Programmazione di Processori *MultiCore* – III lezione

Federico Massaioli (**federico.massaioli@caspur.it**)

CASPUR e Università degli Studi di Roma "La Sapienza"
Laurea Magistrale in Informatica
Anno accademico 2008-2009

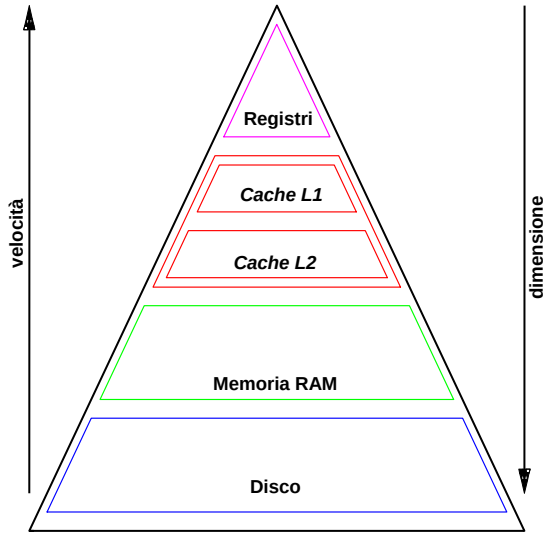
Il sistema di memoria

- Capacità di calcolo delle CPU: 2× ogni 18 mesi
- Velocità di accesso alla RAM: 2× ogni 120 mesi!!

- Inutile ridurre numero o costo delle operazioni se i dati non arrivano dalla memoria

- Soluzione: memorie intermedie veloci (Von Neumann, 1947!!)
- Il sistema di memoria è una struttura profondamente gerarchica
- La gerarchia è trasparente all'applicazione, i suoi effetti no

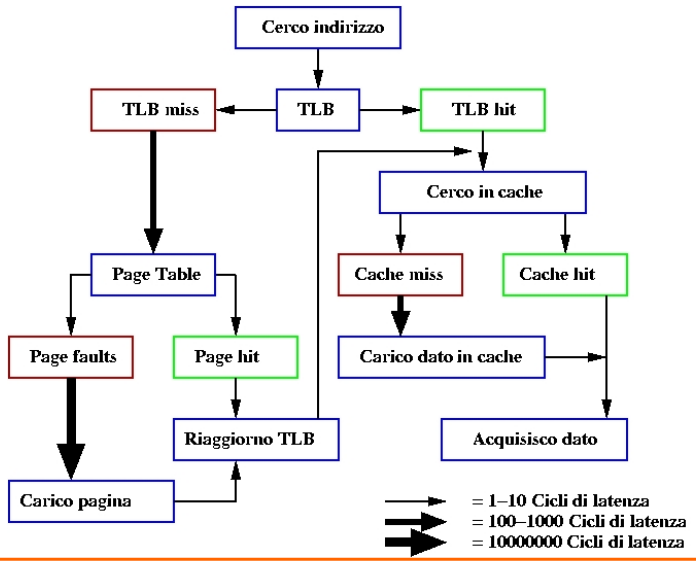
La gerarchia di memoria



Indirizzi logici e virtuali

- Il processo in esecuzione vede la memoria come uno spazio di indirizzamento lineare
- Lo spazio di indirizzamento è frammentato in pagine (di solito 4÷32 kB, 256÷4096 kB in casi speciali)
- Le pagine di memoria fisica (*frame*) non devono essere necessariamente contigue
- Vantaggi
  - protezione reciproca dei processi
  - maggiore efficienza nella gestione della memoria
  - possibilità di spostare “pagine” su disco in situazione critica
- Svantaggi
  - traduzione degli indirizzi, possibili rallentamenti (non ne parleremo…)
  - lo *swapping* di pagine su disco rallenta di ordini di grandezza la velocità di esecuzione (da evitare assolutamente!!)

La memoria virtuale



topas (con *swapping*)

```
Topas Monitor for host:    pwr520
Mon Jan 30 11:24:23 2006   Interval:  2

Kernel    0.1    |#                          |
User      0.0    |                           |
Wait     87.4    |#########################  |
Idle     12.5    |####                       |

Network  KBPS    I-Pack  O-Pack    KB-In  KB-Out
en0       0.2       0.5     0.5      0.0     0.2
sn0       0.0       0.0     0.0      0.0     0.0
en2       0.0       0.0     0.0      0.0     0.0
sn1       0.0       0.0     0.0      0.0     0.0
lo0       0.0       0.0     0.0      0.0     0.0
ml0       0.0       0.0     0.0      0.0     0.0

Disk     Busy%     KBPS      TPS KB-Read KB-Writ
hdisk0     0.0   2166.0   305.5  1104.0  1062.0
hdisk1     0.0      0.0     0.0     0.0     0.0

Name            PID  CPU%  PgSp Owner
bgk3d.0.     741490   0.1 622.6 amatig
syncd        372972   0.0   0.6 root
lrud         274566   0.0   0.2 root
getty        589858   0.0   0.5 root
rpc.lock     336114   0.0   0.2 root
xmgc         290958   0.0   0.1 root
bgk3d.0.     634992   0.0 622.6 amatig
bgk3d.0.     581842   0.0 622.6 amatig
bgk3d.0.     692358   0.0 622.6 amatig
bgk3d.0.     749686   0.0 622.6 amatig
bgk3d.0.     704606   0.0 622.6 amatig
bgk3d.0.     725098   0.0 622.6 amatig
bgk3d.0.     585824   0.0 622.6 amatig
errdemon     331952   0.0   0.7 root

EVENTS/QUEUES    FILE/TTY
Cswitch      762 Readch      182
Syscall       73 Writech     384
Reads          0 Rawin         0
Writes         1 Ttyout      182
Forks          0 Igets         0
Execs          0 Namei         0
Runqueue     0.0 Dirblk        0
Waitqueue    8.1

PAGING           MEMORY
Faults       308 Real,MB    7743
Steals       274 % Comp    100.4
PgspIn       275 % Noncomp   0.5
PgspOut      265 % Client    0.5
PageIn       275
PageOut      265 PAGING SPACE
Sios         536 Size,MB   32768
                 % Used     14.5
NFS (calls/sec)  % Free     85.4
ServerV2       0
ClientV2       0   Press:
ServerV3       0   "h" for help
ClientV3       0   "q" to quit
```

topas (senza *swapping*)

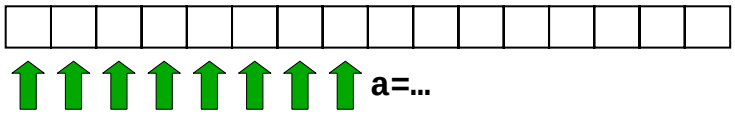
```
Topas Monitor for host:    pwr520
Mon Feb  6 14:32:42 2006   Interval:  2

Kernel    0.0   |                            |
User    100.0   |############################|
Wait      0.0   |                            |
Idle      0.0   |                            |

Network  KBPS    I-Pack  O-Pack    KB-In  KB-Out
en0       0.3       0.5     0.5      0.0     0.3
sn0       0.0       0.0     0.0      0.0     0.0
en2       0.0       0.0     0.0      0.0     0.0
sn1       0.0       0.0     0.0      0.0     0.0
lo0       0.0       0.0     0.0      0.0     0.0
ml0       0.0       0.0     0.0      0.0     0.0

Disk    Busy%     KBPS     TPS KB-Read KB-Writ
hdisk1    0.0      0.0     0.0      0.0     0.0
hdisk0    0.0      0.0     0.0      0.0     0.0

Name            PID  CPU%  PgSp Owner
bgk3d.0.     663710  12.6 625.3 amatig
bgk3d.0.     622836  12.6 625.3 amatig
bgk3d.0.     671804  12.6 625.3 amatig
bgk3d.0.     557238  12.5 625.3 amatig
bgk3d.0.     659618  12.5 625.3 amatig
bgk3d.0.     614488  12.5 625.3 amatig
bgk3d.0.     602244  12.5 625.3 amatig
bgk3d.0.     667678  12.5 625.3 amatig
getty        589858   0.0   0.5 root

EVENTS/QUEUES    FILE/TTY
Cswitch      151  Readch      213
Syscall       65  Writech     497
Reads          0  Rawin         0
Writes         2  Ttyout      213
Forks          0  Igets         0
Execs          0  Namei         0
Runqueue     8.0  Dirblk        0
Waitqueue    0.0

PAGING           MEMORY
Faults         0  Real,MB    7743
Steals         0  % Comp     92.0
PgspIn         0  % Noncomp   1.0
PgspOut        0  % Client    1.1
PageIn         0
PageOut        0  PAGING SPACE
Sios           0  Size,MB   32768
                  % Used      0.7
NFS (calls/sec)  % Free     99.2
ServerV2       0
ClientV2       0   Press:
ServerV3       0   "h" for help
ClientV3       0   "q" to quit
```

vmstat (con *swapping* e senza)

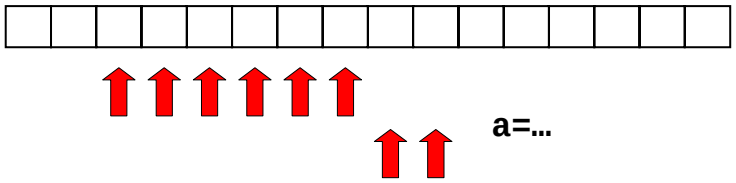
```
<amatig@pwr520 /scratch/amatig/BENCH_CASPUR/RUN>vmstat 5 10
System Configuration: lcpu=8 mem=7744MB
kthr     memory             page              faults        cpu
----- ----------- ------------------------ ------------ -----------
 r  b   avm     fre  re  pi  po  fr    sr  cy  in    sy  cs us sy id wa
 2  8 2843904   845   0   3  12  12    49   0  933   193 181  1  0 98  1
 0  8 2843908   945   0 285 305 284   972   0 1259  496 793  0  0 12 87
 0  8 2843908   958   0 287 289 288   609   0 1247  415 783  0  0 12 87
 0  8 2843908   957   0 284 284 284   827   0 1246  417 780  0  0 12 87
 0  8 2843908   956   0 279 277 279   713   0 1242  419 776  0  0 12 87
 0  8 2843908   957   0 280 280 280 1791   0 1241  418 774  0  0 12 87
 0  8 2843908   964   0 281 280 281   563   0 1242  418 770  0  0 12 87
 0  8 2843908   945   0 279 271 279   702   0 1238  416 767  0  0 12 87
 0  8 2843908   957   0 283 279 284   609   0 1244  419 783  0  0 12 87
 0  8 2843908   954   0 284 279 284 1897   0 1245  417 779  0  0 12 87
...

<amatig@pwr520 /scratch/amatig/BENCH_CASPUR/RUN>vmstat 5 10
System Configuration: lcpu=8 mem=7744MB
kthr     memory             page              faults        cpu
----- ----------- ------------------------ ------------ -----------
 r  b      avm   fre  re  pi  po  fr    sr  cy  in    sy  cs us sy id wa
 5  1 2058980 158519   0   0   0   0     6   0 928    22 153  0  0 99  0
 8  0 2058985 158513   0   0   0   0     0   0 931   426 152 99  0  0  0
 8  0 2058985 158513   0   0   0   0     0   0 929   417 152 99  0  0  0
 8  0 2058985 158513   0   0   0   0     0   0 929   417 152 99  0  0  0
 8  0 2058985 158513   0   0   0   0     0   0 929   419 151 99  0  0  0
 8  0 2058985 158513   0   0   0   0     0   0 930   418 153 99  0  0  0
 8  0 2058985 158513   0   0   0   0     0   0 930   418 152 99  0  0  0
 8  0 2058986 158508   0   0   0   0     0   0 931   457 157 99  0  0  0
 8  0 2058986 158224   0  56   0   0     0   0 985   418 264 99  0  0  0
 9  0 2058986 158131   0  18   0   0     0   0 949   420 191 99  0  0  0
```

Allineamento naturale dei dati

- Accesso allineato: **`double a;`**



- Accesso disallineato : **`double a;`**



Accessi disallineati

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

Accessi allineati e disallineati (1)

- Test “sintetico”: i tempi si riferiscono ad un PentiumIII a 700 Mhz…

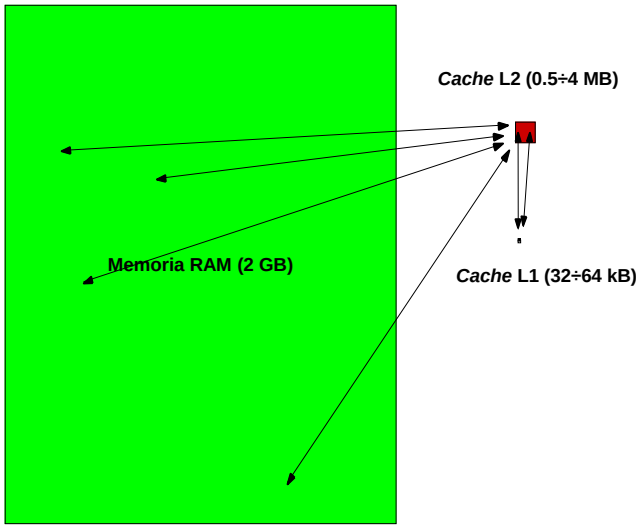
```
   #define ND 1000000
   double *a, *b; // sizeof(double) == 8 !!
....
   // alloco opportunamente aree puntate da a e b
   for(j = 0; j < 300; ++j)
      for(i = 0; i < ND; ++i)
         somma1 += (a[i]-b[i]);
....
```

- **a** multiplo di 8, **b** multiplo dispari di 4 → **t = 2.74”**
- **a** e **b** multipli dispari di 4 → **t = 3.75”**
- **a** e **b** multipli di 8 → **t = 1.41”**

Accessi allineati e disallineati (2)

- Codice reale: flusso attraverso una valvola cardiaca
- Intel Itanium2@1.5 Ghz, compilatore Intel 9.0
  - variabili non allineate: → t = 2100’’
  - variabili allineate a mano → t = 161’’
  - con opzione di compilazione (align all) → t = 162’’
- IBM Power5@1.9 Ghz
  - variabili non allineate: → t = 145’’
  - variabili allineate a mano → t = 142’’
- Dipende molto da architettura e compilatore…

Memoria RAM e *cache*



Trend tecnologici (1)

- SDRAM (memoria *cache*)

	1980	1985	1990	1995	2000	
$/MB	19200	2900	320	256	100	÷192
latenza	300	150	35	15	3	÷100
MB	0	0.004	0.256	0.512	1	-

- DRAM (memoria RAM)

	1980	1985	1990	1995	2000	
$/MB	8000	880	100	30	1	÷8000
latenza	375	200	100	70	60	÷6
MB	0.064	0.256	4	16	64	×1000

Trend tecnologici (2)

- CPU (Intel)

	1980	1985	1990	1995	2000	
CPU	8080	80286	80386	Pentium	P-III	
Clock MHz	1	6	20	150	600	×600
Cycle time ns	1000	166	50	6	1.6	÷600

- Dati tratti da Bryant & O'Hallaron

La *cache*

- La *cache* è composta di uno (o più) livelli di memoria intermedia, abbastanza veloce ma piccola (kB ÷ MB)
- Principio fondamentale: si lavora sempre su un sottoinsieme ristretto dei dati
  - dati che servono nella memoria ad accesso veloce
  - dati che (per ora) non servono nei livelli più lenti
- **Località temporale**: ogni variabile è usata più volte
- **Località spaziale**: se si accede una locazione di memoria, a breve si accederanno quelle vicine
- Limitazioni:
  - accesso casuale senza riutilizzo
  - non è mai abbastanza grande…
  - più è veloce, più scalda e… costa → gerarchia di livelli intermedi

Cache: qualche stima quantitativa

Latenze per IBM Power3 (*clock* 375 MHz!):

Livello	Dimensioni	Costo di un accesso
L1	64+32kB	1 cicli di *clock*
L2	4 MB	7 cicli di *clock*
RAM	> 1 GB	36 cicli di *clock*

- 100 accessi con 100% *cache* hit: → t = 100
- 100 accessi con 5% *cache miss* in L1 → t = 130
- 100 accessi con 10% *cache miss* in L1 → t = 160
- 100 accessi con 10% *cache miss* in L2 → t = 450
- 100 accessi con 100% *cache miss* in L2 → t = 3600

17

Cache: a *clock* più alti?

- I *clock* delle CPU sono ora di alcuni GHz
- La *cache* L1 è sullo stesso *chip*, fortemente accoppiata
- *Cache* L2 ed L3 sullo stesso *chip*, RAM esterna
  Intel Itanium 2@1.5GHz:

Livello	Dimensioni	Costo di un accesso
L1	16 + 16KB	3 cicli di *clock*
L2	256 KB	7 cicli di *clock*
L3	1.5 ÷ 24 MB	14 cicli di *clock*
RAM	> 1 GB	210 cicli di *clock*

- 100 accessi con 100% *cache* hit: → t = 300
- 100 accessi con 5% *cache miss* in L1 → t = 320
- 100 accessi con 10% *cache miss* in L2 → t = 410
- 100 accessi con 10% *cache miss* in L3 → t = 2370
- 100 accessi con 100% *cache miss* in L3 → t = 21000

18

Cache miss in tutti i livelli

1. cerco due dati, **A** e **B**
2. cerco **A** nella *cache* di primo livello (L1) 3 cicli
3. cerco **A** nella *cache* di secondo livello (L2) 4 cicli
4. trovo l'indirizzo fisico di A (fingiamo 0 cicli)
5. copio **A** dalla RAM alla L2 alla L1 ai registri 210 cicli
6. cerco **B** nella *cache* di primo livello (L1) 3 cicli
7. cerco **B** nella *cache* di secondo livello (L2) 4 cicli
8. trovo l'indirizzo fisico di A (fingiamo 0 cicli)
9. copio **B** dalla RAM alla L2 alla L1 ai registri 210 cicli
10. eseguo l'operazione richiesta

Almeno 430 cicli di *overhead*!!!

Cache hit in L1

1. cerco due dati, **A** e **B**
2. cerco **A** nella *cache* di primo livello (L1) 3 cicli

6. cerco **B** nella *cache* di primo livello (L1) 3 cicli

10. eseguo l'operazione richiesta

Solo 6 cicli di *overhead*!!!

Dimensione e riutilizzo dei dati

- Prodotto matrice-matrice in doppia precisione
- Versioni alternative, differenti chiamate di libreria BLAS
- Prestazioni in MFlops, un processore IBM Power4

Dimensioni	1 DGEMM	*N* DGEMV	N^2 DDOT
125	2050	993	401
250	2475	1411	389
500	2628	701	106
750	2802	675	100
1000	2709	552	61
1500	2725	681	50
2000	2959	613	47

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

Località spaziale: ordine di accesso

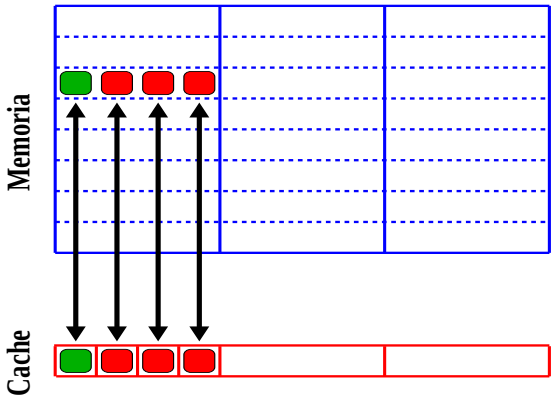
Prodotto di matrici in doppia precisione, 512×512
MFlops misurati su IBM Power4

Ordine indici	Fortran	C
`i,j,k`	15	17
`i,k,j`	7.3	372
`k,i,j`	7.5	191
`k,j,i`	190	7.6
`j,i,k`	17	17
`j,k,i`	373	7.4

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

Località spaziale e righe di *cache*

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

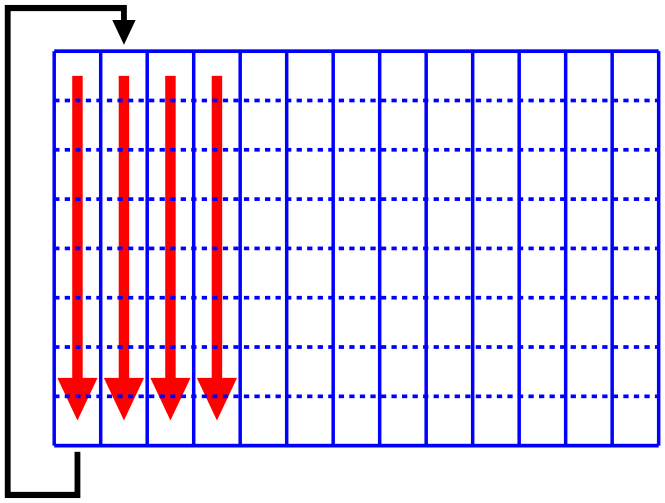


Layout degli *array* in memoria

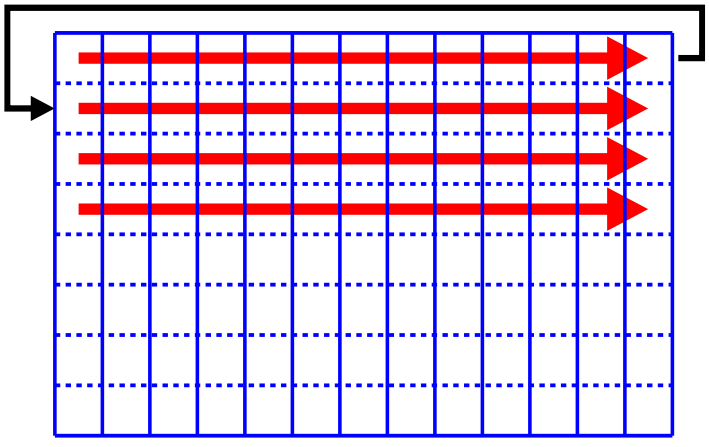
- Memoria → sequenza lineare di locazioni elementari
- Matrice *A*, elemento a_{ij}: *i* è l'indice di riga, *j* di colonna
- Le matrici sono rappresentate con *array*

- 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)`
- *Stride*
  - distanza tra due dati successivamente acceduti, indipendentemente dalla dimensione del tipo
  - due elementi consecutivi di un *array* sono "a *stride* 1"

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)
   enddo
enddo
```

- C

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
         d[i] = a[i]+b[i][j]*c[j];
```

Il compilatore: *index reordering*

Per un *loop nest* “semplice” ci pensa il compilatore

- HP alpha EV6@500

```
for(i = 0; i < n; ++i)
   for(k = 0; k < n; ++k)
      for(j = 0; j < n; ++j)
         c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Tempi in secondi:
  - `i,k,j = 5.23''`
  - `j,i,k = 5.21''`
  - `j,k,i = 5.26''`

Il compilatore: *index reordering*

Per un *loop nest* “complicato”, no

- HP alpha EV6@500

```
for(ii = 0; ii < n; ii+=step)
   for(kk = 0; kk < n; kk+=step)
      for(jj = 0; jj < n; jj+=step)
         for(i = ii; i < ii+step; ++i)
            for(k = kk; k < kk+step; ++k)
               for(j = jj; j < jj+step; ++j)
                  c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Tempi in secondi:
  - `i,k,j = 7.51’’` ☺☺☺
  - `j,k,i = 2147’’` ☹☹☹

Un caso reale in C

- Codice per campi elettromagnetici (cellulari)
- Aggiorna in sequenza tutti gli elementi di una matrice a tre indici

- Qual‘è lo *stride*?
- L’ordine dei *loop* è corretto?

```
/* declared for n_Na & Na_rate & Initialize_Na*/
double    Na_rn_exp[5][MAX_NCH][MAX_SEG];
double    Na_nint_t[5][MAX_NCH][MAX_SEG];
double    Na_nint_t1_next[5][MAX_NCH][MAX_SEG];
double    Na_num[MAX_ND][MAX_SEG];
double    I_Na[MAX_ND][MAX_SEG];
...

for (k=0; k < MAX_SEG; k++)
     for (j=0; j < MAX_NCH; j++)
          for (i=0; i < 5; i++) {
               Na_rn_exp[i][j][k] = ....
               ....
}
```

Con poco sforzo, un fattore…

- Tempo di esecuzione del codice originale: **1073''**
- Tempo di esecuzione del codice modificato: **102''**

```
/* declared for n_Na & Na_rate & Initialize_Na*/
double    Na_rn_exp[MAX_SEG][MAX_NCH][5];
double    Na_nint_t[MAX_SEG][MAX_NCH][5];
double    Na_nint_t1_next[MAX_SEG][MAX_NCH][5];
double    Na_num[MAX_SEG][MAX_ND];
double    I_Na[MAX_SEG][MAX_ND];
...

for (k=0; k < MAX_SEG; k++)
     for (j=0; j < MAX_NCH; j++)
          for (i=0; i < 5; i++) {
               Na_rn_exp[k][j][i] = ....
               ....
}
```

Località spaziale: sistema lineare

Soluzione sistema triangolare

- $\boldsymbol{Lx} = \boldsymbol{b}$
- Dove:
  - $\boldsymbol{L}$ è una matrice $n \times n$ triangolare inferiore
  - $\boldsymbol{x}$ è un vettore di n incognite
  - $\boldsymbol{b}$ è un vettore di n termini noti

- Questo sistema è risolvibile tramite:
  - *forward substitution*
  - partizionamento della matrice

Qual'è più veloce?

Perché?

Partizionamento della matrice

Soluzione sistema triangolare tramite partizionamento della matrice:

```
  .....
for(j = 0; j < n; ++j) {
   b[j] = b[j]/L[j][j];
   for(i = j+1; i < n; ++i)
      b[i] = b[i] – L[i][j]*b[j];
}
.....
```

Forward substitution

Soluzione sistema triangolare tramite *forward substitution*:

```
  .....
for(i = 0; i < n; ++i) {
   for(j = 0; j < i-1; ++j)
      b[i] = b[i] – L[i][j]*b[j];
   b[i] =  b[i]/L[i][i];
}
.....
```

Cosa è cambiato?

- *Forward substitution*

```
for(i = 0; i < n; ++i) {
   for(j = 0; j < i-1; ++j)
      b[i] = b[i] – L[i][j]*b[j];
   b[i] =  b[i]/L[i][i];
}
```

- Partizionamento della matrice

```
for(j = 0; j < n; ++j) {
   b[j] = b[j]/L[j][j];
   for(i = j+1; i < n; ++i)
      b[i] = b[i] – L[i][j]*b[j];
}
```

- Stesso numero di operazioni, ma tempi molto differenti (più di un fattore 6)
- 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

mentre in Fortran è memorizzata:

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

Dimensioni del problema

Le dimensioni del problema posso avere impatti sulle prestazioni

- Prestazioni di un codice cinetico
- Itanium2@1300 Mhz, 6MB cache
- Milioni di elementi aggiornati al secondo (Mlups)

Reticolo	500*100	550*100	600*120	650*130	700*140	750*150	800*160	850*170
Size (MB)	3.5	4.1	4.9	5.8	6.7	7.7	8.7	10
Mlups	30.2	30.2	29.6	29.0	26.2	22.5	17.8	15.4

Dimensione e riutilizzo dei dati

- Prodotto matrice-matrice in doppia precisione
- Versioni alternative, differenti chiamate di libreria BLAS
- Prestazioni in MFlops, un processore IBM Power4

Dimensioni	1 DGEMM	*N* DGEMV	N^2 DDOT
125	2050	993	401
250	2475	1411	389
500	2628	701	106
750	2802	675	100
1000	2709	552	61
1500	2725	681	50
2000	2959	613	47

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

Località spaziale e temporale

- Trasposizione di matrice:

```
for(i = 0; i < n; ++i)
   for(j = 0; j < n; ++j)
      a[i][j] = b[j][i];
```

- Quale è l'ordine dei 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 (v. anche prodotto matrice-matrice)

Cache blocking

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

```
for(ii = 0; ii < n; ii += step)
   for(jj = 0; jj < n; jj += step)
      for(i = ii; i < ii+step; ++i)
         for(j = jj; j < jj+step; ++j)
            a[i][j] = b[j][i];
```

Cache blocking: vantaggi

- Trasposizione matrice 4096×4096
- Tempi di esecuzione su IBM Power4 per due livelli di ottimizzazione (in secondi):

`step`	`-04`	`-05`
Unblocked	2.00	0.45
1	3.24	3.52
2	1.83	1.98
4	0.73	0.94
8	0.70	0.83
16	0.61	0.69
32	0.49	0.54
64	0.45	0.46
128	1.61	1.83

41

Prodotto di matrice con *blocking*

Matrici in doppia precisione, 1024×1024
MFlops misurati su IBM Power4

Block-size	MFlops
Unblocked	390
1	94
2	257
4	253
8	273
16	390
32	567
64	600
128	381
256	349

42

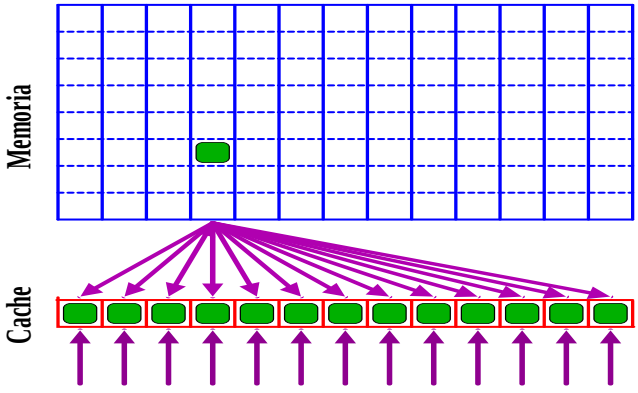
Cache: capacity miss & thrashing

- 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 *thrashing*:
  - per caricare nuovi dati si getta via una riga prima che sia stata completamente utilizzata
  - è più lento che non avere *cache*
- Capita quando più flussi di dati/istruzioni insistono sulle stesse righe di *cache*

- Dipende dal *mapping* memoria ←→ *cache*
  - *fully associative cache*
  - *direct mapped cache*
  - *N-way set associative cache*

Fully associative cache

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



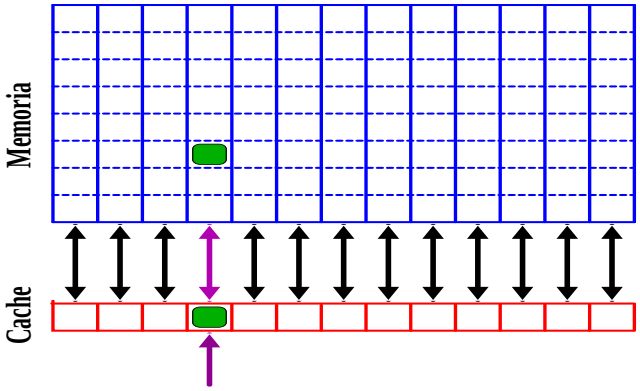
Fully associative cache

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

45

Direct mapped cache

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



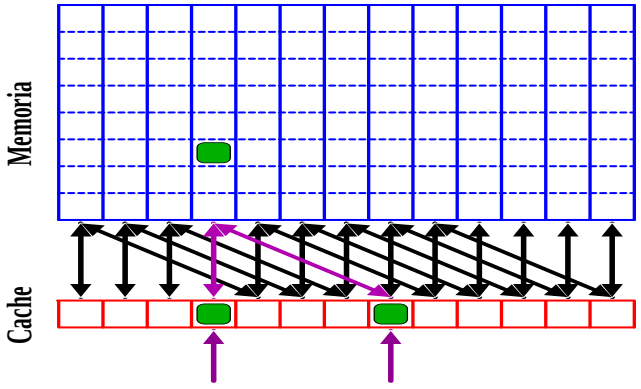
46

Direct mapped cache

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

N-way set associative cache

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



N-way set associative cache

- Pro:
  - è un punto di bilanciamento intermedio
    - *N* = 1 → *direct mapped*
    - *N* = numero di righe di *cache* → *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 thrashing*

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 *thrashing* vanno affrontati
  - le tecniche sono le stesse
  - controllo della disposizione dei dati in memoria
  - controllo delle sequenze di accessi in memoria

- Le *cache* L1 lavorano 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 ad esecuzione
  - si controllano a livello di sistema operativo

Cache trashing

- Problemi di *layout* ed accesso dei 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 thrashing: `C[i] = A[i]+B[i]`

- Iterazione i=0
  1. Cerco `A[0]` nella cache di primo livello (L1) -> **cache miss**
  2. Recupero `A[0]` nella memoria RAM
  3. Copio da `A[0]` a `A[7]` nella L1
  4. Copio `A[0]` in un registro
  5. Cerco `B[0]` nella cache di primo livello (L1) -> **cache miss**
  6. Recupero `B[0]` nella memoria RAM
  7. Copio da `B[0]` a `B[7]` nella L1
  8. Copio `B[0]` in un registro
  9. Eseguo somma
- Iterazione i=1
  1. Cerco `A[1]` nella cache di primo livello (L1) -> **cache hit**
  2. Copio `A[1]` in un registro
  3. Cerco `B[1]` nella cache di primo livello (L1) -> **cache hit**
  4. Copio `B[1]` in un registro
  5. Eseguo somma
- Iterazione i=2

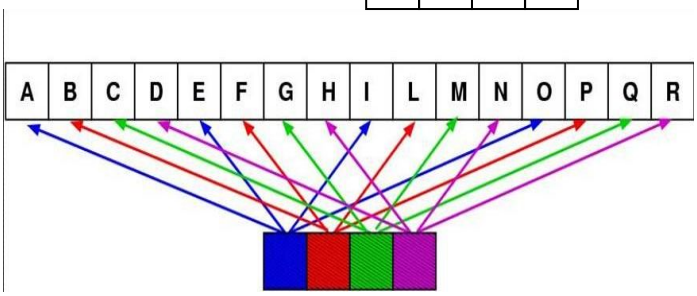
Thrashing: `C[i] = A[i]+B[i]`

- Iterazione i=0
  1. Cerco **`A[0]`** nella cache di primo livello (L1) -> **cache miss**
  2. Recupero **`A[0]`** nella memoria RAM
  3. Copio da **`A[0]`** a **`A[7]`** nella L1
  4. Copio **`A[0]`** in un registro
  5. Cerco **`B[0]`** nella cache di primo livello (L1) -> **cache miss**
  6. Recupero **`B[0]`** nella memoria RAM
  7. Scarico la riga di cache che contiene **`A[1]-A[8]`**
  8. Copio da **`B[0]`** a **`B[7]`** nella L1
  9. Copio **`B[0]`** in un registro
  10. Eseguo somma
- Iterazione i=1
  1. Cerco **`A[1]`** nella cache di primo livello (L1) -> **cache miss**
  2. Recupero **`A[1]`** nella memoria RAM
  3. Scarico la riga di cache che contiene **`B[1]-B[8]`**
  4. Copio da **`A[0]`** a **`A[7]`** nella L1
  5. Copio **`A[1]`** in un registro
  6. Cerco **`B[1]`** nella cache di primo livello (L1) -> **cache miss**
  7. Recupero **`B[1]`** nella memoria RAM
  8. Scarico la riga di cache che contiene **`A[1]-A[8]`**
  9. Copio da **`B[0]`** a **`B[7]`** nella L1
  10. Copio **`B[1]`** in un registro
  11. Eseguo somma
- Iterazione i=2

Cache thrashing: no padding

Succede tipicamente con *array* le cui dimensioni sono potenze di 2

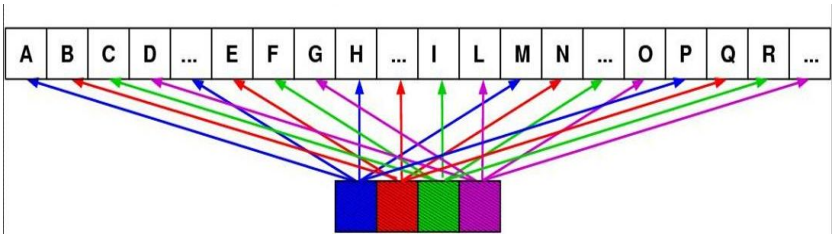
A	**B**	**C**	**D**
E	**F**	**G**	**H**
I	**L**	**M**	**N**
O	**P**	**Q**	**R**



Padding → no thrashing

Elementi "*dummy*" aggiunti ad ogni colonna "distruggono" il *pattern* di accesso inefficiente

A	**B**	**C**	**D**	**…**
E	**F**	**G**	**H**	**…**
I	**L**	**M**	**N**	**…**
O	**P**	**Q**	**R**	**…**



Come identificarlo?

Effetto variabile in funzione della dimensione del *data set*

Codice cinetico:
- Reticolo $(n+2)^3$
- Tempo per sito costante

n	Tempo/sito
57	1.65e-6
58	1.66e-6
59	1.66e-6
60	1.66e-6
61	1.67e-6
62	2.32e-6
63	1.65e-6
64	1.66e-6
65	1.65e-6
66	1.66e-6
67	1.66e-6

Equazione di Helmoltz

Dati misurati su IBM Power4 @ 1100 Mhz:

Dimensioni	Iterazioni	*Cache miss* (%)	secondi
509×509	48991	0.74	217''
510×510	48847	0.70	218''
511×511	48781	0.84	243''
512×512	48715	1.04	267''
513×513	48647	0.84	250''
514×514	48578	0.71	230''
515×515	48508	0.75	228''

512 (pad=8)	48715	0.80	214''
512 (pad=0)	48715	1.04	267''

Prodotto di matrici: *blocking&padding*

Matrice in doppia precisione, 1024×1024
MFlops su IBM Power4
Padding di 16 elementi

block-size	*padding* = 0	*padding* = 16
unblocked	390	388
1	94	99
2	257	305
4	253	395
8	273	524
16	390	623
32	567	713
64	600	692
128	381	459
256	349	454

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: HPMToolkit
  - Intel: VTune
- Esistono strumenti di misura multipiattaforma
  - PAPI
  - PCL

– Analizzatori di accessi in memoria: Valgrind, SLO

Cache thrashing con *HW counter*

Cache thrashing → aumento dei *cache miss* (codice cinetico)

n	Tempo/sito	miss/sito
57	1.65e-6	2.48
58	1.66e-6	2.48
59	1.66e-6	2.47
60	1.66e-6	2.47
61	1.67e-6	2.47
62	2.32e-6	3.37
63	1.65e-6	2.47
64	1.66e-6	2.47
65	1.65e-6	2.47
66	1.66e-6	2.47
67	1.66e-6	2.46

Con gli *array* è facile…

- … ma con strutture dati complesse?
  - *linked list*
  - alberi
  - *hash table*
- Singolo “nodo” più piccolo di una riga di *cache*
  - valutare uso di *array* o *heap*
  - non malloc() nodo per nodo, ma di blocchi di nodi
  - *array* di puntatori piuttosto che liste
  - liste di *array* di nodi piuttosto che liste di nodi
  - …
- Dipende tutto dai “*pattern*” di accesso ai dati
  - valutare possibilità di riorganizzazioni periodiche

La *cache* è una memoria

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