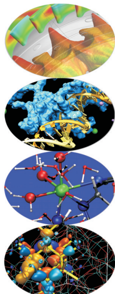
CINECA SCAI
SuperComputing Applications and Innovation

SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)



Roma, 2 Aprile 2014

CINECA

Outline

Pipeline

Profilers

CPU: parallelismo interno?

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

Pipeline

- 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

Esempio

- Somma di reali
  1. Allineare esponente
  2. Sommare mantissa
  3. normalizzare il risultato
  4. arrotondamento
- Esempio: $9.752 \cdot 10^4 + 4.876 \cdot 10^3$
  - $(1) \rightarrow 9.752 \cdot 10^4 + 0.4876 \cdot 10^4$
  - $(2) \rightarrow 10.2396 \cdot 10^4$
  - $(3) \rightarrow 1.02396 \cdot 10^5$
  - $(4) \rightarrow 1.024 \cdot 10^5$

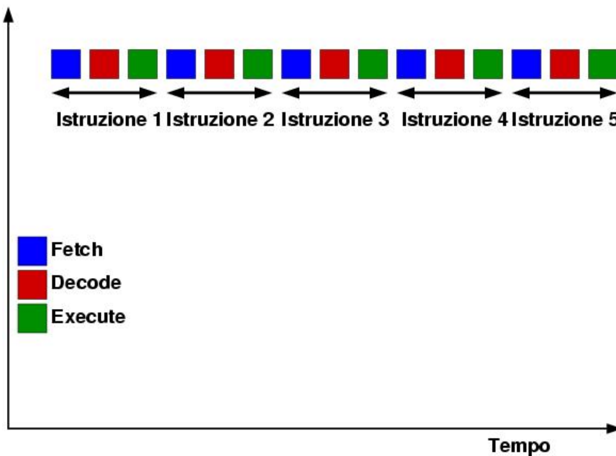
Pipeline funzionale

- I passi da fare per eseguire un'istruzione:
  1. Prendere l'istruzione dalla memoria
  2. Decodificare l'istruzione
  3. Inizializzare registri e prendere dati dalla memoria
  4. Eseguire l'istruzione (per esempio: somma di reali)
  5. Scrivere i risultati nella memoria.
- Come sfruttare la sequenzialità delle operazioni?
  - Perché non eseguire le singole sotto-operazioni di più operazioni differenti insieme (purché sfalsate di un passo)?

Pipeline funzionale

- Nell'ipotesi che:
  1. un'operazione si possa dividere in più sotto-operazioni;
  2. le singole sotto-operazioni siano tra di loro logicamente indipendenti;
  3. le singole sotto-operazioni impieghino (più o meno) lo stesso tempo;

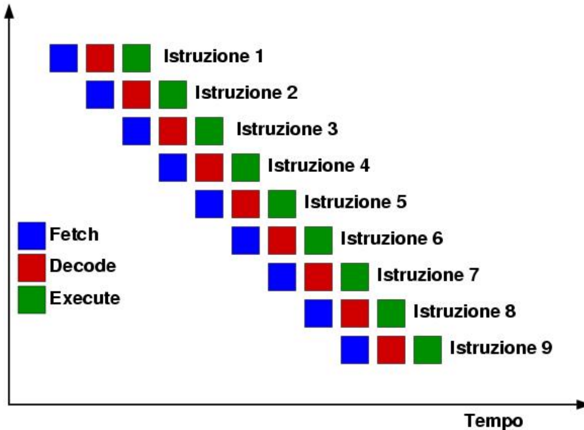
CPU sequenziale



CPU sequenziale: esempio

- ATTENZIONE: è solo un esempio indicativo!!!!
- Ogni quadrato $\rightarrow$ 1 ciclo di clock.
- Un'istruzione è composta da tre passi;
- Nell'ipotesi che un passo venga completato in un ciclo:
  - Esegue 1 istruzione ogni tre cicli;
- Esempio (istruzione = calcolo floating point)
  - Clock = 100 Mhz
  - Mflops = 33
  - Bandwidth = 8*33 $\rightarrow$ 264 MB/s

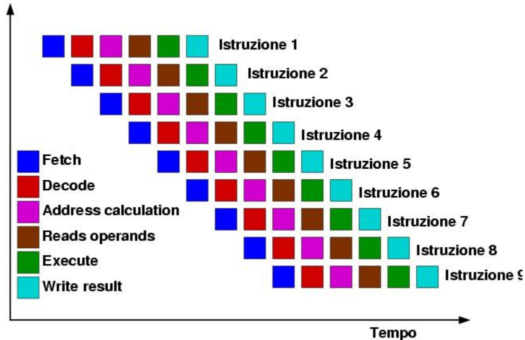
CPU pipelined



CPU pipelined: esempio

- Un’istruzione è composta da tre passi;
- Nell’ipotesi che un passo venga completato in un ciclo:
  - + Esegue 1 istruzione ogni ciclo (a pipeline piena)
  - - Esegue 1 istruzione ogni 3 cicli (a pipeline vuota)
  - - Più complessa da realizzare
  - - Richiede una bandwidth maggiore
- Esempio (istruzione = calcolo floating point)
  - Clock = 100 Mhz
  - Mflops = 100/33 (pipeline piena/in stallo)
  - Bandwidth = 8*100 $\rightarrow$ 800 MB/s a pipeline piena

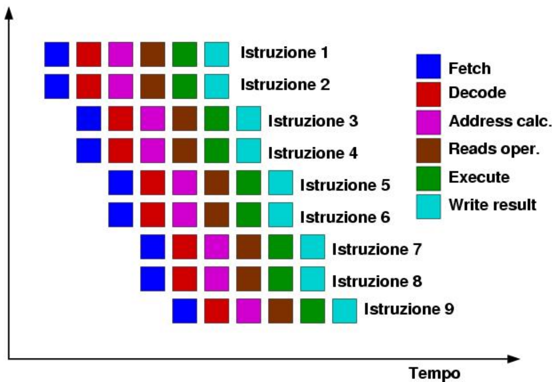
CPU superpipelined



CPU superpipelined: esempio

- Un'istruzione è composta da sei passi
- Nell'ipotesi che un passo venga completato in un ciclo:
  - \+ Esegue 1 istruzione ogni ciclo (a pipeline piena)
  - \+ Permette di aumentare il clock
  - \- Esegue 1 istruzione ogni 6 cicli (a pipeline vuota)
  - \- Più complessa da realizzare
  - \- Richiede una bandwidth maggiore
- Esempio (istruzione = calcolo floating point)
  - Clock = 200 Mhz
  - Mflops = 200/33 (pipeline piena/in stallo)
  - Bandwidth = 8*200 $\rightarrow$ 1.6 GB/s

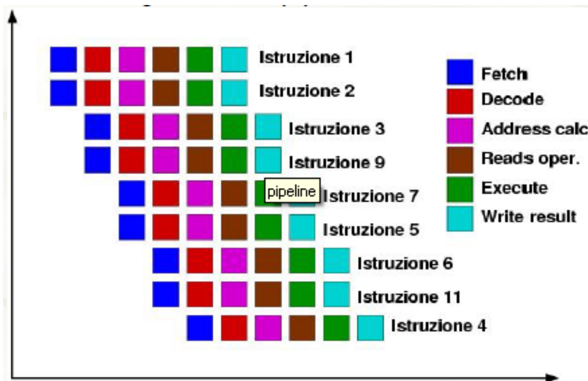
CPU superscalare



CPU superscalare: esempio

- Un'istruzione è composta da sei passi;
- Nell'ipotesi che un passo venga completato in un ciclo:
  - \+ Esegue 2 istruzioni ogni ciclo (a pipeline piena)
  - \- Esegue 1 istruzione ogni 6 cicli (a pipeline vuota)
  - \- Più complessa da realizzare
  - \- Richiede una bandwidth maggiore ed indipendenza tra le istruzioni
- Esempio (istruzione = calcolo floating point)
  - Clock = 200 Mhz
  - Mflops = 400/33 (pipeline piena/in stallo)
  - Bandwidth = 8*2*200 $\rightarrow$ 3.2 GB/s

Out of Order execution CPU



OoO CPU: esempio

- 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

- Un'istruzione è composta da sei passi;
- Nell'ipotesi che un passo venga completato in un ciclo:
  - \+ Esegue 2 istruzioni ogni ciclo (a pipeline piena)
  - \+ Può ridurre lo stallo della pipeline
  - \- Esegue 1 istruzione ogni 6 cicli (a pipeline vuota)
  - \- Estremamente complessa da realizzare
- Esempio (istruzione = calcolo floating point)
  - Clock = 200 Mhz
  - Mflops = 400/ 33 (pipeline piena/in stallo)
  - Bandwidth = 8*2*200 $\rightarrow$ 3.2 GB/s

Pipeline

- Vantaggi:
  - Aumento potenza (di picco): da 33 a 400 Mflops.
  - Possibilità di diminuire il clock: da 100 a 200 Mhz.
- Problemi:
  - dipendenza tra i dati (Pipeline di calcolo)
  - dipendenza tra le istruzioni (Pipeline funzionale)
  - Bandwidth necessaria: da 264 MB/s a 3200 MB/s
- Da evitare assolutamente:
  - DO WHILE, dipendenze inutili, procedure ricorsive

Pipeline Funzionale

- L'esecuzione di un'istruzione presenta una pipeline (funzionale).
  1. Prendere l'istruzione dalla memoria
  2. Decodificare l'istruzione
  3. Inizializzare registri e prendere dati dalla memoria
  4. Eseguire l'istruzione (per esempio: somma di reali)
  5. Scrivere i risultati nella memoria.
- In questo caso limitano il riempimento della pipeline:
  - salti nel programma (function, subroutine, goto)
  - clausole IF-THEN-ELSE (eccezioni)
- Cosa fare:
  - inlining esplicito o automatico, test negli IF quasi sempre **`false`**

Riempimento pipeline

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

SCAI SuperComputing Applications and Innovation

Esempio ..

```
      do 1000 z=1,nz
         do 1000 y=1,ny
            do 1000 x=1,nx/2
               hr(x,y,z,1)=hr(x,y,z,1)*norm      !! primo loop
               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
1000  continue
      do 2000 z=1,nz
         do 2000 y=1,ny
            do 2000 x=1,nx/2
               ur(x,y,z,1)=ur(x,y,z,1)*norm      !! secondo loop
               ur(x,y,z,1)=ur(x,y,z,1)*norm
               ur(x,y,z,2)=ur(x,y,z,2)*norm
               ui(x,y,z,2)=ui(x,y,z,2)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
2000  continue
```

Esempio


```
      do 3000 z=1,nz
          k3=beta(z)
          do 3000 y=1,ny
             k2=eta(y)
             do 3000 x=1,nx/2
                k1=alfa(x,1)                                   !! terzo loop
                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
  usertime   1+2+3 = 1.646515
```

Esempio: loop merging


```
      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            !! primo loop
               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
               ur(x,y,z,1)=ur(x,y,z,1)*norm           !! secondo loop
               ur(x,y,z,1)=ur(x,y,z,1)*norm
               ur(x,y,z,2)=ur(x,y,z,2)*norm
               ui(x,y,z,2)=ui(x,y,z,2)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
               k1=alfa(x,1)                           !! terzo loop
               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

 usertime   0.983780  (1+2+3 = totale: 1.646515)
```

Loop splitting

- L'operazione inversa al loop merging è il loop splitting
  - si separa un singolo loop con molte istruzioni con più loop con meno istruzioni
  - Può favorire il lavoro del compilatore consentendogli di fare unrolling e/o blocking (operazione fattibile con loop semplici);
  - Semplifica il flusso delle istruzioni;
  - Il vantaggio/svantaggio è difficilmente quantificabile a priori.

Loop unrolling

- All'interno di un loop, si sviluppa parzialmente il ciclo.

```
do j = 1, nj                 -> do j = 1, nj
do i = 1, ni                 -> do i = 1, ni, 2
 a(i,j)=a(i,j)+c*b(i,j)      ->  a(i  ,j)=a(i  ,j)+c*b(i  ,j)
                             ->  a(i+1,j)=a(i+1,j)+c*b(i+1,j)
```

  - Maggiore riempimento della pipeline;
  - Riduce le strutture di controllo;
  - Non usabile quando c'è dipendenza tra i dati;
  - In genere è gestito dal compilatore;
  - Esiste un unrolling ideale (dipende da problema, macchina, etc...);
  - Implica un movimento più veloce di dati da e per la memoria (richiede maggiore bandwidth);

Esempio


```
      do 2000 z=1,nzp
         do 2000 y=1,nyp
            do 2000 x=1,nxp/2
              do 3000 i=1,3
                 ar(i)=ur(x,y,z,i)
                 ai(i)=ui(x,y,z,i)
                 br(i)=hr(x,y,z,i)
                 bi(i)=hi(x,y,z,i)
3000 continue
              hr(x,y,z,1)=ar(2)*br(3)-ar(3)*br(2)
              hr(x,y,z,2)=ar(3)*br(1)-ar(1)*br(3)
              hr(x,y,z,3)=ar(1)*br(2)-ar(2)*br(1)
              hi(x,y,z,1)=ai(2)*bi(3)-ai(3)*bi(2)
              hi(x,y,z,2)=ai(3)*bi(1)-ai(1)*bi(3)
              hi(x,y,z,3)=ai(1)*bi(2)-ai(2)*bi(1)
2000  continue

usertime    2.01301
```

Esempio: unrolling


```
      do 2000 z=1,nzp
         do 2000 y=1,nyp
            do 2000 x=1,nxp/2
               ar(1)=ur(x,y,z,1)
               ai(1)=ui(x,y,z,1)
               br(1)=hr(x,y,z,1)
               bi(1)=hi(x,y,z,1)
               ar(2)=ur(x,y,z,2)
               ai(2)=ui(x,y,z,2)
               ...
               hr(x,y,z,1)=ar(2)*br(3)-ar(3)*br(2)
               hr(x,y,z,2)=ar(3)*br(1)-ar(1)*br(3)
               hr(x,y,z,3)=ar(1)*br(2)-ar(2)*br(1)
               hi(x,y,z,1)=ai(2)*bi(3)-ai(3)*bi(2)
               ...
2000  continue

usertime    1.41762
```

Unroll loop esterno


```
        do j= 1, n     !caso 1
           do i= 1, n
              y(j) = y(j) + x(i)*a(i,j)
           enddo
        enddo
.......
        do j= 1, n, 4  !caso 2
           do i= 1, n
              y(j+0) = y(j+0) + x(i)*a(i,j+0)
              y(j+1) = y(j+1) + x(i)*a(i,j+1)
              y(j+2) = y(j+2) + x(i)*a(i,j+2)
              y(j+3) = y(j+3) + x(i)*a(i,j+3)
           enddo
        enddo
Tempi (f77 -O2 (-O5)):
caso 1: 0.8488270  (0.3282020)
caso 2: 0.3540250  (0.3215380) -> unrolling di 4
caso 2: 0.3248700  (0.2915500) -> unrolling di 8
```

Reduction & Unroll


```
    do j = i, nj ! caso normale 1)
       do i = i, ni
          somma = somma + a(i,j)
       end do
    end do
......
    do j = i, nj !reduction a 4 elementi.. 2)
       do i = i, ni, 4
          somma_1 = somma_1 + a(i+0,j)
          somma_2 = somma_2 + a(i+1,j)
          somma_3 = somma_3 + a(i+2,j)
          somma_4 = somma_4 + a(i+3,j)
       end do
    end do
    somma = somma_1 + somma_2 + somma_3 + somma_4
f77 -native -O2 (-O4)
tempo 1) ---> 4.49785 (2.94240)
tempo 2) ---> 3.54803 (2.75964)
```

Unrolling: commenti I

- In genere l'unrolling è gestito dal compilatore
- Si può pure inibire il compilatore non facendogli capire quando le istruzioni sono indipendenti.
- Cosa può fare qui il compilatore?

```
do j = 1, nj
   do i = 1, ni
      a(low(i),up(j)) = a(low(i),up(j)) + b(i,j) * c(i,j)
   enddo
enddo
```

- Se non c'è dipendenza tra `low(i),up(j)` allora si può fare l'unrolling (a mano ...)

Unrolling: commenti (II)

- e qui?

```
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?
- 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

Unrolling: commenti (III)

- Inibisce inoltre l'unrolling (ed in genere il riempimento della pipeline):
  - Salti condizionali (`if` ...)
  - Chiamate a funzioni anche intrinseche o di libreria (sin, exp, ...)
  - Chiamate a procedure all'interno di un loop
  - Operazioni di I/0 all'interno di un loop

Opzioni del compilatore

- Come posso sapere cosa può fare il compilatore?
- Come posso sapere cosa effettivamente ha fatto il compilatore?

Opzioni del compilatore

- Come posso sapere cosa può fare il compilatore?
- Come posso sapere cosa effettivamente ha fatto il compilatore?
- Consulto il manuale.

Opzioni del compilatore

- Come posso sapere cosa può fare il compilatore?
- Come posso sapere cosa effettivamente ha fatto il compilatore?
- Consulto il manuale.
- Provate, ad esempio, il compilatore intel con la flag **-opt-report**.

Esercitazione: sommario

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

src/eser_4/fortran/mm.f90


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

src/eser_4/c/mm.c


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

Unrolling loop esterno

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

Unrolling	Fortran	C
2		
4		
8		
16		

Unrolling loop esterno e padding

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

Unrolling	Fortran	C
2		
4		
8		
16		

Ultimo sforzo ...

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

Outline

Pipeline

Profilers
- Introduzione
- gprof

Outline

Pipeline

Profilers
- Introduzione
- gprof

Perché misurare le prestazioni?

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

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

La misura delle prestazioni però:

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

time

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

un tipico output:

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

time: output

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

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

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

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.

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

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

man top


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

cpu_time

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

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

clock

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

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

tic toc

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

```
tic
k = 1:1:n;
    j = 1:1:n;
        i = 1:1:n;
        q1(i,j) = q1(i,j) + a(i,k)*b(k,j);
toc

tic
q2=a*b;
toc
```

Timing multipiattaforma

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

Sintassi:

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

gettimeofday

Definizione funzione: dwalltime.c

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

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

Profiling di un programma

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

Profiling di un programma (II)

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

Profiling di un programma (III)

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

Outline

Pipeline

Profilers

- Introduzione
- gprof

gprof

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

gprof: sintassi


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

gprof: flat profile output


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

gprof: flat profile output

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

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

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

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

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

gprof: call graph


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

index % time    self  children    called     name
                1.27   57.38       1/1           main [2]
[1]    100.0    1.27   57.38       1         MAIN__ [1]
                3.07   36.96    1000/1000        sl_ [3]
                0.00   17.34       1/1           cfl_ [4]
                0.00    0.01       1/1           input_ [11]
                0.00    0.00    1000/1001        cc_ [13]
                0.00    0.00    1000/1000        output_ [14]
                0.00    0.00       2/2           dwalltime00_ [15]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   58.65                 main [2]
                1.27   57.38       1/1           MAIN__ [1]
-----------------------------------------------
                3.07   36.96    1000/1000        MAIN__ [1]
[3]     68.3    3.07   36.96    1000         sl_ [3]
               12.09    0.00 251001000/251001000     ccsl_ [7]
               10.07    0.00 251001000/251001000     interpbl_ [8]
                7.47    0.00 251001000/502002000     fcvx_ [5]
                7.33    0.00 251001000/502002000     fcvy_ [6]
-----------------------------------------------
                0.00   17.34       1/1           MAIN__ [1]
[4]     29.6    0.00   17.34       1         cfl_ [4]
                1.29    7.47       1/1           maxfx_ [9]
                1.25    7.33       1/1           maxfy_ [10]
                0.00    0.00       1/1           maxdhx_ [16]
                0.00    0.00       1/1           maxdhy_ [17]
-----------------------------------------------
.....................
```

gprof: call graph


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

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

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

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

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

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

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

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

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

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

gprof: possibili problemi

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

gprof: caso intrusivo


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

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

gprof: e le librerie matematiche? (linkare statico)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 68.13    121.61   121.61                             sin
  8.44    136.67    15.06 502002000     0.00     0.00  fcvx_
  6.97    149.11    12.44 502002000     0.00     0.00  fcvy_
  6.23    160.24    11.13 251001000     0.00     0.00  ccsl_
  5.46    169.98     9.74 251001000     0.00     0.00  interpbl_
  1.71    173.04     3.06     1000     0.00     0.04  sl_
  0.82    174.50     1.46        1     1.46     7.68  maxfy_
  0.70    175.75     1.25        1     1.25     8.78  maxfx_
  0.69    176.99     1.24        1     1.24    55.45  MAIN__
  0.12    177.21     0.22                             vfprintf
  0.10    177.38     0.17                             ___printf_fp
  0.08    177.52     0.14                             __mpn_mul_1
  0.07    177.64     0.12                             fd_alloc_w_at
  0.06    177.75     0.11                             _IO_default_xsputn
  0.05    177.84     0.09                             memcpy
  0.04    177.91     0.07                             __write_nocancel
  0.03    177.97     0.06                             ___vsnprintf_chk
  0.03    178.02     0.05     1001     0.00     0.00  cc_
  0.02    178.06     0.04                             ____strtol_l_internal
  0.02    178.10     0.04                             mempcpy
  0.02    178.13     0.03                             _IO_str_init_static_internal
  0.02    178.16     0.03                             _gfortran_transfer_real
  0.02    178.19     0.03                             _gfortrani_next_format
  0.02    178.22     0.03                             formatted_transfer
  0.02    178.25     0.03                             formatted_transfer_scalar
  0.02    178.28     0.03                             output_float
```

gprof: line by line profiling

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

```
-l            --->  per il profiling line by line
-l -A -x      --->  per il listing del codice
```

gprof: line by line profiling

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call name
 69.00    123.21   123.21                             sin
  6.28    134.43    11.21                             fcvx (FCVX.F90:28@805e8ba)
  5.29    143.88     9.46                             fcvy (FCVY.F90:28@805ca92)
  3.02    149.27     5.39                             ccsl (CCSL.F90:29@805ea53)
  2.09    153.00     3.73                             interpbl (INTERBL.F90:18@804d578)
  1.37    155.45     2.45                             interpbl (INTERBL.F90:18@804d5a7)
  1.16    157.52     2.07                             interpbl (INTERBL.F90:12@804d4fe)
  1.15    159.57     2.05                             ccsl (CCSL.F90:30@805ea92)
  1.05    161.45     1.88 502002000     3.75     3.75 fcvx_ (FCVX.F90:8@805e420)
  0.73    162.76     1.31                             ccsl (CCSL.F90:29@805ea8d)
```

gprof: line by line profiling

```
     Call graph

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

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

gprof: annotated source


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


Top 10 Lines:
     Line      Count
        8  502002000
Execution Summary:
        2   Executable lines in this file
        2   Lines executed
   100.00   Percent of the file executed
502002000   Total number of line executions
251001000.00   Average executions per line
```

Line level profiling

- Il tool GNU **gcov** fornisce un profiling per linea
  - utile per vedere quali linee vengono maggiormente accedute
  - utile per capire quali istruzioni sono più lente delle altre
  - attenzione è intrusivo
- Utilizzo:
  - compilare con l'opzione

    ```
    -fprofile-arcs -ftest-coverage
    ```
  - eseguire il programma
  - invocare:

    ```
    gcov codice\_sorgente
    ```

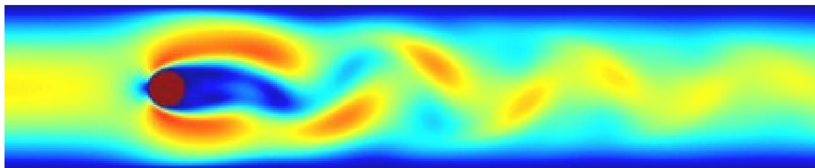
  - Le informazione saranno raccolte nei file ***.gcov**

Gprof: Esercitazione

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

```
eser_gprof
```

  - Flusso attraverso un cilindro



Esercitazione

- Come scala il codice al variare delle dimensioni?

Dimensioni	250*50	500*100	1000*200

- Quali sono le quattro subroutine più impegnative?

Subroutine	Tempo	%	Num. chiamate

Alcuni consigli pratici

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