Lezione 5

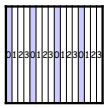
Prodotto $C=\beta C+\alpha AB$ su sistema a memoria distribuita

- Consideriamo un sistema parallelo a memoria distribuita
  - Es. Cluster, sistema MPP
  - Ogni processore ha una sua memoria privata
  - La comunicazione avviene mediante spedizione/ricezione di messaggi su una rete
  - Utilizzo di librerie di message passing
  - Es. MPI, BLACS, PVM
- Prima domanda: come distribuire le matrici tra le memorie dei differenti processori?

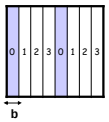
2

Alcuni esempi di distribuzione

1) Distribuzione 1D a blocchi di colonne

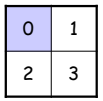


2) Distribuzione ciclica 1D di colonne

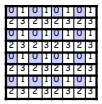


3) Distr. Ciclica 1D a blocchi dicolonne

4) Versioni a righe delle precedenti distribuzioni



5) Distribuzione 2D a blocchi



Generalizza le precedenti

6) Distribuzione ciclica 2D a blocchi

Prodotto matrici in parallelo

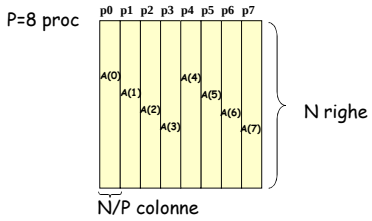
- Per semplicita' $C=C+A*B$
- Compl. Comput.: $N_{flop} = 2*N^3$
- L'algoritmo dipende da:
  - Distribuzione dei dati
  - Topologia del sistema
  - Algoritmo di comunicazione
- Semplice modello per le comunicazioni
  - **Time = latenza + dati * tempo-per-dato**

    $= \alpha + N * \beta$
- Efficienza:
  - Efficienza = Speedup/P = $T_1 / (P * T_P)$
  - Speedup lineare $\leftrightarrow$ efficienza = 1

4

1) Distribuzione 1D a blocchi colonne

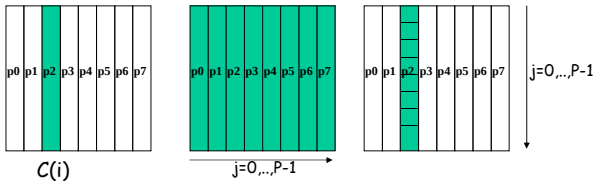
- Matrici A, B e C di ordine N, divisibile per P (per semplicita')



- A(i) e' il blocco di N/P colonne che Pi possiede (stesso per B(i) e C(i))
- B(i,j) e' un (sotto)blocco di N/P x N/P elementi di B(i)
  - Righe da j*N/P fino a (j+1)*N/P - 1

L'algoritmo

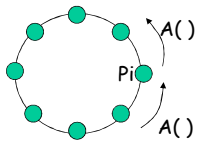
- Ogni Pi calcola
  $C(i) = C(i) + A*B(i)$ cioe' $C(i) + \Sigma_j A(j)*B(j,i)$



- Ogni processore Pi
  - possiede C(i) e B(i)
  - ma ha bisogno di tutte le colonne di A
    - (in possesso degli altri Pj)

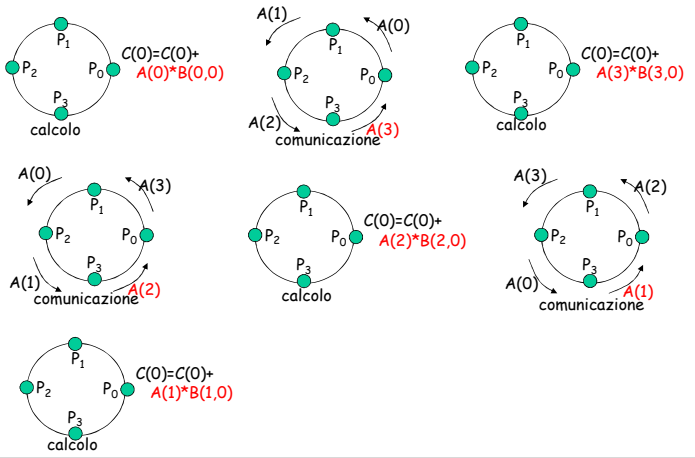
La topologia del sistema

- Consideriamo un sistema con i processori disposti logicamente ad anello (Ogni processore puo' comunicare solo con i propri vicini)
- La comunicazione puo' avvenire simultaneamente
- Ad ogni step, P_i riceve un blocco di colonne di A da P_{i-1} e invia un blocco a P_{i+1} (ovviamente … mod P)



- In P-1 step, tutte le colonne raggiungono tutti i P processori

Esempio P=4



Mettendo tutto insieme…

Algoritmo del generico Pi (modello SPMD)

```
copia  A( i ) in Tmp
C( i ) = C( i ) + Tmp*B( i  , i )
for j = 1 to P-1 {
    Send ( Tmp , i+1(mod P) )
    Receive ( Tmp , i-1 (mod p) )
    C( i ) = C( i ) + Tmp*B( i-j mod p , i )
}
```

- Tmp ha dimensione N x N/P
- B(i-j mod Pm i) ha dimensione $(N/P)^2$

tempo del ciclo interno = $2*(\alpha + \beta*N^2/P) + 2*N*(N/P)^2$

comm calc

Analisi dell'algoritmo

- Tempo del ciclo interno = $2*(\alpha + \beta*N^2/P) + 2*N*(N/P)^2$
- Tempo totale = $2*N*(N/P)^2$ + (P-1) * Tempo del ciclo interno
  $\approx 2*N^3/P + 2*P*\alpha + 2*\beta*N^2$

- Efficienza = $T_1/(P\,T_P)$ = $2*N^3$ / (P * Tempo totale)

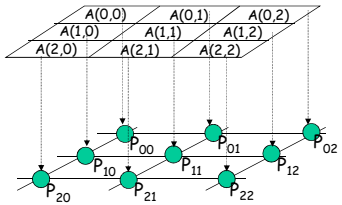
  $= 1/(1 + \alpha * P^2/N^3 + \beta * P/N)$

(N/P=cost) $= \frac{1}{1+O(P/N)}$

- Tende a 1 se P/N diminuisce

C=C+AB con distribuzione a blocchi 2D

- Consideriamo una griglia 2D di P processori periodica
  - (per semplicita' quadrata ➔ P = s x s)
- Ogni processore comunica con i 4 vicini
- Al processore $P_{i,j}$ viene assegnato il blocco A(i,j)



Distribuzione a blocchi 2D di A (analogamente per B e C)

Esempio: P = 9 = 3x3

Algoritmo di Cannon

- Disposizione iniziale

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

Ad esempio: il Proc P_{21} deve calcolare

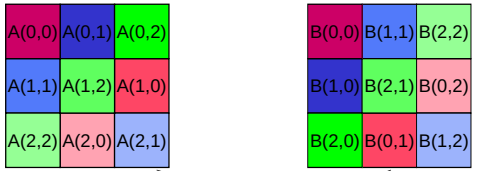
C(2,1) = C(2,1) + A(2,0)B(0,1) + A(2,1)B(1,1) + A(2,2)B(2,1)

il prodotto avviene sempre con blocchi dello stesso colore !!

E' necessario:
- spostare di 2 posizioni a sinistra i blocchi della riga 2
- spostare di 1 posizione in alto i blocchi della colonna 1

Algoritmo di Cannon

- Disposizione dopo l' "inclinazione" iniziale dei blocchi

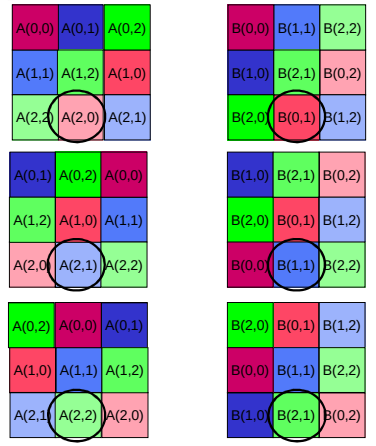


$C(2,1) = C(2,1) + A(2,0)B(0,1) + A(2,1)B(1,1) + A(2,2)B(2,1)$

Osservare:
- diagonali con lo stesso colore

A(2,0)B(0,1) + A(2,1)B(1,1) + A(2,2)B(2,1)

- P_{21} calcola A(2,0)B(0,1)
- Shift dei blocchi di A a sinistra e dei blocchi di B in alto
- P_{21} calcola A(2,1)B(1,1)
- Shift dei blocchi di A a sinistra e dei blocchi di B in alto
- P_{21} calcola A(2,2)B(2,1)



Algoritmo di Cannon (SPMD)

Ogni P_{ij} calcola

$C(i,j) = C(i,j) + \sum_k A(i,k)*B(k,j)$ s = sqrt(p) intero

ALGORITMO

shift a sinistra A(i,j) di i posizioni
cosi' che A(i,j) e' sovrascritto da A(i,(j+i)mod s)

shift in alto B(i,j) di j posizioni
cosi' che B(i,j) e' sovrascritto da B((i+j)mod s), j)

for k=0 to s-1
  C(i,j) = C(i,j) + A(i,j)*B(i,j)
  shift a sinistra le righe di A di 1 posizione
  shift sopra le colonne di B di 1 posizione
}

Analisi dell'algoritmo di Cannon

shift a sinistra A(i,j) di i posizioni	... costo ≤ $s*(\alpha + \beta*N^2/P)$
shift in alto B(i,j) di i posizioni	... costo ≤ $s*(\alpha + \beta*N^2/P)$
for k=0 to s-1	
$C(i,j) = C(i,j) + A(i,j)*B(i,j)$	... costo = $2*(N/s)^3 = 2*N^3/P^{3/2}$
shift a sinistra blocchi di A	... costo = $\alpha + \beta*N^2/P$
shift in alto blocchi di B	... costo = $\alpha + \beta*N^2/P$
endfor	

tempo totale $= 2*N^3/P + 4*s*\alpha + 4*\beta*N^2/s$

Efficienza $= 2*N^3 / (P * \text{tempo totale})$
$= 1/(1 + \alpha * 2*(s/N)^3 + \beta * 2*(s/N))$

$= 1/(1 + O(\text{sqrt}(P)/N))$

Confronto distrib. 1D - Cannon

Distrib. 1 D

$$E_P = \frac{1}{1+O(P/N)}$$

Alg. Cannon

$$E_P = \frac{1}{1+O(\sqrt{P}/N)}$$

Poiche' $P/N > \sqrt{P}/N$ si ha che l'algoritmo di Cannon e' piu' efficiente

Vantaggi e svantaggi dell'alg. di Cannon

- Quello che e' ottimale e' anche "veloce"?
  - Si: il prodotto dei blocchi avviene localmente con moduli otimizzati sequenziali
- E' difficile da generalizzare
  - P diverso da un quadrato perfetto (es. 12=3x4)
  - A e B non quadrate
  - Dimensioni di A, B non perfettamente divisibili per s=sqrt(p)
  - Dimensioni dei blocchi dipendenti da P
  - Distribuzione ciclica a blocchi
- Memoria aggiuntiva per le copie locali dei blocchi

18

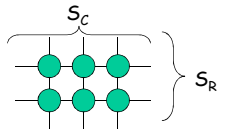
SUMMA Algorithm

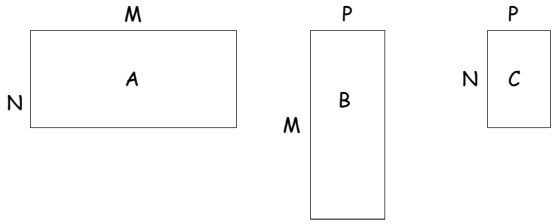
- SUMMA = Scalable Universal Matrix Multiply
- Leggermente meno efficiente dell'algoritmo di Cannon ma piu' facile da generalizzare
  - Puo' gestire ogni distribuzione, dimensione e allineamento
  - Utilizza broadcast di blocchi invece dello shift circolare
  - log p volte piu' comunicazioni dell'alg. di Cannon
  - Usa meno memoria
- usata in PBLAS = Parallel BLAS
  - www.netlib.org/lapack/lawns/lawn{96,100}.ps

19

SUMMA (caso generale)

- Sia una griglia di $P=S_R \times S_C$ processori (es: $S_R=2$ $S_C=3$)
- Matrici A, B e C di dimensioni NxM MxP NxP

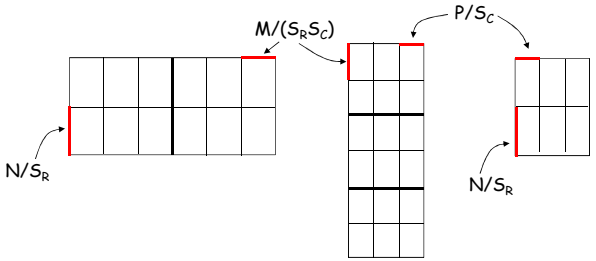




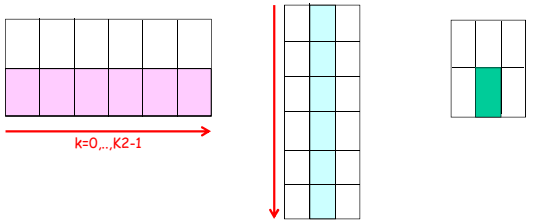
suddivisione delle matrici

- Le matrici vengono divise in:
  - matrice A : K1 x K2 blocchi — K1 = S_R — K2 = S_R S_C
  - matrice B : K2 x K3 blocchi — K2 = S_R S_C — K3 = S_C
  - matrice C : K1 x K3 blocchi — K1 = S_R — K3 = S_C

Si distribuiscono tra i nodi in maniera ciclica a blocchi rispettando la topologia dei nodi



SUMMA con distribuzione a blocchi 2D



- solo A(i,j), B(i,j) e C(i,j) sono in possesso di P_{ij}
- Ogni nodo P_{ij} calcola C(i,j) (in parallelo)
- $C(i,j) = \Sigma_k A(i,k)*B(k,j)$
- k2 passi sequenziali

Osservazione

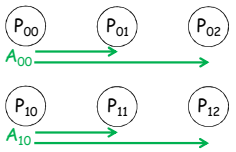
$$
\begin{aligned}
P_{00} &\rightarrow C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30} + A_{04}B_{40} + A_{05}B_{50}\\
P_{01} &\rightarrow C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31} + A_{04}B_{41} + A_{05}B_{51}\\
P_{02} &\rightarrow C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} + A_{03}B_{32} + A_{04}B_{42} + A_{05}B_{52}\\
P_{10} &\rightarrow C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} + A_{13}B_{30} + A_{14}B_{40} + A_{15}B_{50}\\
P_{11} &\rightarrow C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41} + A_{15}B_{51}\\
P_{12} &\rightarrow C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} + A_{14}B_{42} + A_{15}B_{52}
\end{aligned}
$$

k=0,..,K2-1

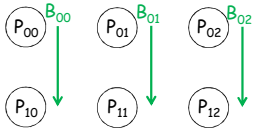
Rossi = Dati disponibili
Neri = Dati non disponibili

per k = 0 non tutti i nodi possono iniziare il calcolo

iterazione k = 0



broadcast di un blocco di A lungo la riga di nodi



broadcast di un blocco di B lungo la colonna di nodi

Verde = Nuovi dati disponibili

k=0

$$
\begin{aligned}
P_{00} &\rightarrow C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30} + A_{04}B_{40} + A_{05}B_{50}\\
P_{01} &\rightarrow C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31} + A_{04}B_{41} + A_{05}B_{51}\\
P_{02} &\rightarrow C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} + A_{03}B_{32} + A_{04}B_{42} + A_{05}B_{52}\\
P_{10} &\rightarrow C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} + A_{13}B_{30} + A_{14}B_{40} + A_{15}B_{50}\\
P_{11} &\rightarrow C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41} + A_{15}B_{51}\\
P_{12} &\rightarrow C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} + A_{14}B_{42} + A_{15}B_{52}
\end{aligned}
$$

tutti i nodi possono eseguire il calcolo per k = 0

iterazione k = 1

Verde = Nuovi dati disponibili

k=1

$P_{00}\rightarrow C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30} + A_{04}B_{40} + A_{05}B_{50}$
$P_{01}\rightarrow C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31} + A_{04}B_{41} + A_{05}B_{51}$
$P_{02}\rightarrow C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} + A_{03}B_{32} + A_{04}B_{42} + A_{05}B_{52}$
$P_{10}\rightarrow C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} + A_{13}B_{30} + A_{14}B_{40} + A_{15}B_{50}$
$P_{11}\rightarrow C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41} + A_{15}B_{51}$
$P_{12}\rightarrow C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} + A_{14}B_{42} + A_{15}B_{52}$

tutti i nodi possono eseguire il calcolo per k = 1

Algoritmo SUMMA (SPMD)

```
for k = 0 to K2-1
    c = k mod Sc
    r = k mod SR
    P ic broadcast A(i,k) lungo la riga i          (i = 0,.., K1-1)
    P rj broadcast B(k,j) lungo la colonna j       (j = 0,.., K3-1)

   Receive A(i,k) in Acol
   Receive B(k,j) in Brow
   C  = C  +  Acol * Brow
}
```

Analisi dell'algoritmo SUMMA

Per semplicita' s = sqrt(p)= S_R=S_C M=N=P

```
for k = 0 to K2-1
    s = k mod Sc
    t = k mod SR
    Pik broadcast A(i,k) lungo la riga i
                          ... tempo = log s *( α + β * (N/s)²), using a tree
    Pkj broadcast B(k,j) lungo la colonna j
                          ... tempo = log s *( α + β * (N/s)²), using a tree

   Receive A(i,k) in Acol
   Receive B(k,j) in Brow
   C  = C  + Acol * Brow        ... tempo = 2*(N/s)³
}
```

Tempo totale = $2*N^3/p + 2s * \alpha * \log s + 2\beta * \log s * N^2/s$

SUMMA performance

- tempo totale =
  $2*N^3/P + 2\alpha * \log s * s + 2\beta * \log s * N^2/s$
- Parallel Efficiency = $2*N^3 / (P * \text{tempo totale})$

$1/(1 + \alpha * \log s * (s/N)^3 + \beta * \log s * (s/N)) =$

$$E_P = \frac{1}{1 + O(\log(\sqrt{P})\sqrt{P}/N)}$$

SUMMA performance

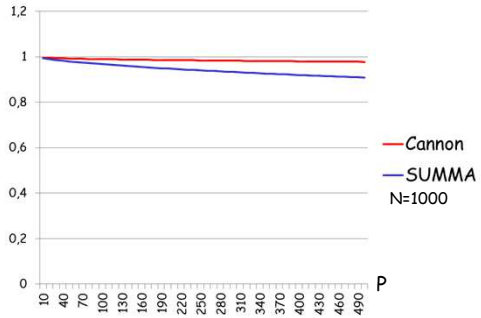
SUMMA

$$E_P = \frac{1}{1 + O(log(\sqrt{P})\sqrt{P}/N)}$$

Alg. Cannon

$$E_P = \frac{1}{1 + O(\sqrt{P}/N)}$$

- stessa efficienza dell'algoritmo di Cannon, tranne che per il fattore log(sqrt(P))
- ma log(sqrt(P)) cresce molto lentamente



Il precedente algoritmo

- massimizza la granularita' del calcolo
- bilancia il carico tra i processori
- ha una efficiente strategia di comunicazione

E' alla base della routine
PDGEMM di PBLAS

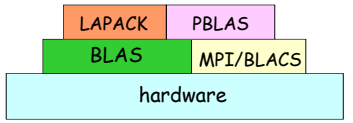
riassumendo

- Distribuzione 1D
  - Algoritmo ad anello: Efficiency = 1/(1 + O(p/n))
- Distribuzione 2D
  - Cannon
    - Efficienza = $1/(1 + \alpha * (\text{sqrt}(p)/N)^3 + \beta * \text{sqrt}(p)/N)$ – ottimale !
    - Difficile da generalizzare per generici P, N, distribuzione ciclica
  - SUMMA
    - Efficienza = $1/(1 + \alpha * \log s * (s/N)^3 + \beta * \log s * (s/N))$
    - Piu' generale dell'algoritmo di Cannon
    - Usato concretamente (PBLAS)

31

Parallel BLAS

- Versione parallela di BLAS1, BLAS2 e BLAS 3
  - Circa 34 routine, ognuna in 4 versioni
- Operazioni di base su matrici e vettori memorizzate secondo la distribuzione ciclica a blocchi
- Utilizza BLAS localmente in ogni processore
- Interfacce C e Fortran
- www.netlib.org/scalapack



Descrittore di matrici

- PBLAS utilizza un array per la descrivere le matrici distribuite

INTEGER DESCA(9)

- **DESCA(1): (DTYPE)** 1
- **DESCA(2): (CTXT)** contesto BLACS
- **DESCA(3): (M)** numero di righe nella matrice globale
- **DESCA(4): (N)** numero di colonne nella matrice globale
- **DESCA(5): (MB)** numero di righe in un blocco
- **DESCA(6): (NB)** numero di colonne in un blocco
- **DESCA(7): (RSRC)** indice di riga del proprietario di A(1, 1)
- **DESCA(8): (CSRC)** indice di colonna del proprietario di A(1, 1)
- **DESCA(9): (LLD)** Leading dimension della matrice locale

DGEMM vs PDGEMM

- **BLAS 3**
  - **CALL DGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)**
- PBLAS
  - **CALL PDGEMM(TRANSA, TRANSB, M, N, K,ALPHA, A, IA, JA, DESC_A, B, IB, JB, DESC_B, BETA, C, IC, JC, DESC_C)**
- DESC_A, DESC_B e DESC_C sostituiscono LDA, LDB e LDC della subroutine DGEMM