  - inlining di funzioni $\rightarrow$ evito interruzioni di flusso di istruzioni

Un semplice loop

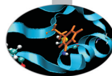


- Prodotto matrice-matrice

```
do j = 1, n
    do k = 1, n
     do i = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
     end do
    end do
end do
```

- 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

Loop unrolling

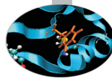


- Prodotto matrice-matrice con unrolling del loop esterno

```
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

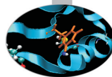


- Il compilatore sa fare automaticamente l'unrolling
- Ma non lo applica se lo ritiene pericoloso!
- Esempio: aliasing

```
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

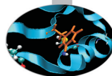


- Dichiarando che non vi sarà aliasing:

```
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

Loop fusion



- Dipendenze tra le iterazioni successive impediscono l’unrolling:

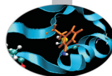
```
do i = 2,n
    a(i)= a(i-1) + 1.0
end do
do i = 2,n
    b(i)= b(i-1)*2.0
end do
```

- 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
```

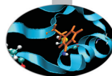
A volte il compilatore riesce a farlo da solo.

Loop splitting



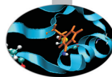
- Il corpo del loop può essere troppo grande
  - molte istruzioni e variabili di appoggio temporaneo
  - poche istruzioni ma espressioni lunghe e complicate
- In questi casi il compilatore può:
  - rinunciare
  - tentare l'unrolling e peggiorare le cose
- Register spilling
  - le variabili temporanee 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

Salti condizionali



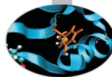
- I salti condizionali (loop, if) interrompono il flusso regolare delle pipeline finché il valore della condizione non è nota
- Esecuzione speculativa
  - la Branch Prediction Unit accumula statistica sulle più recenti istruzioni di salto condizionato, e predice cosa avverrà la prossima volta
  - predizione sbagliata $\rightarrow$ 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"

Inlining di funzioni



- 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

Outline



Introduzione

Architetture

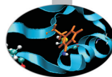
La cache e il sistema di memoria

Pipeline

Profilers
- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

SuperComputing Applications and Innovation

Outline



Introduzione

Architetture

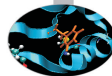
La cache e il sistema di memoria

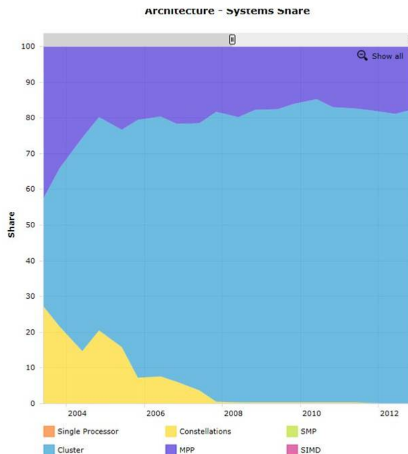
Pipeline

Profilers
- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

CINECA SCAI
SuperComputing Applications and Innovation

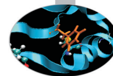
Il trend architetturale (Top500 list)

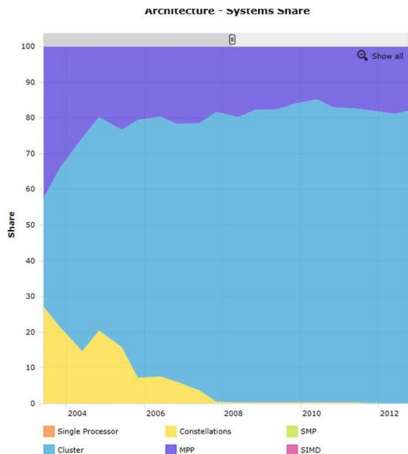




CINECA

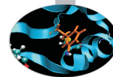
Il trend architetturale (Top500 list)

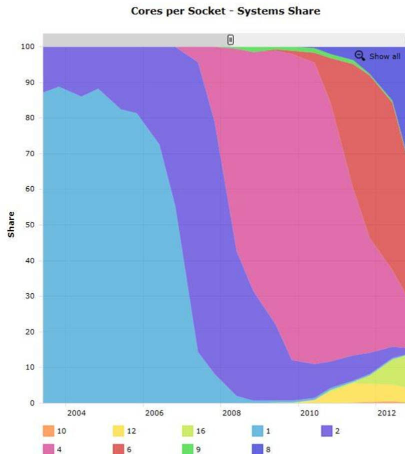




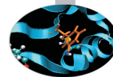
I cluster dominano il mercato dell'High Performance Computing

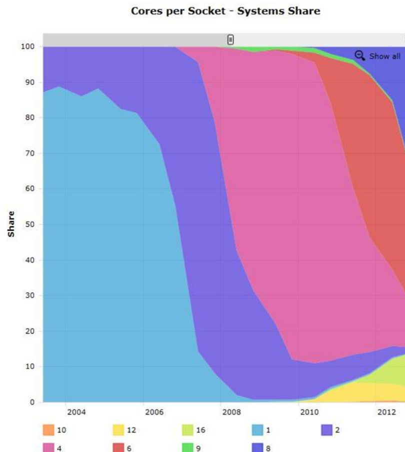
Il trend dei processori





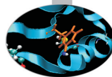
Il trend dei processori





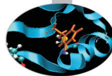
sempre piu processori "multicore" per "socket"

Perché misurare le prestazioni?



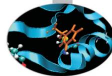
- Architetture sempre più complesse:

Perché misurare le prestazioni?



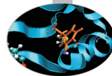
- Architetture sempre più complesse:
  - "bandwidth" verso la memoria ridotta

Perché misurare le prestazioni?



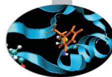
- Architetture sempre più complesse:
  - "bandwidth" verso la memoria ridotta
  - quantità di memoria per "core" ridotta

Perché misurare le prestazioni?



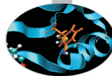
- Architetture sempre più complesse:
  - "bandwidth" verso la memoria ridotta
  - quantità di memoria per "core" ridotta
  - gerarchia di memoria sempre più complessa

Perché misurare le prestazioni?



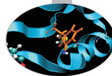
- Architetture sempre più complesse:
  - "bandwidth" verso la memoria ridotta
  - quantità di memoria per "core" ridotta
  - gerarchia di memoria sempre più complessa
- La programmazione su queste macchine non è semplice

Perché misurare le prestazioni?



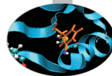
- Architetture sempre più complesse:
  - "bandwidth" verso la memoria ridotta
  - quantità di memoria per "core" ridotta
  - gerarchia di memoria sempre più complessa
- La programmazione su queste macchine non è semplice
- "Spremere" le prestazioni dei codici di calcolo non è semplice

Perché misurare le prestazioni?



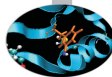
- Architetture sempre più complesse:
  - "bandwidth" verso la memoria ridotta
  - quantità di memoria per "core" ridotta
  - gerarchia di memoria sempre più complessa
- La programmazione su queste macchine non è semplice
- "Spremere" le prestazioni dei codici di calcolo non è semplice
- Risulta pertanto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

Il ciclo vita delle prestazioni



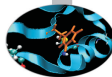
Modules/Ciclo.png

Profiling



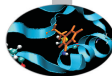
- Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:

Profiling



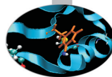
- Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - trovare quelle parti dove viene speso gran parte del tempo

Profiling



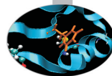
- Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - trovare quelle parti dove viene speso gran parte del tempo
  - trovare il "grafo" e le dipendenze delle varie parti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

Profiling



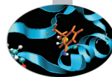
- Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - trovare quelle parti dove viene speso gran parte del tempo
  - trovare il "grafo" e le dipendenze delle varie parti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

Profiling



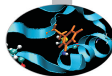
- Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - trovare quelle parti dove viene speso gran parte del tempo
  - trovare il "grafo" e le dipendenze delle varie parti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

Profiling



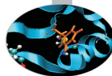
- Una tipica applicazione seriale o parallela è composta da una serie di procedure.
- Per ottimizzare o parallelizzare un codice (che può essere anche molto complesso) è fondamentale:
  - trovare quelle parti dove viene speso gran parte del tempo
  - trovare il "grafo" e le dipendenze delle varie parti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

Profiling



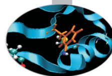
- Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:

Profiling



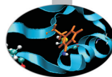
- Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - semplicità o meno di utilizzo

Profiling



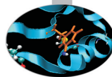
- Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - semplicità o meno di utilizzo
  - proprietari o di pubblico dominio

Profiling



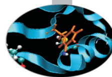
- Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
    - semplicità o meno di utilizzo
    - proprietari o di pubblico dominio
    - intrusivi o no

Profiling



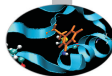
- Esistono una gran varietà di strumenti di "Profiling" che si differenziano, essenzialmente, per:
  - semplicità o meno di utilizzo
  - proprietari o di pubblico dominio
  - intrusivi o no
  - etc, etc

Profiling



- 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.

Outline



Introduzione

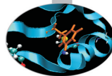
Architetture

La cache e il sistema di memoria

Pipeline

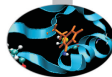
Profilers
- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

time: caratteristiche



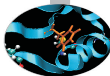
- Presente in tutte le architetture *Unix /Linux*.

time: caratteristiche



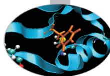
- Presente in tutte le architetture *Unix /Linux*.
- Fornisce il tempo totale di esecuzione di un programma ed altre utili informazioni.

time: caratteristiche



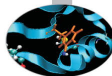
- Presente in tutte le architetture *Unix /Linux*.
- Fornisce il tempo totale di esecuzione di un programma ed altre utili informazioni.
- Non ha bisogno che il programma sia compilato con particolari opzioni di compilazione (assolutamente non intrusivo).

time: caratteristiche



- 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>

time: caratteristiche

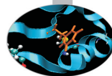


- Presente in tutte le architetture *Unix /Linux*.
- Fornisce il tempo totale di esecuzione di un programma ed altre utili informazioni.
- Non ha bisogno che il programma sia compilato con particolari opzioni di compilazione (assolutamente non intrusivo).
- time <nome_eseguibile>

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
```

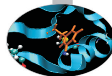
time: output



9.29u

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

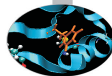
time: output



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 dell programma.

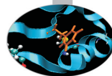
time: output



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 dell programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").

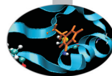
time: output



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 dell programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").
4. La percentuale di CPU impiegata nel processo.

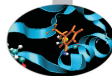
time: output



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 dell 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).

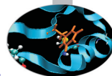
time: output



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 dell 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).

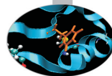
time: output



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

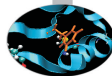
1. (*User time*) Il tempo di CPU (in secondi) impiegato dall'eseguibile a girare.
2. (*System time*) Il tempo di CPU (in secondi) impiegato dal processo in chiamate di sistema durante l'esecuzione dell programma.
3. (*Elapsed time*) Il tempo (ore:minuti:secondi) effettivamente impiegato ("elapsed time").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

time-output



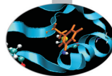
- L’uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:

time-output



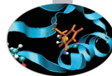
- L’uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - (*Lo "user" time è confrontabile con il "sys" time*)

time-output



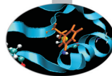
- L’uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - (*Lo "user" time è confrontabile con il "sys" time*)
  - (*La percentuale di utilizzo della CPU è quasi del 100%*)

time-output



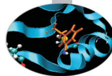
- L'uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - (*Lo "user" time è confrontabile con il "sys" time*)
  - (*La percentuale di utilizzo della CPU è quasi del 100%*)
  - (*Non è presente I/O*)

time-output



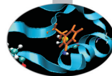
- L'uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - (*Lo "user" time è confrontabile con il "sys" time*)
  - (*La percentuale di utilizzo della CPU è quasi del 100%*)
  - (*Non è presente I/O*)
  - (*Non vi sono quasi per nulla "page-faults"*)

time-output



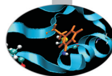
- L’uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - (*Lo "user" time è confrontabile con il "sys" time*)
  - (*La percentuale di utilizzo della CPU è quasi del 100%*)
  - (*Non è presente I/O*)
  - (*Non vi sono quasi per nulla "page-faults"*)
  - (*L’area dati (massima) durante l’esecuzione è di circa* 18*Gbytes*)

time-output



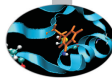
- L'uso di time su questo eseguibile ci ha dato alcune informazioni interessanti:
  - (*Lo "user" time è confrontabile con il "sys" time*)
  - (*La percentuale di utilizzo della CPU è quasi del 100%*)
  - (*Non è presente I/O*)
  - (*Non vi sono quasi per nulla "page-faults"*)
  - (*L'area dati (massima) durante l'esecuzione è di circa* 18*Gbytes*)
  - 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 gia in "Kbytes".
- Il valore corretto per il nostro eseguibile è dunque circa 4 Gbytes.

Page-faults



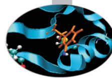
- se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto piu grande (circa 8 milioni). Cosa sta succedendo?

Page-faults



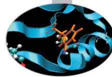
- se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto piu grande (circa 8 milioni). Cosa sta succedendo?
- Un "page-fault" è un segnale generato dalla CPU, con conseguente reazione del sistema operativo, quando un programma tenta di accedere ad una pagina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

Page-faults



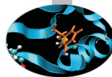
- se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto piu grande (circa 8 milioni). Cosa sta succedendo?
- Un "page-fault" è un segnale generato dalla CPU, con conseguente reazione del sistema operativo, quando un programma tenta di accedere ad una pagina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

Page-faults



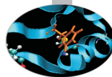
- se cambiamo la taglia del file di input della nostra simulazione il numero di "page-faults" è molto piu grande (circa 8 milioni). Cosa sta succedendo?
- Un "page-fault" è un segnale generato dalla CPU, con conseguente reazione del sistema operativo, quando un programma tenta di accedere ad una pagina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

time: Analisi output



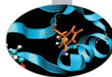
- Per questo codice *System time* $\sim$ *User time*.

time: Analisi output



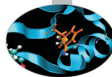
- Per questo codice *System time* $\sim$ *User time*.
- è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.

time: Analisi output



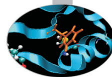
- Per questo codice *System time* $\sim$ *User time*.
- è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.
  il programma "alloca" e "dealloca" nel corso dell'esecuzione una serie di matrici: questa cosa è altamente sconsigliata.

time: Analisi output



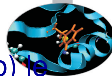
- Per questo codice *System time* ∼ *User time*.
- è indice di molti page-faults o di cattivo uso della memoria e, nel caso specifico, di molte chiamate a sistema.
  il programma "alloca" e "dealloca" nel corso dell'esecuzione una serie di matrici: questa cosa è altamente sconsigliata.
- *System time*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

time: Analisi output



- 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.

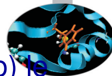
time: Analisi output



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

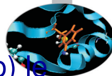


Cambiando la struttura del programma (eliminando le allocazioni e deallocazioni durante l'esecuzione dello stesso) le cose migliorano drasticamente:

```
[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



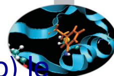
Cambiando la struttura del programma (eliminando le allocazioni e deallocazioni durante l'esecuzione dello stesso) le cose migliorano drasticamente:

```
[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
```

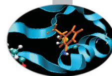
e ora correttamente *System time* $<<$ *User time*.

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:

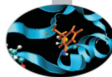
```
12973.38user 1915.82system 20:55.80elapsed 1185%CPU (0avgtext+0avgdata 2597648maxresident)k
19608inputs+10649880outputs (147major+223489935minor)pagefaults 0swaps
```

time:Analisi output



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

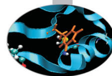
time:Analisi output



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 soprenderci, dato che stiamo utilizzando 48 cores di PLX).

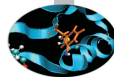
time:Analisi output



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 soprenderci, 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.

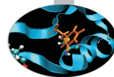
time:Analisi output



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 soprenderci, 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

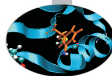
time:Analisi output



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 soprenderci, 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.

Outline



Introduzione

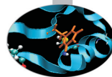
Architetture

La cache e il sistema di memoria

Pipeline

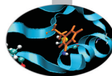
Profilers
- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

Top



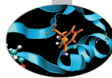
L'informazione che ritorna dal comando time è utile ma non descrive dinamicamente (nel tempo) e con una buona approssimazione il "comportamento " della nostra applicazione.

Top



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

Top



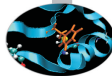
L'informazione che ritorna dal comando time è utile ma non descrive dinamicamente (nel tempo) e con una buona approssimazione il "comportamento " della nostra applicazione. Inoltre, time non ci fornisce alcuna informazione dello stato della macchina (o iinsieme di macchine) su cui stiamo in esecuzione 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:

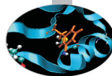
top [options ...]

Esempio di output di top



```
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
```

Outline



Introduzione

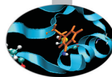
Architetture

La cache e il sistema di memoria

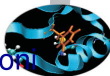
Pipeline

Profilers
- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

gprof: caratteristiche principali

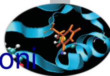


gprof: caratteristiche principali



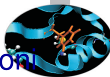
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull’esecuzione della nostra applicazione.

gprof: caratteristiche principali



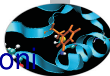
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).

gprof: caratteristiche principali



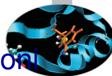
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull’esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- in prima battuta può servire uno strumento che ci dia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

gprof: caratteristiche principali



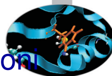
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- in prima battuta può servire uno strumento che ci dia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



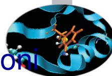
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- in prima battuta può servire uno strumento che ci dia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



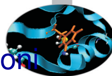
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull’esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- in prima battuta può servire uno strumento che ci dia 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



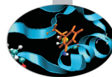
- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- in prima battuta può servire uno strumento che ci dia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

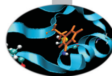


- time è uno strumento anche efficace per ottenere informazioni "complessive" e "a grana grossa" sull'esecuzione della nostra applicazione.
- chiaramente non risponde a tutte le esigenze, soprattutto per codici realistici (come COSMO ad esempio).
- in prima battuta può servire uno strumento che ci dia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

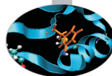


Gprof "Sampling"



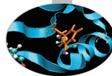
- La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.

Gprof "Sampling"



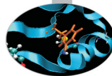
- La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- Gprof si basa su un Time Based Sampling : ad intervalli di tempo fissati si interroga il "program counter"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

Gprof "Sampling"



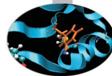
- La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- Gprof si basa su un Time Based Sampling : ad intervalli di tempo fissati si interroga il "program counter"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



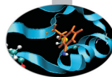
- La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- Gprof si basa su un Time Based Sampling : ad intervalli di tempo fissati si interroga il "program counter"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

Gprof "Sampling"

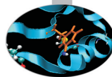


- La tecnica del "Sampling" viene utilizzata da Gprof (e in generale da strumenti di Profiling) per raccogliere informazioni relative al "timing" della nostra applicazione durante la sua esecuzione.
- Gprof si basa su un Time Based Sampling : ad intervalli di tempo fissati si interroga il "program counter" 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"
- Il vantaggio è che essendo meno intrusivo, l'esecuzione non dovrebbe risentire eccessivamente dell'uso del profiling.

Gprof "Instrumentation"

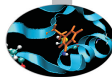


Gprof "Instrumentation"



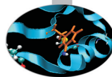
- essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.

Gprof "Instrumentation"



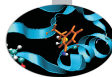
- essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.
- può risultare, come detto, maggiormente invasivo e pertanto inficiare le prestazioni del nostro eseguibile di partenza.

Gprof "Instrumentation"



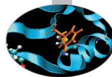
- essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.
- può risultare, come detto, maggiormente invasivo e pertanto inficiare le prestazioni del nostro eseguibile di partenza.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

Gprof "Instrumentation"



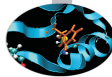
- essenzialmente "Instrumentare" un codice vuol dire aggiungere al programma istruzioni vere e proprie (dunque in modo intrusivo) che servono a raccogliere le informazioni richieste.
- può risultare, come detto, maggiormente invasivo e pertanto inficiare le prestazioni del nostro eseguibile di partenza.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

Gprof:how-to



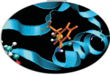
- Per "attivare" Gprof, occorre compilare e linkare il codice (scritto in Fortran, C, C++) con l'opzione -pg
- Utilizzo
  <compiler> -pg programma.f -o nome_eseguibile
  ./nome_eseguibile
  gprof nome_eseguibile

Gprof:how-to



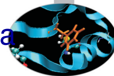
- Per "attivare" Gprof, occorre compilare e linkare il codice (scritto in Fortran, C, C++) con l'opzione -pg
- Utilizzo
  <compiler> -pg programma.f -o nome_eseguibile
  ./nome_eseguibile
  gprof nome_eseguibile
- dopo un run andato a buon fine (da non sottovalutare!!!), viene generato il file gmon.out

Gprof:how-to



- 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.

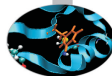
gprof: Flat profile



- 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);
   }
```

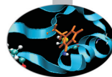
gprof: Flat profile



- Flat profile: continua....

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

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

gprof: Flat profile

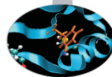


- Flat profile: continua....

```
   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.

gprof: Flat profile

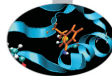


- Flat profile: continua....

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

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

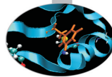
- compiliamo e linkiamo il sorgente C con Gprof e analizziamo il Flat profile.
- ci aspettiamo che la routine b() pesi circa 4 volte la routine a():

gprof: flat profile



```
/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              self     total
 time   seconds   seconds    calls  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
```

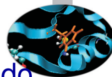
Flat profile:descrizione



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:

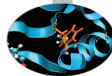
```
int cinsideb(int d) {
  {
  }
  return d;
}
```

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

```
int b(void) {
  int i=0,g=0;
  while(i++<400000)
  {
    g+=cinsideb(i);
  }
  return g;
}
```

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

gprof: flat profile

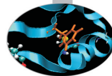


Nuovo Flaf profile:

```
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:

gprof: flat profile



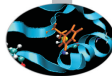
Nuovo Flaf profile:

```
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:

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

gprof: flat profile



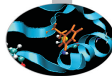
Nuovo Flaf profile:

```
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:

- 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()

gprof: flat profile



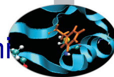
Nuovo Flaf profile:

```
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:

- 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.

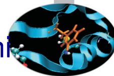
gprof: call tree profile



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

SCAI
SuperComputing Applications and Innovation

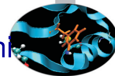
gprof: call tree profile



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.

CINECA

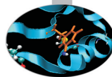
gprof: call tree profile



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:

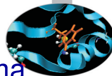
```
                     Call graph (explanation follows)

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

index % time    self  children    called     name
                                                                <spontaneous>
[1]    100.0    0.00    7.40                 main [1]
                      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]
-----------------------------------------------
...
```

Call tree profile:descrizione



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.

gprof: limitazioni



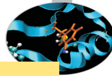
Consideriamo a titolo esemplificativo un semplice programma che realizza il classico prodotto matrice-matrice in due modi:

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.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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_
```

gprof: limitazioni



versione con le BLAS:

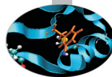
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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:

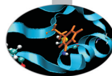
- Per la versione MKL Gprof non ci fornisce alcuna informazione utile, se non chiamate a funzioni di libreria ausiliarie.
- La versione BLAS correttamente riporta la chiamata alla funzione *dgemm_* che è responsabile della quasi totalità del tempo.
- Fortunatamente possiamo ritenere che le funzioni di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

SuperComputing Applications and Innovation

gprof: altre limitazioni

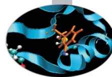


gprof: altre limitazioni



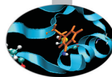
- Gprof ha "granularità abbastanza grande. Per codici complessi non è semplice capire, anche partendo da una data routine/function, dove mettere le mani per migliorare le prestazioni del codice.

gprof: altre limitazioni



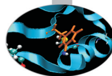
- Gprof ha "granularità abbastanza grande. Per codici complessi non è semplice capire, anche partendo da una data routine/function, dove mettere le mani per migliorare le prestazioni del codice.
- Gprof puo risultare molto intrusivo. Verificare, sperimentalmente, che l'esecuzione del codice senza Gprof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

Gprof: Esercitazione

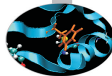


- Profiling di un codice seriale per la risoluzione di un'Equazione alle Derivate Parziali.
  - Utilizzare il tool Gprof per effettuare un Profiling del codice al variare della taglia del problema.
  - Eseguire i seguenti comandi:

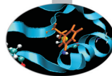
```
cp /gpfs/scratch/userinternal/planucar/CORSO2013/gprof.tar.gz .
tar xvfz gprof.tar.gz
cd GPROF/JACOBI
```

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

Funzioni per misurare il tempo

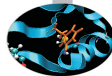


Funzioni per misurare il tempo



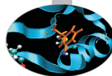
- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.

Funzioni per misurare il tempo



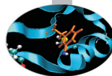
- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l’utilizzo di ulteriori strumenti.

Funzioni per misurare il tempo



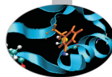
- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l’utilizzo di ulteriori strumenti.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

Funzioni per misurare il tempo



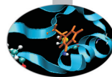
- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.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



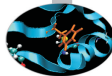
- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.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



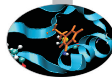
- dopo aver utilizzato Gprof e verificato che una determinata routine è computazionalmente onerosa, può risultare più semplice "instrumentare" il nostro codice con funzioni per misurare il tempo.
- è una tecnica che ci consente di raffinare la nostra analisi "a mano" senza l'utilizzo di ulteriori strumenti.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

Un esempio C/C++



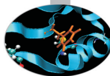
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
clock_t time1, time2;
double dub_time;
int main(){
int i,j,k,nn=1000;
double c[nn][nn],a[nn][nn],b[nn][nn];
...
time1 = clock();
for (i = 0; i < nn; i++)
for (k = 0; k < nn; k++)
for (j = 0; j < nn; j ++)
c[i][j] = c[i][j]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



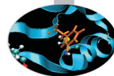
```
real(8)::a(1000,1000),b(1000,1000),c(1000,1000)
real(8) ::t1,t2
integer :: time_array(8)
a=0;b=0;c=0;n=1000
...
...
call date_and_time(values=time_array)
t1 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
do j = 1,n
do k = 1,n
do i = 1,n
c(i,j) = c(i,j) + a(i,k)*b(k,j)
enddo
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
```

Outline



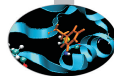
- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
  - Motivazioni
  - time
  - top
  - gprof
  - Papi
  - Scalasca
  - Consigli finali

Papi



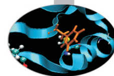
- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.

Papi



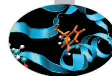
- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione,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

Papi



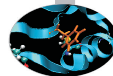
- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione,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

Papi



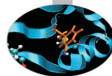
- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione,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

Papi



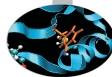
- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione,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



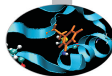
- PAPI fornisce un’interfaccia agli *Hardware Counters* essenzialmente di due tipi:

Papi-cont.



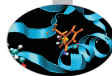
- PAPI fornisce un’interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)

Papi-cont.



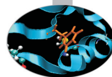
- PAPI fornisce un’interfaccia agli *Hardware Counters* essenzialmente di due tipi:
    - *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)
    - *Low level interface*, che fornisce informazioni specifiche dell’hardware a disposizione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

Papi-cont.



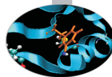
- PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)
  - *Low level interface*, che fornisce informazioni specifiche dell'hardware a disposizione per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

PAPI Preset Events



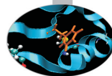
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni

PAPI Preset Events



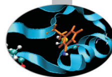
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:

PAPI Preset Events



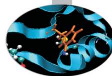
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali

PAPI Preset Events



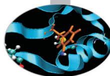
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS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

PAPI Preset Events



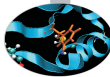
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point

PAPI Preset Events



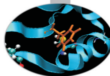
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1

PAPI Preset Events



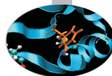
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1
  - PAPI_L1_DCM - cache misses L1

PAPI Preset Events



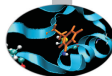
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS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



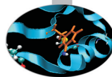
- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei pù importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS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



- 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 pù 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

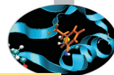
Papi: interfaccia C e Fortran



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)
...
```

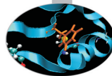
Papi: interfaccia C e Fortran



un esempio in C:

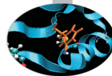
```
%\begin{lstlisting}
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"

#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



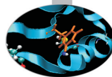
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:

PAPI: tutte le funzioni di alto livello



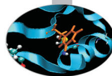
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili

PAPI: tutte le funzioni di alto livello



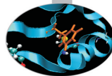
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate

PAPI: tutte le funzioni di alto livello



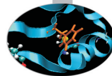
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate

PAPI: tutte le funzioni di alto livello



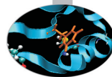
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time

PAPI: tutte le funzioni di alto livello



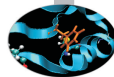
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters

PAPI: tutte le funzioni di alto livello



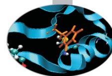
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters

PAPI: tutte le funzioni di alto livello



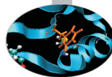
- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters
  - PAPI_start_counters - start counting hw events

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
  - PAPI_start_counters - start counting hw events
  - PAPI_stop_counters - stop counters return current counts

PAPI: Esercitazione

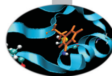


- Profiling con PAPI del codice seriale che utilizza chiamate alle BLAS per alcune operazioni fondamentali di "linear algebra".
  - Eseguire i seguenti comandi:

```
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.

Outline



Introduzione

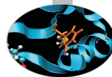
Architetture

La cache e il sistema di memoria

Pipeline

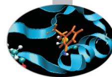
Profilers
- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

Scalasca: caratteristiche



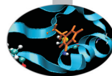
- Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.

Scalasca: caratteristiche



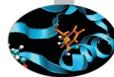
- Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- In realtà nasce come il successore di un altro tool di successo (KOJAK)
- È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).

Scalasca: caratteristiche



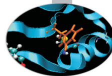
- Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- In realtà nasce come il successore di un altro tool di successo (KOJAK)
- È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).
- Utilizzabile sulla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

Scalasca: caratteristiche



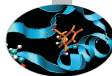
- Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- In realtà nasce come il successore di un altro tool di successo (KOJAK)
- È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).
- Utilizzabile sulla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

Scalasca: caratteristiche



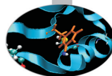
- Supporta applicazioni scritte in Fortran, C e C++.

Scalasca: caratteristiche



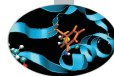
- Supporta applicazioni scritte in Fortran, C e C++.
- In pratica, Scalasca consente due tipi di analisi:

Scalasca: caratteristiche



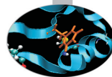
- Supporta applicazioni scritte in Fortran, C e C++.
- In pratica, Scalasca consente due tipi di analisi:
  - una modalità "summary" che consente di ottenere informazioni aggregate per la nostra applicazione (ma sempre dettagliate a livello di singola istruzione)
  - pause una modalità "tracing" che 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"

Scalasca: caratteristiche



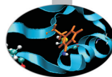
- Supporta applicazioni scritte in Fortran, C e C++.
- In pratica, Scalasca consente due tipi di analisi:
  - una modalità "summary" che consente di ottenere informazioni aggregate per la nostra applicazione (ma sempre dettagliate a livello di singola istruzione)
  - pause una modalità "tracing" che 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 (instrumentato) Scalasca è in grado di caricare i files in memoria ed analizzarli in parallelo usando lo stesso numero di "cores" della nostra applicazione.

Scalasca: istruzioni pratiche per l’uso



L’intero processo avviene in tre passi:

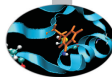
Scalasca: istruzioni pratiche per l'uso



L'intero processo avviene in tre passi:

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

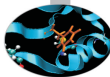
Scalasca: istruzioni pratiche per l’uso



L’intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  ifort -openmp [altre_opzioni] <codice_sorgente>

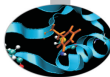
Scalasca: istruzioni pratiche per l'uso



L'intero processo avviene in tre passi:

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

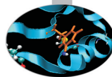
Scalasca: istruzioni pratiche per l’uso



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>

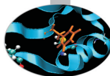
Scalasca: istruzioni pratiche per l'uso



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>

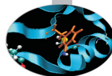
Scalasca: istruzioni pratiche per l’uso



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:

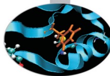
Scalasca: istruzioni pratiche per l’uso



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:
  <codice_eseguibile>

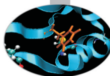
Scalasca: istruzioni pratiche per l’uso



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>

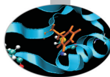
Scalasca: istruzioni pratiche per l’uso



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>

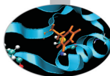
Scalasca: istruzioni pratiche per l’uso



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>

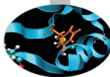
Scalasca: istruzioni pratiche per l’uso



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]

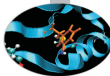
Scalasca: istruzioni pratiche per l’uso



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:

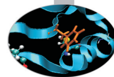
Scalasca: istruzioni pratiche per l’uso



L’intero processo avviene in tre passi:

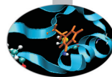
- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
 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

Scalasca: un esempio



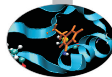
- Codice sismologia elementi finiti (fem.F).

Scalasca: un esempio



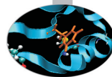
- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando il tool OpenMP.

Scalasca: un esempio



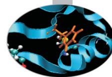
- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando il tool OpenMP.
- Eseguito su 8 processori (parallelismo "moderato") su singolo nodo

Scalasca: un esempio



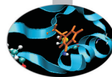
- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando il tool OpenMP.
- Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- Compilatore intel (comando ifort -O3 -free -openmp...)

Scalasca: un esempio



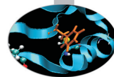
- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando il tool OpenMP.
- Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- Compilatore intel (comando ifort -O3 -free -openmp...)
- Alcuni numeri:

Scalasca: un esempio



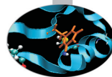
- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando il tool OpenMP.
- Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- Compilatore intel (comando ifort -O3 -free -openmp...)
- Alcuni numeri:
  - Numeri dei nodi della griglia 2060198.

Scalasca: un esempio



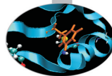
- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando il tool OpenMP.
- Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- Compilatore intel (comando ifort -O3 -free -openmp...)
- Alcuni numeri:
  - Numeri dei nodi della griglia 2060198.
  - Numeri degli elementi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

Scalasca: compilazione ed esecuzione



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

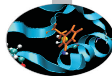
Scalasca: compilazione ed esecuzione



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

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

Scalasca: compilazione ed esecuzione

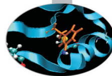


```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp  *.F
```

```
[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
.....
```

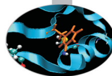
Scalasca: analisi dei risultati



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

SCAI
SuperComputing Applications and Innovation

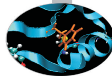
Scalasca: analisi dei risultati



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

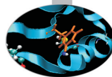
```
cube3_score ./epik_fem_Ox8_sum/epitome.cube
Reading epik_fem_Ox8_sum/epitome.cube... done.
            .....
Estimated aggregate size of event trace (total_tbc): 1102182744 bytes
Estimated size of largest process trace (max_tbc):   1090211280 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc 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



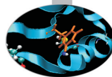
1. Per tipologia:

Scalasca output: breve descrizione



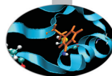
1. Per tipologia:
   - ANY: tutte le routines che compongono il programma

Scalasca output: breve descrizione



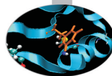
1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).

Scalasca output: breve descrizione



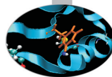
1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizzazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
   - COM:tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.

Scalasca output: breve descrizione



1. Per tipologia:
    - ANY: tutte le routines che compongono il programma
    - OMP: quelle che contengono costrutti di parallellizzazione (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.

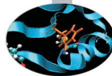
Scalasca output: breve descrizione



1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (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).

CINECA SCAI SuperComputing Applications and Innovation

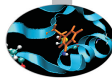
Scalasca output: breve descrizione



1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (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.

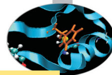
CINECA

Scalasca output: breve descrizione



1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (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.

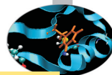
Scalasca output: maggior dettaglio



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

SuperComputing Applications and Innovation

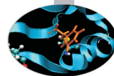
Scalasca output: maggior dettaglio



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

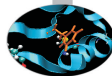
```
...
      USR   889293768       68.51    1.12 expand_
      USR    98889504        9.96    0.16 ordinamento_
      USR    98747208       12.60    0.21 elem2d_
      USR      730224        0.05    0.00 elem_ij_      OMP      349200         0.48
0.01 !$omp do @solutore_parallelo.F:157
      OMP      349200        0.34    0.01 !$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 "instrumentazione" manuale



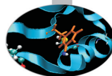
```
        program fem
        implicit none
#include "epik_user.inc"
...
...
...
EPIK_USER_REG(r_write, "<<write>>")
        real*8,allocatable :: csi(:),eta(:)
...
...
EPIK_USER_START(r_write)
         do i=1,13
            write(i+5000,*) t,csi(2*p(i)-1),eta(2*p(i)-1)
            write(i+6000,*)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

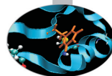


Cube 3.2 QT: epik_fem_0x8_sum/epitome.cube <@neo178>

File Display Topology Help

Absolute	Absolute	Absolute
Metric tree	Call tree / Flat view	System tree / Topology 0
4.59e7 Visits	4382.53 MAIN__	4382.53 Linux Intel
0 Collective synchronizatic		
0 P2P send synchronizatic		
0 P2P recv synchronizatio		
0 Collective exchange cor		
0 Collective communicatio		
0 Collective communicatio		
0 P2P send communicatic		
0 P2P recv communicatioi		
0 Collective bytes outgoin		
0 Collective bytes incomin		
0 P2P bytes sent		
0 P2P bytes received		
0.60 OMP thread fork time		
0.76 OMP thread manage		
4382.53 Time		
0.00 4382.53 (100.00%) 4382.53	0.00 4382.53 (100.00%) 4382.53	0.00 4382.53 (100.00%) 4382.53

Scalasca: output grafico



Cube 3.2 QT: epik_fem_0x8_sum/epitome.cube <@neo178>

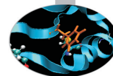
File Display Topology Help

Own root percent	Own root percent	Own root percent
Metric tree	Call tree / Flat view	System tree / Topology 0
100.00 Visits	1.70 MAIN__	100.00 Linux Intel
0.00 Collective synchronizations	0.00 dwalltime00_	
0.00 P2P send synchronizations	0.00 inittime_	
0.00 P2P recv synchronizations	0.01 printtime_	
0.00 Collective exchange communicati	0.03 ordinamento_	
0.00 Collective communications as sou	0.05 preliminari_solutore_parallelo_	
0.00 Collective communications as des	0.00 q_compute_	
0.00 P2P send communications	0.73 costruzione_	
0.00 P2P recv communications	0.00 abc03_	
0.00 Collective bytes outgoing	0.04 !$omp parallel @fem.F:2177	
0.00 Collective bytes incoming	0.15 presolutore_parallelo_	
0.00 P2P bytes sent	0.01 !$omp parallel @fem.F:2373	
0.00 P2P bytes received	34.73 !$omp parallel @fem.F:2511	
100.00 OMP thread fork time	0.00 funzione_gr_	
100.00 OMP thread management time	19.17 !$omp parallel @fem.F:2589	
100.00 Time	40.92 solutore_parallelo_	
	2.47 !$omp parallel @fem.F:2725	
0.00 100.00 100.00	0.00 1.70 100.00	0.00 100.00 100.00
0.00 4382.53 (100.00%) 4382.53	0.00 74.70 (1.70%) 4382.53	0.00 74.70 (100.00%) 74.70

Selected "100.00 MAIN__"

CINECA SCAI SuperComputing Applications and Innovation

Scalasca: output grafico



Cube 3.2 QT: epik_fem_0x8_sum/epitome.cube <@neo178>

File Display Topology Help

Absolute	Absolute	Own root percent
Metric tree	Call tree / Flat view	System tree / Topology 0

Metric tree

- 4.59e7 Visits
- 0 Collective synchronizations
- 0 P2P send synchronizations
- 0 P2P recv synchronizations
- 0 Collective exchange communications
- 0 Collective communications as source
- 0 Collective communications as destinati
- 0 P2P send communications
- 0 P2P recv communications
- 0 Collective bytes outgoing
- 0 Collective bytes incoming
- 0 P2P bytes sent
- 0 P2P bytes received
- 0.60 OMP thread fork time
- 0.76 OMP thread management time
- 4382.53 Time

Call tree

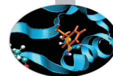
- 74.70 MAIN__
  - 0.00 dwalltime00_
  - 0.01 inittime_
  - 0.32 printtime_
  - 1.23 ordinamento_
  - 2.34 preliminari_solutore_parallelo_
  - 0.00 q_compute_
  - 31.79 costruzione_
  - 0.02 abc03_
  - 1.86 !$omp parallel @fem.F:2177
  - 6.55 presolutore_parallelo_
  - 0.31 !$omp parallel @fem.F:2373
  - 1522.13 !$omp parallel @fem.F:2511
  - 0.00 funzione_gr_
  - 840.07 !$omp parallel @fem.F:2589
  - 1793.11 solutore_parallelo_
  - 108.09 !$omp parallel @fem.F:2725

0.00	4382.53 (100.00%)	4382.53

0.00	840.07 (19.17%)	4382.53

0.00	12.50	100.00
105.01	105.01 (86.53%)	105.01

Scalasca: output grafico



Cube 3.2 QT: epik_fem_0x8_sum/epitome.cube <@neo178>

File Display Topology Help

Absolute | Absolute | Absolute

Metric tree | Call tree | Flat view | System tree | Topology 0

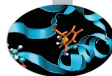
- 4.59e7 Visits
- 0 Collective synchronizations
- 0 P2P send synchronizations
- 0 P2P recv synchronizations
- 0 Collective exchange communicatior
- 0 Collective communications as sourc
- 0 Collective communications as destir
- 0 P2P send communications
- 0 P2P recv communications
- 0 Collective bytes outgoing
- 0 Collective bytes incoming
- 0 P2P bytes sent
- 0 P2P bytes received
- 0.60 OMP thread fork time
- 0.76 OMP thread management time
- 4382.53 Time

- 74.70 MAIN__
  - 0.00 dwalltime00_
  - 0.01 inittime_
  - 0.32 printtime_
  - 1.23 ordinamento_
  - 2.34 preliminari_solutore_parallelo_
  - 0.00 q_compute_
  - 31.79 costruzione_
  - 0.02 abc03_
  - 1.86 !$omp parallel @fem.F:2177
  - 6.55 presolutore_parallelo_
  - 0.31 !$omp parallel @fem.F:2373
  - 1522.13 !$omp parallel @fem.F:25
  - 0.00 funzione_gr_
  - 840.07 !$omp parallel @fem.F:258
  - 1793.11 solutore_parallelo_
  - 108.09 !$omp parallel @fem.F:272

0.00 4382.53 (100.00%) 4382.53 | 0.00 0.00 (0.00%) 4382.53 | 0.00 0.00 (100.00%) 0.00

Selected "0.00 funzione_gr_"

Scalasca: output grafico



Cube 3.2 QT: epik_fem_0x8_sum/epitome.cube <@neo178>

File Display Topology Help

Absolute | Absolute | Own root percent

Metric tree | Call tree | Flat view | System tree | Topology 0

Metric tree
- 4.59e7 Visits
- 0 Collective synchronizations
- 0 P2P send synchronizations
- 0 P2P recv synchronizations
- 0 Collective exchange communicatior
- 0 Collective communications as sourc
- 0 Collective communications as destir
- 0 P2P send communications
- 0 P2P recv communications
- 0 Collective bytes outgoing
- 0 Collective bytes incoming
- 0 P2P bytes sent
- 0 P2P bytes received
- 0.60 OMP thread fork time
- 0.76 OMP thread management time
- 4382.53 Time

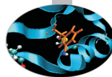
Call tree
- 74.70 MAIN__
  - 0.00 dwalltime00_
  - 0.01 inittime_
  - 0.32 printtime_
  - 1.23 ordinamento_
  - 2.34 preliminari_solutore_parallelo_
  - 0.00 q_compute_
  - 31.79 costruzione_
  - 0.02 abc03_
  - 1.86 !$omp parallel @fem.F:2177
  - 6.55 presolutore_parallelo_
  - 0.31 !$omp parallel @fem.F:2373
  - 1522.13 !$omp parallel @fem.F:25
  - 0.00 funzione_gr_
  - 840.07 !$omp parallel @fem.F:258
  - 1793.11 solutore_parallelo_
  - 108.09 !$omp parallel @fem.F:272

System tree
- Linux Intel
  - neo178
    - Process 0
      - 12.50 Thread 0
      - 12.50 Thread 1
      - 12.50 Thread 2
      - 12.50 Thread 3
      - 12.50 Thread 4
      - 12.50 Thread 5
      - 12.50 Thread 6
      - 12.50 Thread 7

0.00 4382.53 (100.00%) 4382.53 | 0.00 1.86 (0.04%) 4382.53 | 0.00 12.50 100.00 / 0.00 0.23 (12.50%) 1.86

Ready

Outline



Introduzione

Architetture

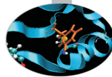
La cache e il sistema di memoria

Pipeline

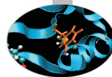
Profilers

- Motivazioni
- time
- top
- gprof
- Papi
- Scalasca
- Consigli finali

Profiling...

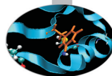


Profiling...



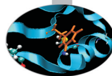
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...

Profiling...



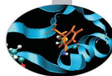
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!

Profiling...



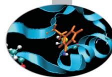
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- alcune considerazioni "pratiche":

Profiling...



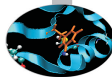
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- alcune considerazioni "pratiche":
  - usare più casi test cercando di attivare tutte le parti del codice

Profiling...



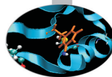
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- alcune considerazioni "pratiche":
  - usare più casi test cercando di attivare tutte le parti del codice
  - scegliere casi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

Profiling...



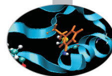
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- alcune considerazioni "pratiche":
  - usare più casi test cercando di attivare tutte le parti del codice
  - scegliere casi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

Profiling...



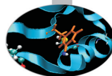
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- alcune considerazioni "pratiche":
  - usare più casi test cercando di attivare tutte le parti del codice
  - scegliere casi test il più possibile realistici
  - usare casi test caratterizzati da diverse "dimensioni" del problema
  - attenzione alla fase di input/output

Profiling...



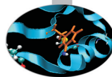
- la lezione non pretendeva di essere esaustiva sugli strumenti di Profiling mostrati...
- ..e non pretendeva di essere esaustiva su tutti i possibili strumenti di Profiling!
- alcune considerazioni "pratiche":
  - usare più casi test cercando di attivare tutte le parti del codice
  - scegliere casi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

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":
  - 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.

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

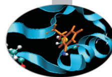
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

Floating Point Computing

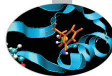
Linguaggi



- Esiste un'infinità di linguaggi differenti
- http://foldoc.org/contents/language.html

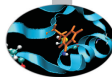
```
20-GATE; 2.PAK; 473L Query; 51forth; A#; A-0; a1; a56;
Abbreviated Test Language for Avionics Systems; ABC;
ABC ALGOL; ABCL/1; ABCL/c+; ABCL/R; ABCL/R2; ABLE;
ABSET; abstract machine; Abstract Machine Notation;
abstract syntax; Abstract Syntax Notation 1;
Abstract-Type and Scheme-Definition Language; ABSYS;
Accent; Acceptance, Test Or Launch Language; Access;
ACOM; ACOS; ACT++; Act1; Act2; Act3; Actalk; ACT ONE;
Actor; Actra; Actus; Ada; Ada++; Ada 83; Ada 95; Ada 9X;
Ada/Ed; Ada-O; Adaplan; Adaplex; ADAPT; Adaptive Simulated
Annealing; Ada Semantic Interface Specification;
Ada Software Repository; ADD 1 TO COBOL GIVING COBOL;
ADELE; ADES; ADL; AdLog; ADM; Advanced Function Presentation;
Advantage Gen; Adventure Definition Language; ADVSYS; Aeolus;
AFAC; AFP; AGORA; A Hardware Programming Language; AIDA;
AIr MAterial COmmand compiler; ALADIN; ALAM; A-language;
A Language Encouraging Program Hierarchy; A Language for Attributed ...
```

Linguaggi



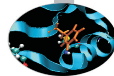
- Linguaggi interpretati
  - il linguaggio viene "tradotto" statement per statement dall'interprete durante l'esecuzione
  - impossibili ottimizzazioni tra differenti statement
  - migliore gestione degli errori semantici
  - linguaggi di scripting, Java (bytecode),. . .
- Linguaggi compilati
  - il programma viene "tradotto" dal compilatore prima dell'esecuzione
  - possibili ottimizzazioni tra differenti statement
  - gestione minimale degli errori semantici
  - Fortran, C, C++

Le CPU



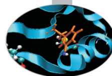
- É composta di:
  - registri (operandi delle istruzioni)
  - unità funzionali (eseguono le istruzioni)
- Unità funzionali:
  - aritmetica intera
  - operazioni logiche bitwise
  - aritmetica floating-point
  - calcolo di indirizzi
  - lettura e scrittura in memoria (load & store)
  - previsione ed esecuzione di "salti" (branch) nel flusso di esecuzione

Le CPU



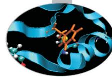
- RISC: Reduced Instruction Set CPU
  - istruzioni semplici
  - formato regolare delle istruzioni
  - decodifica ed esecuzione delle istruzioni semplificata
  - codice macchina molto "verboso"
- CISC: Complex Instruction Set CPU
  - istruzioni di semantica "ricca"
  - formato irregolare delle istruzioni
  - decodifica ed esecuzione delle istruzioni complicata
  - codice macchina molto "compatto"
- Differenza non più rilevante quanto a prestazioni: le CPU CISC di oggi convertono le istruzioni in micro operazioni RISC-like

Architettura vs. Implementazione



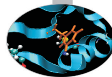
- Architettura:
  - set di istruzioni
  - registri architetturali interi, floating point e di stato
- Implementazione
  - registri fisici (2.5$\div$ 20 $\times$ registri architetturali)
  - frequenza di clock e tempo di esecuzione delle istruzioni
  - numero di unità funzionali
  - dimensione, numero, caratteristiche delle cache
  - Out Of Order execution, Simultaneous Multi-Threading
- Una architettura, più implementazioni:
  - Power: Power4, Power5, Power6, ...
  - x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Core, Core2, Athlon, Opteron, ...
  - prestazioni differenti
  - "regole" diverse per ottenere alte prestazioni

Il compilatore



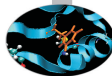
- Traduce il codice sorgente in codice macchina
- Rifiuta codici sintatticamente errati
- Segnala (alcuni) potenziali problemi semantici
- Può tentare di ottimizzare il codice
  - ottimizzazioni indipendenti dal linguaggio
  - ottimizzazioni dipendenti dal linguaggio
  - otttimizzazioni dipendenti dalla CPU
  - ottimizzazioni dipendenti dall'implementazione della CPU
  - ottimizzazioni dell'uso della memoria e della cache
  - suggerimenti al processore su cosa probabilmente farà il codice
- É uno strumento potente
  - potente: può risparmiare lavoro al programmatore
  - complesso: a volte può fare cose sorprendenti o controproducenti
  - limitato: è un sistema esperto, ma non ha l'intelligenza di un essere umano, non può capire pienamente il codice

HPC e Linguaggi



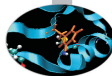
- Linguaggi ideali per l'High Performance Computing?
  - adatti all'implementazione di algoritmi "scientifici"
  - devono permettere elevati livelli di ottimizzazione
  - integrandosi nelle recenti architetture di supercalcolo
- Quali sono?
  - Fortran/C/C++, ci limitiamo a Fortran e C con qualche cenno specifico a C++
  - il Python é in ascesa, ma per le parti computazionalmente intensive si usa appoggiarsi a C o Fortran
- I compilatori piú usati per l'HPC (non sono molti)
  - GNU (gfortran, gcc, g++): "libero"
  - Intel (ifort, icc, icpc)
  - IBM (xlf, xlc, xlC)
  - Portland Group (pgf90, pgcc, pgCC)
  - PathScale, Oracle/Solaris, Fujitsu, Nag, Microsoft,...
- Linux, Windows or Mac OS X?
  - discorso complesso, ma la grande maggioranza delle architetture di supercalcolo ad oggi gira su piattaforma Linux

Building di un Programma



- Creare un eseguibile dai sorgenti é in generale un processo a tre fasi
- Pre-processing:
  - ogni sorgente é letto dal pre-processore
    - sostituire (**`#define`**) MACROs
    - inserire codice per gli statement **`#include`**
    - inserire o cancellare codice valutando **`#ifdef`**, **`#if`** ...
- Compilazione:
  - ogni sorgente é tradotto in un codice oggetto
    - un file oggetto é una collezione organizzata di simboli che si riferiscono a variabili e funzioni definite o usate nel sorgente
- Linking:
  - file oggetti sono combinati per costruire il singolo eseguibie finale
  - ogni simbolo deve essere risolto
    - i simboli possono essere definiti nei file oggetto
    - o disponibili in altri codici oggetti (librerie esterne)

Compilare con GNU gfortran



- Quando si dá il comando:

```
user@cineca$> gfortran dsp.f90 dsp_test.f90
```

vengono eseguiti automaticamente i tre passi

- Pre-processing

```
user@cineca$> gfortran -E -cpp dsp.f90
user@cineca$> gfortran -E -cpp dsp_test.f90
```

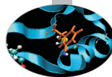
  - l'opzione `-E -cpp` dice a `gfortran` di fermarsi after pre-process
  - semplicemente chiama `cpp` (automaticamente chiamata se l'estensione é `F90`)

- Compilazione dei sorgenti

```
user@cineca$> gfortran -c dsp.f90
user@cineca$> gfortran -c dsp_test.f90
```

  - l'opzione `-c` dice `gfortran` di compilare solo i sorgenti
  - da ogni sorgente viene prodotto un file oggetto `.o`

Linking con GNU gfortran



- Linkare oggetti tra di loro

```
user@cineca$> gfortran dsp.o dsp_test.o
```

- Per risolvere i simboli definiti in librerie esterne specificare:
  - le librerie da usare (opzione `-l`)
  - le directory in cui stanno (opzione `-L`)
- Come linkare `libblas.a` nella cartella `/opt/lib`

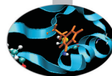
```
user@cineca$> gfortran file1.o file2.o -L/opt/lib -ldsp
```

- Come creare e linkare una libreria statica

```
user@cineca$> gfortran -c dsp.f90
ar curv libdsp.a dsp.o
ranlib libdsp.a
gfortran test_dsp.f90 -L. -ldsp
```

  - `ar` create the archive `libdsp.a` containing `dsp.o`
  - `ranlib` generare un indice per l'archiviazione

Prendere confidenza



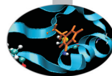
- Il manuale in linea, e.g.

```
man gcc
```

  riporta le opzioni del compilatore C di GNU e il loro significato

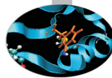
- Attenzione: **man gfortran** dá solo le opzioni ulteriori rispetto a gcc, mentre per gli altri compilatori solitamente i man sono replicati nelle parti comuni
- Il numero di opzioni é notevole e purtroppo differisce da compilatore a compilatore
- Tipologia di opzioni
  - linguaggio: sullo standard (o sul dialetto) da seguire
  - ottimizzazione: argomento delle prossime slide...
  - target: per l'integrazione con l'architettura di calcolo
  - debugging: la piú importante, **-g**, crea i simboli di debugging necessari per l'uso di debugger
  - warning: per avere informazioni sulla compilazione, utile per debugging e/o ottimizzazione

Il compilatore cosa sa fare



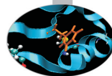
- Esegue trasformazioni del codice come:
  - Register allocation
  - Register spilling
  - Copy propagation
  - Code motion
  - Dead and redundant code removal
  - Common subexpression elimination
  - Strength reduction
  - Inlining
  - Index reordering
  - Loop pipelining , unrolling, merging
  - Cache blocking
  - ...
- Lo scopo è massimizzare le prestazioni

Il compilatore cosa non sa fare



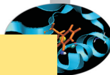
- Analizzare ed ottimizzare globalmente codici molto grossi (a meno di abilitare l'IPO, molto costoso in tempo e risorse)
- Capire dipendenze tra dati con indirizzamenti indiretti
- Strenght reduction di potenze non intere, o maggiori di 2÷4
- Common subexpression elimination attraverso chiamate a funzione
- Unrolling, Merging, Blocking con:
  - chiamate a funzioni e(o) subroutine
  - chiamate o statement di Input-Output in mezzo al codice
- Fare inlining di funzioni se non viene detto esplicitamente
- Sapere a run-time i valori delle variabili per i quali alcune ottimizzazioni sono inibite

Livelli di ottimizzazione



- I compilatori forniscono dei livelli di ottimizzazione "predefiniti" utilizzabili con la semplice opzione **`-O<n>`**
  - **`n`** accresce il livello di ottimizzazione, da 0 a 3 (a volte fino a 5)
- Fortran IBM:
  - **`-O0`**: nessuna ottimizzazione (utile insieme a **`-g`** in debugging)
  - **`-O2, -O`** : ottimizzazioni locali, compromesso tra velocità di compilazione, ottimizzazione e dimensioni dell'eseguibile
  - **`-O3`**: ottimizzazioni memory-intensive, può alterare la semantica del programma (da qui in poi da considerare l'uso si **`-qstrict`** per evitare risultati errati)
  - **`-O4`**: ottimizzazioni aggressive (**`-qarch=auto`**, **`-qhot`**, **`-qipa`**, **`-qtune=auto`**, **`-qcache=auto`**, **`-qsimd=auto`**)
  - **`-O5`**: come **`-O4`** con **`-qipa=level=2`** aggressiva e lenta
- Alcuni compilatori hanno **`-fast`**, che include **`On`** e altro
- Per GNU una scelta comune insieme a **`-O3`** é **`-funroll-loops`**
- Attenzione all'ottimizzazione di default: per GNU é **`-O0`** mentre per gli altri di solito é **`-O2`**

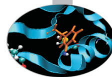
Loop con -O3 (Compilatore Intel)



```
icc (or ifort) -O3
```

- Automatic vectorization (use of packed SIMD instructions)
- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Prefetching (for patterns not recognized by h/w prefetcher)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop versioning (for loop count; data alignment; runtime dependency tests)
- Memcpy recognition (call Intel's fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar replacement (reduce array accesses by scalar temps)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

Dal sorgente all’eseguibile

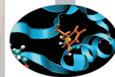


- La macchina astratta "vista" a livello di sorgente è molto diversa da quella reale
- Esempio: prodotto di matrici

```
do j = 1, n
do k = 1, n
do i = 1, n
   c(i,j) = c(i,j) + a(i,k)*b(k,j)
end do
end do
end do
```

- Il “nocciolo”
  - carica dalla memoria tre valori
  - fa una moltiplicazione ed una somma
  - immagazzina il risultato

Hands-on: cluster PLX



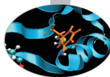
- Accedere al cluster PLX con le proprie credenziali

```
ssh -X <user_name>@login.plx.cineca.it
```

  - in questo modo si accede al cosiddetto nodo di front-end
  - il front-end guida l'utente per operare sui nodi di calcolo attraverso un sistema di code

```
Model: IBM iDataPlex DX360M3
Architecture: Linux Infiniband Cluster
Processors Type:
-Intel Xeon (Esa-Core Westmere) E5645 2.4 GHz (Compute)
-Intel Xeon (Quad-Core Nehalem) E5530 2.66 GHz (Service & Login)
Number of nodes: 274 Compute + 1 Login + 1 Service + 8 Fat +
                 6 RVN + 8 Storage  + 2 Management
Number of cores: 3288 (Compute)
Number of GPUs:528 nVIDIA Tesla M2070 + 20 nVIDIA Tesla M2070Q
RAM: 14 TB (48 GB/Compute node + 128GB/Fat node)
```

Hands-on: PLX e code



- Per testare le performance nei casi reali occorre usare i nodi calcolo (diversi da quelli di front-end!)
  - occorre preparare un file di script con in comandi, e.g. **qsub.script**

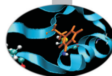
```
#!/bin/bash
#PBS -A train_cspR2013
#PBS -W group_list="train_cspR2013"
#PBS -l walltime=00:10:00 # or other needed time
#PBS -l select=1:ncpus=1  # or ncpus=N, N is the number of CORES
#PBS -q private
cd $PBS_O_WORKDIR
# (a) Load Modules, e.g.
module load profile/advanced
module load gnu/4.7.2
# (b) Run executables
./matmul >& matmul.out
```

- Il file di script viene sottomesso in coda con il comando:

`qsub qsub.script`

- La coda private permette di accedere ai nodi `fat`

Hands-on: PLX e moduli

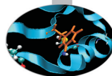


- Il software installato é organizzato in moduli
  - per poter usare i moduli della lista completa advanced (farlo come prima cosa!): **`module load profile/advanced`**
  - per avere la lista dei moduli disponibili: **`module av`**
  - per avere la lista dei moduli caricati: **`module li`**
  - per caricare un modulo, e.g: **`module load gnu/4.7.2`**
  - per scaricare un modulo, e.g: **`module unload gnu/4.7.2`**
  - per scaricare tutti i moduli caricati, e.g: **`module purge`**

- I codici per le esercitazioni si trovano in

```
/gpfs/scratch/userinternal/fsalvado/OPT_2013/Exercises
```

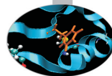
Hands-on: Livelli di Ottimizzazione



- Prodotto matrice-matrice, 1024×1024, doppia precisione
- Ordine dei loop ottimale per la cache
- Architettura considerata per la prove, PLX:
  - 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz per nodo
- Per caricare i compilatori: (`module load profile/advanced`):
  - GNU: `module load gnu/4.7.2`
  - Intel: `module load intel/cs-xe-2013--binary`
  - PGI: `module load pgi/12.10`
  - Fare `unload` di un compilatore prima di caricarne un altro

	GNU	Intel	PGI	GNU	Intel	PGI
Opzione	secondi	secondi	secondi	GFlops	GFlops	GFlops
-O0						
-O1						
-O2						
-O3						
-O3 -funroll-loops		——	——		——	——
-fast	——			——		

Prodotto di matrici: performance



- Prodotto matrice-matrice, 1024×1024, doppia precisione
- Ordine dei loop ottimale per la cache
- Scritto in Fortran
- Architetture considerate per le prove
  - FERMI: IBM Blue Gene/Q system, nodi da 16 core single-socket PowerA2 a 1.6 GHz di frequenza
  - PLX: 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz per nodo

FERMI - xlf

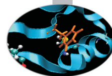
Opzione	secondi	Mflops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX - ifort

Opzione	secondi	MFlops
-O0	8.94	240
-O1	1.41	1514
-O2	0.72	2955
-O3	0.33	6392
-fast	0.32	6623

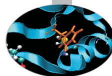
- Perché tanta varietà di risultati?
- Basta passare da -On a -On+1?

Cosa fa il compilatore: report



- Cosa accade ai diversi livelli di ottimizzazione?
  - Perché il compilatore IBM su Fermi al livello **`-O4`** degrada cosí le performance?
- Utilizzare le opzioni di report é un buon modo di capire cosa sta facendo il compilatore
- Su IBM **`-qreport`** mostra che per **`-O4`** l'ottimizzazione prende un percorso completamente diverso dagli altri casi
  - il compilatore riconosce il pattern del prodotto matrice-matrice e sostituisce le righe di codice con la chiamata a una funzione di libreria BLAS **`__xl_dgemm`**
  - che peró si rivela molto lenta perché non fa parte delle librerie matematiche ottimizzate da IBM (ESSL)
  - anche il compilatore Intel fa questo per dgemm, ma invoca le efficienti MKL
- Aumentando il livello di ottimizzazione, solitamente le performance migliorano
  - ma é bene testare questo miglioramento per il proprio codice

Uno sguardo all'assembler

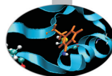


- Esempio datato, utile peró per capire

Matrix Multiply inner loop code with -qnoopt

38 instructions, 31.4 cycles per iteration

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

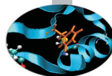


Matrix Multiply inner loop code with -qnoopt

necessary instructions

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Loop control

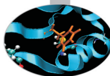


Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

L'indirizzamento!

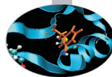


Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control   addressing code

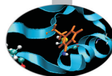
```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Cosa fare?



- Le operazioni dominanti sono quelle di conversione indici indirizzo di memoria
- Osservazioni:
  - il loop “percorre” la memoria sequenzialmente
  - gli indirizzi degli elementi successivi sono calcolabili facilmente sommando una costante
  - sfruttare una conversione indice indirizzo per piú elementi successivi
- Puó essere fatto automaticamente?

Ottimizziamo



Matrix Multiply inner loop code with -O3 -qtune=pwr4

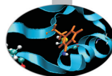
```
__L1:
    fmadd   fp6,fp12,fp13,fp6
    lfdux   fp12,r12,r7
    lfd     fp13,8(r11)
    fmadd   fp7,fp8,fp9,fp7
    lfdux   fp8,r12,r7
    lfd     fp9,16(r11)
    lfdux   fp10,r12,r7
    lfd     fp11,24(r11)
    fmadd   fp1,fp12,fp13,fp1
    lfdux   fp12,r12,r7
    lfd     fp13,32(r11)
    fmadd   fp0,fp8,fp9,fp0
    lfdux   fp8,r12,r7
    lfd     fp9,40(r11)
    fmadd   fp2,fp10,fp11,fp2
    lfdux   fp10,r12,r7
    lfd     fp11,48(r11)
    fmadd   fp4,fp12,fp13,fp4
    lfdux   fp12,r12,r7
    lfd     fp13,56(r11)
    fmadd   fp3,fp8,fp9,fp3
    lfdux   fp8,r12,r7
    lfdu    fp9,64(r11)
    fmadd   fp5,fp10,fp11,fp5
    bdnz    __L1
```

unrolled by 8

dot product accumulated in
8 interleaved parts (fp0-fp7)
(combined after loop)

3 instructions, 1.6 cycles per iteration
2 loads and 1 fmadd per iteration

Ottimizziamo di piú



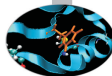
Matrix multiply inner loop code with -O3 -qhot -qtune=pwr4

```
__L1:
    fmadd   fp1,fp4,fp2,fp1
    fmadd   fp0,fp3,fp5,fp0
    lfdux   fp2,r29,r9
    lfdu    fp4,32(r30)
    fmadd   fp10,fp7,fp28,fp10
    fmadd   fp7,fp9,fp7,fp8
    lfdux   fp26,r27,r9
    lfd     fp25,8(r29)
    fmadd   fp31,fp30,fp27,fp31
    fmadd   fp6,fp11,fp30,fp6
    lfd     fp5,8(r27)
    lfd     fp8,16(r28)
    fmadd   fp30,fp4,fp28,fp29
    fmadd   fp12,fp13,fp11,fp12
    lfd     fp3,8(r30)
    lfd     fp11,8(r28)
    fmadd   fp1,fp4,fp9,fp1
    fmadd   fp0,fp13,fp27,fp0
    lfd     fp4,16(r30)
    lfd     fp13,24(r30)
    fmadd   fp10,fp8,fp25,fp10
    fmadd   fp8,fp2,fp8,fp7
    lfdux   fp9,r29,r9
    lfdu    fp7,32(r28)
    fmadd   fp31,fp11,fp5,fp31
    fmadd   fp6,fp26,fp11,fp6
    lfdux   fp11,r27,r9
    lfd     fp28,8(r29)
    fmadd   fp12,fp3,fp26,fp12
    fmadd   fp29,fp4,fp25,fp30
    lfd     fp30,-8(r28)
    lfd     fp27,8(r27)
    bdnz      L1
```

unroll-and-jam 2x2
inner unroll by 4
interchange "i" and "j" loops

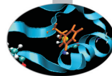
2 instructions, 1.0 cycles per iteration
balanced: 1 load and 1 fmadd per iteration

Istruzioni macchina



- Istruzioni per `c(i,j) = c(i,j) + a(i,k)*b(k,j)`
- -O0: 24 istruzioni
  - 3 load/1 store
  - 1 floating point multiply+add Flop/istruzione 2/24
- -O2: 9 istruzioni (riuso calcolo indirizzi)
  - 4 load/1 store
  - 2 floating point multiply+add Flop/istruzione 4/9
- -O3: 150 istruzioni (unrolling)
  - 68 load/ 34 store
  - 48 floating point multiply+add Flop/istruzione 96/150
- -O4: 344 istruzioni (unrolling&blocking)
  - 139 load / 74 store
  - 100 floating point multiply+add Flop/istruzione 200/344

Fa tutto il compilatore?



- **`-fast`** realizza uno speed-up di 30 volte rispetto a **`-O0`** per il caso matrice-matrice (ifort su PLX)
  - mette in atto una vasta gamma di ottimizzazioni piú o meno complicate
- **Ha senso ottimizzare il codice anche manualmente?**
- Il compilatore sa fare automaticamente
  - Dead code removal: per esempio rimuovere un if

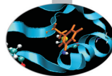
```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
    ......
}
```

  - Redudant code removal

```
integer, parameter :: c=1.0
f=c*f
```

- Ma la vita non è sempre così facile

Loop counters: attenzione ai tipi



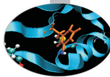
- Usare sempre i tipi corretti
- Usare un real per l'indice dei loop implica una trasformazione implicita reale → intero ...
- Secondo i recenti standard Fortran si tratta di un vero e proprio errore, ma i compilatori tendono a tollerarlo

```
real :: i,j,k
....
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
```

Risultati in secondi

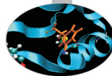
Compilazione	integer	real
(PLX) gfortran -O0	9.96	8.37
(PLX) gfortran -O3	0.75	2.63
(PLX) ifort -O0	6.72	8.28
(PLX) ifort -fast	0.33	1.74
(PLX) pgif90 -O0	4.73	4.85
(PLX) pgif90 -fast	0.68	2.30
(FERMI) bgxlf -O0	64.78	104.10
(FERMI) bgxlf -O5	0.64	12.38

Il compilatore può fare tutto?



- Il compilatore può fare molto . . . ma non è un essere umano
- È piuttosto facile intralciare il suo lavoro
  - corpo del loop troppo lungo
  - loop con i due estremi di iterazione variabili
  - uso eccessivo di costrutti condizionali(if)
  - uso eccessivo di puntatori ed indici
  - uso improprio di variabili intermedie
- Importante:
  - due codici semanticamente uguali possono avere prestazioni ben diverse
  - il compilatore può fare assunzioni erronee ed alterare la semantica

Il compilatore: index reordering



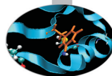
- Per un loop nest semplice ci pensa il compilatore
  - a patto di usare un opportuno livello di ottimizzazione

```
do i=1,n
do k=1,n
do j=1,n
   c(i,j) = c(i,j) + a(i,k)*b(k,j)
end do
end do
end do
```

- Tempi in secondi

Compilazione	j-k-i	i-k-j
(PLX) ifort -O0	6.72	21.8
(PLX) ifort -fast	0.34	0.33

Il compilatore: index reordering / 2



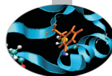
- Per loop nesting piú complicati il compilatore a volte no...
  - anche al piú alto livello di ottimizzazione
  - conoscere il meccanismo di cache é quindi utile!

```
do jj = 1, n, step
   do kk = 1, n, step
      do ii = 1, n, step
         do j = jj, jj+step-1
            do k = kk, kk+step-1
               do i = ii, ii+step-1
                  c(i,j) = c(i,j) + a(i,k)*b(k,j)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

- Tempi in secondi

Compilazione	j-k-i	i-k-j
(PLX) ifort -O0	10	11.5
(PLX) ifort -fast	1.	2.4

Inlining



- Ottimizzazione manuale o effettuata dal compilatore che sostituisce una funzione col suo corpo
  - elimina il costo della chiamata e potenzialmente l'instruction cache
  - rende piú facile l'ottimizzazione interprocedurale
- In C e C++ la keyword **`inline`** é un “suggerimento”
- Non ogni funzione é “inlinable” e in ogni caso dipende dalle capacitá del compilatore
  - oltre che dalle capacitá del programmatore
- Intel (n: 0=disable, 1=secondo la keyword, 2=se opportuno)

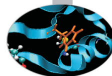
```
-inline-level=n
```

- GNU (n: size, default is 600):

```
-finline-functions
-finline-limit=n
```

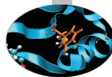
- In alcuni compilatori automaticamente attivate ad alti livelli di ottimizzazione

Common Subexpression Elimination



- Per i calcoli intermedi si riusano spesso alcune espressioni: può essere vantaggioso riciclare quantità già calcolate:
  A= B+C+D
  E= B+F+C
- Richiede: 4 load,2 store, 4 somme
  A=(B+C) + D
  E=(B+C) + F
- Richiede: 4 load, 2 store, 3 somme
- Attenzione: dal punto di vista numerico il risultato non é necessariamente identico
- Se la locazione di un array é acceduta piú di una volta puó convenire effettuare lo “Scalar replacement”
  - ad opportune ottimizzazioni il compilatore puó farlo

Funzioni e Side Effects



- Lo scopo “primo” di una funzione é in genere dare un valore in ritorno
  - a volte, per vari motivi, peró non é cosí
  - la modifica di variabili passate, o globali o anche l’I/O si chiamano comunque side effects (effetti collaterali)
- La presenza di funzioni con side effects può inibire il compilatore dal fare ottimizzazioni
- Se:

```
function f(x)
   f=x+dx
end
```

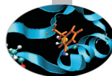
allora `f(x)+f(x)+f(x)` puó essere valutato come `3*f(x)`

- Se:

```
function f(x)
   x=x+dx
   f=x
end
```

allora la precedente valutazione non é piú corretta

CSE e chiamate a funzione



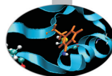
- Alterando l'ordine delle chiamate il compilatore non sa se si altera il risultato (possibili effetti collaterali)
- 5 chiamate a funzioni, 5 prodotti:

```
x=r*sin(a)*cos(b);
y=r*sin(a)*sin(b);
z=r*cos(a);
```

- 4 chiamate a funzioni, 4 prodotti (1 variabile temporanea):

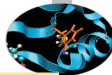
```
temp=r*sin(a)
x=temp*cos(b);
y=temp*sin(b);
z=r*cos(a);
```

CSE: limitazioni



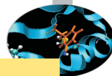
- Core loop troppo grossi:
  - il compilatore lavora su finestre di dimensioni finite: potrebbe non accorgersi di una grandezza da riutilizzare
- Funzioni:
  - se altero l'ordine delle chiamate ottengo lo stesso risultato?
- Ordine e valutazione:
  - solo ad alti livelli di ottimizzazione il compilatore altera l'ordine delle operazioni (**`-qnostrict`** per IBM)
  - per inibirla in certe espressioni: mettere le parantesi (il programmatore ha sempre ragione)
- Aumenta l'uso di registri per l'appoggio dei valori intermedi ("register spilling")

Cosa può fare il compilatore?



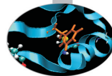
```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         acc =1./(1.-coe*aciv(i)*(1.-int(forclo(nve,i,j,k))))
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         ami(jj,i)=-coe*amiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         fi(jj,i)=qcap(i,j,k)*acc
     enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         acc =1./(1.-coe*ackv(k)*(1.-int(forclo(nve,i,j,k))))
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         amk(jj,k)=-coe*amkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

Probabilmente questo



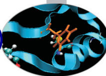
```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*aciv(i)*temp)
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*temp
         ami(jj,i)=-coe*amiv(i)*acc*temp
         fi(jj,i)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

Ma non questo



```
do k=1,n3m
   do j=n2i,n2do
      do i=1,n1m
         temp_fact(i,j,k) = 1.-int(forclo(nve,i,j,k))
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = temp_fact(i,j,k)
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
!  idem per l'altro loop
```

Array syntax



- Traslazione dei primi due indici di un array a tre indici (512^3)
- Il “caro vecchio” loop (stile Fortran 77): 0.19 secondi
  - l’ordine dei cicli negli indici che traslano é inverso alla traslazione per evitare di “sporcare” i dati della matrice

```
do k = nd, 1, -1
  do j = nd, 1, -1
    do i = nd, 1, -1
      a03(i,j,k) = a03(i-1,j-1,k)
    enddo
  enddo
enddo
```

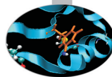
- Array syntax (stile Fortran 90): 0.75 secondi
  - secondo lo standard, le cose vanno “come se il membro a destra fosse tutto valutato prima di effettuare le operazioni richieste”

```
a03(1:nd, 1:nd, 1:nd) = a03(0:nd-1, 0:nd-1, 1:nd)
```

- Array syntax con un hint al compilatore: 0.19 secondi

```
a03(nd:1:-1,nd:1:-1,nd:1:-1) = a03(nd-1:0:-1, nd-1:0:-1, nd:1:-1)
```

Report di ottimizzazione



- Per capirne di piú in questo come in altri casi é bene attivare le flag di report di ottimizzazione
- Con Intel ifort attivare

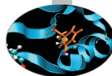
```
-opt-report[n]     n=0(none),1(min),2(med),3(max)
-opt-report-file<file>
-opt-report-phase<phase>
-opt-report-routine<routine>
```

- Le tre modalitá sono alle righe 55,64,69 rispettivamente

```
Loop at line:55 simple MEMOP Intrinsic disabled-->SIMPLE reroll
Loop at line:64 memcopy generated
Loop at line:69 simple MEMOP Intrinsic disabled-->SIMPLE reroll
```

- Tutto questo é ovviamente molto dipendente dal compilatore

Report di ottimizzazione / 2



- Purtroppo non é presente un opzione equivalente con compilatori GNU
  - La migliore alternativa é specificare

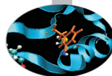
    ```
    -fdump-tree-all
    ```

    in modo che vengano stampate tutte le fase intermedie di compilazione
  - ma la lettura non é decisamente agevole
- Con il compilatore PGI

  ```
  -Minfo=accel,inline,ipa,loop,lre,mp,opt,par,unified,vect
  ```

  oppure senza opzioni per averle tutte

Dare informazioni al compilatore



- Estremi del loop noti a compile time o solo a run-time:
  - puó inibire alcune ottimizzazioni, tra cui l'unrolling

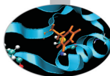
```
real a(1:1024,1:1024)
real b(1:1024,1:1024)
real c(1:1024,1:1024)
...
read(*,*) i1,i2
read(*,*) j1,j2
read(*,*) k1,k2
...
do j = j1, j2
do k = k1, k2
do i = i1, i2
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

- Tempi in secondi (Loop Bounds Compile-Time o Run-Time)

Compilazione	LB-CT	LB-RT
(PLX) ifort -O0	6.72	9
(PLX) ifort -fast	0.34	0.75

- Molto dipendente dal tipo di loop, dal compilatore, etc.

Allocazione statica o dinamica ?

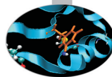


- Potenzialmente l’allocazione statica puó dare al compilatore piú informazioni per ottimizzare
  - a prezzo di un codice piú rigido
  - l’elasticitá permessa dall’allocazione dinamica é particolarmente utile nel calcolo parallelo

```
integer :: n
parameter(n=1024)
real a(1:n,1:n)
real b(1:n,1:n)
real c(1:n,1:n)
```

```
real, allocatable, dimension(:,:) :: a
real, allocatable, dimension(:,:) :: b
real, allocatable, dimension(:,:) :: c
print*,’Enter matrix size’
read(*,*) n
allocate(a(n,n),b(n,n),c(n,n))
```

Allocazione statica o dinamica ? / 2



- Per i compilatori recenti peró spesso le prestazioni statica vs dinamica si equivalgono
  - per il semplice matrice-matrice l'allocazione dinamica gestisce meglio i loop bounds letti da input

Compilazione	statica	dinamica	dinamica-LBRT
(PLX) ifort -O0	6.72	18.26	18.26
(PLX) ifort -fast	0.34	0.35	0.36

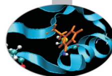
- L'allocazione statica viene fatta nella memoria cosiddetta "stack"
  - in compilazione possono esserci dei limiti di utilizzo per cui occorre specificare l'opzione **-mcmodel=medium**
  - a run-time assicurarsi che sul nodo la stack non sia limitata (se si usa bash)

```
ulimit -a
```

ed eventualmente

```
ulimit -s unlimited
```

Allocazione dinamica in C



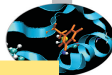
- C non conosce matrici ma array di array
  - l'allocazione statica garantisce allocazione contigua di tutti i valori

```
   double A[nrows][ncols];
```

- Con l'allocazione dinamica occorre fare attenzione
  - "the wrong way" (= non efficiente)

```
/* Allocate a double matrix with many malloc */
double** allocate_matrix(int nrows, int ncols) {
   double **A;
   /* Allocate space for row pointers */
   A = (double**) malloc(nrows*sizeof(double*) );
   /* Allocate space for each row */
   for (int ii=1; ii<nrows; ++ii) {
      A[ii] = (double*) malloc(ncols*sizeof(double));
   }
   return A;
}
```

Allocazione dinamica in C / 2



- Si puó allocare un array lineare

```
/* Allocate a double matrix with one malloc */
double* allocate_matrix_as_array(int nrows, int ncols) {
   double *arr_A;
   /* Allocate enough raw space */
   arr_A = (double*) malloc(nrows*ncols*sizeof(double));
   return arr_A;
}
```

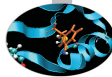
- e usarlo come una matrice (linearizzazione dell'indice)

```
arr_A[i*ncols+j]
```

- le MACROs possono aiutare
- e, eventualmente aggiungere una matrice di puntatori che puntano all'array allocato

```
/* Allocate a double matrix with one malloc */
double** allocate_matrix(int nrows, int ncols, double* arr_A) {
   double **A;
   /* Prepare pointers for each matrix row */
   A = new double*[nrows];
   /* Initialize the pointers */
   for (int ii=0; ii<nrows; ++ii) {
      A[ii] = &(arr_A[ii*ncols]);
   }
   return A;
}
```

Aliasing e Restrict

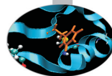


- In C, se due puntatori puntano ad una stessa area di memoria, si parla di “aliasing”
- Il rischio di aliasing puó **molto** limitare l’ottimizzazione del compilatore
  - difficile invertire l’ordine delle operazioni
  - particolarmente per gli argomenti passati a una funzione
- Lo standard C99 introduce la keyword `restrict` per indicare che l’aliasing non é possibile

```
void saxpy(int n, float a, float *x, float* restrict y)
```

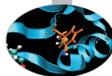
- In C++, si assume che l’aliasing non possa avvenire tra puntatori a tipi diversi (strict aliasing)

Aliasing e Restrict / 2



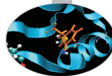
- Il Fortran assume che gli argomenti di procedure non possano puntare a identiche aree di memoria
  - tranne che per gli array per i quali gli indici permettono comunque un'analisi corretta
  - o per i **`pointer`** che peró vengono usati ove necessario
  - un motivo per cui il Fortran spesso ottimizza meglio del C!
- I compilatori permettono di configurare le assunzioni dell'aliasing (vedere il man)
  - GNU (solo strict-aliasing): **`-fstrict-aliasing`**
  - Intel (eliminazione completa): **`-fno-alias`**
  - IBM (no overlap per array): **`-qalias=noaryovrlp`**

Non solo compilatore



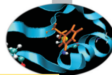
- Il compilatore ha associata una runtime library
- Contiene funzioni chiamate esplicitamente
  - funzioni trigonometriche e trascendenti
  - manipolazioni di bit
  - funzioni Input Output (C)
- Contiene funzioni chiamate implicitamente
  - funzioni Input Output (Fortran)
  - operatori complessi del linguaggio
  - routine di utilità generiche, gestione eccezioni, . . .
  - routine di supporto ad un particolare modello di calcolo (OpenMP, UPC, GAF)
- Può essere fondamentale per le prestazioni
  - qualità dell'implementazione
  - funzioni matematiche accurate vs. veloci

Input/Output



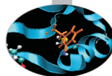
- È sempre mediato dal sistema operativo
  - causa chiamate di sistema
  - comporta lo svuotamento della pipeline
  - distrugge la coerenza dei dati in cache
  - può alterare la priorità di scheduling
  - è lento
- Regolo d'oro n.1: MAI mescolare calcolo intensivo con I/O
- Regolo d'oro n.2: leggere/scrivere i dati in blocco, non pochi per volta
- Attenzione ad I/O nascosti: swapping
  - avviene quando la RAM è insufficiente
  - usa il disco come surrogato
  - unica soluzione: fuggirlo come la peste

Ci sono più modi di fare I/O



```
do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69,*) a(i,j,k)                                   ! formattato
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69) a(i,j,k)                                     ! binario
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n
write(69) (a(i,j,k),i=1,n)                             ! colonne
enddo     ;  enddo

do k=1,n
write(69) ((a(i,j,k),i=1),n,j=1,n)                     ! matrice
enddo

write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n)             ! blocco

write(69) a                                            ! dump
```

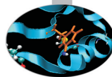
Con prestazioni differenti



Opzione	secondi	Kbyte
formattato	81.6	419430
binario	81.1	419430
colonne	60.1	268435
matrice	0.66	134742
blocco	0.94	134219
dump	0.66	134217

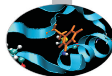
- Il file-system e anche il suo utilizzo hanno un notevole impatto sui tempi

I/O riassumendo



- La lettura/scrittura dei dati formattati è lenta
- Leggere/scrivere i dati in formato binario
- Leggere/scrivere in un blocco e non uno per volta
- Scegliere il file system più efficiente a disposizione
- I buffer di scrittura possono nascondere latenze
- Ma l'impatto sul calcolo sarà comunque devastante
- Attenzione al dump di array in caso di padding
- Soprattutto per il calcolo parallelo:
  - usare librerie di I/O: MPI-I/O, HDF5, NetCDF,...

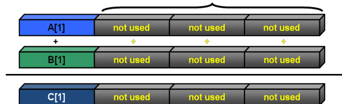
Unitá vettoriali



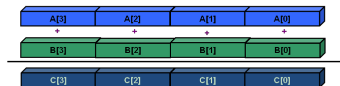
- Da non confondere con le macchine vettoriali!
- Le unitá vettoriali lavorano con set di istruzioni SIMD e circuiti dedicati a operazioni floating-point simultanee
  - Intel MMX (1996), AMD 3DNow! (1998), Intel SSE (1999) che aggiungono nuovi registri e possibilitá floating point
  - Nuove istruzioni (packet) SSE2, SSE3, SSE4, AVX
- Esempio di vettorizzazione: addizione di due array a 4 componenti puó diventare una singola istruzione

```
c(0) = a(0) + b(0)
c(1) = a(1) + b(1)
c(2) = a(2) + b(2)
c(3) = a(3) + b(3)
```

non vettorizzato

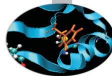


vettorizzato



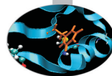
- L'esecuzione SSE é sincrona

Evoluzione unitá SIMD



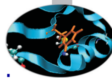
- SSE: registri a 128 bit (intel Core - AMD Opteron)
  - 4 operazioni floating/integer in singola precisione
  - 2 operazioni floating/integer in doppia precisione
- AVX: registri a 256 bit (intel Sandy Bridge - AMD Bulldozer)
  - 8 operazioni floating/integer in singola precisione
  - 4 operazioni floating/integer in doppia precisione
- MIC: registri a 512 bit (Intel Knights Corner - 2013)
  - 16 operazioni floating/integer in singola precisione
  - 8 operazioni floating/integer in doppia precisione

Vettorizzazione



- La vettorizzazione dei loop puó incrementare drammaticamente le performance
- Ma per essere vettorizzabili, i loop devono obbedire a certi criteri
- E il programmatore deve aiutare il compilatore a verificarli
- Anzitutto, l'assenza di dipendenza tra i dati di diverse iterazioni
  - circostanza frequente ma non troppo in ambito HPC
- Altri criteri
  - Countable (numero delle iterate costante)
  - Single entry-single exit (nessun break or exit)
  - Straight-line code (nessun branch a meno di implementazioni come assegnazione di mask)
  - Deve essere il loop interno di nest
  - Nessuna chiamata a funzione (eccetto quelle matematiche o quelle inlined)
- AVX puó essere una sorgente di risultati diversi in calcolo numerico (e.g., Fused Multiply Addiction)

Algoritmi e vettorizzazione



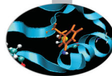
- Differenti algoritmi per lo stesso scopo possono comportarsi diversamente rispetto alla vettorizzazione
  - Gauss-Seidel: dipendenza tra le iterazioni, non vettorizzabile

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i][j] = w0 * a[i][j] +
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

- Jacobi: nessuna dipendenza tra le iterazioni, vettorizzabile

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     b[i][j] = w0*a[i][j] +
       w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     a[i][j] = b[i][j];
```

Ottimizzazione e vettorizzazione



- Alcuni comuni “coding tricks” possono impedire la vettorizzazione
  - vettorizzabile

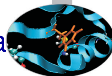
```
for( i = 0; i < n-1; ++i ){
   b[i] = a[i] + a[i+1];
}
```

  - **x** a una certa iterazione é necessaria per lo step successivo

```
x = a[0];
for( i = 0; i < n-1; ++i ){
  y = a[i+1];
  b[i] = x + y;
  x = y;
}
```

- Quando si compila é bene controllare se la vettorizzazione é stata attivata
- In caso contrario, si puó provare ad aiutare il compilatore
  - modificando il codice per renderlo vettorizzabile
  - inserendo direttive per forzare la vettorizzazione

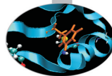
Direttive di vettorizzazione



- Se il programmatore ha certezza che una certa dipendenza riscontrata dal compilatore sia in realtá solo apparente puó forzare la vettorizzazione con direttive “compiler dependent”
  - Intel Fortran: **`!DIR$ simd`**
  - Intel C: **`#pragma simd`**
- Poiché **`inow`** é diverso da **`inew`**, la dipendenza é solo apparente

```
62     do k = 1,n
63!DIR$ simd
        do i = 1,l
...
66          x02 = a02(i-1,k+1,inow)
67          x04 = a04(i-1,k-1,inow)
68          x05 = a05(i-1,k  ,inow)
            x06 = a06(i  ,k-1,inow)
            x11 = a11(i+1,k+1,inow)
            x13 = a13(i+1,k-1,inow)
72          x14 = a14(i+1,k  ,inow)
73          x15 = a15(i  ,k+1,inow)
74          x19 = a19(i  ,k  ,inow)
75
76          rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
...
126         a05(i,k,inew) = x05 - omega*(x05-e05) + force
127         a06(i,k,inew) = x06 - omega*(x06-e06)
```

Vettorizzazione manuale

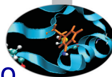


- É possibile istruire direttamente il codice delle funzioni vettoriali da utilizzare
- In pratica, si tratta di scrivere un loop che fa quattro iterazioni alla volta usando registri e operazioni vettoriali, ed essere pratici con le “mask”

```
void scalar(float* restrict result,
            const float* restrict v,
            unsigned length)
{
  for (unsigned i = 0; i < length; ++i)
  {
    float val = v[i];
    if (val >= 0.f)
      result[i] = sqrt(val);
    else
      result[i] = val;
  }
}
```

```
void sse(float* restrict result,
         const float* restrict v,
         unsigned length)
{
  __m128 zero = _mm_set1_ps(0.f);

  for (unsigned i = 0; i <= length - 4; i += 4)
  {
    __m128 vec  = _mm_load_ps(v + i);
    __m128 mask = _mm_cmpge_ps(vec, zero);
    __m128 sqrt = _mm_sqrt_ps(vec);
    __m128 res  =
         _mm_or_ps(_mm_and_ps(mask, sqrt),
         _mm_andnot_ps(mask, vec));
    _mm_store_ps(result + i, res);
  }
}
```

Parallelizzazione automatica



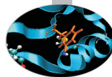
- Alcuni compilatori offrono opzioni per sfruttare il parallelismo architetturale delle macchina (e.g., i cores) senza modificare il codice sorgente
- Shared Memory Parallelism (solo intra-nodo)
- Simile a OpenMP ma non richiede direttive
  - performance attese piú limitate
- Intel:

```
-parallel
-par-threshold[n] - set loop count threshold
-par-report{0|1|2|3}
```

- IBM:

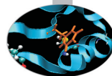
```
-qsmp                    la abilita automaticamente
-qsmp=openmp:noauto      per disabilitare la
                         parallelizzazione automatica
```

Outline



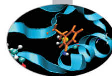
- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
- Compilatori e ottimizzazione
- **Librerie scientifiche**
- Floating Point Computing

Librerie statiche e dinamiche



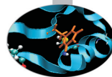
- Una libreria puó essere statica o dinamica
  - entrambe sono richieste in fase di compilazione aggiungendo **`-L<directory_libreria> -l<nome_libreria>`**
- Libreria statica:
  - estensione **`.a`**
  - tutti i simboli oggetti della libreria vengono inclusi nell'eseguibile al momento del linking
  - se si crea una libreria che si appoggia ad un'altra non include i suoi simboli: l'eseguibile dovrá linkare tutte le librerie in cascata
  - eseguibile piú efficiente
- Libreria dinamica:
  - estensione **`.so`**
  - richiede di specificare la directory in cui cercare la librerie in fase di esecuzione (per esempio settando la variabile di ambiente **`LD_LIBRARY_PATH`**)
  - **`ldd <nome_eseguibile>`** dice le librerie dinamiche richieste dall'eseguibile
  - eseguibile piú snello

Librerie scientifiche



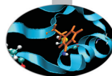
- Le librerie sono insiemi di funzioni che implementano una varietá di algoritmi, spesso numerici.
- Operazioni aritmetiche di basso livello (e.g. prodotto scalare o numeri casuali), ma anche algoritmi piú complicati (trasformata di Fourier o diagonalizzazione di matrici)
- Le performance delle migliori librerie sono difficilmente superabili da un utente ordinario
- A volte ottimizzate in assembler
- Libere o proprietarie
- Occorre fare attenzione ad utilizzare la migliore libreria possibile per una certa coppia compilatore-libreria

Vantaggi e svantaggi



- Vantaggi:
  - migliorano la modularitá
  - standardizzazione
  - portabilitá
  - efficienza
  - pronte all'uso
- Svantaggi:
  - dettagli nascosti
  - spesso non si sa cosa si usa
  - troppa fiducia nell'implementazione

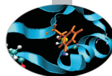
Panomarica



- Difficile avere una panoramica completa
  - tante tipologie
  - e uno scenario in continua evoluzione
  - anche per le nuove architetture in arrivo (GPU)

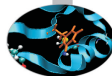
- Tipologie di uso comune
  - algebra lineare
  - fft
  - input-output
  - calcolo parallelo
  - mesh decomposition
  - suite

Algebra lineare



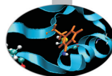
- Per applicazioni massive é cruciale il tipo di parallelizzazione
  - alcune sono giá fornite multi-threaded o anche parallele a memoria distribuita
- Memoria condivisa
  - BLAS
  - GOTOBLAS
  - LAPACK/CLAPACK/LAPACK++
  - ATLAS
  - PLASMA
  - SuiteSparse
- Memoria distribuita
  - Blacs (solo suddivisione)
  - ScaLAPACK
  - PSBLAS
  - Elemental

BLAS



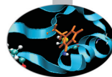
- BLAS: Basic Linear Algebra Subprograms
  - la Basic Linear Algebra Subprograms é tra le prime librerie scritte (1979), in origine per calcolatori con architettura vettoriale
  - comprende operazioni elementari tra vettori e matrici come il prodotto scalare e la moltiplicazione tra scalari, vettori, matrici, anche in forma trasposta
  - é utilizzata da numerose librerie di livello piú alto, perció ne sono state prodotte diverse versioni, ottimizzate per varie piattaforme di calcolo
- 3 livelli
  - BLAS liv. 1 subroutine Fortran per il calcolo di operazioni di base scalare-vettore. Sono la conclusione di un progetto terminato nel 1977
  - BLAS liv. 2 operazioni vettore-matrice. Scritte tra il 1984 e 1986
  - BLAS liv. 3 subroutine Fortran per operazioni matrice-matrice, disponibili dal 1988

BLAS / 2



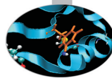
- Le subroutine BLAS si applicano a dati reali e complessi, in semplice o doppia precisione
- Operazioni scalare-vettore (O(n))
  - SWAP scambio vettori
  - COPY copia vettori
  - SCAL cambio fattore di scala
  - NRM2 norma L2
  - AXPY somma: Y + A*X
- Operazioni vettore-matrice ($O(n^2)$)
  - GEMV prodotto vettore/matrice generica
  - HEMV prodotto vettore/matrice hermitiana
  - SYMV prodotto vettore/matrice simmetrica

BLAS / 3



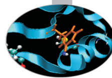
- Operazioni matrice-matrice ($O(n^3)$)
  - GEMM prodotto matrice/matrice generica
  - HEMM prodotto matrice/matrice hermitiana
  - SYMM prodotto matrice/matrice simmetrica
- GOTOBLAS
  - Kazushige Goto, un ricercatore del Texas Advanced Computing Center (University of Texas at Austin), ha ottimizzato manualmente in assembler le subroutine BLAS per diversi supercomputer

LAPACK & Co.



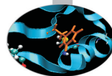
- LAPACK: Linear Algebra PACKage
  - evoluzione di LINPACK e EISPACK
  - soluzione di problemi di algebra lineare, tra cui sistemi di equazioni lineari, problemi di minimi quadrati, autovalori
- ATLAS: Automatically Tuned Linear Algebra Software
  - implementazione BLAS e di alcune routine di LAPACK efficiente grazie alla procedura di autotuning che avviene durante l'installazione
- PLASMA: Parallel Linear Algebra Software for Multi-core Architectures
  - soluzione di sistemi lineari, progettate per essere efficienti su processori multi-core, funzionalitá simili a LAPACK ma piú limitate
- SuiteSparse
  - collezione di pacchetti per matrici sparse

Ancora algebra lineare



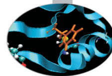
- Calcolo di autovalori/autovettori
  - EISPACK: calcolo di autovalori e autovettori, con versioni specializzate per matrici di diversi tipi, reali e complesse, hermitiane, simmetriche, tridiagonali
  - ARPACK: problemi agli autovalori di grandi dimensioni. La versione parallela é un'estensione della libreria classica e usa le librerie BLACS e MPI
- Algebra lineare a memoria distribuita
  - BLACS: linear algebra oriented message passing interface
  - ScaLAPACK: Scalable Linear Algebra PACKage
  - Elemental: framework per algebra lineare densa
  - PSBLAS: Parallel Sparse Basic Linear Algebra Subroutines
  - SLEPc: problemi agli autovalori

Input/Output



- Le librerie di I/O risultano particolarmente utili per
  - interoperabilitá: C/Fortran, Little Endian/Big Endian,...
  - visualizzazione
  - analisi di sub-set
  - metadati
  - I/O parallelo
- HDF5: “is a data model, library, and file format for storing and managing data”
- NetCDF: “NetCDF is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data”
- VTK: “open-source, freely available software system for 3D computer graphics, image processing and visualization”

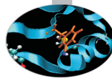
Librerie per calcolo parallelo e mesh



- MPI: Message Passing Interface
  - standard piú diffuso per la parallelizzazione in memoria distribuita
  - implementazioni piú importanti come librerie: MPICH e OpenMPI
- Decomposizione di mesh
  - METIS e ParMETIS: “can partition a graph, partition a finite element mesh, or reorder a sparse matrix”
  - Scotch e PT-Scotch: “sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hypergraph partitioning, and sequential and parallel sparse matrix block ordering”

SCAI SuperComputing Applications and Innovation

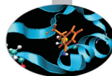
Suite per il calcolo scientifico



- Trilinos
  - collezione di algoritmi in stile object-oriented per la soluzione di problemi scientifici e ingegneristici multi-fisica
  - struttura a due livelli concepita attorno a collezione di “package” sviluppati da diversi esperti in materia (precondizionatori, solutori non lineari,...)
- PETSc
  - strutture dati e funzioni per la soluzione su calcolatori paralleli di equazioni alle derivate parziali
  - include solutori per equazioni lineari, non lineari e integratori ODE
  - utilizza MPI per realizzare il parallelismo e permette di sviluppare programmi che richiedono ingenti risorse computazionali

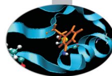
CINECA

Versioni specializzate



- MKL: Intel Math Kernel Library
  - Major functional categories include Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics. Cluster-based versions of LAPACK and FFT are also included to support MPI-based distributed memory computing.
- ACML: AMD Core Math Library
  - Libreria di funzioni altamente ottimizzate per processori AMD. Include tra l'altro BLAS, LAPACK, FFT, Random Generators
- GSL: GNU Scientific Library
  - The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.
- ESSL (IBM): Engineering and Scientific Subroutine library
  - BLAS, LAPACK, ScaLAPACK, solutori sparsi, FFT e altro. La versione parallela utilizza MPI.

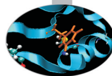
Chiamare le librerie



- Per utilizzare le librerie nei programmi é necessario innanzitutto che la sintassi di chiamata delle funzioni sia corretta
- Inoltre in fase di generazione dell'eseguibile é indispensabile fornire tutte le indicazioni necessarie per l'individuazione della versione corretta della libreria
- Spesso nel caso di librerie proprietarie esistono modalitá di compilazione e linking specifiche
- Puó non essere banale, e.g. Intel ScaLAPACK

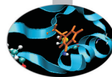
```
mpif77 <programma> -L$MKLROOT/lib/intel64 \
   -lmkl_scalapack_lp64  -lmkl_blacs_openmpi \
   -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core \
   -liomp5 -lpthread
```

Librerie: Interoperabilitá



- Molte librerie scientifiche sono scritte in C, molte in Fortran
- Chiamare le funzioni di libreria da un linguaggio diverso dall'originario puó dare qualche difficoltá
  - matching dei tipi: l'**`int`** del C non é garantito corrispondere all'**`integer`** del Fortran
  - matching dei simboli: Fortran e C++ deformano i nomi dei simboli sorgenti nel produrre gli oggetti
- Problema affrontato in modo un po' "rozzo" fino a non molto tempo fa
  - tentando di matchare i tipi e aggiungendo gli _ necessari per matchare con gli oggetti della libreria
  - il comando **`nm <file_oggetto>`** elenca i simboli ivi contenuti
  - alcune librerie fornivano a questo scopo wrapper giá pronti
- Problema affrontato in modo efficace nello standard Fortran 2003 (modulo **`iso_c_binding`**)
  - librerie piú importanti forniscono le interfacce Fortran 2003
- In C++ vedere il comando **`extern "C"`**

Librerie: Interoperabilitá / 2



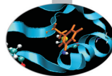
- Per chiamare librerie scritte in C dal Fortran o viceversa
- **`mpi`** scritta in C/C++:
  - vecchia modalitá : **`include "mpif.h"`**
  - nuova modalitá: **`use mpi`**
  - non sono del tutto equivalenti: usare il modulo vuol dire avere il check dei tipi a compile-time
- **`fftw`** scritta in C
  - legacy : **`include "fftw3.f"`**
  - modern:

```
use iso_c_binding
include 'fftw3.f03'
```

  - di nuovo: la versione moderna offre maggiori potenzialitá
- **`BLAS`** scritte in Fortran
  - legacy : chiamare **`dgemm_`**
  - modern: chiamare **`cblas_dgemm`**
- Purtroppo ancora poca standardizzazione
  - studiare il manuale e tentare di essere standard o almeno portabile

SCAI
SuperComputing Applications and Innovation

BLAS: Interoperabilitá / 1



- Argomenti delle funzioni sul sito “netlib”

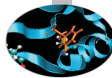
```
http://www.netlib.org/blas/
```

- Poiché le BLAS sono scritte in Fortran 77, anche dal Fortran alcuni compilatori rendono disponibili interfacce per chiamarle in sicurezza (check dei tipi) e con le features del Fortran 95 (assumed shape arrays e argomenti opzionali)
  - Con Intel e MKL

```
use mkl95_blas
```

CINECA

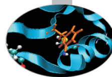
BLAS: Interoperabilitá / 2



- C (modalitá legacy):
  - aggiungere l'underscore ai nomi delle funzioni
  - poiché il Fortran passa tutti gli argomenti per reference, é necessario sempre passare i puntatori
  - assumere matching dei tipi (compiler dependent): probabilmente **`double`**, **`int`**, **`char`** per **`double precision`**, **`integer`**, **`character`**
  - ma l'ordinamento di array multidimensionali verrá trasposto!
- C (modalitá moderna)
  - usare le interafcce **`cblas`**: le GSL di GNU o le MKL di Intel le forniscono
  - in effetti le GSL includono anche l'implementazione delle BLAS (non solo wrapper)
  - includere l'header **`#include <gsl.h>`** o **`#include<mkl.h>`**
  - rispettare la sintassi (si puó specificare l'ordinamento delle matrici)

http://www.gnu.org/software/gsl/manual/html_node/GSL-CBLAS-Library.html

Hands-on: BLAS



- Sostituire il codice matrice-matrice di matrixmul con una chiamata a **`DGEMM`**, la routine BLAS che esegue il prodotto matrice-matrice in doppia precisione
- **`DGEMM`** esegue (l'operatore **`op`** consente trasposizioni)

```
C := alpha*op( A )*op( B ) + beta*C,
```

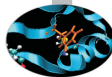
- Argomenti della **`DGEMM`** : **`http://www.netlib.org/blas/dgemm.f`**
- Fortran: GNU, BLAS efficienti sono le **`acml`**, nelle versioni:
  - **`gfortran64`** (seriali)
  - **`gfortran64_mp`** (multi-thread)

```
module load profile/advanced
module load gnu/4.7.2 acml/5.3.0--gnu--4.7.2
gfortran -O3 -L$ACML_HOME/gfortran64/lib/ -lacml matrixmulblas.F90
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACML_HOME/gfortran64/lib/
```

- Fortran: Intel, BLAS efficienti sono le proprietarie MKL
  - **`sequential`** (seriali)
  - **`parallel`** (multi-thread)

```
module load profile/advanced
module load intel/cs-xe-2013--binary
ifort -O3 -mkl=sequential matrixmulblas.F90
```

Hands-on: BLAS / 2



- C: compilatore Intel (MKL con cblas)
  - includere l’header file `#include<mkl.h>` nel sorgente
  - provare `-mkl=sequential` e `-mkl=parallel`

```
module load profile/advanced
module load intel/cs-xe-2013--binary
icc -O3 -mkl=sequential matrixmulblas.c
```

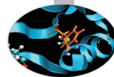
- C: compilatore GNU (GSL con cblas)
  - includere l’header file `#include <gsl/gsl_cblas.h>` nel sorgente

```
module load profile/advanced
module load gnu/4.7.2 gsl/1.15--gnu--4.7.2
gcc -O3 -L$GSL_HOME/lib -lgslcblas matrixmulblas.c -I$GSL_INCLUDE
```

  - Confrontare le performance con quelle ottenibili con `-o3`/`-fast`
- Provare anche le versioni multithreaded: osservazioni?
- Riportare i **GFlop** e considerare matrici 4096x4096 (risultati piú stabili)

GNU -O3	Intel -fast	GNU-ACML/GSL seq	Intel-MKL seq
—	Intel -fast -parallel	GNU-ACML par	Intel-MKL par
—			

Si puó andare oltre?



- GPU...

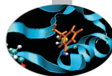
blas + cublas MPI
blas + cublas
cublas 3.2 (magmablas 0.3)
cublas 3.1
shared memory hand-made GPU
naive hand-made GPU (cache a 48Kb)
PGI accelerator
mkl, 12 thread
unrolling+padding, ifort -fast
naive 12 thread, ifort -fast
mkl
unrolling+padding, ifort -fast
naive, ifort -fast

0 50 100 150 200 250 300 350 400 450

SCAI

SuperComputing Applications and Innovation

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

Profilers

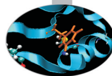
Compilatori e ottimizzazione

Librerie scientifiche

Floating Point Computing

CINECA

Why talking about data formats?

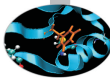


- The “numbers” used in computers are different from the “usual” numbers

- Some differences have known consequences
  - size limits
  - numerical stability
  - algorithm robustness

- Other differences are often misunderstood
  - portability
  - exceptions
  - surprising behaviours with arithmetic

Bits and Integers



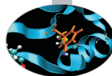
- Computers usually handle bits
- An integer number n may be stored as a sequence of bits
- Of course, you have a range

$$-2^{r-1} \leq n \leq 2^{r-1} - 1$$

- Two common sizes
  - 32 bit: range $-2^{31} \leq n \leq 2^{31} - 1$
  - 64 bit: range $-2^{63} \leq n \leq 2^{63} - 1$
- Languages allow for declaring different flavours of integers
  - select the type you need compromizing on avoiding overflow and saving memory
- Is it difficult to have an integer overflow?
  - consider a cartesian discretization mesh $(1536 \times 1536 \times 1536)$ and a linearized index i

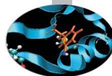
$$0 \leq i \leq 3623878656 > 2^{31} = 2147483648$$

Bits and Integers / 2



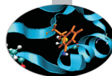
- Fortran “officially” does not let you specify the size of declared data
  - you request **`kind`** and the language do it for you
  - in principle very good, but interoperability must be considered with attention
  - and the underlying types are usually just a few of “well known” types
- C standard types do not match exact sizes, too
  - look for **`int`**, **`long int`**, **`unsigned int`**, ...
  - **`char`** is an 8 bit integer
  - unsigned integers available, doubling the maximum value $0 \leq n \leq 2^r - 1$

Bits and Reals



- Note: From now on, some examples will consider base 10 numbers just for readability
- Representing reals using bits is not natural
- Fixed size approach
  - select a fixed point corresponding to comma
  - e.g., with 8 digits and 5 decimal places 36126234 gets interpreted as 361.26234
- Cons:
  - limited range: from 0.00001 to 999.99999, spanning 10^8
  - only numbers having at most 5 decimal places can be exactly represented
- Pros:
  - constant resolution, i.e. the distance from one point to the closest one (0.00001)

Floating point approach

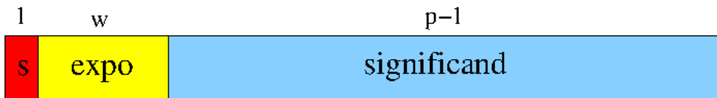


- Consider scientific notation

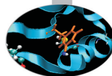
$$n = (-1)^s \cdot m \cdot \beta^e$$

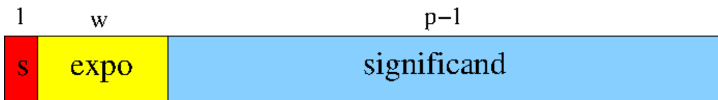
$$0.0046367 = (-1)^0 \cdot 4.6367 \cdot 10^{-3}$$

- Represent it using bits reserving
  - one digit for sign s
  - “p-1” digits for significand (mantissa) m
  - “w” digits for exponent e



Exponent and Significand

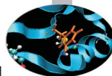


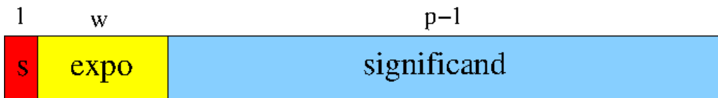


- Exponent
  - unsigned biased exponent
  - $e_{min} \leq e \leq e_{max}$
  - e_{min} must be equal to $(1 - e_{max})$
- Mantissa
  - precision p, the digits x_i are $0 \leq x_i < \beta$
  - “hidden bit” format used for normal values: 1.xx...x

IEEE Name	Format	Storage Size	w	p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

Mantissa



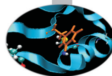


- Cons:
  - only "some" real numbers are floating point numbers (see later)
- Pros:
  - constant relative resolution (relative precision), each number is represented with the same *relative error* which is the distance from one point to the closest one divided by the number (see later)
  - wide range: "normal" positive numbers from $10^{e_{min}}$ to $9,999..9 \cdot 10^{e_{max}}$
- The representation is unique assuming the mantissa is

$$1 \leq m < \beta$$

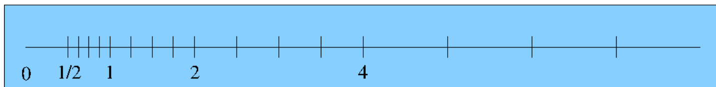
i.e. using "normal" floating-point numbers

Resolution

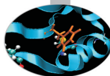


- The distance among “normal” numbers is not constant
- E.g., $\beta = 2$, $p = 3$, $e_{min} = -1$ and $e_{max} = 2$:
  - 16 positive “normalized” floating-point numbers

```
e = -1 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/8:5/8:6/8:7/8]
e =  0 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/4:5/4:6/4:7/4]
e = +1 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/2:5/2:6/2:7/2]
e = +2 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/1:5/1:6/1:7/1]
```



Relative Resolution



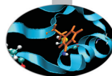
- What does it mean “constant relative resolution”?
- Given a number $N = m \cdot \beta^e$ the nearest number has distance

$$R = \beta^{-(p-1)}\beta^e$$

  - E.g., given $3.536 \cdot 10^{-6}$, the nearest (larger) number is $3.537 \cdot 10^{-6}$ having distance $0.001 \cdot 10^{-6}$
- The relative resolution is (nearly) constant (considering $m \simeq \beta/2$)

$$\frac{R}{N} = \frac{\beta^{-(p-1)}}{m} \simeq 1/2\beta^{-p}$$

Intrinsic Error

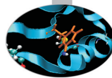


- Not any real number can be expressed as a floating point number
  - because you would need a larger exponent
  - or because you would need a larger precision
- The resolution is directly related to the intrinsic error
  - if $p = 4$, 3.472 may approximate numbers between 3.4715 and 3.4725, its intrinsic error is 0.0005
  - the instrinsic error is (less than) $(\beta/2)\beta^{-p}\beta^{e}$
  - the relative intrinsic error is

$$\frac{(\beta/2)\beta^{-p}}{m} \leq (\beta/2)\beta^{-p} = \varepsilon$$

  - The intrinsic error ε is also called “machine epsilon” or “relative precision”

Measuring error



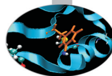
- When performing calculations, floating-point error may propagate and exceed the intrinsic error

```
real value              = 3.14145
correctly rounded value = 3.14
current value           = 3.17
```

- The most natural way to measure rounding error is in “ulps”, i.e. units in the last place
  - e.g., the error is 3 ulps
- Another interesting possibility is using “machine epsilon”, which is the relative error corresponding to 0.5 ulps

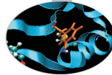
```
relative error   = 3.17-3.14145 = 0.02855
machine epsilon  = 10/2*0.001   = 0.005
relative error   = 5.71 ε
```

Handling errors



- Featuring a constant relative precision is very useful when dealing with rescaled equations
- Beware:
  - 0.2 has just one decimal digit using radix 10, but is periodic using radix 2
  - periodicity arises when the fractional part has prime factors not belonging to the radix
  - by the way, in Fortran if **`a`** is double precision, **`a=0.2`** is badly approximated (use **`a=0.2d0`** instead)
- Beware overflow!
  - you think it will not happen with your code but it may happen
  - exponent range is symmetric: if possibile, perform calculations around 1 is a good idea

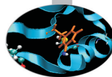
Types features



IEEE Name	min	max	ε	C	Fortran
Binary32	1.2E-38	3.4E38	5.96E-8	float	real
Binary64	2.2E-308	1.8E308	1.11E-16	double	real(kind(1.d0))
Binary128	3.4E-4932	1.2E4932	9.63E-35	long double	real(kind=...)

- There are also “double extended” type and parametrized types
- Extended and quadruple precision devised to limit the round-off during the double calculation of trascendental functions and increase overflow
- Extended and quad support depends on architecture and compiler: often emulated and, hence, slow!
- Decimal with 32, 64 and 128 bits are defined by standards, too
- FPU are usually “conformant” but not “compliant”
- To be safe when converting binary to text specify 9 decimals for single precision and 17 decimal for double

Error propagation



- Assume $p = 3$ and you have to compute the difference $1.01 \cdot 10^1 - 9.93 \cdot 10^0$
- To perform the subtraction, usually a shift of the smallest number is performed to have the same exponent
- First idea: compute the difference exactly and then round it to the nearest floating-point number

$$x = 1.01 \cdot 10^1 \quad ; \quad y = 0.993 \cdot 10^1$$

$$x - y = 0.017 \cdot 10^1 = 1.70 \cdot 10^{-2}$$

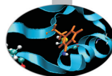
- Second idea: compute the difference with $\mathbf{p}$ digits

$$x = 1.01 \cdot 10^1 \quad ; \quad y = 0.99 \cdot 10^1$$

$$x - y = 0.02 \cdot 10^1 = 2,00 \cdot 10^{-2}$$

the error is 30 ulps!

Guard digit



- A possibile solution: use the guard digit (p+1 digits)

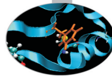
$$x = 1.010 \cdot 10^1$$

$$y = 0.993 \cdot 10^1$$

$$x - y = 0.017 \cdot 10^1 = 1.70 \cdot 10^{-2}$$

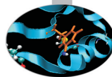
- Theorem: if x and y are floating-point numbers in a format with parameters and p, and if subtraction is done with p + 1 digits (i.e. one guard digit), then the relative rounding error in the result is less than 2 ε.

Cancellation



- When subtracting nearby quantities, the most significant digits in the operands match and cancel each other
- There are two kinds of cancellation: catastrophic and benign
  - benign cancellation occurs when subtracting exactly known quantities: according to the previous theorem, if the guard digit is used, a very small error results
  - catastrophic cancellation occurs when the operands are subject to rounding errors
- For example, consider $b = 3.34$, $a = 1.22$, and $c = 2.28$.
  - the exact value of $b^2 - 4ac$ is 0.0292
  - but b^2 rounds to 11.2 and $4ac$ rounds to 11.1, hence the final answer is 0.1 which is an error by $70 ulps$
  - the subtraction did not introduce any error, but rather exposed the error introduced in the earlier multiplications.

Cancellation / 2

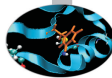


- The expression $x^2 - y^2$ is more accurate when rewritten as $(x - y)(x + y)$ because a catastrophic cancellation is replaced with a benign one
  - replacing a catastrophic cancellation by a benign one may be not worthwhile if the expense is large, because the input is often an approximation
- Eliminating a cancellation entirely may be worthwhile even if the data are not exact
- Consider second-degree equations

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

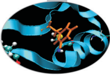
  - if $b^2 >> ac$ then $b^2 - 4ac$ does not involve a cancellation
  - but, if $b > 0$ the addition in the formula will have a catastrophic cancellation.
  - to avoid this, multiply the numerator and denominator of x_1 by $-b - \sqrt{b^2 - 4ac}$ to obtain $x_1 = (2c)/(-b - \sqrt{b^2 - 4ac})$ where no catastrophic cancellation occurs

Rounding and IEEE standards



- The IEEE standards requires correct rounding for:
  - addition, subtraction, mutiplication, division, remainder, square root
  - conversions to/from integer
- The IEEE standards recommends correct rounding for:
  - e^x, $e^x - 1$, 2^x, $2^x - 1$, $\log_\alpha(\phi)$, $1/\sqrt(x)$, $sin(x)$, $cos(x)$, $tan(x)$,....
- Remember: “No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits” (W. Kahan)

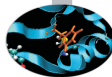
Special values



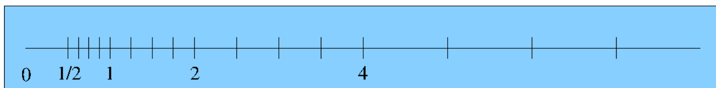
- Zero: signed
- Infinity: signed
  - overflow, divide by 0
  - Inf-Inf, Inf/Inf, $0 \cdot$ Inf $\rightarrow$ NaN (indeterminate)
  - Inf op a $\rightarrow$ Inf if a is finite
  - a / Inf $\rightarrow$ 0 if a is finite
- NaN: not a number!
  - Quiet NaN or Signaling NaN
  - e.g. $\sqrt{a}$ with $a < 0$
  - NaN op a $\rightarrow$ NaN or exception
  - NaNs do not have a sign: they aren't a number
  - The sign bit is ignored
  - NanS can "carry" information

SuperComputing Applications and Innovation

Zero and Denormals

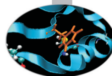


- Considering positve numbers, the smallest "normal" floating point number is $n_{smallest} = 1.0 \cdot \beta^{e_{min}}$
- In the previous example it is $1/2$



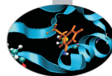
- At least we need to add the zero value
  - there are two zeros: $\mathbf{+0}$ and $\mathbf{-0}$
- When a computation result is less than the minimum value, it could be rounded to zero or to the minimum value

Zero and Denormals / 2



- Another possibility is to use denormal (also called subnormal) numbers
  - decreasing mantissa below 1 allows to decrease the floating point number, e.g. $0.99 \cdot \beta^{e_{min}}$, $0.98 \cdot \beta^{e_{min}}, \ldots, 0.01 \cdot \beta^{e_{min}}$
  - subnormals are linearly spaced and allow for the so called "gradual underflow"
- Pro: $k/(a-b)$ may be safe (depending on k) even is $a - b < 1.0 \cdot \beta^{e_{min}}$
- Con: performance of denormals are significantly reduced (dramatic if handled only by software)
- Some compilers allow for disabling denormals
  - Intel compiler has **`-ftz`**: denormal results are flushed to zero
  - automatically activated when using any level of optimization!

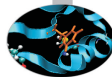
Walking Through



- Double precision: w=11 ; p=53

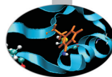
```
0x0000000000000000  +zero
0x0000000000000001  smallest subnormal
...
0x000fffffffffffff  largest subnormal
0x0010000000000000
...
0x001fffffffffffff  smallest normal
0x0020000000000000  2 X smallest normal
...
0x7fefffffffffffff  largest normal
0x7ff0000000000000  +infinity
```

Walking Through



```
0x7ff0000000000001  NaN
...
0x7fffffffffffffff  NaN
0x8000000000000000  -zero
0x8000000000000001  negative subnormal
...
0x800fffffffffffff  'largest' negative subnormal
0x8010000000000000  'smallest' negative normal
...
0xfff0000000000000  -infinity
0xfff0000000000001  NaN
...
0xffffffffffffffff  NaN
```

Error-Free Transformations



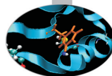
- An error-free transformation (EFT) is an algorithm which determines the rounding error associated with a floating-point operation
- E.g., addition/subtraction

$$a + b = (a \oplus b) + t$$

where $\oplus$ is a symbol for floating-point addition

- Under most conditions, the rounding error is itself a floating-point number
- **An EFT can be implemented using only floating-point computations in the working precision**

EFT for Addition



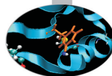
- FastTwoSum: compute $a + b = s + t$ where

$$|a| \geq |b|$$

$$s = a \oplus b$$

```
void FastTwoSum( const double a, const double b,
                 double* s, double* t ) {
            // No unsafe optimizations !
            *s = a + b;
            *t = b - ( *s - a );
            return;
      }
```

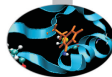
EFT for Addition / 2



- No requirements on *a* or *b*
- Beware: avoid compiler unsafe optimizations!

```
void TwoSum( const double a, const double b,
                 double* s, double* t ) {
            // No unsafe optimizations !
            *s = a + b;
            double z = *s - b;
            *t = (a-z)+(b-s-z));return;
```

Summation techniques

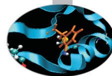


- Condition number

$$C_{sum} = \frac{|\sum a_i|}{\sum |a_i|}$$

- If C_{sum} is “ not too large”, the problem is not ill conditioned and traditional methods may suffice
- But if it is “too large”, we want results appropriate to higher precision without actually using a higher precision
- But if higher precision is available, consider to use it!
  - beware: quadruple precision is nowadays only emulated

Traditional summation

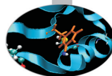


$$s = \sum_{i=0}^{n} x_i$$

```
double Sum( const double* x, const int n ) {
   int i;
   for ( i = 0; i < n; i++ ) {
      Sum += x[ i ];
   }
   return Sum;
}
```

- Traditional Summation: what can go wrong?
  - catastrophic cancellation
  - magnitude of operands nearly equal but signs differ
  - loss of significance
  - small terms encountered when running sum is large
  - the smaller terms don't affect the result
  - but later large magnitude terms may reduce the running sum

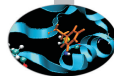
Kahan summation



- Based on FastTwoSum and TwoSum techniques
- Knowledge of the exact rounding error in a floating-point addition is used to correct the summation
- Compensated Summation

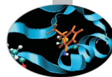
```
double Kahan( const double* a, const int n ) {
   double s = a[ 0 ];          //  sum
   double t = 0.0;             //  correction term
   for(int i=1; i<n ; i++) {
       double y = a[ i ] - t; //  next term "plus" correction
       double z = s + y;      //  add to accumulated sum
       t = ( z - s ) - y;     //  t <- -( low part of y )
       s = z;                 //  update sum
   }
   return s;
}
```

More



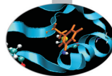
- Many variations known (Knutht, Priest,...)
- Sort the values and sum starting from smallest values (for positive numbers)
- Other techniques (distillation)
- Use a greater precision or emulate it (long accumulators)
- Similar problems for Dot Product, Polynomial evaluation,...

Exceptions (IEEE 754-2008)



- Underflow
  - Absolute value of a non zero result is less than the minimum value (i.e., it is subnormal or zero)
- Overflow
  - Magnitude of a result greater than the largest finite value
  - Result is $\pm\infty$
- Division by zero
  - a/b where a is finite and non zero and b=0
- Inexact
  - Result, after rounding, is not exact
- Invalid
  - an operand is sNaN, square root of negative number or combination of infinity

Handling exceptions



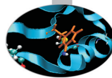
- Let us say you may produce a NaN
- What do you want to do in this case?

- First scenario: go on, there is no error and my algorithm is robust
- E.g., the function **maxfunc** compute the maximum value of a scalar function $f(x)$ testing each function value corresponding to the grid points **g(i)**

```
        call maxfunc(f,g)
```

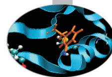
  - to be safe I should pass the domain of f but the it could be difficult to do
  - I may prefer to check each grid point **g(i)**
  - if the function is not defined somewhere, I will get a NaN (or other exception) but I do not care: the maximum value will be correct

Handling exceptions / 2



- Second scenario: ops, something went wrong during the computation...
- (Bad) solution: complete your run and check the results and, if you see NaN, throw it away
- (First) solution: trap exceptions using compiler options (usually systems ignore exception as default)
- Some compilers allow to enable or disable floating point exceptions
  - Intel compiler: **`-fpe0`**: Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted.
  - GNU compiler: **`-ffpe-trap=zero,overflow,invalid,underflow`**
- very useful, but the performance loss may be material!
- use only in debugging, not in production stage

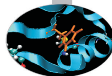
Handling exceptions / 3



- (Second) solution: check selectively
  - each N_{check} time-steps
  - the most dangerous code sections
- Using language features to check exceptions or directly special values (NaNs,...)
  - the old print!
  - Fortran (2003): from module **ieee_arithmetic**, **ieee_is_nan(x)**, **ieee_is_finite(x)**
  - C: from **<math.h>**, **isnan** or **isfinite**, from C99 look for **fenv.h**
  - do not use old style checks (compiler may remove them):

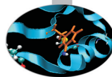
```
int IsFiniteNumber(double x) {
    return (x <= DBL_MAX && x >= -DBL_MAX);
}
```

Floating-point control



- Why doesn't my application always give the same answer?
  - inherent floating-point uncertainty
  - we may need reproducibility (porting, optimizing,...)
  - accuracy, reproducibility and performance usually conflict!
- Compiler safe mode: transformations that could affect the result are prohibited, e.g.
  - $x/x = 1.0$, false if $x = 0.0, \infty, NaN$
  - $x - y = -(y - x)$ false if $x = y$, zero is signed!
  - $x - x = 0.0$ ...
  - $x * 0.0 = 0.0$ ...

Floating-point control / 2

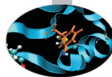


- An important case: reassociation is not safe with floating-point numbers
  - $(x+y)+z = x+(y+z)$: reassociation is not safe
  - compare

$$-1.0+1.0e{-13}+1.0 \;=\; 1.0-1.0+1.0e{-13} \;=\; 1.0e{-13}+1.0-1.0$$

  - $a*b/c$ may give overflow while $a*(b/c)$ does not
- Best practice:
  - select the best expression form
  - promote operands to the higher precision (operands, not results)

Floating-point control / 3



- Compilers allow to choose the safety of floating point semantics
- GNU options (high-level):

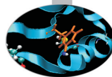
```
-f[no-]fast-math
```

  - It is off by default (different from icc)
  - Also sets abrupt/gradual underflow (FTZ)
  - Components control similar features, e.g. value safety (`-funsafe-math-optimizations`)

- For more detail

```
http://gcc.gnu.org/wiki/FloatingPointMath
```

Floating-point control / 4



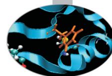
- Intel options:

```
-fp-model <type>
```

  - fast=1: allows value-unsafe optimizations (**default**)
  - fast=2: allows additional approximations
  - precise: value-safe optimizations only
  - strict: precise + except + disable fma
- Also pragmas in C99 standard

```
#pragma STDC FENV_ACCESS etc
```

Endianness

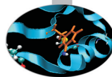


- Which is the ordering of bytes in memory? E.g.,

```
-1267006353 ===> 10110100011110110000010001101111
```

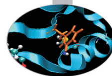
  - Big endian: `10110100 01111011 00000100 01101111`
  - Little endian: `01101111 00000100 01111011 10110100`
  - Other exotic layouts (VAX,...) nowadays unusual
  - Limits portability
- Possibile solutions
  - conversion binary to text and text to binary
  - compiler extensions(Fortran):
    - HP Alpha, Intel: -convert big_endian | little_endian
    - PGI: -byteswapio
    - Intel, NEC: F_UFMTENDIAN (variabile di ambiente)
  - explicit reoredering
  - conversion libraries

C and Fortran data portability



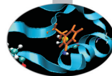
- For C Standard Library a file is written as a stream of byte
- In Fortran file is a sequence of records:
  - each read/write refer to a record
  - there is record marker before and after a record (32 or 64 bit depending on file system)
  - remember also the different array layout from C and Fortran
- Possible portability solutions:
  - read Fortran records from C
  - perform the whole I/O in the same language (usually C)
  - use Fortran 2003 `access='stream'`
  - use I/O libraries

How much precision do I need?



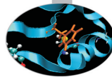
- Single, Double or Quad?
  - maybe single is too much!
  - computations get (much) slower when increasing precision, storage increases and power supply too
- Famous story
  - Patriot missile incident (2/25/91) . Failed to stop a scud missile from hitting a barracks, killing 28
  - System counted time in 1/10 sec increments which doesn't have an exact binary representation. Over time, error accumulates.
  - The incident occurred after 100 hours of operation at which point the accumulated errors in time variable resulted in a 600+ meter tracking error.
- **Wider floating point formats turn compute bound problems into memory bound problems!**

How much precision do I need? / 2



- Programmers should conduct mathematically rigorous analysis of their floating point intensive applications to validate their correctness
- Training of modern programmers often ignores numerical analysis
- Useful tricks
  - Repeat the computation with arithmetic of increasing precision, increasing it until a desired number of digits in the results agree
  - Repeat the computation in arithmetic of the same precision but rounded differently, say Down then Up and perhaps Towards Zero, then compare results
  - Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results vary

Interval arithmetic

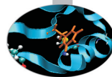


- A “correct” approach
- Interval number: possible values within a closed set

$$\mathbf{x} \equiv [x_L, x_R] := \{x \in \mathbb{R} | x_L \leq x \leq x_R\}$$

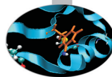
  - e.g., 1/3=0.33333 ; 1/3 $\in$ [0.3333,0.3334]
- Operations
  - Addition x + y = [a, b] + [c, d] = [a + c, b + d]
  - Subtraction x + y = [a, b] + [c, d] = [a -d, b -c]
  - ...
- Properties are interesting and can be applied to equations
- Interval Arithmetic has been tried for decades, but often produces bounds too loose to be useful
- A possible future
  - chips supporting variable precision and uncertainty tracking
  - runs software at low precision, tracks accuracy and reruns computations automatically if the error grows too large.

References



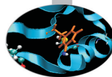
- N.J. Higham, Accuracy and Stability of Numerical Algorithms 2nd ed., SIAM, capitoli 1 e 2
- D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM C.S., vol. 23, 1, March 1991 http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- W. Kahan http://www.cs.berkeley.edu/ wkahan/
- Standards: http://grouper.ieee.org/groups/754/

Outline



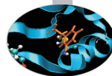
- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
- Compilatori e ottimizzazione
- Librerie scientifiche
- Floating Point Computing

HPC development tools



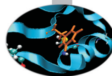
- What do I need to develop my HPC application? At least:
  - A compiler, e.g. GNU, Intel, PGI, PathScale, Sun
  - Code editor
- Several tools may help you (even for non HPC applications)
  - Debugger, e.g. gdb, TotalView, DDD
  - Profiler, e.g. gprof, Scalasca, Tau, Vampir
  - Project management, e.g. make, projects
  - Revision control, e.g. svn, git, cvs, mercurial
  - Generating documentation, e.g. doxygen
  - Source code repository, e.g. sourceforge, github, google.code
  - Data repository, currently under significant increase
  - and more . . .

Code editor



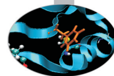
- You can select the code editor among a very wide range
  - from the light and fast text editors, e.g. the wonderful VIM, emacs
  - to the more sophisticated Integrated development environment (IDE), e.g. Ecplise
  - or you have intermediate options, e.g. Geany
- The choice obviously depends on the complexity and on the software tasks
- ...but also on your personal taste

Project management



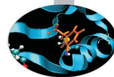
- Non trivial programs are hosted in several source files and link libraries
- Different types of files require different compilation
  - different optimization flags
  - different languages may be mixed, too
  - compilation and linking require different flags
  - and the code could work on different platforms
- During development (and debugging) several recompilations are needed, and we do not want to recompile all the source files but only the modified ones
- How to deal with it?
  - use the IDE (with plug-ins) and their project files to manage the content (e.g. Eclipse)
  - use language-specific compiler features
  - use external utilities, e.g. Make!

GNU Make



- "Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files"
- Make gets its knowledge from a file called the makefile, which lists each of the non-source files and how to compute it from other files
- When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program
- GNU Make has some powerful features for use in makefiles, beyond what other Make versions have

Preparing and Running Make

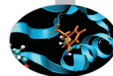


- To prepare to use make, you have to write a file that describes:
  - the relationships among files in your program
  - commands for updating each file
- Typically, the executable file is updated from object files, which are in turn made by compiling source files
- Once a suitable makefile exists, each time you change some source files, the shell command

```
make -f <makefile_name>
```

  suffices to perform all necessary recompilations
- If **-f** option is missing, the default names **makefile** or **Makefile** are used

Rules

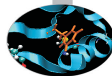


- A simple makefile consists of “rules”:

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

  - a **target** is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as clean
  - a **prerequisite** is a file that is used as input to create the target. A target often depends on several files.
  - a **recipe** is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Recipe commands must be preceded by a tab character.
- By default, make starts with the first target (default goal)

My first rule

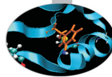


- A simple rule:

```
foo.o : foo.c defs.h
        gcc -c -g foo.c
```

- This rule says two things
  - how to decide whether foo.o is out of date: it is out of date if it does not exist, or if either foo.c or defs.h is more recent than it
  - how to update the file foo.o: by running gcc as stated. The recipe does not explicitly mention defs.h, but we presume that foo.c includes it, and that that is why defs.h was added to the prerequisites.
- Remember the tab character before starting the recipe lines!

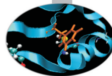
A simple example in C



- The main program is in laplace2d.c file
  - includes two header files: timing.h and size.h
  - calls functions in two source files: update_A.c and copy_A.c
- update_A.c and copy_A.c includes two header files: laplace2d.h and size.h
- A possible (naive) Makefile

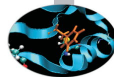
```
laplace2d_exe: laplace2d.o update_A.o copy_A.o
        gcc -o laplace2d_exe laplace2d.o update_A.o copy_A.o

laplace2d.o: laplace2d.c timing.h size.h
        gcc -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        gcc -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        gcc -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

How it works



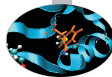
- The default goal is (re-)linking laplace2d_exe
- Before make can fully process this rule, it must process the rules for the files that edit depends on, which in this case are the object files
- The object files, according to their own rules, are recompiled if the source files, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist
- Note: in this makefile .c and .h are not the targets of any rules, but this could happen it they are automatically generated
- After recompiling whichever object files need it, make decides whether to relink edit according to the same “updating” rules.
- Try to follow the path: what happens if, e.g., laplace2d.h is modified?

A simple example in Fortran



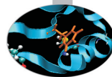
- The main program is in laplace2d.f90 file
  - uses two modules named prec and timing
  - calls subroutines in two source files: update_A.f90 and copy_A.f90
- update_A.f90 and copy_A.f90 use only prec module
- sources of prec and timing modules are in the prec.f90 and timing.f90 files
- Beware of the Fortran modules:
  - program units using modules require the mod files to exist
  - a target may be a list of files: e.g., both timing.o and timing.mod depend on timing.f90 and are produced compiling timing.f90
- Remember: the order of rules is not significant, except for determining the default goal

A simple example in Fortran / 2



```
laplace2d_exe: laplace2d.o update_A.o copy_A.o prec.o timing.o
	gfortran -o laplace2d_exe prec.o timing.o laplace2d.o update_A.o copy_A.o

prec.o prec.mod: prec.f90
	gfortran -c prec.f90

timing.o timing.mod: timing.f90
	gfortran -c timing.f90

laplace2d.o: laplace2d.f90 prec.mod timing.mod
	gfortran -c laplace2d.f90

update_A.o: update_A.f90 prec.mod
	gfortran -c update_A.f90

copy_A.o: copy_A.f90 prec.mod
	gfortran -c copy_A.f90

.PHONY: clean
clean:
	rm -f laplace2d_exe *.o *.mod
```

Phony Targets and clean



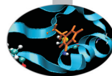
- A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.
  - avoid target name clash
  - improve performance
- **`clean`**: an ubiquitous target

```
.PHONY: clean
clean:
        rm *.o temp
```

- Another common solution: since **`FORCE`** has no prerequisite, recipe and no corresponding file, make imagines this target to have been updated whenever its rule is run

```
clean: FORCE
        rm *.o temp
FORCE:
```

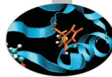
Variables



- The previous makefiles have several duplications
  - error-prone and not expressive
- Use variables!
  - define
    `objects = laplace2d.o update_A.o copy_A.o`
  - and use as **`$(objects)`**

```
objects := laplace2d.o update_A.o copy_A.o

laplace2d_exe: $(objects)
        gcc -o laplace2d_exe $(objects)

laplace2d.o: laplace2d.c timing.h size.h
        gcc -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        gcc -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        gcc -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

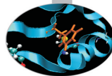
More Variables



- Use more variables to enhance readability and generality
- Modifying the first four lines it is easy to modify compilers and flags

```
CC       := gcc
CFLAGS   := -O2
CPPFLAGS :=
LDFLAGS  :=

objects  := laplace2d.o update_A.o copy_A.o

laplace2d_exe: $(objects)
	$(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_exe $(objects) $(LDFLAGS)

laplace2d.o: laplace2d.c timing.h size.h
	$(CC) $(CFLAGS) $(CPPFLAGS) -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
	$(CC) $(CFLAGS) $(CPPFLAGS) -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
	$(CC) $(CFLAGS) $(CPPFLAGS) -c copy_A.c

.PHONY: clean
clean:
	rm -f laplace2d_exe *.o
```

Implicit rules



- There are still duplications: each compilation needs the same command except for the file name
  - immagine what happens with hundred of files!
- What happens if Make does not find a rule to produce one or more prerequisite (e.g., and object file)?
- Make searches for an implicit rule, defining default recipes depending on the processed type
  - C programs: n.o is made automatically from n.c with a recipe of the form

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c
```

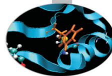
  - C++ programs: n.o is made automatically from n.cc, n.cpp or n.C with a recipe of the form

```
$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c
```

  - Fortran programs: n.o is made automatically from n.f, n.F (`$(CPPFLAGS)` only for .F)

```
$(FC) $(FFLAGS) $(CPPFLAGS) -c
```

Pattern rules



- Implicit rules allow for saving many recipe lines
  - but what happens is not clear reading the Makefile
  - and you are forced to use a predefined structure and variables
  - to clarify the types to be processed, you may define **`.SUFFIXES`** variable at the beginning of Makefile

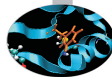
```
.SUFFIXES:
.SUFFIXES: .o .f
```

- You may use re-define an implicit rule by writing a pattern rule
  - a pattern rule looks like an ordinary rule, except that its target contains one character `%`
  - usually written as first target, does not become the default target

```
%.o : %.c
        $(CC) -c $(OPT_FLAGS) $(DEB_FLAGS) $(CPP_FLAGS) $< -o $@
```

- Automatic variables are usually exploited
  - `$@` is the target
  - `$<` is the first prerequisite (usually the source code)
  - `$^` is the list of prerequisites (useful in linking stage)

Running make targets

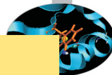


- It is possible to select a specific target to be updated, instead of the default goal (remember **clean**)

```
make copy_A.o
```

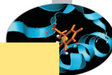
  - of course, it will update the chain of its prerequisite
  - useful during development when the full target has not been programmed, yet
- And it is possible to set target-specific variables as (repeated) target prerequisites
- Consider you want to write a Makefile considering both GNU and Intel compilers
- Use a default goal which is a help to inform that compiler must be specified as target

C example



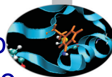
```
CPPFLAGS  :=
LDFLAGS   :=

objects   := laplace2d.o update_A.o copy_A.o

.SUFFIXES :=
.SUFFIXES := .c .o

%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<

help:
        @echo "Please select gnu or intel compilers as targets"

gnu: CC       := gcc
gnu: CFLAGS   := -O3
gnu: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_gnu $(objects) $(LDFLAGS)

intel: CC     := icc
intel: CFLAGS := -fast
intel: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_intel $(objects) $(LDFLAGS)

laplace2d.o: laplace2d.c timing.h size.h
update_A.o : update_A.c laplace2d.h size.h
copy_A.o   : copy_A.c laplace2d.h size.h

.PHONY: clean
clean:
        rm -f laplace2d_gnu laplace2d_intel $(objects)
```

Fortran example



```
LDFLAGS :=
objects := prec.o timing.o laplace2d.o update_A.o copy_A.o
.SUFFIXES:
.SUFFIXES: .f90 .o .mod

%.o: %.f90
	$(FC) $(FFLAGS) -c $<
%.o %.mod: %.f90
	$(FC) $(FFLAGS) -c $<

help:
	@echo "Please select gnu or intel compilers as targets"
gnu: FC       := gfortran
gnu: FCFLAGS  := -O3
gnu: $(objects)
	$(FC) $(FCFLAGS) -o laplace2d_gnu $^ $(LDFLAGS)
intel: FC      := ifort
intel: FCFLAGS := -fast
intel: $(objects)
	$(FC) $(CFLAGS) -o laplace2d_intel $^ $(LDFLAGS)

prec.o prec.mod:     prec.f90
timing.o timing.mod: timing.f90
laplace2d.o:         laplace2d.f90 prec.mod timing.mod
update_A.o:          update_A.f90 prec.mod
copy_A.o:            copy_A.f90 prec.mod

.PHONY: clean
clean:
	rm -f laplace2d_gnu laplace2d_intel $(objects) *.mod
```

Defining variables



- Another way to support different compilers or platforms is to include a platform specific file (e.g., make.inc) containing the needed definition of variables

```
include make.inc
```

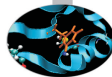
- Common applications feature many make.inc.<platform_name> which you have to select and copy to make.inc before compiling
- When invoking **make**, it is also possible to set a variable

```
make OPTFLAGS=-O3
```

  - this value will override the value inside the Makefile
  - unless **override** directive is used
  - but override is useful when you want to add options to the user defined options, e.g.

```
override CFLAGS += -g
```

Variable Flavours



- The variables considered until now are called *simply expanded* variables, are assigned using := and work like variables in most programming languages.
- The other flavour of variables is called *recursively expanded*, and is assigned by the simple =
  - recursive expansion allows to make the next assignments working as expected

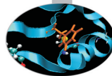
```
CFLAGS       = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

  - but may lead to unpredictable substitutions or even impossibile circular dependencies

```
CFLAGS       = $(CFLAGS) -g
```

- You may use **+=** to add more text to the value of a variable
  - acts just like normal **=** if the variable in still undefined
  - otherwise, exactly what **+=** does depends on what flavor of variable you defined originally
- Use recursive variables only if needed

Wildcards



- A single file name can specify many files using wildcard characters: `*`, `?` and `[...]`
- Wildcard expansion depends on the context
  - performed by make automatically in targets and in prerequisites
  - in recipes, the shell is responsible for
  - what happens typing **`make print`** in the example below? (The automatic variable **`$?`** stands for files that have changed)

```
print: *.c
       lpr -p $?
       touch print
```

  - if you define

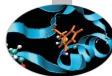
```
objects = *.o
foo : $(objects)
       gcc -o foo $(objects)
```

    it is expanded only when is used and it is not expanded if no .o file exists: in that case, foo depends on a oddly-named `.o` file
  - use instead the `wildcard` function:

```
objects := $(wildcard *.o)
```

SCAI SuperComputing Applications and Innovation

Conditional parts of Makefile



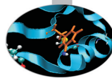
- Environment variables are automatically transformed into make variables
- Variables could be not enough to generalize rules
  - e.g., you may need non-trivial variable dependencies
- Immagine your application has to be compiled using GNU on your local machine mac_loc, and Intel on the cluster mac_clus
- You can catch the hostname from shell and use a conditional statement (**$SHELL** is not exported)

```
SHELL := /bin/sh
HOST  := $(shell hostname)
ifeq ($(HOST),mac_loc)
   CC     := gcc
   CFLAGS := -O3
endif
ifeq ($(HOST),mac_clus)
   CC     := icc
   CFLAGS := -fast
endif
```

- Be careful on Windows systems!

CINECA

Directories for Prerequisites



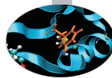
- For large systems, it is often desirable to put sources and headers in separate directories from the binaries
- Using Make, you do not need to change the individual prerequisites, just the search paths
- A **vpath** pattern is a string containing a `%` character.
  - **%.h** matches files that end in **.h**

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

will look for a file ending in `.c` in foo, then blish, then bar

- using **vpath** without specifying directory clears all search paths associated with patterns

Directories for Prerequisites / 2



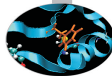
- When using directory searching, recipe generalizing is mandatory

```
vpath %.c src
vpath %.h ../headers
foo.o : foo.c defs.h hack.h
        gcc -c $< -o $@
```

- Again, automatic variables solve the problem
- And implicit or pattern rules may be used, too
- Directory search also works for linking libraries using prerequisites of the form `-lname`
- `make` will search for the file libname.so and, if not found, for libname.a first searching in vpath and then in system directory

```
foo : foo.c -lcurses
      gcc $^ -o $@
```

Advanced topics



- Functions, also user-defined
  - e.g., define objects as the list of file which will be produced from all .c files in the directory

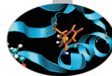
```
objects := $(patsubst %.c,%.o,$(wildcard *.c))
```

  - e.g., sorts the words of list in lexical order, removing duplicate words

```
headers := $(sort math.h stdio.h timer.h math.h)
```

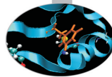
- Recursive make, i.e. make calling chains of makes
  - MAKELEVEL variable keeps the level of invocation

Standard Targets (good practice)



- all → Compile the entire program. This should be the default target
- install → Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use.
- uninstall → Delete all the installed files
- clean → Delete all files in the current directory that are normally created by building the program.
- distclean → Delete all files in the current directory (or created by this makefile) that are created by configuring or building the program.
- check → Perform self-tests (if any).
- install-html/install-dvi/install-pdf/install-ps → Install documentation
- html/dvi/pdf/ps → Create documentation

Much more...



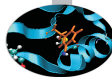
- Compiling a large application may require several hours
- Running make in parallel can be very helpful, e.g. to use 8 processes

```
make -j8
```

  - but not completely safe (e.g., recursive make compilation)
- There is much more you could know about **`make`**
  - this should be enough for your in-house application
  - but probably not enough for understanding large projects you could encounter

```
http://www.gnu.org/software/make/manual/make.html
```

Outline



Introduzione

Architetture

La cache e il sistema di memoria

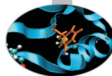
Pipeline

Profilers

Compilatori e ottimizzazione

Librerie scientifiche

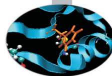
Floating Point Computing



As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.!

Maurice Wilkes discovers debugging, 1949.

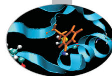
Testing-Debugging



- TESTING: finds errors.
- DEBUGGING: localizes and repairs them.

TESTING DEBUGGING CYCLE:
we test, then debug, then repeat.

Testing-Debugging

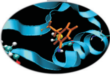


- TESTING: finds errors.
- DEBUGGING: localizes and repairs them.

TESTING DEBUGGING CYCLE:
we test, then debug, then repeat.

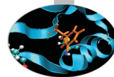
Program testing can be used to show the presence of bugs, but never to show their absence!
Edsger Dijkstra

What is a bug?



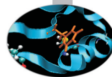
- Defect: An incorrect program code

What is a bug?



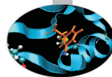
- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.

What is a bug?



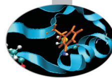
- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state

What is a bug?



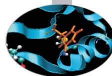
- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state $\Longrightarrow$ a bug in the state.

What is a bug?



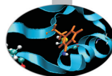
- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state $\Longrightarrow$ a bug in the state.
- Failure: An osservable incorrect program behaviour

What is a bug?



- Defect: An incorrect program code $\Longrightarrow$ a bug in the code.
- Infection: An incorrect program state $\Longrightarrow$ a bug in the state.
- Failure: An osservable incorrect program behaviour $\Longrightarrow$ a bug in the behaviour.

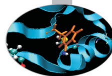
Infection chain



Defect

- The programmer creates a defect in the program code (also known as bug or fault).

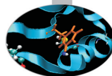
Infection chain



Defect $\Longrightarrow$ Infection

- The programmer creates a defect in the program code (also known as bug or fault).
- The defect causes an in infection in the program state.

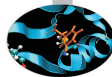
Infection chain



Defect $\Longrightarrow$ Infection $\Longrightarrow$ Failure

- The programmer creates a defect in the program code (also known as bug or fault).
- The defect causes an in infection in the program state.
- The infection creates a failure - an externally observable error.

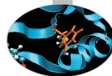
Tracking down defect



Failure

- A Failure is visible to the end user of a program. For example, the program prints an incorrect output.

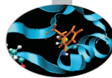
Tracking down defect



Infection $\Longleftarrow$ Failure

- A Failure is visible to the end user of a program. For example, the program prints an incorrect output.
- Infection is the underlying state of the program at runtime that leads to a Failure. For example, the program might display the incorrect output because the wrong value is stored in avariable.

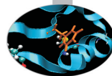
Tracking down defect



Defect $\Longleftarrow$ Infection $\Longleftarrow$ Failure

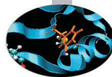
- A Failure is visible to the end user of a program. For example, the program prints an incorrect output.
- Infection is the underlying state of the program at runtime that leads to a Failure. For example, the program might display the incorrect output because the wrong value is stored in avariable.
- A Defect is the actual incorrect fragment of code that the programmer wrote; this is what must be changed to fix the problem.

First of all..



THE BEST WAY TO DEBUG A PROGRAM IS TO MAKE NO MISTAKES

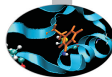
First of all..



THE BEST WAY TO DEBUG A PROGRAM IS TO MAKE NO MISTAKES

- Preventing bugs.
- Detecting/locating bugs.

First of all..



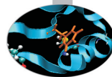
THE BEST WAY TO DEBUG A PROGRAM IS TO MAKE NO MISTAKES

- Preventing bugs.
- Detecting/locating bugs.

Ca. 80 percent of software development costs spent on identifying and correcting defects.

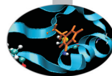
It is much more expensive (in terms of time and effort) to detect/locate exisisting bugs, than prevent them in the first place.

IfSQ Institute for Software Quality



www.ifsq.org

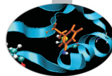
IfSQ Institute for Software Quality



www.ifsq.org

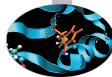
- We have an agreed common goal: to raise the standard of software (and software development) around the world by promoting Code Inspection as a prerequisite to Software Testing in the production and delivery cycle.
- Our strong hope is that eventually the idea of inspecting code before testing will be as self-evident as testing software before using it.

The Fundamental question



How can I prevent Bugs?

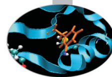
The Fundamental question



How can I prevent Bugs?

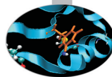
- Design.
- Good writing.
- Self-checking code.
- Test scaffolding.

Preventing bugs via design



Programming is a design activity.
It’s a creative act, not mechanical code generation.

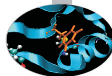
Preventing bugs via design



Programming is a design activity.
It’s a creative act, not mechanical code generation.

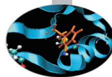
- Good modularization.
- Strong encapsulation/information hiding.
- Clear,simple pre- and post-processing.
- Requirements of operations should testable.

What is a module?



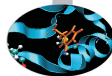
- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.

What is a module?



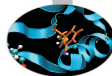
- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.
  39% of all errors are caused by internal
  interface errors / errors in communication between routines.

What is a module?



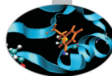
- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.
  39% of all errors are caused by internal
  interface errors / errors in communication between routines.
- Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand.
- The art of software engineering is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.

What is a module?



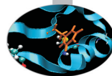
- A module is a discrete, well-defined, small component of a system.
- The smaller the component, the easier it is to understand.
- Every component has interfaces with other components, and all interfaces are sources of confusion.
  39% of all errors are caused by internal
  interface errors / errors in communication between routines.
- Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand.
- The art of software engineering is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.
  A study showed that when routines averaged 100 to 150 lines each, code was more stable and required less changes.

Good modularization...



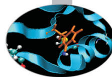
- reduces complexity
- avoids duplicate code
- facilitates reusable code
- limits effects of the changes
- facilitates test
- results in easier implementation

Strong encapsulation/information hiding



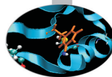
- The principle of information hiding suggests that modules be characterized by design decisions that each hides from all others.
- The information contained within a module is inaccesible to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- Because most data and procedure are hidden from other parts of software , inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Cohesion & Coupling



- Cohesion: a measure of how closely the instructions in a module are related to each other.
- Coupling: a measure of how closely a modules' execution depends upon other modules.

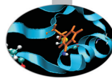
Cohesion & Coupling



- Cohesion: a measure of how closely the instructions in a module are related to each other.
- Coupling: a measure of how closely a modules' execution depends upon other modules.

- Avoid global variables.
- As a general rule, you should always aim to create modules that have strong cohesion and weak coupling.
- The routines with the highest coupling-to-cohesion ratios had 7 times more errors than those with the lowest ratios.

Cohesion & Coupling

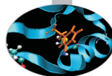


- Cohesion: a measure of how closely the instructions in a module are related to each other.
- Coupling: a measure of how closely a modules' execution depends upon other modules.

- Avoid global variables.
- As a general rule, you should always aim to create modules that have strong cohesion and weak coupling.
- The routines with the highest coupling-to-cohesion ratios had 7 times more errors than those with the lowest ratios.

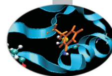
Spaghetti code is characterized by very strong coupling.

Preventing bugs via good writing



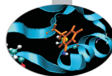
- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.

Preventing bugs via good writing



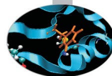
- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.
      Studies have shown that few people can understand nesting of conditional statements to more than 3 levels.

Preventing bugs via good writing



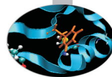
- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.
      Studies have shown that few people can understand nesting of conditional statements to more than 3 levels.
- Comments,comments,comments.
- Small size of functions,routines.

Preventing bugs via good writing



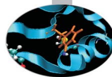
- Clarity is more important than efficiency: clarity of writing and style.
  - Use simple expressions, not complex
  - Use meaningful names
  - Clarity of purpose for:
    - Functions.
    - Loops.
    - Nested constructs.
      Studies have shown that few people can understand nesting of conditional statements to more than 3 levels.
- Comments,comments,comments.
- Small size of functions,routines.
  A study at IBM found that the most error-prone routines were those larger than 500 lines of code.

Preventing bugs via good writing 2



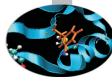
- Use the smallest acceptable scope for:
  - Variable names.
  - Static/local functions.
- Use named intermediate values.
- Avoid "Magic numbers".
- Generalize your code.
- Document your program.
- Write standard language.

Preventing bugs via Self-checking code



- Checking assumptions.
- "assert" macro.
- Custom "assert" macros.
- Assertions about intermediate values.
- Preconditions,postcondition.

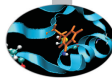
Preventing bugs via Self-checking code



- Checking assumptions.
- "assert" macro.
- Custom "assert" macros.
- Assertions about intermediate values.
- Preconditions,postcondition.

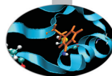
Defensive programming.

Checking assumptions



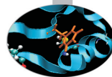
- That an input parameter's value falls within its expected range (or an output parameters' value does)
- That the value of an input-only variable is not changed by a routine.
- That a pointer is non-NULL.
- That an array or other container passed into a routine can contain at least X data number of data elements.
- That a table has been initialized to contain real values.
- Etc.

Assertions



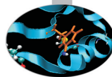
- An assertion is a code (usually routine or macro) that allows a program to check itself as runs.
- When an assertion is true, that means everything is operating as expected
- When it's false, that means it has detected an unexpected error in the code.
- Use error handling code for conditions you expected to occur.

Error handling technique



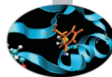
- Return a neutral value.
- Substitute the closest legal value.
- Log a warning message to a file.
- Return an error code.
  - Set a value of a status variable.
  - Return status as the function's return value.
  - Throw an exception using the language's build-in exception mechanism.
- Display an error message wherever the error is encountered.
- Handle the error in whatever way works best locally.
- Shutdown.

Outline



- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
- Compilatori e ottimizzazione
- Librerie scientifiche
- Floating Point Computing

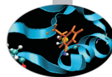
Testing: a simple program



Input

Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Testing: a simple program



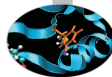
Input

Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Output

Tell whether the triangle is:
Scalene
Isosceles
Equilater.

Testing: a simple program



Input

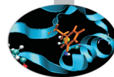
Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Output

Tell whether the triangle is:
Scalene
Isosceles
Equilater.

Create a set of test cases for this program.

Testing: a simple program



Input

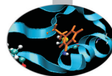
Read three integer values from the command line.
The three values represent the lengths of the sides of a triangle.

Output

Tell whether the triangle is:
Scalene
Isosceles
Equilater.

Create a set of test cases for this program.
("The art of sotware testing" G.J. Myers)

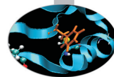
Classification of errors



Syntax

- Definition: errors in grammar (violations of the "rules" for forming legal language statements).
- Examples: undeclared identifiers, mismatched parentheses in an expression, an opening brace without a matching closing brace, etc.
- Occur: an error that is caught in the process of compiling the program.

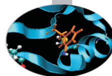
Classification of errors



Syntax

- Definition: errors in grammar (violations of the "rules" for forming legal language statements).
- Examples: undeclared identifiers, mismatched parentheses in an expression, an opening brace without a matching closing brace, etc.
- Occur: an error that is caught in the process of compiling the program.
  The easiest errors to spot by a compiler or some aid in checking the syntax.
  As you get more practice using a language, you naturally make fewer errors, and will be able to quickly correct those that do occur.

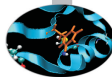
Classification of errors



Runtime

- Definition: "Asking the computer to do the impossible!"
- Examples: division by zero, taking the square root of a negative number, referring to the 101th on a list of only 100 items, deferencing a null pointer, etc.
- Occur: an error that is not detected until the program is executed, and then causes a processing error to occur.

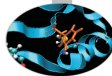
Classification of errors



Runtime

- Definition: "Asking the computer to do the impossible!"
- Examples: division by zero, taking the square root of a negative number, referring to the 101th on a list of only 100 items, deferencing a null pointer, etc.
- Occur: an error that is not detected until the program is executed, and then causes a processing error to occur.
  They are not easy to spot because they are not syntax or grammar errors, they are subtle errors and develop in the course of program's execution.
  Avoiding exceptions and correcting program behaviour is also largely a matter of experience.
  To prevent use defensive programming.

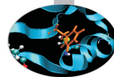
Classification of errors



Logic (semantic, meaning)

- Definition: the program compiles (no syntax errors) and runs to a normal completion (no runtime erros), but the output is wrong. A statement may be syntactically correct, but mean something other than what we intended. Therefore it has a different effect, causing the program output to be wrong.
- Examples: improper initialization of variables, performing arithmetic operations in the wrong order, using an incorrect formula to compute a value, etc.
- Occur: an error that affect how the code works.

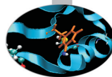
Classification of errors



Logic (semantic, meaning)

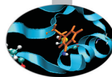
- Definition: the program compiles (no syntax errors) and runs to a normal completion (no runtime erros), but the output is wrong. A statement may be syntactically correct, but mean something other than what we intended. Therefore it has a different effect, causing the program output to be wrong.
- Examples: improper initialization of variables, performing arithmetic operations in the wrong order, using an incorrect formula to compute a value, etc.
- Occur: an error that affect how the code works.
  It is a type of error that only the programmer can recognize.Finding and correcting logic errors in a program is known as debugging.

Outline



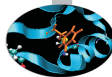
- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
- Compilatori e ottimizzazione
- Librerie scientifiche
- Floating Point Computing

Definitions



Static analysis refers to a method of examinig software that allows developers to discover dangerous programming pratices or potential errors in source code, without actually run the code.

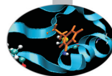
Definitions

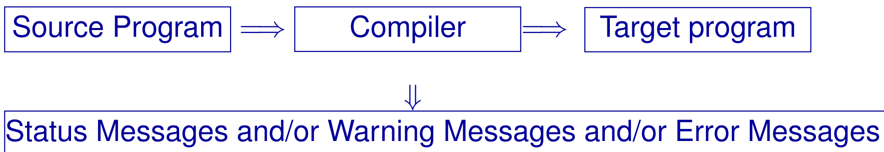


Static analysis refers to a method of examinig software that allows developers to discover dangerous programming pratices or potential errors in source code, without actually run the code.

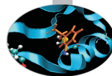
- Using compiler options.
- Using static analyzer.

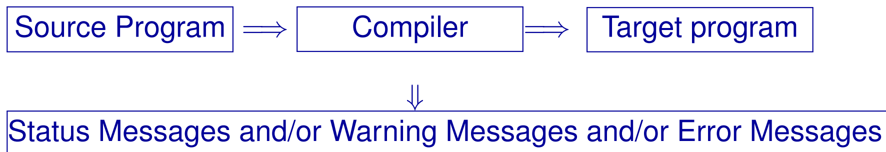
Compiler and errors





Compiler and errors

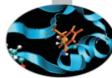




Compiler checks:

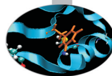
- Syntax.
- Semantic.

Lexical (Syntactic) errors



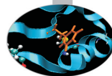
- Characters in the source that aren't in the alphabet of the language.
- Words in the source that aren't in the vocabulary of the language.

Syntactic errors



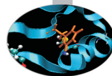
- Comment delimiters that have been put in wrong place or omitted.
- Literal delimiters that have been put in wrong place or omitted.
- Keywords that have been misspelled.
- Required punctuation that is missing.
- Construct delimiters such as parentheses or braces that have been missplaced.
- Blank or tab characters that are missing.
- Blank or tab characters that shouldn't occur where they're found.

Semantic errors



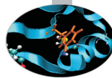
- Names that aren't declared.
- Operands of the wrong type for the operator they're used with.
- Values that have the wrong type for the name to which they're assigned.
- Procedures that are invoked with the wrong number of arguments.
- Procedures that are invoked with the wrong type of arguments.
- Function return values that are the wrong type for the context in which they're used.

Semantic errors



- Code blocks that are unreachable.
- Code blocks that have no effect.
- Local variables that are used before being initialized or assigned.
- Local variables that are initialized or assigned but not used.
- Procedures that are never invoked.
- Procedures that have no effect.
- Global variables that are used before being initialized or assigned.
- Global variables that are initialized or assigned, but not used.

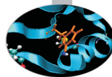
Compilers



- Not all compilers find the same defects.
- The more information a compilers has, the more defects it can find.
- Some compilers operate in "forgiving" mode but have "strict" or "pedantic" mode, if you request it.

SCAI
SuperComputing Applications and Innovation

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

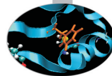
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

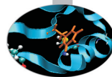
Floating Point Computing

Static analysis:example 1 part one



```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f\n", 4);
5    return 0;
6  }
```

Static analysis:example 1 part one



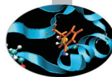
```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f\n", 4);
5    return 0;
6  }
```

```
<ruggiero@shiva ~/CODICI>gcc bad.c
```

```
<ruggiero@shiva ~/CODICI>./a.out
```

```
Two plus two is 0.000000
```

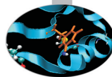
Static analysis:example 1 part two



```
<ruggiero@shiva ~/CODICI>gcc -Wall bad.c
```

```
bad.c: In function main:
bad.c:4: warning: format ’%f’ expects type ’double’,
but argument 2 has type ’int’
```

Static analysis:example 1 part two

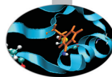


```
<ruggiero@shiva ~/CODICI>gcc -Wall bad.c
```

```
bad.c: In function main:
bad.c:4: warning: format '%f' expects type 'double',
but argument 2 has type 'int'
```

```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f \n", 4);
5    return 0;
6  }
```

Static analysis:example 1 part two

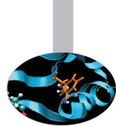


```
<ruggiero@shiva ~/CODICI>gcc -Wall bad.c
```

```
bad.c: In function main:
bad.c:4: warning: format '%f' expects type 'double',
but argument 2 has type 'int'
```

```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %d\n", 4);
5    return 0;
6  }
```

gcc options

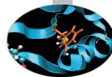


-Wall

turns on the most commonly-used compiler warnings option

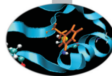
-Waddress -Warray-bounds (only with -O2) -Wc++0x-compat -Wchar-subscripts -Wimplicit-int -Wimplicit-function-declaration -Wcomment -Wformat -Wmain (only for C/ObjC and unless -ffreestanding) -Wmissing-braces -Wnonnull -Wparentheses -Wpointer-sign -Wreorder -Wreturn-type -Wsequence-point -Wsign-compare (only in C++) -Wstrict-aliasing -Wstrict-overflow=1 -Wswitch -Wtrigraphs -Wuninitialized -Wunknown-pragmas -Wunused-function -Wunused-label -Wunused-value -Wunused-variable -Wvolatile-register-var

Oher compilers



```
[ruggiero@matrix1 ~]$ pgcc bad.c
```

Oher compilers

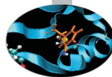


```
[ruggiero@matrix1 ~]$ pgcc bad.c
```

```
[ruggiero@matrix1 ~]$ icc bad.c
```

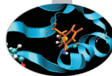
```
bad.c(4): warning #181: argument is incompatible
with corresponding format string conversion
    printf ("Two plus two is %f\n", 4);
                                    ^
```

check.c:source



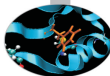
```
1  #include <stdio.h>
2
3  int main (void)
4  {
5            return 0;
6  }
7  void checkVal(unsigned int n) {
8            if (n < 0) {
9                        /* Do something... */
10                     }
11             else if (n >= 0) {
12                         /* Do something else... */
13                       }
14 }
```

check.c:compilation



```
<ruggiero@shiva:~> gcc -Wall check.c
```

check.c:compilation

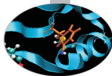


```
<ruggiero@shiva:~> gcc -Wall check.c
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra check.c
```

```
check.c: In function ?checkVal?:
check.c:8: warning: comparison of unsigned expression < 0 is always false
check.c:11: warning: comparison of unsigned expression >= 0 is always true
```

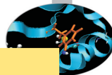
gcc options



-Wextra (-W)
reports the most common programming errors and less-serious but potential problem

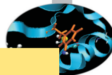
```
-Wclobbered -Wempty-body
-Wignored-qualifiers -Wmissing-field-initializers
-Wmissing-parameter-type (C only) -Wold-style-declaration (C only)
-Woverride-init -Wsign-compare
-Wtype-limits -Wuninitialized (only with -O1 and above)
-Wunused-parameter (only with -Wunused or -Wall)
```

check1.c:source



```
1   #include <stdio.h>
2    int main (void)
3    {
4            double x = 10.0;
5            double y = 11.0;
6            double z = 0.0;
7            if  (x == y)  {
8                        z = x * y;
9            }
10             return 0;
11   }
```

check1.c:source



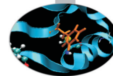
```
1  #include <stdio.h>
2   int main (void)
3   {
4           double x = 10.0;
5           double y = 11.0;
6           double z = 0.0;
7           if (x == y) {
8                   z = x * y;
9           }
10            return 0;
11  }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra  check1.c
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra -Wfloat-equal check1.c
```

```
check1.c: In function main:
check1.c:8: warning: comparing  floating point
with == or != is unsafe
```

gcc: other compiler options

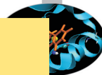


-Wno-div-by-zero -Wsystem-headers -Wfloat-equal -Wtraditional (C only) -Wdeclaration-after-statement (C only) -Wundef -Wno-endif-labels -Wshadow -Wlarger-than-len -Wpointer-arith -Wbad-function-cast (C only) -Wcast-align -Wwrite-strings -Wconversion -Wsign-compare -Waggregate-return -Wstrict-prototypes (C only) -Wold-style-definition (C only) -Wmissing-prototypes (C only) -Wmissing-declarations (C only) -Wcast-qual -Wmissing-field-initializers -Wmissing-noreturn -Wmissing-format-attribute -Wno-multichar -Wno-deprecated-declarations -Wpacked -Wpadded -Wredundant-decls -Wnested-externs (C only) -Wvariadic-macros -Wunreachable-code -Winline -Wno-invalid-offsetof (C++ only) -Winvalid-pch -Wlong-long -Wdisabled-optimization -Wno-pointer-sign

testinit.c:source

```
1  int main() {
2          int v[16];
3          int i,j,k;
4          j=i;
5          v[i]= 42;
6          return 0 ;
7  }
```

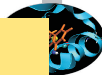
testinit.c:source



```
1   int main() {
2           int v[16];
3           int i,j,k;
4           j=i;
5           v[i]= 42;
6           return 0 ;
7   }
```

```
ruggiero@shiva:~> gcc -Wall -Wextra testinit.c
```

testinit.c:source



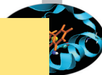
```
1 int main() {
2         int v[16];
3         int i,j,k;
4         j=i;
5         v[i]= 42;
6         return 0 ;
7 }
```

```
ruggiero@shiva:~> gcc -Wall -Wextra testinit.c
```

```
ruggiero@shiva:~> gcc -O1 -Wall -Wextra -Wuninitialized testinit.c
```

```
testinit.c: In function main
testinit.c:4: warning: unused variable k
testinit.c:5: warning: i is used uninitialized
in this function
```

testinit.c:source



```
1  int main() {
2          int v[16];
3          int i,j,k;
4          j=i;
5          v[i]= 42;
6          return 0 ;
7  }
```

```
ruggiero@shiva:~> gcc -Wall -Wextra testinit.c
```

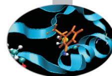
```
ruggiero@shiva:~> gcc -O1 -Wall -Wextra -Wuninitialized testinit.c
```

```
testinit.c: In function main
testinit.c:4: warning: unused variable k
testinit.c:5: warning: i is used uninitialized
in this function
```

```
[ruggiero@matrix1 ~]$ icc testinit.c
```

```
testinit.c(5): warning #592: variable "i" is used
               before its value is set j=i;
```

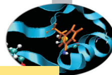
Compilation: default options



```
program par
implicit none
integer,parameter  :: hacca=10

call sub(hacca)

end

subroutine sub(hacca)
implicit none
integer hacca

hacca=hacca+1
write(*,*) hacca

return

end
```

SuperComputing Applications and Innovation

Compilation: default options

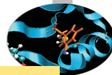


```
> xlf par.f -o par.x
```

```
> ./par.x
```

11

Compilation: default options



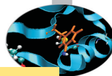
```
> xlf par.f -o par.x
```

```
> ./par.x
```

```
11
```

```
<ruggiero@ife2 ~>ifort par.f -o par.x
```

Compilation: default options



```
> xlf par.f -o par.x
```

```
> ./par.x
```

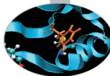
```
11
```

```
<ruggiero@ife2 ~>ifort par.f -o par.x
```

```
<ruggiero@ife2 ~> ./par.x
```

```
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
par.x              0000000000402688  Unknown            Unknown     Unknown
par.x              0000000000402670  Unknown            Unknown     Unknown
par.x              000000000040262A  Unknown            Unknown     Unknown
libc.so.6          00000036FF81C40B  Unknown            Unknown     Unknown
par.x              000000000040256A  Unknown            Unknown     Unknown
```

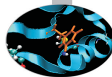
Compilation: default options



```
<ruggiero@ife2 ~> man ifort
```

```
? -assume noprotect_constants

Tells the compiler to pass a copy of a constant actual argument.
This copy  can  be  modified  by  the called routine, even though
the Fortran standard prohibits such modification. The calling
routine does not see any modification to the constant. The default
is -assume protect_constants, which  passes  the constant actual
argument.  Any attempt to modify it results in an error.
```

Compilation: default options

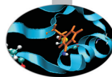


```
<ruggiero@ife2 ~> man ifort
```

```
? -assume noprotect_constants

Tells the compiler to pass a copy of a constant actual argument.
This copy  can  be  modified  by  the called routine, even though
the Fortran standard prohibits such modification. The calling
routine does not see any modification to the constant. The default
is -assume protect_constants, which  passes  the constant actual
argument.  Any attempt to modify it results in an error.
```

```
<ruggiero@ife2 ~>ifort par.f -assume noprotect_constants -o par.x
```

Compilation: default options



```
<ruggiero@ife2 ~> man ifort
```

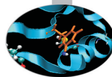
```
? -assume noprotect_constants

Tells the compiler to pass a copy of a constant actual argument.
This copy  can  be  modified  by  the called routine, even though
the Fortran standard prohibits such modification. The calling
routine does not see any modification to the constant. The default
is -assume protect_constants, which  passes  the constant actual
argument.  Any attempt to modify it results in an error.
```

```
<ruggiero@ife2 ~>ifort par.f -assume noprotect_constants -o par.x
```

```
<ruggiero@ife2 ~>./par.x
```

```
11
```

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

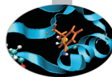
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

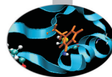
Floating Point Computing

splint



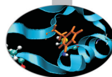
- Open Source Static Analysis Tool developed at University of Virginia by Professor Dave Evans
- Based on Lint
- www.splint.org
- splint [-option -option ...] filename [filename ...]

splint can detect with just source code...



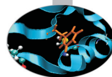
- Unused declarations.
- Type inconsistencies.
- Variables used before being assigned.
- Function return values that are ignored.
- Execution paths with no return.
- Switch cases that fall through.
- Apparent infinite loops.

splint can detect with annotation information



- Dereferencing pointers with possible null values.
- Using storage that is undefined or partly undefined.
- Returning storage that is undefined or partly defined.
- Type mismatches.
- Using deallocated storage.

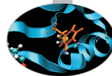
splint can detect with annotation information



- Memory leaks.
- Inconsistent modification of caller visible states.
- Violations of information hiding.
- Undefined program behaviour due to evaluation order, incomplete logic, infinite loops, statements with no effect, and so on.
- Problematic uses of macros.

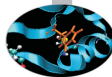
SCAI SuperComputing Applications and Innovation

Static analysis:example 1 part one



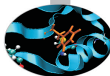
```
1  #include <stdio.h>
2  int main (void)
3  {
4    printf ("Two plus two is %f\n", 4);
5    return 0;
6  }
```

check.c:source



```
1   #include <stdio.h>
2
3   int main (void)
4   {
5               return 0;
6   }
7   void checkVal(unsigned int n) {
8               if (n < 0) {
9                           /* Do something... */
10                          }
11                else if (n >= 0) {
12                            /* Do something else... */
13                            }
14  }
```

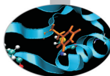
splint:examples



```
<ruggiero@shiva:~> splint-3.1.2/bin/splint bad.c
```

```
Finished checking --- no warnings
```

splint:examples



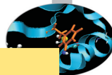
```
<ruggiero@shiva:~> splint-3.1.2/bin/splint bad.c
```

```
Finished checking --- no warnings
```

```
<ruggiero@shiva:~> splint-3.1.2/bin/splint check.c
```

```
splint 3.1.2 --- 28 Mar 2008
check.c: (in function checkVal)
check.c:8:15: Comparison of unsigned value involving zero: n < 0
 An unsigned value is used in a comparison with zero in a way that is
 a bug or confusing. (Use -unsignedcompare to inhibit warning)
check.c:11:22: Comparison of unsigned value involving zero: n >= 0
Finished checking --- 2 code warnings
```

check1.c:source



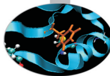
```
1  #include <stdio.h>
2   int main (void)
3   {
4           double x = 10.0;
5           double y = 11.0;
6           double z = 0.0;
7           if  (x == y) {
8                       z = x * y;
9           }
10            return 0;
11  }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra  check1.c
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra -Wfloat-equal check1.c
```

```
check1.c: In function main:
check1.c:8: warning: comparing  floating point
with == or != is unsafe
```

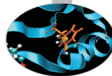
splint:examples



```
<ruggiero@shiva:~> splint-3.1.2/bin/splint check1.c
```

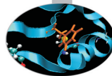
```
Splint 3.1.2 --- 28 Mar 2008

check1.c: (in function main)
check1.c:8:14: Dangerous equality comparison involving
double types: x == y Two real (float, double, or long double)
values are compared directly using == or != primitive. This
may produce unexpected results since floating point representations
are inexact. Instead, compare the difference to FLT_EPSILON
or DBL_EPSILON. (Use -realcompare to inhibit warning)

Finished checking --- 1 code warning
```

check1.c: rewritten



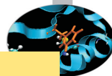
```
1   int main (void)
2     {
3               double x = 10.0;
4               double y = 11.0;
5               double z = 0.0;
6               double norma=0.1e-16;
7               double epsilon;
8              epsilon=x-y;
9              if  (epsilon < norma) {
10                         z = x * y;
11             }
12
13               return 0;
14   }
```

average.c: source



```
1   #include <stdio.h>
2   int main (void) {
3     int size = 5;
4     int a;
5     float total;
6     float art[size];
7
8    for(a = 0 ; a < size; ++ a) {
9      art[a] = (float) a;
10     total  += art[a]; }
11    printf(" %f\n" , total / (float) size);
12    return 0;
13  }
```

average.c: compilation e run

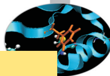


```
<ruggiero@shiva:~> gcc -Wall -Wextra average.c
```

```
<ruggiero@shiva:~> ./a.out
```

```
1.999994
```

average.c: compilation e run



```
<ruggiero@shiva:~> gcc -Wall -Wextra average.c
```

```
<ruggiero@shiva:~> ./a.out
```

```
1.999994
```

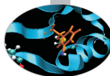
```
<ruggiero@shiva:~> gcc -Wall  -Wextra -O2 average.c
```

```
average.c: In function main:
average.c:5: warning: total may be used uninitialized
in this function
```

```
<ruggiero@shiva:~> ./a.out
```

```
nan
```

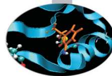
average.c: splint



```
<ruggiero@shiva:~> splint-3.1.2/bin/splint average.c
```

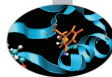
```
Splint 3.1.2 --- 28 Mar 2008

average.c: (in function main)
average.c:10:4 Variable total used before definition
An rvalue is used that may not be initialized to a value
on some execution path. (Use -usedef to inhibit warning)
average.c:11:20: Variable total used before definition

Finished checking --- 2 code warnings
```

average.c: new version



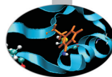
```
1  #include <stdio.h>
2  int main (void) {
3    int size = 5;
4    int a;
5    float total;
6    float art[size];
7     total=0;
8   for(a = 0 ; a < size; ++ a) {
9     art[a] = (float) a;
10    total  += art[a]; }
11   printf(" %f\n" , total / (float) size);
12   return 0;
13 }
```

assign.c: source and compilation



```
1  #include <stdio.h>
2  main()
3  {
4          int a=0;
5          while (a=1)
6                  printf("hello\n");
7          return 0;
8  }
```

assign.c: source and compilation

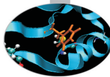


```
1  #include <stdio.h>
2  main()
3  {
4          int a=0;
5          while (a=1)
6               printf("hello\n");
7          return 0;
8  }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra assign.c
```

```
assigna.c:3: warning: return type defaults to int
assign.c: In function main:
assign.c:5: warning: suggest parentheses around assignment used as
truth value
```

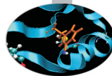
assign.c: splint



```
<ruggiero@shiva:~>splint-3.1.2/bin/splint assign.c
```

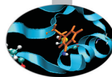
```
Splint 3.1.2 --- 28 Mar 2008
assign.c: (in function main)
assign.c:5:14: Test expression for while is assignment expression:a=1
The condition test is an assignment expression. Probably, you mean
to use == instead of =. If an assignment is intended, add an extra
parentheses nesting (e.g., if ((a = b)) ...) to suppress this message.
(Use -predassign to inhibit warning)
assign.c:5:14: Test expression for while not boolean, type int: a=1
Test expression type is not boolean or int. (Use -predboolint
to inhibit warning)
Finished checking --- 2 code warnings
```

memory.c: source and compilation



```
1
2   #include <stdlib.h>
3   int main()
4   {
5           int *p = malloc(5*sizeof(int));
6
7
8
9           *p = 1;
10          free(p);
11          return 0;
12  }
```

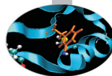
memory.c: source and compilation



```
1
2   #include <stdlib.h>
3   int main()
4   {
5           int *p = malloc(5*sizeof(int));
6
7
8
9           *p = 1;
10          free(p);
11          return 0;
12  }
```

```
<ruggiero@shiva:~> gcc -Wall -Wextra memory.c
```

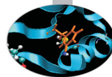
memory.c: splint



```
<ruggiero@shiva:~> splint-3.1.2/bin/splint memory.c
```

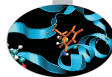
```
Splint 3.1.2 --- 28 Mar 2008

memory.c: (in function main)
memory.c:9:10: Dereference of possibly null pointer p: *p
A possibly null pointer is dereferenced.  Value is either
the result of a function which may return null (in which
case, code should check it is not null), or a global,
parameter or structure field declared with the null
qualifier. (Use -nullderef to inhibit warning)
memory.c:5:18: Storage p may become null

Finished checking --- 1 code warning
```

memory.c: new version



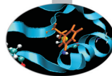
```
1   #include <stdlib.h>
2   #include <stdio.h>
3   int main()
4   {
5           int *p = malloc(5*sizeof(int));
6           if (p == NULL) {
7               fprintf(stderr, "error in malloc");
8               exit(EXIT_FAILURE);
9           } else  *p = 1;
10          free(p);
11          return 0;
12  }
```

out_b.c: source



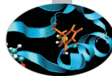
```
1   #include<stdio.h>
2   #define N 5
3   int main (void)
4   {
5           int t[N];
6           int i;
7
8           i=6;
9           t[i] = i+1;
10
11          return 0;
12  }
```

out_b.c: splint



```
<ruggiero@shiva:~> splint-3.1.2/bin/splint out_b.c
```

out_b.c: splint

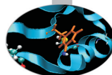


```
<ruggiero@shiva:~> splint-3.1.2/bin/splint out_b.c
```

```
<ruggiero@shiva:~> splint-3.1.2/bin/splint +bounds out_b.c
```

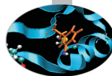
```
Splint 3.1.2 --- 28 Mar 2008

out_b.c: (in function main)
out_b.c:9:9: Likely out-of-bounds store: t[i]
    Unable to resolve constraint:
    requires 4 >= 6
     needed to satisfy precondition:
    requires maxSet(t @ out_b.c:9:9) >= i @ out_b.c:9:11
  A memory write may write to an address beyond the allocated buffer. (Use
  -likelyboundswrite to inhibit warning)

Finished checking --- 1 code warning
```

error.f: source



```
1           REAL FUNCTION COMPAV(SCORE,COUNT)
2               INTEGER SUM,COUNT,J,SCORE(5)
3               DO 30 I = 1,COUNT
4                   SUM = SUM + SCORE(I)
5    30         CONTINUE
6               COMPAV = SUM/COUNT
7               write(*,*) compav
8           END
9           PROGRAM AVENUM
10            PARAMETER(MAXNOS=10)
11            INTEGER I, COUNT
12            REAL NUMS(MAXNOS), AVG
13            COUNT = 0
14            DO 80 I = 1,MAXNOS
15                READ (5,*,END=100) NUMS(I)
16                COUNT = COUNT + 1
17  80         CONTINUE
18  100        AVG = COMPAV(NUMS, COUNT)
19          END
```

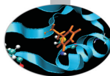
error.f: compilation



```
<ruggiero@ife2 /ftnchek>gfortran -Wall -Wextra -O2 error.f
```

```
In file error.f:2

          INTEGER SUM,COUNT,J,SCORE(5)
                            1
CWarning: Unused variable j declared at (1)
error.f: In function compav:
error.f:2: warning: sum may be used uninitialized in this function
```

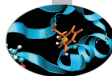
error.f: compilation



```
<ruggiero@ife2/ftnchek>pgf90 -Minform=inform error.f
```

```
PGF90-I-0035-Predefined intrinsic sum loses intrinsic property (error.f: 5)
PGF90-I-0035-Predefined intrinsic count loses intrinsic property (error.f: 16)
```

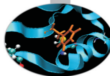
error.f: compilation



```
<ruggiero@ife2 /ftnchek>ifort -warn all error.f
```

```
fortcom: Warning: error.f, line 4: This name has not been given an explicit type.   [I]
             DO 30 I = 1,COUNT
-------------------^
fortcom: Info: error.f, line 2: This variable has not been used.   [J]
             INTEGER SUM,COUNT,J,SCORE(5)
-------------------------------^
fortcom: Warning: error.f, line 13: This name has not been given an explicit type.   [MAXNOS]
           PARAMETER(MAXNOS=10)
---------------------^
fortcom: Warning: error.f, line 21: This name has not been given an explicit type.   [COMPAV]
100         AVG = COMPAV(NUMS, COUNT)
------------------^
```

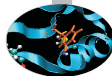
error.f: compilation



<ruggiero@ife2 /ftnchek>g95 -Wall -Wextra error.f

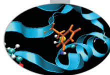
```
In file error.f:6

 30          CONTINUE
                    1
Warning (142): Nonblock DO statement at (1) is obsolescent
In file error.f:1

      REAL FUNCTION COMPAV(SCORE,COUNT)
                          1
Warning (163): Actual argument 'score' at (1) does not have an INTENT
In file error.f:1

      REAL FUNCTION COMPAV(SCORE,COUNT)
                                1
Warning (163): Actual argument 'count' at (1) does not have an INTENT
In file error.f:2

             INTEGER SUM,COUNT,J,SCORE(5)
                                1
Warning (137): Variable 'j' at (1) is never used and never set
```

error.f: compilation



```
In file error.f:20

80          CONTINUE
                    1
Warning (142): Nonblock DO statement at (1) is obsolescent
In file error.f:21

100         AVG = COMPAV(NUMS, COUNT)
                  1
Warning (165): Implicit interface 'compav' called at (1)
In file error.f:15

           REAL NUMS(MAXNOS), AVG
                             1
Warning (112): Variable 'avg' at (1) is set but never used
In file error.f:21

100         AVG = COMPAV(NUMS, COUNT)
                         1
In file error.f:1

      REAL FUNCTION COMPAV(SCORE,COUNT)
                           2
Warning (155): Inconsistent types (REAL(4)/INTEGER(4))
in actual argument lists at (1) and (2)
```

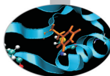
www.dsm.fordham.edu/~ftnchek/



```
<ruggiero@ife2/ftnchek>./bin/ftnchek error.f
```

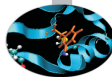
```
FTNCHEK Version 3.3 November 2004

File error.f:
      7               COMPAV = SUM/COUNT
                                 ^
Warning near line 7 col 21 file error.f:
 integer quotient expr SUM/COUNT  converted to real

Warning in module COMPAV in file error.f:
 Variables declared but never referenced:
    J declared at line 2 file error.f

Warning in module COMPAV in file error.f:
 Variables may be used before set:
    SUM used at line 5 file error.f
    SUM set at line 5 file error.f

Warning in module AVENUM in file error.f:
 Variables set but never used:
    AVG set at line 21 file error.f
```

www.dsm.fordham.edu/~ftnchek/



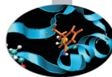
```
 0 syntax errors detected in file error.f
 4 warnings issued in file error.f

Warning: Subprogram COMPAV argument data type mismatch at position 1:
    Dummy arg SCORE in module COMPAV line 1 file
 error.f is type intg
    Actual arg NUMS in module AVENUM line 21 file
 error.f is type real

Warning: Subprogram COMPAV argument arrayness mismatch at position 1:
    Dummy arg SCORE in module COMPAV line 1 file
 error.f has 1 dim size 5
    Actual arg NUMS in module AVENUM line 21 file
 error.f has 1 dim size 10
```

Static analyzer for Fortran



- Forcheck can handle up Fortran 95 and some Fortran 2003. item Cleanscape FortranLint can handle up to Fortran 95.
- plusFORT is a multi-purpose suite of tools for analyzing and improving Fortran programs.
- ...

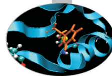
Static analyzer: errors and warnings



- 40% false positive reports of correct code.
- 40% multiple occurence of same problem.
- 10% minor or cosmetic problems.
- 10% serious bugs, very hard to find by other methods.

From **"The Developer's Guide to Debugging"** T. Grotker, U. Holtmann, H. Keding, M. Wloka

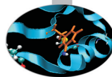
Static analyzer: Lessons learned



- Do not ignore compiler warnings, even if they appear to be harmless.
- Use multiple compilers to check the code.
- Familiarize yourself with a static checker.
- Reduce static checker errors to (almost) zero.
- Rerun all test cases after a code cleanup.
- Doing regular sweeps of the source code will pay off in long term.

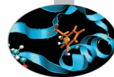
From **"The Developer's Guide to Debugging"** T. Grotker, U. Holtmann, H. Keding, M. Wloka

Outline



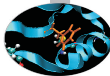
- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
- Compilatori e ottimizzazione
- Librerie scientifiche
- Floating Point Computing

Runtime signals



- When a job terminates abnormally, it usually tries to send a signal (exit code) indicating what went wrong.
- The exit code from a job is a standard OS termination status.
- Typically, exit code 0 (zero) means successful completion.
- Your job itself calling exit() with a non-zero value to terminate itself and indicate an error.
- The specific meaning of the signal numbers is platform-dependent.

Runtime signals

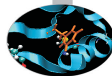


You can find out why the job was killed using:

```
[ruggiero@matrix1 ~]$ kill -l
```

```
 1) SIGHUP   2) SIGINT   3) SIGQUIT   4) SIGILL
 5) SIGTRAP   6) SIGABRT   7) SIGBUS   8) SIGFPE
 9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT
17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU
25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH
29) SIGIO 30) SIGPWR 31) SIGSYS 34) SIGRTMIN
....
```

Runtime signals

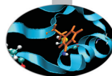


To find out what all the "kill -l" words mean:

```
[ruggiero@matrix1 ~]$ man 7 signal
```

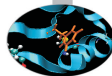
```
.....
Signal     Value     Action   Comment
-------------------------------------------------------------------------
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
......
```

Action description



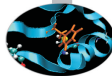
Term	Default action is to terminate the process.
Ign	Default action is to ignore the signal.
Core	Default action is to terminate the process and dump the core.
Stop	Default action is to stop the process.
Cont	Default action is to continue the process if is currently stopped.

Common runtime signals



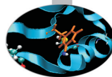
Signal name	OS signal name	Description
Floating point exception	SIGFPE	The program attempted an arithmetic operation with values that do not make sense
Segmentation fault	SIGSEGV	The program accessed memory incorrectly
Aborted	SIGABRT	Generated by the runtime library of the program or a library it uses, after having detecting a failure condition.

FPE example



```
1  main()
2  {
3    int a = 1.;
4    int b = 0.;
5    int c = a/b;
6  }
```

FPE example



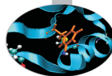
```
1  main()
2  {
3    int a = 1.;
4    int b = 0.;
5    int c = a/b;
6  }
```

```
[ruggiero@matrix1 ~]$ gcc fpe_example.c
```

```
[ruggiero@matrix1 ~]$ ./a.out
```

```
Floating exception
```

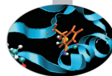
SEGV example



```
1  main()
2  {
3  int array[5]; int i;
4       for(i = 0; i < 255; i++) {
5            array[i] = 10;}
6      return 0;
7  }
```

CINECA SCAI
SuperComputing Applications and Innovation

SEGV example



```
1  main()
2  {
3  int array[5]; int i;
4       for(i = 0; i < 255; i++) {
5               array[i] = 10;}
6      return 0;
7  }
```

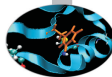
```
[ruggiero@matrix1 ~]$ gcc segv_example.c
```

```
[ruggiero@matrix1 ~]$ ./a.out
```

```
Segmentation fault
```

CINECA

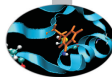
ABORT example



```
1  #include <assert.h>
2  main()
3  {
4    int i=0;
5    assert(i!=0);
6  }
```

CINECA SCAI SuperComputing Applications and Innovation

ABORT example



```c
1  #include <assert.h>
2  main()
3  {
4    int i=0;
5    assert(i!=0);
6  }
```

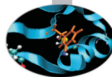
```
[ruggiero@matrix1 ~]$ gcc abort_example.c
```

```
[ruggiero@matrix1 ~]$ ./a.out
```

```
a.out: abort_example.c:5: main: Assertion `i!=0' failed.
Abort
```

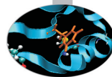
CINECA

Common runtime errors



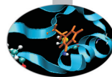
- Allocation Deallocation errors (AD).
- Array conformance errors (AC).
- Array Index out of Bound (AIOB).
- Language specific errors (LS).
- Floating Point errors (FP).
- Input Output errors (IO).
- Memory leaks (ML).
- Pointer errors (PE).
- String errors (SE).
- Subprogram call errors (SCE).
- Uninitialized Variables (UV).

Useful link



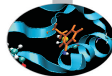
- Iowa State University's High Performance Computing Group
- Run Time Error Detection Test Suites for Fortran, C, and C++
- http://rted.public.iastate.edu

Grading Methodology: score



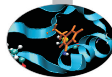
- 0.0: is given when the error was not detected.
- 1.0: is given for error messages with the correct error name.
- 2.0: is given for error messages with the correct error name and line number where the error occurred but not the file name where the error occurred.
- 3.0: is given for error messages with the correct error name, line number and the name of the file where the error occurred.
- 4.0: s given for error messages which contain the information for a score of 3.0 but less information than needed for a score of 5.0 .
- 5.0: is given in all cases when the error message contains all the information needed for the quick fixing of the error.

Grading Methodology : an example



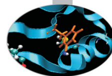
```
!*****************************************************************************
!  copyright (c) 2005 Iowa State University, Glenn Luecke, James Coyle,
!  James Hoekstra, Marina Kraeva, Olga Taborskaia, Andre Wehe, Ying Xu,
!  and Ziyu Zhang, All rights reserved.
!  Licensed under the Educational Community License version 1.0.
!  See the full agreement at http://rted.public.iastate.edu/ .
!*****************************************************************************
!*****************************************************************************
!
!     Name of the test:  F_H_1_1_b.f90
!
!     Summary:           allocation/deallocation error
!
!     Test description:  deallocation twice
!                        for allocatable array in a subroutine
!                        contains in a main program
!
!     Support files:     Not needed
!
!     Env. requirements: Not needed
!
!     Keywords:          deallocation error
!                        subroutine contains in a main program
```

Grading Methodology: an example



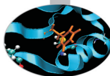
```
!
!     Last modified:     1/17/2005
!
!     Programmer:        Ying Xu, Iowa State Univ.
!***************************************************************************

program tests
  implicit none
  integer :: n=10, m=20
  double precision :: var

  call sub(n,m,var)
  print *,var
  contains
    subroutine sub(n,m,var)
      integer, intent(in) :: n,m
      double precision, intent(inout) :: var
      double precision, allocatable :: arr(:,:)    ! DECLARE
```

Grading Methodology: an example



```
      integer :: i,j
      allocate(arr(1:n,1:m))
      do i=1,n
         do j=1,m
            arr(i,j) = dble(i*j)
         enddo
      end do
      var = arr(n,m)
      deallocate(arr)
      deallocate(arr)   ! deallocate second time here. ERROR
      return
    end subroutine sub
end program tests
```

Grading Methodology: an example



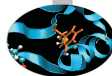
Real message (grade 1.0)

```
Fortran runtime error: Internal: Attempt to DEALLOCATE
unallocated memory.
```

Ideal message (grade 5.0)

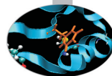
```
ERROR: unallocated array
At line 52 column 17 of subprogram 'sub'
in file 'F_H_1_1_b.f90', the argument
'arr' in the DEALLOCATE statement is an
unallocated array.  The variable is declared
in line 41 in subprogram 'sub' in file 'F_H_1_1_b.f90'.
```

Fortran Results



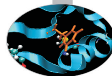
Compiler	AC	A D	AIOB	LS	FP	IO
gcc-4.3.2	1	0.981481	3.40025	2.88235	0	2.33333
gcc-4.3.2	1	1.38889	3.40025	2.88235	0	2.33333
gcc-4.4.3	1	1.38889	3.40025	2.88235	0	2.33333
gcc-4.6.3	1	1.27778	0.969504	0.94117	0	2.33333
gcc-4.7.0	1	1.38889	0.969504	0.94117	0.714286	2.33333
g95.4.0.3	0.421053	1.22222	3.60864	3.82353	0.571428	2.66667
intel-10.1.021	0.421053	1.42593	3.45362	2.82353	0.571428	2.11111
intel-11.0.074	0.421053	1.68519	3.446	2.82353	0.571428	2.11111
intel-12.0.2	0.421053	1.62963	3.44727	2.82353	0.571428	2.11111
open64.4.2.3	3	0.888889	2.63405	3	0	1
pgi-7.2-5	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-8.0-1	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-10.3	0.421053	0.388889	3.8526	3.82353	0	2.44444
pgi-11.8	0.421053	0.388889	3.8526	3.82353	0	2.44444
sun.12.1	3	2.77778	3.00381	3	2	2.16667
sun.12.1+bcheck	3	2.77778	3.03431	3	0.285714	2.16667

Fortran Results



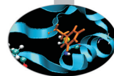
Compiler	ML	PE	SE	SCE	UV
gcc-4.3.2	0	3.49609	3.25	0	0.0159236
gcc-4.3.2	0	3.49609	3.25	0	0.0286624
gcc-4.4.3	0	3.49609	3.25	0	0.0286624
gcc-4.6.3	0	1	3.25	0.166667	0.22293
gcc-4.7.0	0	1	3.25	0.166667	0.130573
g95.4.0.3	1	3	3.43333	0	0.0159236
intel-10.1.021	0	3.5625	0	0.166667	0.286624
intel-11.0.074	0	3.55469	0	0.166667	0.299363
intel-12.0.2	0	3.55469	0	0.166667	0.292994
open64.4.2.3	0	3.00391	0	0	0.0127389
pgi-7.2-5	0	4	0	0	0.022293
pgi-8.0-1	0	4	0	0	0.022293
pgi-10.3	0	4	0	0	0.0254777
pgi-11.8	0	4	0	0	0.0127389
sun.12.1	0	3.03125	3	0	0.022293
sun.12.1+bcheck	1.25	3.03125	3	1	0.640127

C Results



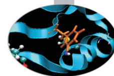
Compiler	AD	AIoB	LS	FP	IO
gcc-4.3.2	0.44	0.00925926	0.0416667	0.2	0.0666667
gcc-4.4.3	0.44	0.00925926	0.0416667	0.2	0.0666667
gcc-4.6.3	0.44	0.00925926	0.0416667	0.2	0.2
gcc-4.7.0	0.44	0.00925926	0.0416667	0.2	0.2
intel-10.1.021	0.52	0.00925926	0.0416667	0	0.2
intel-11.0.074	0.52	0.00925926	0.0416667	0	0.2
intel-12.0.2	0.44	0.00925926	0.0416667	0	0.2
open64-4.2.3	0.52	0.00925926	0.0416667	0	0.2
pgi-7.2-5	0.44	0.00925926	0.0416667	0	0.2
pgi-8.0-1	0.44	0.00925926	0.0416667	0	0.2
pgi-10.3	0.44	0.00925926	0.0416667	0	0.2
pgi-11.8	0.44	0.00925926	0.0416667	0	0.2
sun-12.1	0.44	0.00925926	0	0	0.2
sun-12.1+bcheck	0.16	0	0	0	0

C Results



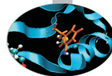
Compiler	ML	PE	SE	SCE	UV
gcc-4.3.2	0.0166667	0.0166667	0.05	0	0
gcc-4.4.3	0.0166667	0.0166667	0.05	0	0
gcc-4.6.3	0.0166667	0.0166667	0.05	0	0
gcc-4.7.0	0.0166667	0.0166667	0.05	0	0
intel-10.1.021	0.0666667	0.0166667	0.05	0	0
intel-11.0.074	0.0666667	0.0166667	0.05	0	0
intel-12.0.2	0.0666667	0.0166667	0.05	0	0
open64-4.2.3	0.0666667	0.0166667	0.05	0	0
pgi-7.2-5	0.0666667	0.0166667	0.05	0	0
pgi-8.0-1	0.0666667	0.0166667	0.05	0	0
pgi-10.3	0.0666667	0.0166667	0.05	0	0
pgi-11.8	0.0666667	0.0166667	0.05	0	0
sun-12.1	0.0666667	0.0166667	0.05	0	0
sun-12.1+bcheck	1.13333	0.025	0	0	0

C++ Results



Compiler	AD	AIoB	FP	IO	ML	PE	SE	UV
gcc-4.3.2	0.44	0.00925926	0	0	0.0666667	0.0166667	0.05	0
gcc-4.4.3	0.44	0.00925926	0	0	0.0666667	0.0166667	0.05	0
gcc-4.6.3	0.44	0.00925926	0	0.2	0.0666667	0.0166667	0.05	0
gcc-4.7.0	0.44	0.00925926	0	0.2	0.0666667	0.0166667	0.05	0
intel-10.1.021	0.330275	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
intel-11.0.074	0.330275	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
intel-12.0.2	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
open64-4.2.3	0.330275	0.00903614	0	0	0.047619	0.0254777	0.05	0
pgi-7.2.5	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-8.0.1	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-10.3	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
pgi-11.8	0.311927	0.00903614	0	0.0714286	0.047619	0.0254777	0.05	0
sun-12.1	0.311927	0.00903614	0	0	0.047619	0.0254777	0.05	0
sun-12.1+bcheck	0.247706	0.0150602	0	0	1.16667	0.0191083	0	0

Grading Methodology: Used options



Fortran

gcc	-frange-check -O0 -fbounds-check -g -ffpe-trap=invalid,zero,overflow -fdiagnostics-show-location=every-line
g95	-O0 -fbounds-check -g -ftrace=full
intel	-O0 -C -g -traceback -ftrapuv -check
open64	-C -g -O0
pgi	-C -g -Mchkptr -O0
sun	-g -C -O0 -xs -ftrap=%all -fnonstd -xcheck=%all

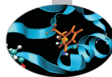
C

gcc	-O0 -g -fbounds-check -ftrapv
intel	-O0 -C -g -traceback
open64	-g -C -O0
pgi	-g -C -Mchkptr -O0
sun	-g -C -O0 -xs -ftrap=%all -fnonstd -xcheck=%all

C++

gcc	-O0 -g -fbounds-check -ftrapv
intel	-O0 -C -g -traceback
open64	-g -C -O0
pgi	-g -C -Mchkptr -O0
sun	-g -C -O0 -xs -ftrap=%all -fnonstd -xcheck=%all

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

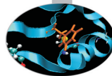
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

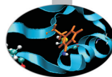
Floating Point Computing

Fixing memory problems



- Memory leaks are data structures that are allocated at runtime, but not deallocated once they are no longer needed in the program.
- Incorrect use of the memory management is associated with incorrect calls to the memory management: freeing a block of memory more than once, accessing memory after freeing...
- Buffer overruns are bugs where memory outside of the allocated boundaries is overwritten, or corrupted.
- Uninitialized memory bugs: reading uninitialized memory.

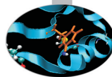
Valgrind



- Open Source Software, available on Linux for x86 and PowerPc processors.
- Interprets the object code, not needed to modify object files or executable, non require special compiler flags, recompiling, or relinking the program.
- Command is simply added at the shell command line.
- No program source is required (black-box analysis).

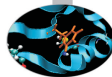
www.valgrind.org

Valgrind:tools



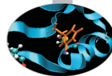
- Memcheck: a memory checker.
- Callgrind: a runtime profiler.
- Cachegrind: a cache profiler.
- Helgrind: find race conditions.
- Massif: a memory profiler.

Why should use I use Valgrind?



- Valgrind will tell you about tough to find bugs.
- Valgrind is very through.
- You may be tempted to think that Valgrind is too picky, since your program may seem to work even when valgrind complains. It is users' experience that fixing ALL Valgrind complaints will save you time in the long run.

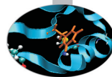
Why should use I use Valgrind?



- Valgrind will tell you about tough to find bugs.
- Valgrind is very through.
- You may be tempted to think that Valgrind is too picky, since your program may seem to work even when valgrind complains. It is users' experience that fixing ALL Valgrind complaints will save you time in the long run.

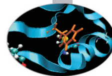
But...
Valgrind is kind-of like a virtual x86 interpeter. So your program will run 10 to 30 times slower than normal.
Valgrind won't check static arrays.

Use of uninitialized memory:test1.c



- Local Variables that have not been initialized.
- The contents of malloc's blocks, before writing there.

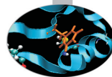
Use of uninitialized memory:test1.c



- Local Variables that have not been initialized.
- The contents of malloc's blocks, before writing there.

```
1  #include <stdlib.h>
2  int main()
3  {
4        int p,t,b[10];
5        if(p==5)
6            t=p+1;
7            b[p]=100;
8        return 0;
9  }
```

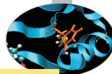
Use of uninitialized memory:test1.c



- Local Variables that have not been initialized.
- The contents of malloc's blocks, before writing there.

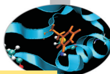
```
1  #include <stdlib.h>
2  int main()
3  {
4        int p,t,b[10];
5        if(p==5)      ERROR
6            t=p+1;
7            b[p]=100;  ERROR
8        return 0;
9  }
```

Valgrind output



```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t1
```

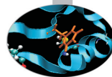
Valgrind output



ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t1

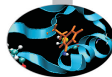
```
==7879== Memcheck, a memory error detector.
         ....
==7879== Conditional jump or move depends on uninitialised value(s)
==7879==    at 0x8048399: main (test1.c:5)
==7879==
==7879== Use of uninitialised value of size 4
==7879==    at 0x80483A7: main (test1.c:7)
==7879==
==7879== Invalid write of size 4
==7879==    at 0x80483A7: main (test1.c:7)
==7879==  Address 0xCEF8FE44 is not stack'd, malloc'd or (recently) free'd
==7879==
==7879== Process terminating with default action of signal 11 (SIGSEGV)
==7879==  Access not within mapped region at address 0xCEF8FE44
==7879==    at 0x80483A7: main (test1.c:7)
==7879==
==7879== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 3 from 1)
==7879== malloc/free: in use at exit: 0 bytes in 0 blocks.
==7879== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==7879== For counts of detected errors, rerun with: -v
==7879== All heap blocks were freed -- no leaks are possible.
Segmentation fault
```

Illegal read/write test2.c



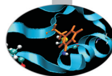
```
1   #include <stdlib.h>
2   int main()
3   {
4         int *p,i,a;
5         p=malloc(10*sizeof(int));
6             p[11]=1;
7             a=p[11];
8             free(p);
9         return 0;
10  }
```

Illegal read/write test2.c



```
1  #include <stdlib.h>
2  int main()
3  {
4       int *p,i,a;
5       p=malloc(10*sizeof(int));
6           p[11]=1; ERROR
7           a=p[11]; ERROR
8           free(p);
9       return 0;
10 }
```

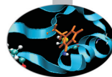
Illegal read/write: Valgrind output



ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t2

```
       .....
==8081== Invalid write of size 4
==8081==    at 0x804840A: main (test2.c:6)
==8081==  Address 0x417B054 is 4 bytes after a block of size 40 alloc'd
==8081==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8081==    by 0x8048400: main (test2.c:5)
==8081==
==8081== Invalid read of size 4
==8081==    at 0x8048416: main (test2.c:7)
==8081==  Address 0x417B054 is 4 bytes after a block of size 40 alloc'd
==8081==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8081==    by 0x8048400: main (test2.c:5)
==8081==
==8081== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 1)
==8081== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8081== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==8081== For counts of detected errors, rerun with: -v
==8081== All heap blocks were freed -- no leaks are possible.
```

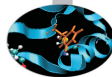
Invalid free:test3.c



```
1   #include <stdlib.h>
2   int main()
3   {
4         int *p,i;
5         p=malloc(10*sizeof(int));
6               for(i=0;i<10;i++)
7                   p[i]=i;
8               free(p);
9               free(p);
10        return 0;
11  }
```

CINECA SCAI
SuperComputing Applications and Innovation

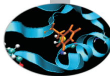
Invalid free:test3.c



```
1   #include <stdlib.h>
2   int main()
3   {
4        int *p,i;
5        p=malloc(10*sizeof(int));
6            for(i=0;i<10;i++)
7               p[i]=i;
8            free(p);
9            free(p);     ERROR
10       return 0;
11  }
```

CINECA

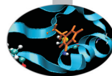
Invalid free: Valgrind output



```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t3
```

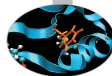
```
        .....
==8208== Invalid free() / delete / delete[]
==8208==    at 0x40231CF: free (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8208==    by 0x804843C: main (test3.c:9)
==8208==  Address 0x417B028 is 0 bytes inside a block of size 40 free'd
==8208==    at 0x40231CF: free (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8208==    by 0x8048431: main (test3.c:8)
==8208==
==8208== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==8208== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8208== malloc/free: 1 allocs, 2 frees, 40 bytes allocated.
==8208== For counts of detected errors, rerun with: -v
==8208== All heap blocks were freed -- no leaks are possible.
```

Mismatched use of functions:test4.cpp



- If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free.**
- If allocated with **new[]**, you must dealloacate with **delete[].**
- If allocated with **new**, you must deallocate with **delete.**

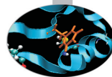
Mismatched use of functions:test4.cpp



- If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- If allocated with **new[]**, you must dealloacate with **delete[]**.
- If allocated with **new**, you must deallocate with **delete**.

```
1   #include <stdlib.h>
2   int main()
3   {
4        int *p,i;
5        p=(int*)malloc(10*sizeof(int));
6            for(i=0;i<10;i++)
7               p[i]=i;
8            delete(p);
9        return 0;
10  }
```

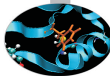
Mismatched use of functions:test4.cpp



- If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- If allocated with **new[]**, you must dealloacate with **delete[]**.
- If allocated with **new**, you must deallocate with **delete**.

```
1  #include <stdlib.h>
2  int main()
3  {
4       int *p,i;
5       p=(int*)malloc(10*sizeof(int));
6           for(i=0;i<10;i++)
7              p[i]=i;
8           delete(p);    ERROR
9       return 0;
10 }
```

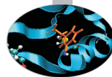
Mismatched use of functions: Valgrind output



```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t4
```

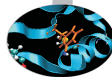
```
        .....
==8330== Mismatched free() / delete / delete []
==8330== at 0x4022EE6: operator delete(void*) (in /usr/lib/valgrind/x86-linux/vgpreload_memche
==8330==by 0x80484F1: main (test4.c:8)
==8330==Address 0x4292028 is 0 bytes inside a block of size 40 alloc'd
==8330==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8330==by 0x80484C0: main (test4.c:5)
==8330==
==8330==ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==8330==malloc/free: in use at exit: 0 bytes in 0 blocks.
==8330==malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==8330==For counts of detected errors, rerun with: -v
==8330==All heap blocks were freed -- no leaks are possible.
```

Invalid system call parameter:test5.c



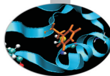
```
1  #include <stdlib.h>
2  #include <unistd.h>
3  int main()
4  {
5       int *p;
6       p=malloc(10);
7        read(0,p,100);
8           free(p);
9       return 0;
10 }
```

Invalid system call parameter:test5.c



```
1   #include <stdlib.h>
2   #include <unistd.h>
3   int main()
4   {
5         int *p;
6         p=malloc(10);
7          read(0,p,100); ERROR
8             free(p);
9         return 0;
10  }
```

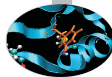
Invalid system call parameter: Valgrind output



ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t5

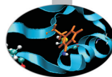
```
...
==18007== Syscall param read(buf) points to unaddressable byte(s)
==18007==    at 0x4EEC240: __read_nocancel (in /lib64/libc-2.5.so)
==18007==    by 0x40056F: main (test5.c:7)
==18007==  Address 0x517d04a is 0 bytes after a block of size 10 alloc'd
==18007==    at 0x4C21168: malloc (vg_replace_malloc.c:236)
==18007==    by 0x400555: main (test5.c:6)
...
```

Memory leak detection:test6.c



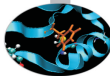
```
1  #include <stdlib.h>
2           int main()
3           {
4               int *p,i;
5               p=malloc(5*sizeof(int));
6               for(i=0; i<5;i++)
7                   p[i]=i;
8
9                   return 0;
10          }
```

Memory leak detection:test6.c



```
1  #include <stdlib.h>
2          int main()
3          {
4              int *p,i;
5              p=malloc(5*sizeof(int));
6              for(i=0; i<5;i++)
7                  p[i]=i;
8              free(p);
9                  return 0;
10         }
```

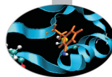
Memory leak detection: Valgrind output



```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t6
```

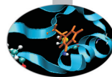
```
    .....
==8237== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 3 from 1)
    ==8237== malloc/free: in use at exit: 20 bytes in 1 blocks.
    ==8237== malloc/free: 1 allocs, 0 frees, 20 bytes allocated.
    ==8237== For counts of detected errors, rerun with: -v
    ==8237== searching for pointers to 1 not-freed blocks.
    ==8237== checked 65,900 bytes.
    ==8237==
    ==8237== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
    ==8237==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8237==    by 0x80483D0: main (test6.c:5)
    ==8237==
    ==8237== LEAK SUMMARY:
    ==8237==    definitely lost: 20 bytes in 1 blocks.
    ==8237==      possibly lost: 0 bytes in 0 blocks.
    ==8237==    still reachable: 0 bytes in 0 blocks.
    ==8237==         suppressed: 0 bytes in 0 blocks.
```

What won’t Valgrind find?



```
1   int main()
2   {
3       char x[10];
4       x[11]='a';
5   }
```

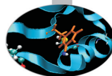
What won’t Valgrind find?



```
1   int main()
2   {
3       char x[10];
4       x[11]='a';
5   }
```

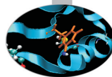
- Valgrind doesn’t perform bound checking on static arrays (allocated on stack).
- Solution for testing purposes is simply to change static arrays into dinamically allocated memory taken from the heap, where you will get bounds-checking, though this could be a message of unfreed memory.

sum.c: source



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main (int argc, char* argv[])  {
4      const int size=10;
5      int n, sum=0;
6      int* A = (int*)malloc( sizeof(int)*size);
7
8      for(n=size; n>0; n--)
9          A[n] = n;
10     for(n=0; n<size; n++)
11         sum+=A[n];
12     printf("sum=%d\n", sum);
13     return 0;
14 }
```

sum.c: compilation and run

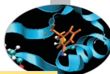


```
ruggiero@shiva:~> gcc -O0 -g  -fbounds-check -ftrapv sum.c
```

```
ruggiero@shiva:~> ./a.out
```

```
sum=45
```

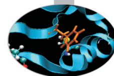
Valgrind:example



```
ruggiero@shiva:~> valgrind --leak-check=full --tool=memcheck ./a.out
```

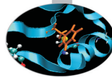
```
==21579== Memcheck, a memory error detector.
...
==21791==Invalid write of size 4
==21791==at 0x804842A: main (sum.c:9)
==21791==Address 0x417B050 is 0 bytes after a block of size 40 alloc'd
==21791==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==21791==by 0x8048410: main (sum.c:6)
==21791==Use of uninitialised value of size 4
==21791== at 0x408685B: _itoa_word (in /lib/libc-2.5.so)
==21791==by 0x408A581: vfprintf (in /lib/libc-2.5.so)
==21791==by 0x4090572: printf (in /lib/libc-2.5.so)
==21791==by 0x804846B: main (sum.c:12)
==21791==
==21791==Conditional jump or move depends on uninitialised value(s)
==21791==at 0x4086863: _itoa_word (in /lib/libc-2.5.so)
==21791==by 0x408A581: vfprintf (in /lib/libc-2.5.so)
==21791==by 0x4090572: printf (in /lib/libc-2.5.so)
==21791==by 0x804846B: main (sum.c:12)
==21791==40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==21791==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==21791==by 0x8048410: main (sum.c:6)
==21791==
```

outbc.c



```
1   #include <stdio.h>
2   #include<stdlib.h>
3           int main (void)
4           {
5               int i;
6               int *a = (int*) malloc( 9*sizeof(int));
7
8               for ( i=0; i<=9; ++i){
9                  a[i] = i;
10                  printf ("%d\n ", a[i]);
11              }
12
13              free(a);
14              return 0;
15          }
```

outbc.c: compilation and run

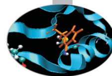


```
ruggiero@shiva:~> icc –C –g outbc.c
```

```
ruggiero@shiva:~> ./a.out
```

```
0
1
2
3
4
5
6
7
8
9
```

outbc.c: compilation and run

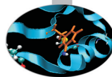


```
ruggiero@shiva:~> pgcc -C -g outbc.c
```

```
ruggiero@shiva:~> ./a.out
```

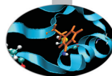
```
0
1
2
3
4
5
6
7
8
9
```

Electric Fence



- Electric Fence (efence) stops your program on the exact instruction that overruns (or underruns) a malloc() memory buffer.
- GDB will then display the source-code line that causes the bug.
- It works by using the virtual-memory hardware to create a red-zone at the border of each buffer - touch that, and your program stops.
- Catch all of those formerly impossible-to-catch overrun bugs that have been bothering you for years.

Electric Fence

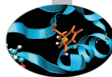


```
ruggiero@shiva:~> icc -g outbc.c libefence.a -o outbc -lpthread
```

```
ruggiero@shiva:~> ./outbc
```

```
0
1
2
3
4
5
6
7
8
Segmentation fault
```

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

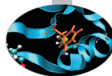
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

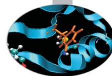
Floating Point Computing

My program fails!



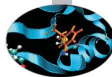
- Erroneous program results.
- Execution deadlock.
- Run-time crashes.

The ideal debugging process



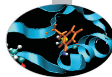
- Find origins.
  - Identify test case(s) that reliably show existence of fault (when possible). Duplicate the bug.
- Isolate the origins of infection.
  - Correlate incorrect behaviour with program logic/code error.
- Correct.
  - Fixing the error, not just a symptom of it.
- Verify.
  - Where there is one bug, there is likely to be another.
  - The probability of the fix being correct is not 100 percent.
  - The probability of the fix being correct drops as the size of the program increases.
  - Beware of the possibility that an error correction creates a new error.

Bugs that can’t be duplicated



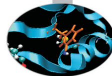
- Dangling pointers.
- Initializations errors.
- Poorly syinchronized threads.
- Broken hardware.

Isolate the origins of infection



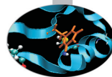
- Divide and conqueror.
- Change one thing at time.
- Determine what you changed since the last time it worked.
- Write down what you did, in what order, and what happened as a result.
- Correlate the events.

Why is debugging so difficult?



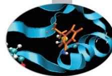
- The symptom and the cause may be geographically remote.
- The symptom may disappear (temporarily) when another error is corrected.
- The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
- It may be difficult to accurately reproduce input conditions (e.g, a real time application in which input ordering is indeterminate).
- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Eight elements



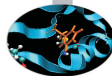
- Reproducibility.
- Reduction.
- Deduction.
- Experimentation.
- Experience.
- Tenacity.
- Good tools.
- A bit of luck.

Why use a Debugger?



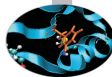
- No need for precognition of what the error might be.
- Flexible.
  - Allows for ”live” error checking (no need to re–write and re–compile when you realize a certain type of error may be occuring).
- Dynamic.
  - Execution Control Stop execution on specified conditions: breakpoints
  - Interpretation Step-wise execution code
  - State Inspection Observe value of variables and stack
  - State Change Change the state of the stopped program.

Why people don't use debuggers?



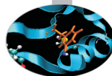
- With simple errors, may not want to bother with starting up the debugger environment.
  - Obvious error.
  - Simple to check using prints/asserts.
- Hard-to-use debugger environment.
- Error occurs in optimized code.
- Changes execution of program
  (error doesn't occur while running debugger).

Why don't use print?



- Cluttered code.
- Cluttered output.
- Slowdown.
- Time consuming.
- And can be misleading.
  - Moves things around in memory, changes execution timing, etc.
  - Common for bugs to hide when print statements are added, and reappear when they're removed.

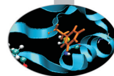
Remember



Finding bug is a process of confirming the many things you believe are true, until you find one which is not true.

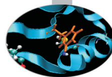
- You believe that at a certain point in your source file, a certain variable has a certain value.
- You believe that in given if-then-else statment, the ”else” part is the one that is excuted.
- You believe that when you call a certain function, the function receives its parameters correctly.
- You believe ...

Strategy



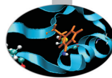
- Observe the failure.
- Invent a hypothesis.
- Use the hypothesis to make predictions.
- Test hypothesis by experiments.

Outline



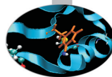
- Introduzione
- Architetture
- La cache e il sistema di memoria
- Pipeline
- Profilers
- Compilatori e ottimizzazione
- Librerie scientifiche
- Floating Point Computing

What is gdb?



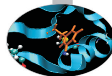
- The GNU Project debugger, is an open-source debugger.
- Protected by GNU General Public License (GPL).
- Runs on many Unix-like systems.
- Was first written by Richard Stallmann in 1986 as part of his GNU System.
- Is an Workstation Application Code extremely powerful all-purpose debugger.
- Its text-based user interface can be used to debug programs written in C, C++, Pascal, Fortran, and several other languages, including the assembly language for every micro-processor that GNU supports.
- www.gnu.org/software/gdb

prime-numbers finding program



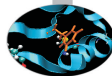
- Print a list of all primes which are less than or equal to the user-supplied upper bound *UpperBound*.
- See if J divides $K \leq$ *UpperBound*, for all values J which are
  - themselves prime (no need to try J if it is nonprime)
  - less than or equal to sqrt(K) (if K has a divisor larger than this square root, it must also have a smaller one, so no need to check for larger ones).
- *Prime[I]* will be 1 if I is prime, 0 otherwise.

Main.c



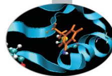
```
1   #include <stdio.h>
2   #define MaxPrimes 50
3   int Prime[MaxPrimes],UpperBound;
4   int main()
5   {
6       int N;
7       printf("enter upper bound\n");
8       scanf("%d",UpperBound);
9       Prime[2] = 1;
10      for (N = 3; N <= UpperBound; N += 2)
11          CheckPrime(N);
12      if (Prime[N]) printf("%d is a prime\n",N);
13      return 0;
14  }
```

CheckPrime.c



```
1   #define MaxPrimes 50
2   extern int Prime[MaxPrimes];
3   void CheckPrime(int K)
4   {
5       int J; J = 2;
6       while (1)  {
7           if (Prime[J] == 1)
8               if (K % J == 0)  {
9                   Prime[K] = 0;
10                  return;
11              }
12              J++;
13      }
14      Prime[K] = 1;
15  }
```

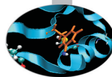
Compilation and run



```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

SuperComputing Applications and Innovation

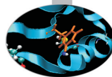
Compilation and run



```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

Compilation and run

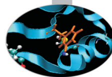


```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
```

Compilation and run



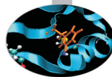
```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
```

20

Compilation and run

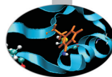


```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

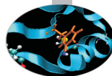
```
enter upper bound
20
Segmentation fault
```

Compilation options for gdb



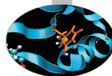
- You will need to compile your program with the appropriate flag to enable generation of symbolic debug information, the -g option is used for this.
- Don't compile your program with optimization flags while you are debugging it.
  Compiler optimizations can "rewrite" your program and produce machine code that doesn't necessarily match your source code.
  Compiler optimizations may lead to:
  - Misleading debugger behaviour.
    - Some variables you declared may not exist at all
    - some statements may execute in different places because they were moved out of loops
  - Obscure the bug.

Lower optimization level



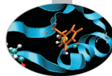
- When your program has crashed, disable or lower optimization to see if the bug disappears.
  (optimization levels are not comparable between compilers, not even -O0).
- If the bug persists $\Longrightarrow$ you can be quite sure there's something wrong in your application.
- If the bug disappears,without a serious performance penalty $\Longrightarrow$ send the bug to your computing center and continue your simulations.

Lower optimization level



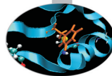
- When your program has crashed, disable or lower optimization to see if the bug disappears.
  (optimization levels are not comparable between compilers, not even -O0).
- If the bug persists $\Longrightarrow$ you can be quite sure there's something wrong in your application.
- If the bug disappears,without a serious performance penalty $\Longrightarrow$ send the bug to your computing center and continue your simulations.
- But your program may still contain a bug that simply doesn't show up at lower optimization $\Longrightarrow$ have some checks to verify the correctness of your code.

Starting gdb



```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o trova_primi
```

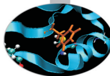
Starting gdb



```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o trova_primi
```

```
<ruggiero@matrix2 ~>gdb trova_primi
```

Starting gdb



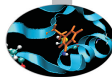
```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o trova_primi
```

```
<ruggiero@matrix2 ~>gdb trova_primi
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

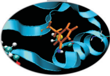
```
(gdb)
```

gdb: Basic commands



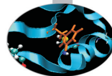
- run (r): start debugged program.
- help (h): print list of commands.
- she: execute the rest of the line as a shell command.
- where, backtrace (bt): print a backtrace of entire stack.
- kill (k): kill the child process in which program is running under gdb.
- list (l) linenum: print lines centered around line number lineum in the current source file.
- quit(q): exit gdb.

prime-number finding program



```
(gdb) r
```

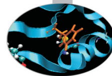
prime-number finding program



```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound
```

prime-number finding program

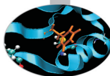


```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program



```
(gdb) r
```

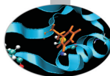
```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

prime-number finding program



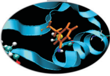
```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

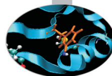
```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

prime-number finding program



```
(gdb) list Main.c:8
```

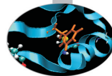
prime-number finding program



```
(gdb) list Main.c:8
```

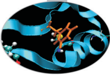
```
3    int Prime[MaxPrimes],UpperBound;
5      main()
6      {  int N;
7         printf("enter upper bound\n");
8         scanf("%d",UpperBound);
9         Prime[2] = 1;
10        for (N = 3; N <= UpperBound; N += 2)
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
```

Main.c :new version



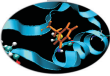
```
1   #include <stdio.h>
2   #define MaxPrimes 50
3   int Prime[MaxPrimes],UpperBound;
4   int main()
5   {
6          int N;
7          printf("enter upper bound\n");
8          scanf("%d",  UpperBound);
9          Prime[2] = 1;
10         for (N = 3; N <= UpperBound; N += 2)
11            CheckPrime(N);
12            if (Prime[N]) printf("%d is a prime\n",N);
13         return 0;
14  }
```

Main.c :new version



```
1   #include <stdio.h>
2   #define MaxPrimes 50
3   int Prime[MaxPrimes],UpperBound;
4   int main()
5   {
6         int N;
7         printf("enter upper bound\n");
8         scanf("%d", &UpperBound);
9         Prime[2] = 1;
10        for (N = 3; N <= UpperBound; N += 2)
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
13        return 0;
14  }
```

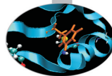
Main.c :new version



```
1  #include <stdio.h>
2  #define MaxPrimes 50
3  int Prime[MaxPrimes],UpperBound;
4  int main()
5  {
6        int N;
7        printf("enter upper bound\n");
8        scanf("%d", &UpperBound);
9        Prime[2] = 1;
10       for (N = 3; N <= UpperBound; N += 2)
11          CheckPrime(N);
12          if (Prime[N]) printf("%d is a prime\n",N);
13       return 0;
14 }
```

In other shell COMPILATION

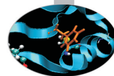
prime-number finding program



```
(gdb) kill
```

Kill the program being debugged? (y or n)

prime-number finding program



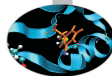
```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

prime-number finding program



```
(gdb) kill
```

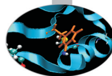
Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program



```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

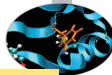
```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7                if (Prime[J] == 1)
```

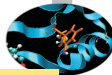
prime-number finding program



```
(gdb) p J
```

SuperComputing Applications and Innovation

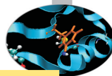
prime-number finding program



```
(gdb) p J
```

```
$1 = 1008
```

prime-number finding program

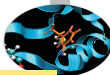


```
(gdb) p J
```

```
$1 = 1008
```

```
gdb l CheckPrime.c:7
```

prime-number finding program



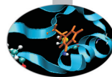
```
(gdb) p J
```

```
$1 = 1008
```

```
gdb l CheckPrime.c:7
```

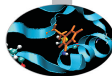
```
2        extern int Prime[MaxPrimes];
3          CheckPrime(int K)
4          {
5              int J; J = 2;
6              while (1)  {
7                 if (Prime[J] == 1)
8                    if (K % J == 0)   {
9                       Prime[K] = 0;
10                      return;
11                   }
```

CheckPrime.c:new version



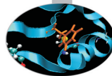
```
1   #define MaxPrimes 50
2   extern int Prime[MaxPrimes];
3   void CheckPrime(int K)
4       {
5         int J; J = 2;
6          while (1){
7              if (Prime[J] == 1)
8                 if (K % J == 0)   {
9                    Prime[K] = 0;
10                   return;
11                }
12         J++;
13         }
14         Prime[K] = 1;
15      }
```

CheckPrime.c:new version



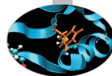
```
1   #define MaxPrimes 50
2   extern int Prime[MaxPrimes];
3   void CheckPrime(int K)
4      {
5        int J;
6            for (J = 2; J*J <= K; J++)
7            if (Prime[J] == 1)
8               if (K % J == 0)   {
9                  Prime[K] = 0;
10                 return;
11              }
12
13
14        Prime[K] = 1;
15     }
```

prime-number finding program



In other shell COMPILATION

prime-number finding program

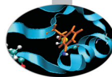


In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n)

prime-number finding program



In other shell COMPILATION

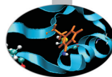
```
(gdb)
```

Kill the program being debugged? (y or n)   y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

prime-number finding program



In other shell COMPILATION

```
(gdb)
```

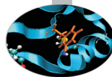
Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program



In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n)   y

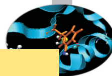
```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

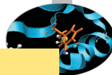
```
Program exited normally.
```

gdb commands



```
(gdb) help break
```

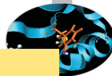
gdb commands



(gdb) help break

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

gdb commands

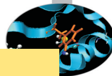


(gdb) help break

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

(gdb) help display

gdb commands



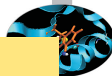
```
(gdb) help break
```

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

```
(gdb) help display
```

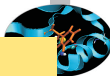
```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

gdb commands



```
(gdb) help next
```

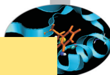
gdb commands



```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

gdb commands

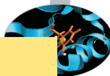


```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help step
```

gdb commands



```
(gdb) help next
```

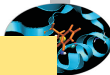
```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help step
```

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) break Main.c:1
```

gdb commands



(gdb) help next

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

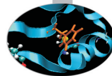
(gdb) help step

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

(gdb) break Main.c:1

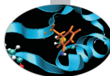
```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```

prime-number finding program



```
(gdb) r
```

prime-number finding program

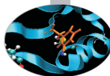


```
(gdb) r
```

```
Starting program: trova_primi
Failed to read a valid object file image from memory.

Breakpoint 1, main () at Main.c:6
6           {   int N;
```

```
(gdb) next
```

prime-number finding program



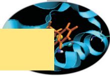
```
(gdb) r
```

```
Starting program: trova_primi
Failed to read a valid object file image from memory.

Breakpoint 1, main () at Main.c:6
6           {   int N;
```

```
(gdb) next
```

```
main () at Main.c:7
7               printf("enter upper bound\n");
```

prime-number finding program



```
(gdb) next
```

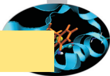
prime-number finding program

```
(gdb) next
```

```
8               scanf("%d",&UpperBound);
```

```
(gdb) next
```

prime-number finding program



```
(gdb) next
```

```
8                scanf("%d",&UpperBound);
```

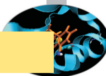
```
(gdb) next
```

20

prime-number finding program

```
(gdb) next

8           scanf("%d",&UpperBound);

(gdb) next
20

9           Prime[2] = 1;

(gdb) next
```

prime-number finding program



```
(gdb) next

8                scanf("%d",&UpperBound);

(gdb) next
20

9                Prime[2] = 1;

(gdb) next

10               for (N = 3; N <= UpperBound; N += 2)

(gdb) next
```

prime-number finding program

```
(gdb) next

8              scanf("%d",&UpperBound);

(gdb) next
20

9              Prime[2] = 1;

(gdb) next

10             for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                CheckPrime(N);
```

prime-number finding program

```
(gdb) display N
```

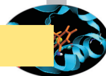
prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

prime-number finding program



```
(gdb) display N
```

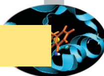
```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6              for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

prime-number finding program



```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6            for (J = 2; J*J <= K; J++)
```

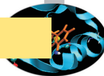
```
(gdb) next
```

```
12           Prime[K] = 1;
```

```
(gdb) next
```

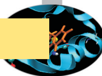
```
13       }
```

prime-number finding program



```
(gdb)  n
```

prime-number finding program

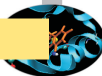


```
(gdb)  n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

prime-number finding program



```
(gdb)  n
```

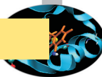
```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11                  CheckPrime(N);
1: N = 5
```

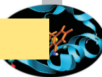
```
(gdb) n
```

prime-number finding program



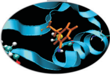
```
(gdb) n

10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}

(gdb) n

11                 CheckPrime(N);
1: N = 5

(gdb) n

10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5

(gdb) n
```

prime-number finding program



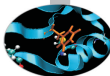
```
(gdb)  n

10             for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}

(gdb) n

11                CheckPrime(N);
1: N = 5

(gdb) n

10             for (N = 3; N <= UpperBound; N += 2)
1: N = 5

(gdb) n

11                CheckPrime(N);
1: N = 7
```

prime-number finding program



```
(gdb) l  Main.c:10
```

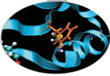
prime-number finding program



```
(gdb) l  Main.c:10
```

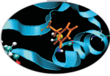
```
5          main()
6          {  int N;
7             printf("enter upper bound\n");
8             scanf("%d",&UpperBound);
9             Prime[2] = 1;
10            for (N = 3; N <= UpperBound; N += 2)
11               CheckPrime(N);
12               if (Prime[N]) printf("%d is a prime\n",N);
13            return 0;
14         }
```

Main.c :new version



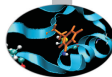
```
1   #include <stdio.h>
2   #define MaxPrimes 50
3    int Prime[MaxPrimes],
4    UpperBound;
5      main()
6      {  int N;
7         printf("enter upper bound\n");
8         scanf("%d",&UpperBound);
9         Prime[2] = 1;
10        for (N = 3; N <= UpperBound; N += 2)
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
13
14        return 0;
15     }
```

Main.c :new version



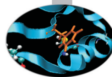
```
1   #include <stdio.h>
2   #define MaxPrimes 50
3    int Prime[MaxPrimes],
4    UpperBound;
5      main()
6      {   int N;
7          printf("enter upper bound\n");
8          scanf("%d",&UpperBound);
9          Prime[2] = 1;
10         for (N = 3; N <= UpperBound; N += 2){
11             CheckPrime(N);
12             if (Prime[N]) printf("%d is a prime\n",N);
13       }
14         return 0;
15     }
```

prime-number finding program



In other shell COMPILATION

prime-number finding program

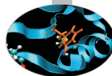


In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n)

prime-number finding program



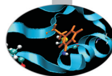
In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) d
```

prime-number finding program



In other shell COMPILATION

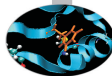
```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n)

prime-number finding program



In other shell COMPILATION

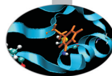
```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) d
```

Delete all breakpoints? (y or n)   y

prime-number finding program



In other shell COMPILATION

```
(gdb) kill
```

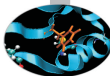
Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb)r
```

prime-number finding program



In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

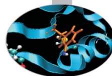
```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

prime-number finding program



In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) d
```

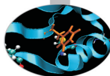
Delete all breakpoints? (y or n)   y

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound
```

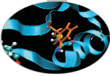
20

prime-number finding program



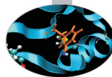
```
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime

Program exited normally.
```

prime-number finding program



```
(gdb) list Main.c:6
```

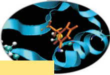
prime-number finding program



```
(gdb) list Main.c:6
```

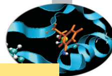
```
1        #include <stdio.h>
2        #define MaxPrimes 50
3         int Prime[MaxPrimes],
4         UpperBound;
5           main()
6           {   int N;
7               printf("enter upper bound\n");
8               scanf("%d",&UpperBound);
9               Prime[2] = 1;
10              for (N = 3; N <= UpperBound; N += 2){
```

prime-number finding program



```
(gdb) break Main.c:8
```

prime-number finding program

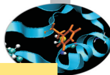


```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

prime-number finding program



```
(gdb) break Main.c:8
```

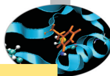
```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8             scanf("%d",&UpperBound);
```

```
(gdb) next
```

prime-number finding program



```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

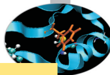
```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8              scanf("%d",&UpperBound);
```

```
(gdb) next
```

20

prime-number finding program



```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

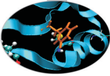
```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8             scanf("%d",&UpperBound);
```

```
(gdb) next
```

20

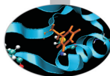
```
9             Prime[2] = 1;
```

prime-number finding program



```
(gdb) set UpperBound=40
(gdb) continue
```

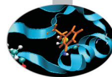
prime-number finding program



```
(gdb) set UpperBound=40
(gdb) continue
```

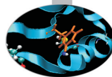
```
Continuing.
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime
23 is a prime
29 is a prime
31 is a prime
37 is a prime

Program exited normally.
```

Debugging post mortem



- When a program exits abnormally the operating system can write out core file, which contains the memory state of the program at the time it crashed.
- Combined with information from the symbol table produced by -g the core file can be used fo find the line where program stopped, and the values of its variables at that point.
- Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- In the GNU Bash shell the command ulimit -c control the maximum size of the core files. If the size limit is set to zero, no core files are produced.

Debugging post mortem

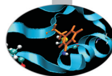


- When a program exits abnormally the operating system can write out core file, which contains the memory state of the program at the time it crashed.
- Combined with information from the symbol table produced by -g the core file can be used fo find the line where program stopped, and the values of its variables at that point.
- Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- In the GNU Bash shell the command ulimit -c control the maximum size of the core files. If the size limit is set to zero, no core files are produced.

```
ulimit -c unlimited
gdb exe_file core
```

CINECA SCAI SuperComputing Applications and Innovation

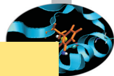
Graphical Debuggers



- gdb -tui or gdbtui
- ddd (data dysplay debugger) is a graphical front-end for command-line debuggers.
- ddt (Distributed Debugging Tool) is a comprehensive graphical debugger for scalar, multi-threaded and large-scale parallel applications that are written in C, C++ and Fortran.
- Etc.

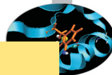
CINECA

Why don’t optmize?



```
1  int main(void)
2  {
3      float q;
4      q=3.;
5      return 0;
6  }
```

Why don't optmize?

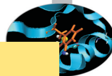


```
1  int main(void)
2  {
3      float q;
4      q=3.;
5      return 0;
6  }
```

```
<ruggiero@shiva ~/CODICI>gcc  opt.c -g -O0 -o opt
<ruggiero@matrix2 ~>gdb opt
(gdb) b main
```

```
Breakpoint 1 at 0x8048395: file opt.c, line 4.
```

Why don't optmize?



```
1  int main(void)
2  {
3      float q;
4      q=3.;
5      return 0;
6  }
```

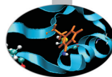
```
<ruggiero@shiva ~/CODICI>gcc  opt.c -g -O0 -o opt
<ruggiero@matrix2 ~>gdb opt
(gdb) b main
```

```
Breakpoint 1 at 0x8048395: file opt.c, line 4.
```

```
<ruggiero@shiva ~/CODICI>gcc  opt.c -g -O3 -o opt
<ruggiero@matrix2 ~>gdb opt
(gdb) b main
```

```
Breakpoint 1 at 0x8048395: file opt.c, line 6.
```

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

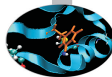
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

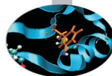
Floating Point Computing

Totalview (www.totalviewtech.com)



- Used for debugging and analyzing both serial and parallel programs.
- Supported languages include the usual HPC application languages:
  - C,C++,Fortran
  - Mixed C/C++ and Fortran
  - Assembler
- Supported many commercial and Open Source Compilers.
- Designed to handle most types of HPC parallel coding (multi-process and/or multi-threaded applications).
- Supported on most HPC platforms.
- Provides both a GUI and command line interface.
- Can be used to debug programs, running processes, and core files.
- Provides graphical visualization of array data.
- Includes a comprehensive built-in help system.
- And more...

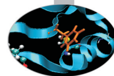
Compilation options for Totalview



- You will need to compile your program with the appropriate flag to enable generation of symbolic debug information. For most compilers, the -g option is used for this.
- It is recommended to compile your program without optimization flags while you are debugging it.
- TotalView will allow you to debug executables which were not compiled with the -g option. However, only the assembler code can be viewed.
- Some compilers may require additional compilation flags. See the *TotalView User's Guide* for details.

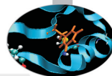
```
ifort [option] -O0 -g file_source.f -o filename
```

Starting Totalview

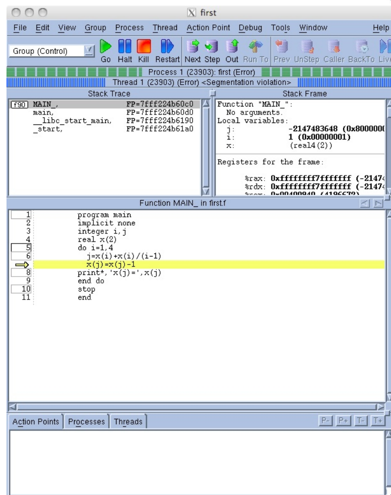


Command	Action
totalview	Starts the debugger. You can then load a program or corefile, or else attach to a running process.
totalview *filename*	Starts the debugger and loads the program specified by *filename*.
totalview *filename corefile*	Starts the debugger and loads the program specified by *filename* and its core file specified by *corefile*.
totalview *filename* -a *args*	Starts the debugger and passes all subsequent arguments (specified by *args*) to the program specified by *filename*. The -a option must appear after all other TotalView options on the command line.

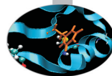
Totalview:panel



1. Stack Trace
   - Call sequence
2. Stack Frame
   - Local variables and their values
3. Source Window
   - Indicates presently executed statement
   - Last statement executed if program crashed
4. Info tabs
   - Informations about processes and action points.

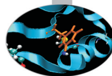


Totalview:Action points



- Breakpoint stops the excution of the process and threads that reach it.
  - Unconditional
  - Conditional: stop only if the condition is satisfied.
  - Evaluation: stop and excute a code fragment when reached.
- Process barrier point synchronizes a set of processes or threads.
- Watchpoint monitors a location in memory and stop execution when its value changes.

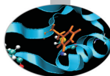
Totalview:Setting Action points

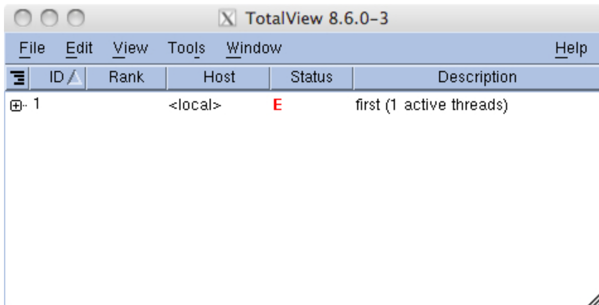


- Breakpoint
  - Right click on a source line → Set breakpoint
  - Click on the line number
- Watchpoint
  - Right click on a variable → Create watchpoint
- Barrier point
  - Right click on a source line → Set barrier
- Edit action point property
  - Rigth click on a action point in the Action Points tab → Properties.

SCAI
SuperComputing Applications and Innovation

Totalview:Status



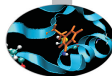


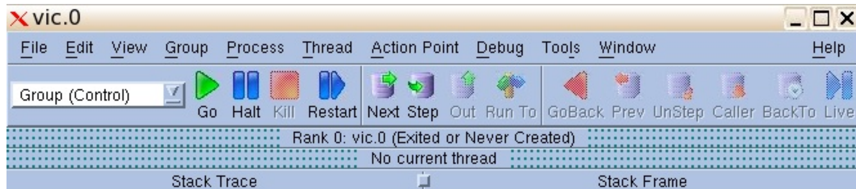
Status Code	Description
T	Thread is stopped
B	Stopped at a breakpoint
E	Stopped because of a error
W	At a watchpoint
H	In a Hold state
M	Mixed - some threads in a process are running and some not
R	Running

CINECA

SuperComputing Applications and Innovation

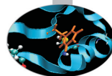
Totalview:Execution control commands





Command	Description
Go	Start/resume excution
Halt	Stop excution
Kill	Terminate the job
Restart	Restarts a running program, or one that has stopped without exiting
Next	Run to next source line or instruction. If the next line/instruction calls a function the entire function will be excuted and control will return to the next source line or instruction.
Step	Run to next source line or instruction. If the next line/instruction calls a function, excution will stop within function.
Out	Excute to the completion of a function. Returns to the instruction after one which called the function.
Run to	Allows you to arbitrarily click on any source line and then run to that point.

Totalview:Mouse buttons

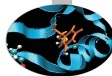


Mouse Button	Purpose	Description	Examples
Left	Select	Clicking on object causes it to be selected and/ or to perform its action	Clicking a line number sets a breakpoint. Clicking on a process/thread name in the root window will cause its source code to appear in the Process Window's source frame.
Middle	Dive	Shows additional information about the object - usually by popping open a new window.	Clicking on an array object in the source frame will cause a new window to pop open, showing the array's values.
Rigth	Menu	Pressing and holding this button a window/frame will cause its associated menu to pop open.	Holding this but ton while the mouse pointer is in the Root Window will cause the Root Window menu to appear. A menu selection can then be made by dragging the mouse pointer while continuing to press the middle button down.

SCAI

SuperComputing Applications and Innovation

I’ve checked everything!

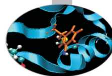


what else could it be?

- Full file system.
- Disk quota exceeded.
- File protection.
- Maximum number of processes exceeded.
- Are all object file are up do date? Use Makefiles to build your projects
- What did I change since last version of my code?
  Use a version control system: CVS,RCS,...
- Does any environment variable affect the behaviour of my program?

CINECA

Outline



Introduzione

Architetture

La cache e il sistema di memoria

Pipeline

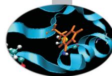
Profilers

Compilatori e ottimizzazione

Librerie scientifiche

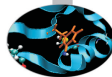
Floating Point Computing

Last resorts



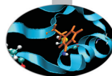
- Ask for help.
- Explain your code to somebody.
- Go for a walk, to the movies,leave it to tomorrow.

Errare humanum est ...



- Where was error made?
- Who made error?
- What was done incorrectly?
- How could the error have been prevented?
- Why wasn’t the error detected earlier?
- How could the error have been detected earlier?

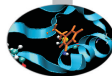
Bibliography



- **Why program fail. A guide to systematic debugging.** A. Zeller *Morgan Kaufmann Publishers* 2005.
- **Expert C Programming deep C secrets** P. Van der Linden *Prentice Hall PTR* 1994.
- **How debuggers works Algorithms, data,structure, and Architecture** J. B. Rosemberg *John Wiley & Sons* 1996.
- **Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code** R. B. Blunden *Apress* 2003.
- **Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems** D.J. Agans *American Management Association* 2002.
- **The Art of Debugging** N. Matloff, P. J. Salzman *No starch press* 2008.

CINECA SCAI SuperComputing Applications and Innovation

Bibliography



- **Debugging With GDB: The Gnu Source-Level Debugger** R.M. Stallmann, R.H. Pesch, S. Shebs *Free Software Foundation* 2002.
- **The Practice of Programming** B.W. Kernighan, R. Pike *Addison-Wesley* 1999.
- **Code Complete** S. McConnell *Microsoft Press* 2004.
- **Software Testing Technique** B. Beizer *The Coriolis Group* 1990.
- **The Elements of Programming Style** B. W. Kernighan P.J. Plauger *Computing Mcgraw-Hill* 1978.
- **The Art of Software testing** K.J. Myers *Kindle Edition* 1979.
- **The Developer's Guide to Debugging** H. GrÃ¶tker, U. Holtmann, H. Keding, M.Wloka *Kindle Edition* 2008.

CINECA