Algoritmi paraleli și distribuiți
Calcul matriceal

Mitică Craus

Universitatea Tehnică "Gheorghe Asachi" din Iași

Cuprins

Introducere
Transpusa unei matrice
- Descriere
- Pseudocod
- Implementare
- Exemplu de execuție
- Complexitatea

Inmulțirea de matrice pătractice
- Descriere
- Pseudocod
- Implementare
- Exemplu de execuție
- Complexitatea

Comentarii bibliografice

Introducere

- Calculul matricial se pretează extrem de bine la procesare paralelă.
- Natura regulată a matricelor constituie un atu de neegalat pentru abordări specifice calculului paralel.
- Vom studia în această lecție problemele transpusei unei matrice și produsului de matrice.

Transpusa unei matrice - formularea problemei

- Dată fiind matricea pătratică $A_{n\times n} = (a_{i,j})_{i,j=0,1\dots,n-1}$, se cere să se calculeze matricea $A^T_{nxn} = (a^T_{i,j})_{i,j=0,1\dots,n-1}$, pentru care

$$a^T_{i,j} = a_{j,i},\ (\forall)i,j = 0,1\dots,n-1$$

.

- Fără calcule.
- Doar mișcări de elemente.
- Timpul secvențial $T_s(n) = O(n^2)$.

Algoritmul secvențial standard - pseudocod

- *Notații*:
  - $A[0..n-1, 0..n-1]$ este un tablou bidimensional, de dimensiune *nxn*.
- *Premise*:
  - Datele de intrare sunt memorate în tabloul A.
  - Datele finale vor fi memorate în tabloul A.

TRANSPUNERE_MATRICE(A, n)

```
for i = 0 to n − 1
do for j = i + 1 to  n − 1
     do INTERSCHIMBA(A[i, j], A[j, i])
```

INTERSCHIMBA$(A[i,j], A[j,i])$

```
temp ← A[i, j]
A[i, j] ← A[[j, i]
A[j, i] ← temp
```

Exemplu de execuție a algoritmului de transpunere a unei matrice pe o plasă de unități de procesare

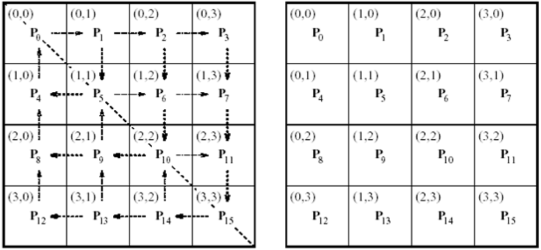


Figura 1: Exemplu de execuție a algoritmului de transpunere a unei matrice 4x4 pe o plasă de unități de procesare

Algoritmul recursiv - pseudocod

- *Notații*:
  - $A[0..n-1, 0..n-1]$ este un tablou bidimensional, de dimensiune nxn.
- *Premise*:
  - $n = 2^m$.
  - Datele de intrare sunt memorate în tabloul A.
  - Matricea A este divizată în patru blocuri: *stânga-sus, dreapta-sus, stânga-jos, dreapta-jos*.
  - Datele finale vor fi memorate în tabloul A.

$\textsc{Transpunere\_Matrice\_Recursiv}(A[0..n-1, 0..n-1])$

$/*$ Interschimbarea blocurilor *stânga-jos* cu *dreapta-sus* $*/$
for $i = \frac{n}{2}$ **to** $n-1$
do for $j = 0$ **to** $\frac{n}{2} - 1$
  do $\textsc{Interschimba}(A[i,j], A[i-\frac{n}{2}, j+\frac{n}{2}],)$
$\textsc{Transpunere\_Matrice\_Recursiv}(A[0..\frac{n}{2}-1, 0..\frac{n}{2}-1])$
$\textsc{Transpunere\_Matrice\_Recursiv}(A[0..\frac{n}{2}-1, \frac{n}{2}..n-1])$
$\textsc{Transpunere\_Matrice\_Recursiv}(A[\frac{n}{2}..n-1, 0..\frac{n}{2}-1])$
$\textsc{Transpunere\_Matrice\_Recursiv}(A[\frac{n}{2}..n-1, \frac{n}{2}..n-1])$

Exemplu de execuție a algoritmului recursiv de transpunere a unei matrice

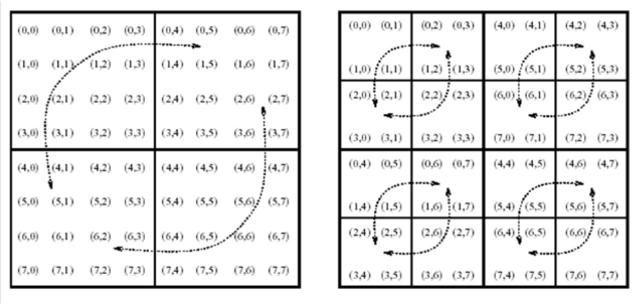


Figura 2: Exemplu de execuție a algoritmului recusrsiv de transpunere a unei matrice 8x8

Exemplu de execuție a algoritmului recursiv de transpunere a unei matrice - continuare

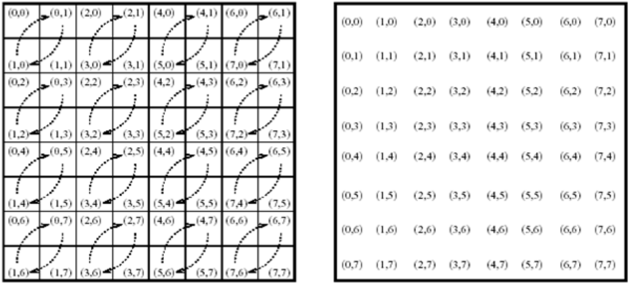


Figura 3: Exemplu de execuție a algoritmului recusrsiv de transpunere a unei matrice 8x8

Implementare pe hipercub

1. Hipercubul este divizat în 4 subcuburi cu $\frac{p}{4}$ procesoare.
2. Sferturile de matrice (quadranții) sunt mapate recursiv pe cele 4 subcuburi astfel:
   - cubul $00*$ conține sfertul *stânga-sus* al matricei A;
   - cubul $01*$ conține sfertul *dreapta-sus* sfert al matricei A;
   - cubul $10*$ conține sfertul *stânga-jos* al matricei A;
   - cubul $11*$ conține sfertul *dreapta-jos* al matricei A.
3. Se interschimbă blocurile *stânga-jos* cu *dreapta-sus*.
4. Se repetă pașii 1-3 în fiecare subcub.

Algoritmul recursiv implementat pe hipercub - pseudocod

- *Notații*:
  - H este un hipercub cu m dimensiuni.
  - i este indicele unității de procesare care executa algoritmul de transpunere a unei matrice A.
  - M este blocul care urmează a fi transmis de o unitate de procesare către partener.
- *Premise*:
  - Inițial, fiecare unitate de procesare p_i deține un bloc B_i din matricea A, conform cu regula de mapare, care urmează a fi transmis unei unități de procesare plasate într-un nod al hipercubului H.

TRANSPUNERE_MATRICE_PE_HIPERCUB(H, m, i, B_i)

$M \leftarrow B_i$
for $k \leftarrow m-1$ **downto** 0 *step* 2
do *partener* $\leftarrow i$ *xor* 2^k
    if $(bit_k(i) + bit_{k-1}(i) = 1)$
        then *trimite M catre partener*
        else *primeste M de la partener*
    partener $\leftarrow i$ *xor* 2^{k-1}
    if $(bit_k(i) + bit_{k-1}(i) \neq 1)$
        then *trimite M catre partener*
        else *primeste M de la partener*
            $B_i \leftarrow M$
TRANSPUNERE_MATRICE($B_i, \frac{n^2}{p}$)

Exemplu de execuție a algoritmului de transpunere a unei matrice pe hipercub

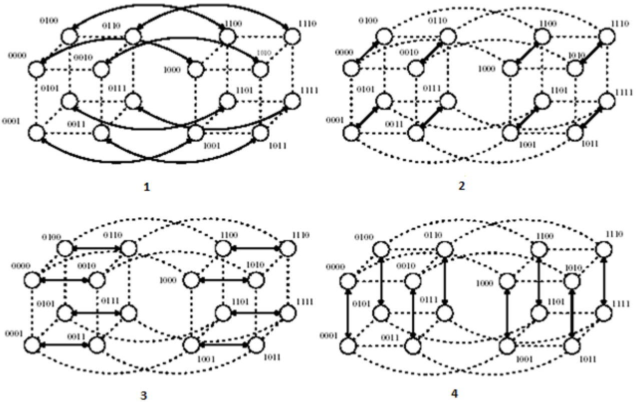


Figura 4: Exemplu de execuție a algoritmului de transpunere pe hipercub a unei matrice 4x4

Complexitatea implementării pe hipercub algoritmului recursiv de transpunere a unei matrice

Teorema (1)

Complexitatea timp a algoritmului recursiv de transpunere a unei matrice A_{nxn}, implementat pe un hipercub cu p unități de procesare este $O(\frac{n^2}{p}\log p)$. Eficiența algoritmului este $O(\frac{1}{\log n})$.

Demonstrație.

Divizarea recursivă se oprește când dimensiunea blocului este $\frac{n^2}{p}$. Numărul de pași de interschimbare de blocuri este $\log_4 p = \frac{log_2 p}{2}$; fiecare pas de interschimbare se realizează pe două dintre dimensiunile hipercubului (două muchii). Timpul de transfer al unui bloc de dimensiune $\frac{n^2}{p}$ este $O(\frac{n^2}{p})$. Transpunerea locală necesită $O(\frac{n^2}{p})$ timp.
Rezultă că timpul total este $O(\frac{n^2}{p}\log p)$. Costul este $O(n^2\log p)$ - nu este optimal. □

Inmulțirea de matrice pătractice - formularea problemei

- Date fiind matricele pătratice $A_{n\times n} = (a_{i,j})_{i,j=0,1\ldots,n-1}$ și $B_{n\times n} = (b_{i,j})_{i,j=0,1\ldots,n-1}$, se cere să se calculeze matricea pătratică $C_{nxn} = (c_{i,j})_{i,j=0,1\ldots,n-1}$, conform cu formula

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} x b_{k,j},\ (\forall) i,j = 0,1\ldots,n-1$$

.

- Timpul secvențial $T_s(n) = O(n^3)$.

Algoritmul secvențial standard - pseudocod

- *Notații*:
  - $A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$ și $C[0..n-1, 0..n-1]$ sunt tablouri bidimensionale, de dimensiune *nxn*.
- *Premise*:
  - Datele de intrare sunt memorate în tablourile A și B; $A[i,j] = a_{i,j},\ B[i,j] = b_{i,j},\ i,j = 0,1,\ldots,n-1$.
  - Datele finale vor fi memorate în tabloul C.

```
INMULTIRE_MATRICE(A, B, n)
for i = 0 to n-1
do for j = 0 to n-1
    do C[i,j] ← 0
        for k = 0 to n-1
        do C[i,j] ← C[i,j] + A[i,k]xB[k,j]
return C
```

Algoritmul înmulțirii de blocuri- pseudocod

- *Notații*:
  - $A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$ și $C[0..n-1, 0..n-1]$ sunt tablouri bidimensionale, de dimensiune *nxn*.
  - Matricele A și B sunt divizate în blocuri $A_{i,k}$ și respectiv $B_{k,j}$, de dimensiuni $\frac{n}{q} \times \frac{n}{q}$.
  - $\text{INMULTIRE\_MATRICE}(A_{i,k}, B_{k,j}, \frac{n}{q})$ semnifică înmulțirea blocului $A_{i,k}$ cu $B_{k,j}$ și returnarea rezultatului $A_{i,k} x B_{k,j}$.
- *Premise*:
  - Datele de intrare sunt memorate în tablourile A și B.
  - Datele finale vor fi memorate în tabloul C.

```
INMULTIRE_MATRICE_BLOCURI(A, B, n, q)
for i = 0 to q − 1
do for j = 0 to  q − 1
    do C[i,j] ← 0
        for k = 0 to  q − 1
        do C[i,j] ← C[i,j] + INMULTIRE_MATRICE(A[i,k], B[k,j], n/q)
return C
```

Implementarea standard

- Arhitectura naturală este cea cu topologie de tip plasă, cu $p = q^2$ unități de procesare.
- Fiecare unitate de procesare $p_{i,j}$ dispune în memoria locală de blocul $A_{i,j}$ și un bloc $B_{i,j}$ și calculează $C_{i,j}$.
- Pentru a calcula $C_{i,j}$, unitatea de procesare $p_{i,j}$ are nevoie de $A_{i,k}$ și $B_{k,j},\ k = 0, 1, \ldots, q-1$.
- Pentru fiecare k, blocurile $A_{i,k}$ și $B_{k,j}$ sunt obținute printr-o difuzie *toți-la-toți* pe linii și apoi pe coloane.

Implementarea lui Cannon

- Similar cu implementarea standard, dar cu consum mai mic de memorie.
- Fiecare bloc este procesat la momente diferite, în locuri diferite. Pentru aceasta blocurile sunt deplasate ciclic.
- Sunt executați $\sqrt{p}-1$ pași de înmulțiri și adunari locale de blocuri, urmate de deplasări ciclice la stânga (în A) și în sus (în B).
- La final este executată o înmulțire și o adunare locală de blocuri.

Algoritmul lui Cannon - pseudocod

- *Notații*:
  - P este o plasă formată din $qxq = \sqrt{p}x\sqrt{p} = p$ unități de procesare.
  - $p_{i,j}$ este unitatea de procesare care execută algoritmul de inmultire de blocuri.
  - M_1 este blocul care urmează a fi transmis de $p_{i,j}$ către $p_{i,j\ominus 1}$.
  - M_2 este blocul care urmează a fi transmis de $p_{i,j}$ către $p_{i\ominus 1,j}$.
  - M_3 este blocul calculat de $p_{i,j}$.
  - $\oplus$ și $\ominus$ semnifică adunarea și scăderea modulo q.
- *Premise*: Inițial, fiecare $p_{i,j}$ deține în memoria locală blocurile $A_{i,j}$ și $B_{i,j}$.
- *Rezultate*: Fiecare unitate de procesare $p_{i,j}$ calculează blocul $C_{i,j}$.

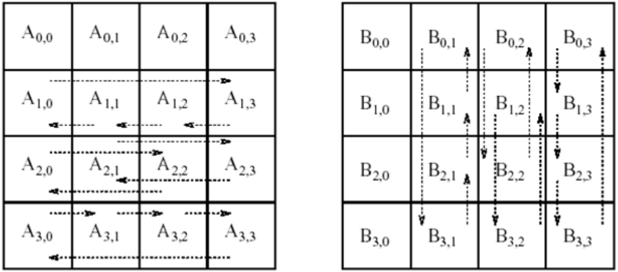
INMULTIRE_MATRICE_CANNON($A_{i,j}, B_{i,j}, n, q, p_{i,j}$)

$M_1 \leftarrow A_{i,j}$; $M_2 \leftarrow B_{i,j}$
ALINIERE_INITIALA($M_1, M_2, n, q, p_{i,j}$)
$M_3 \leftarrow 0$
/∗ $q-1$ pași de înmulțiri și adunari de blocuri /∗
for $k \leftarrow 0$ **to** $q-2$
do $M_3 = M_3 +$ INMULTIRE_MATRICE($M_1, M_2, n/q$)
    trimite M_1 *catre* $p_{i,j\ominus 1}$
    primeste M_1 *de la* $p_{i,j\oplus 1}$
    trimite M_2 *catre* $p_{i\ominus 1,j}$
    primeste M_2 *de la* $p_{i\oplus 1,j}$
$C_{i,j} \leftarrow M_3 +$ INMULTIRE_MATRICE($M_1, M_2, n/q$)
return $C_{i,j}$

ALINIERE_INITIALA($M_1, M_2, n, q, p_{i,j}$)

for $k \leftarrow 1$ **to** i
do *trimite* M_1 *catre* $p_{i,j\ominus 1}$
    primeste M_1 *de la* $p_{i,j\oplus 1}$
for $k \leftarrow 1$ **to** j
do *trimite* M_2 *catre* $p_{i\ominus 1,j}$
    primeste M_2 *de la* $p_{i\oplus 1,j}$

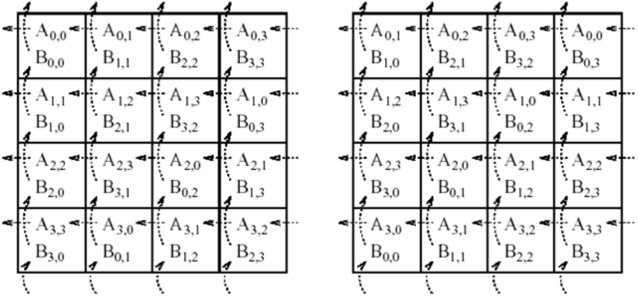
Exemplu de execuție a algoritmului lui Cannon



Alinierea inițială

Figura 5: Exemplu de execuție a algoritmului lui Cannon

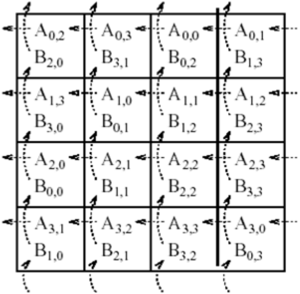
Exemplu de execuție a algoritmului lui Cannon



A si *B* după alinierea inițială   Pozițiile blocurilor după prima deplasare

Figura 6: Exemplu de execuție a algoritmului lui Cannon

Exemplu de execuție a algoritmului lui Cannon



Pozițiile blocurilor după a II-a deplasare

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

Pozițiile blocurilor dupa a III-a deplasare

Figura 7: Exemplu de execuție a algoritmului lui Cannon

Complexitatea algoritmului lui Cannon

Teorema (2)

Complexitatea timp a algoritmului lui Cannon este $O(\frac{n^3}{p})$. Eficiența algoritmului este $O(1)$.

Demonstrație.

Numărul de transmisii și primiri de blocuri M_1 efectuate de $p_{i,j}$ este $i+q-1=i+\sqrt{p}-1$. Numărul de transmisii și primiri de blocuri M_2 efectuate de $p_{i,j}$ este tot $j+q-1=j+\sqrt{p}-1$. Se observă că $p_{q-1,q-1}$ efectuează cele mai multe transmisii și primiri de blocuri, $q-1+q-1=2(q-1)=2(\sqrt{p}-1)$. Rezultă pentru deplasările de blocuri o complexitate timp de $O(\frac{n^2}{p}\sqrt{p})$. Numărul înmulțirilor și adunărilor efectuate de o unitate de procesare este $O((\frac{n}{\sqrt{p}})^3\sqrt{p})$=$O(\frac{n^3}{p})$; $O((\frac{n}{\sqrt{p}})^3)$ pentru fiecare înmulțire și adunare de blocuri. Costul este $O(p)O(\frac{n^3}{p})=O(n^3)$. □

Comentarii bibliografice

- Capitolul *Calcul matricial* are la bază cartea
  V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003
  și ediția mai veche
  V. Kumar, A. Grama A. Gupta & G Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin-Cummings, 1994