Calcolo Parallelo e Distribuiuto mod.B

Lezione 3
Algoritmi a blocchi per l'algebra lineare numerica

Dalla precedente lezione

- Le prestazioni di una applicazione dipendono fortemente dal rapporto $q=N_{mem}/N_{flop}$

$$Perf = \frac{1/t_{flop}}{1+\left(\frac{N_{mem}}{N_{flop}}\frac{t_{mem}}{t_{flop}}\right)} = \frac{Perf\ *}{1+\left(\frac{N_{mem}}{N_{flop}}\frac{t_{mem}}{t_{flop}}\right)}$$

- La disponibilita' di piu' livelli di cache permette di ridurre N_{mem}
- Necessaria una riorganizzazione degli algoritmi

Una macchina virtuale per l'Algebra Lineare

- La disponibilita' di routine efficienti per le operazioni di base ci permette di riorganizzare gli algoritmi in questo contesto

Macchina Virtuale per l'Algebra Lineare

BLAS
(Basic Linear Algebra Subprograms)

BLAS e' un esempio di tecnologia software che ha l'obiettivo di rendere piu' efficienti le applicazioni sovrastanti

Breve storia di BLAS

- Anni '60 – inizio '70: algoritmi "naïve" (sequenze di "for")
  - es. Libreria EISPACK (calcolo di autovalori $Ax=\lambda x$)
- 1973: sviluppo di BLAS (1)
  - Libreria di 15 operazioni su vettori
  - "AXPY" (y = a·x + y), prod.scalare, scale (x = a·x), etc
  - 4 versioni (S/D/C/Z), 46 routines, 3300 linee di codice
  - obiettivi
    - Accesso uniforme alle routine, leggibilita', documentazione
    - Robustezza, attenta scrittura delcodice, attenzione all'aritmetica
    - Portabilita' ed efficienza attraverso implementazioni specifiche
  - Calcolo di q per BLAS 1 : $O(N^1)$ comm su $O(N^1)$ flop ➔ $q \sim O(1)$
  - Usata in librerie come LINPACK (sistemi lineari densi)

Breve storia di BLAS

- Limiti di BLAS-1
  - Valore di q troppo grande (q~1 richiede $t_{mem} \sim t_{flop}$ per avere $\frac{1}{2}$ della Perf*)
  - Utile per calcolatori vettoriali come il CRAY X-MP/12 (SIMD) ma poco utile per i supercalcolatori degli anni '80

- 1984: sviluppo di BLAS-2
  - Libreria di 25 operazioni quasi tutte tipo matrici/vettore
    - "GEMV": $y = \alpha \cdot A \cdot x + \beta \cdot x$, "GER": $A = A + \alpha \cdot x \cdot y^T$, "TRSV": $y = T^{-1} \cdot x$
    - 4 versioni (S/D/C/Z), 66 routines, 18K linee di codice
  - Calcolo di q per BLAS 2 : $O(N^2)$ comm su $O(N^2)$ flop ➔ q~O(1)
    - Ancora poco efficiente per macchine con cache memory

Breve storia di BLAS

- 1987: BLAS-3
  - Libreria di 9 operazioni quasi tutte tipo matrice/matrice
    - "GEMM": $C = \alpha \cdot A \cdot B + \beta \cdot C$, "SYRK": $C = \alpha \cdot A \cdot A^T + \beta \cdot C$, "TRSM": $C = T^{-1} \cdot B$
    - 4 versioni (S/D/C/Z), 30 routines, 10K linee di codice
  - Calcolo di q per BLAS-3: $O(N^2)$ comm su $O(N^3)$ flop
  - $q \sim (4N^2)/(4N^3) \sim 1/N$
    - Possibilita' di ottimizzare l'uso delle cache

- BLAS1/2/3 disponibile su www.netlib.org/blas
  - Solo implementazioni di riferimento non ottimizzate
    - Es: 3 loop innestati per GEMM

riassumendo

BLAS1	BLAS2	BLAS3
q=O(1)	q=O(1)	q=O(1/N)

OBIETTIVO

Nella riorganizzazione degli algoritmi
e' opportuno, ove possibile, utilizzare moduli BLAS3

Possibili implementazioni di C=C+AB

Le diverse combinazioni di indici evidenziano differenti operazioni riconducibili a BLAS

```
for __ = 1 to N
   for __ = 1 to N
      for __ = 1 to N
         C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
      endfor
   endfor
endfor
```

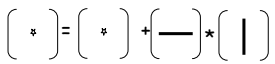
6 combinazioni di indici

- i j k
- j i k
- i k j
- j k i
- k i j
- k j i

Qualche esempio: forma ijk

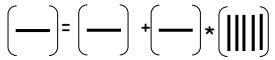
```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

N^2 — DOT (BLAS1)



```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

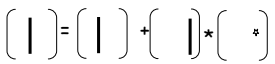
N — matvet (BLAS2)



Qualche esempio: forma jki

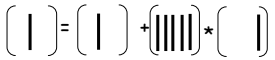
```
for j = 1 to N
    for k = 1 to N
        for i = 1 to N
            C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

N^2 — SAXPY (BLAS1)



```
for j = 1 to N
    for k = 1 to N
        for i = 1 to N
            C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

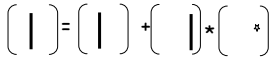
N — matvet (BLAS2)



Qualche esempio: forma kji

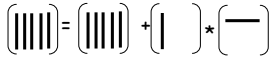
```
for k = 1 to N
    for j = 1 to N
        for i = 1 to N
            C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

N^2 — SAXPY (BLAS1)



```
for k = 1 to N
    for j = 1 to N
        for i = 1 to N
            C (i,j) = β*C(i,j)+ α*A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

N — rank2 (BLAS2)

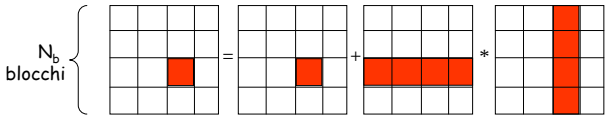


Problema

- Le forme proposte non utilizzano moduli BLAS3
  - Utilizzando BLAS1 e BLAS2 si ha q~O(1)
  - A meno che non impieghiamo 1 sola chiamata alla routine per il prodotto C=C+AB
    - Ma se le matrici non entrano in cache le prestazioni si riducono

Soluzione

Suddividere le matrici a blocchi



$$C(I,J) = \sum_{K=1}^{N_b} A(I,K) * B(K,J)$$

Blocchi di dimensione N/N_b

C=C+AB a blocchi

```
for ii = 1 to Nb
    for jj = 1 to Nb
        for kk = 1 to Nb
            C (ii,jj) = β * C(ii,jj) + α* A(ii,kk) * B(kk,jj)
        endfor
    endfor
endfor
```

N_b^3

DGEMM (BLAS3)

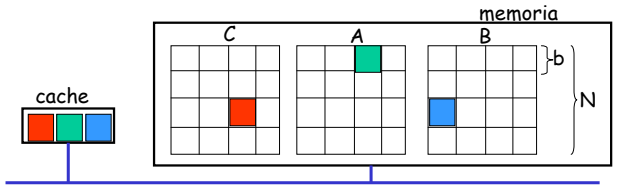
Esempio di algoritmo per l'algebra lineare a blocchi

Analisi dell'algoritmo

- Sia M_{cache} la dimensione della cache memory
- Dividiamo le matrici A, B e C in blocchi di ordine b tali che

$M_{cache} > 3\ b^2$

(cioe' la cache e' capace di contenere 3 blocchi)



Algoritmo per il prodotto di matrici a blocchi

Siano A,B,C matrici di ordine N,
divise in N_bxN_b blocchi di ordine b con N_b=N/ b

```
for ii = 1 to Nb
  for jj = 1 to Nb
    {leggi C(ii,jj) nella cache}
    for kk = 1 to Nb
          {leggi A(ii,kk) nella cache}
          {leggi B(kk,jj) nella cache}
          C(ii,jj) = βC(ii,jj) + αA(ii,kk) * B(kk,jj)
    }
    {scrivi C(i,j) in memoria centrale}
  }
}
```


Analisi dell'algoritmo

N_{mem} : comunicazioni tra cache e memoria
N_{float} e' il numero di operazioni floating point: $4N^3$ nel nostro caso
A, B e C hanno NxN elementi, e N_bxN_b blocchi di ordine b=N/N_b

$N_{mem} = N_b^3 * b^2$ leggi un blocco di B N_b^3 volte
$+ N_b^3 * b^2$ leggi un blocco di A N_b^3 volte
$+ 2N^2$ leggi e scrivi un blocco di C 1 volta
$= 2(N_b + 1) * N^2$

$q = N_{mem} / N_{float} = / (2(N_b + 1) * N^2)/4N^3$
$\approx N_b / 2N = 1/2b$ (per N grande)

E' possibile aumentare le prestazioni aumentando b

Analisi dell'algoritmo

- Le prestazioni crescono con la dimensione del blocco b
- Limite: tre blocchi di A,B,C devono entrare nella cache, cosi' b non puo' essere arbitrariamente grande
- Poiche $3b^2 \le M_{cache}$, si ha $b \le (M_{fast}/3)^{1/2}$

PROBLEMA:
Quanto deve essere grande M_{fast} per avere una prestazione di meta' della peak performance Perf*?

Dimensione della cache

- Si ha

$$M_{cache} \ge 3b^2 \approx \frac{3}{2q^2} = \frac{3}{2}\left(\frac{t_{mem}}{t_{flop}}\right)^2$$

		KB
	t_m/t_f	richiesti
Sun Ultra 2i	24,8	7,2
Sun Ultra 3	14	2,3
Pentium 3	6,25	0,5
Pentium3M	10	1,2
IBM Power3	8,75	0,9
IBM Power4	15	2,6
Itanium1	36	15,2
Itanium2	5,5	0,4

18

Limitazioni

- La precedente analisi mostra che per una operazione $C=\beta C+\alpha AB$ il valore di q e':

$$q \approx 1/2b \le (4M_{fast}/3)^{-1/2}$$

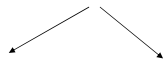
- Esiste un risultato che dstabilisce un limite per q
- Teorema (Hong & Kung, 1981): Qualunque riorganizzazione di questo algoritmo e' limitata a

$$q = O((M_{fast})^{-1/2})$$

19

Automatic performance tuning

- Un problema essenziale nell'utilizzo di BLAS e' definire i valori dei parametri legati alle dimensioni dei livelli di memoria
- Come definire L?



Implementazione sviluppata dai produttori hardware

Ottimizzazione automatica del codice

Implementazione dei produttori

- I maggiori produttori di CPU, sulla base della conoscenza specifica dell'hardware, hanno sviluppato:
  - Libreria Intel Math Kernel Library (MKL)
    - Ottimizzata per processori Intel
    - Contiene BLAS1, 2 e 3, FFT, Sparse Solver, LAPACK, …
  - Libreria AMD Core Mathematical Library (ACML)
    - Ottimizzata per processori AMD
    - Contiene BLAS1, 2 e 3, FFT, LAPACK, …

Ottimizzazione di un codice (tuning)

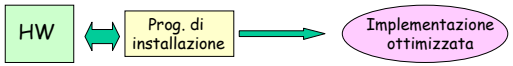
E' una impresa dispendiosa in termini di tempo e di risorse uomo

- Differenze tra le CPU in termini di
  - clock,
  - dimensioni e gestione delle cache,
  - numero e profondita' delle pipeline di calcolo
- necessaria una conoscenza dettagliata e profonda dell'ambiente
- Inoltre per ogni processore e' necessario ripetere il processo di ottimizzazione.

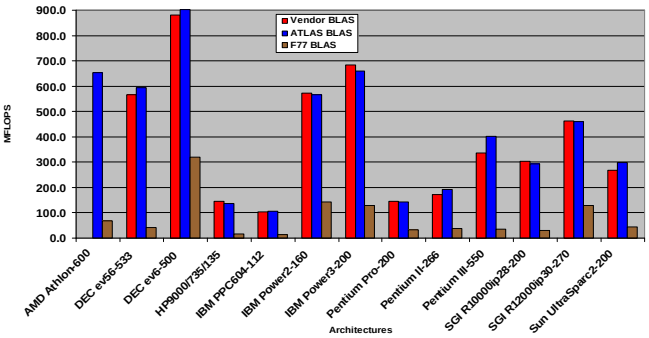
- IDEA: farlo fare alla libreria stessa

Automatic Performance Tuning

- obiettivo: far determinare al programma di installazione le caratteristiche del sistema
  - Numerosi test e benchmark in fase di installazione
  - Eseguiti una sola volta off-line e sempre valida sulla stessa architettura
  - Impiega circa un'ora per definire il codice ottimizzato
  - In generale approccio utile per algoritmi con esecuzione prevedibile e "ben strutturata" che non dipende dai dati (tolleranza, grado di sparsita',…)
  - Es. Automatic Tuning Linear Algebra Subprograms (ATLAS)



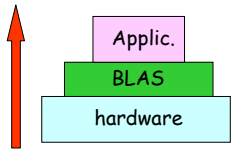
ATLAS (C=C+AB, n=500)



- ATLAS e' piu' veloce di ogni implementazione portabile "scritta a mano" ed ha prestazioni paragonabili a quelle specifiche per la macchina fornite dai vendors
- ATLAS e' scritta in C (www.netlib.org/atlas)

Andando verso l'alto..

- Avendo a disposizione una libreria di base efficiente per l'algebra lineare possiamo pensare di risolvere problemi piu' complessi (LU, LL^T, $Ax=\lambda x$,..)



E' necessario riscrivere le applicazioni in termini di operazioni di base

Possibilita' di sviluppo di Algoritmi a Blocchi

Esempio: fatt. LU in Linpack (1976)

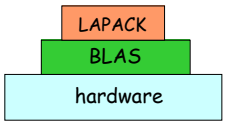
```
.....
for k = 1 to n-1
     max=A(k,k)      L=k
     for i = k+1 to n
          if (A(i,k) > max)
                    max=A(i,k)
                    L=i
          endif
     endfor
     for j = k to n
          t = A(L,j)
          A(L,j) = A(k,j)
          A(k,j) = t
     endfor
     for i = k+1 to n
          A(i,k) = A(i,k) / A(k,k)
     endfor
     for j = k+1 to n
          for i = k+1 to n
               A(i,j) = A(i,j) – A(i,k)*A(k,j)
          endfor
     endfor
endfor
```

```
NM1 = N-1
DO 60 K = 1, NM1
     KP1 = K + 1
     L = ISAMAX(N-K+1,A(K,K),1) + K +
     1
          T = A(L,k)
          A(L,k) = A(k,k)
          A(k,k) = T
          .....

          T = -1.0E0/A(k,k)
          CALL SCAL(N-K,T,A(K+1),1)
     DO 30 J = KP1, N
          T = A(L,J)
          A(L,J) = A(K,J)
          A(K,J) = T
     CALL SAXPY(N-
     K,T,A(K+1,K),1,A(K+1,J),1)
     CONTINUE
     .....
60   CONTINUE
```

Breve storia di LAPACK (dal 1989 ad oggi)

- LAPACK (1989) "Linear Algebra PACKage" - usa BLAS-3
  - Contenuti di LAPACK
    - Algoritmi che possono essere riscritti quasi per intero con BLAS3
      - Sistemi Lineari: calcola x tale che Ax=b
      - Minimi quadrati: calcola x che minimizza minimize $||r||_2 \equiv \sqrt{\Sigma r_i^2}$ dove r=Ax-b
    - Algoritmi che possono essere riscritti per il 50% con BLAS3
      - "autovalori": calcola λ e x tale che $Ax = \lambda x$
      - Singular Value Decomposition (SVD): $A^TAx=\sigma^2x$
    - Numerose varianti (A matrice a banda, $A=A^T$, etc)
    - 1582 routines, 490K linee di codice,
    - Ultima versione nel 2008, scritta per le nuove architetture
  - www.netlib.org/lapack



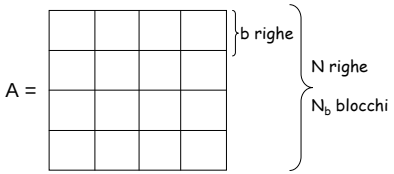
Algoritmo di cholesky a blocchi

- L'algoritmo di Cholesky effettua la fattorizzazione

$A=LL^T$

di una matrice A simmetrica definita positiva di ordine N

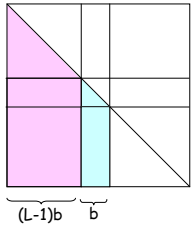
- L e' una matrice triangolare inferiore
- Suddivisione della matrica A a blocchi



Blocchi di ordine b
$N_b \times N_b$ blocchi

$N_b = N/b$

Alg. di Cholesky a blocchi (left looking)



Ogni passo calcola un blocco di colonne

L'algoritmo impiega N_b passi

Supponiamo di aver gia' fatto L-1 passi

(L-1)b colonne gia' calcolate

Bisogna calcolare il successivo blocco di b colonne (L_{22} e L_{32})

$$A=\begin{pmatrix} A_{11} & & \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{pmatrix}=\begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix}*\begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix}=LL^T$$

Alg. di Cholesky a blocchi (left looking)

- Ponendo A=LL^T si ottiene

$$L_{21}L_{21}^T + L_{22}L_{22}^T = A_{22}$$

- Il calcolo di L_{22} consiste quindi di due passi

1. $A'_{22} = A_{22} - L_{21}L_{21}^T$ (aggiornamento di rango k)
2. $L_{22} = Chol(A'_{22})$ (fattorizzazione di A'_{22})

Alg. di Cholesky a blocchi (left looking)

- Sempre da A=LL^T si ottiene

$$L_{31}L_{21}^T + L_{32}L_{22}^T = A_{32}$$

- Il calcolo di L_{32} consiste quindi di due passi

1. $A'_{32} = A_{32} - L_{31}L_{21}^T$ (prodotto matrice-matrice)
2. $L_{32} = A'_{32}\,L_{22}^{-1}$ (sistema triangolare inf multiplo)

BLAS nell'algoritmo di Cholesky

$A'_{22} = A_{22} - L_{21}L_{21}^T$
(aggiornamento di rango k)

- DSYRK (BLAS3)

$L_{22} = Chol(A'_{22})$
(fattorizzazione LL^T)

- DPOTF2 (BLAS2)

$A'_{32} = A_{32} - L_{31}L_{21}^T$
(prodotto matrice-matrice)

- DGEMM (BLAS3)

$L_{32} = A'_{32}\,L_{22}^{-1}$
(sistema triangolare multiplo)

- DTRSM (BLAS3)

Routine DPOTRF (LAPACK)

```
.....
      DO 20 J = 1, N, B
*
*        Update and factorize the current diagonal block and test
*        for non-positive-definiteness.
*
      JB = MIN( B, N-J+1 )
      CALL DSYRK( 'Lower', 'No transpose', JB, J-1, -ONE,
     $            A( J, 1 ), LDA, ONE, A( J, J ), LDA )
      CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
      IF( INFO.NE.0 )
     $   GO TO 30
      IF( J+JB.LE.N ) THEN
*
*        Compute the current block column.
*
        CALL DGEMM( 'No transpose', 'Transpose', N-J-JB+1, JB,
     $              J-1, -ONE, A( J+JB, 1 ), LDA, A( J, 1 ),
     $              LDA, ONE, A( J+JB, J ), LDA )
        CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
     $              N-J-JB+1, JB, ONE, A( J, J ), LDA,
     $              A( J+JB, J ), LDA )
      END IF
   20 CONTINUE
.....
```

Analogamente per l'alg. di Gauss

```
      DO 20 J = 1, MIN( M, N ), NB
            JB = MIN( MIN( M, N )-J+1, NB )
*
*           Factor diagonal and subdiagonal blocks and test for exact singularity.
*
            CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ), IINFO )
*
*           Adjust INFO and the pivot indices.
*
            IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
            DO 10 I = J, MIN( M, J+JB-1 )
              IPIV( I ) = J - 1 + IPIV( I )
   10       CONTINUE
*
*           Apply interchanges to columns 1:J-1.
*
            CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )
*
            IF( J+JB.LE.N ) THEN
*
*              Apply interchanges to columns J+JB:N.
*
               CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1, IPIV, 1 )
*
*              Compute block row of U.
*
               CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
     $                     N-J-JB+1, ONE, A( J, J ), LDA, A( J, J+JB ),
     $                     LDA )
               IF( J+JB.LE.M ) THEN
*
*                 Update trailing submatrix.
*
                  CALL DGEMM( 'No transpose', 'No transpose', M-J-JB+1,  N-J-JB+1, JB, -ONE,
     $                        A( J+JB, J ), LDA, A( J, J+JB ), LDA, ONE, A( J+JB, J+JB ), LDA )
               END IF
            END IF
   20    CONTINUE
```

Fatt. LU
Routine DGETRF