Documentatie Laborator 1

**Analiza cerințelor**

* **Tema:** Se considera o imagine reprezentata printr-o matrice de pixeli, F , de dimensiune (NxM). Dorim sa obtinem imaginea filtrata, aplicand convolutia 2D asupra imaginii originale. Se doreste analiza timpului de executie, atat in varianta secventiala, cat si in cea paralelizata.
* **Cerinte:** Se citeste matricea initiala de pixeli si matricea kernel. Se cere transformarea matricii initiale, aplicand filtrare cu o fereastra kernel definita de multimea de indici

Ind[n,m] = {(k,l) | -n/2<=k<=n/2, -m/2<=l<=m/2}

si de coeficientii *wkl*

Transformarea unui pixel: V[i,j] = {**+** (k,l): -n/2<=k<=n/2, -m/2<=l<=m/2: w[k,l] \* F[i-k ,j-l]

* **Datele de intrare** se citesc dintr-un fisier de intrare “date.txt”.
* Pe prima linie se citesc: N,M,n,m ,unde
  + N- nr de linii a matricii de pixeli F
  + M – nr de coloane a matricii de pixeli F
  + n – nr de linii a matricii kernel w
  + m – nr de coloane a matricii kernel w
* Pe urmatoarele N\*M linii se citesc elementele matricii de pixeli F
* Pe urmatoarele n\*m linii se citesc elementele matricii kernel w

Fisierul trebuie creat anterior prin adaugare de numere generate aleator. Toate rularile trebuie executate cu acelasi fisier.

* **Constrangeri:**

1. n,m impare
2. n<N, m<M.

* **Postconditie:** Matricea rezultat V contine imaginea filtrata a imaginii initiale F (unde V != F)
* **Implementare:**

1. Java
2. C++ ( cel putin C++11 )
   1. matricile sunt alocate static
   2. matricile sunt alocate dinamic

* **Testare:** masurati timpul de executie pentru

1. N=M=10 si n=m=3; p=4;
2. N=M=1000 si n=m=5; p=2,4,8,16
3. N=10 M=10000 si n=m=5; p=2,4,8,16
4. N=10000 M=10 si n=m=5; p=2,4,8,16

**Proiectare**

O imagine care conține text

Descriere generată automat

In functia main introduc manual numarul de threaduri care doresc sa se execute in varianta paralela a programului. Apoi, citesc din fisier dimensiunile si elementele matricii de pixeli F[ ][ ], cat si cele pentru matricea kernel (acestea au fost generate aleator intr-un alt program). Elementele matricii kernel le retin intr-un vector unidimensional ind[ ], in ordinea citirii lor(de exemplu elementul de pe linia 2, coloana 3 se va regasi pe pozitia 6 in ind[ ]). Apoi, tratez 2 cazuri:

* Varianta secventiala: initializez matricea rezultat vSecv[][] si apelez functia secvential() care imi calculeaza timpul de executie a calculului serial. In functia secvential() iau element cu element si calculez rezultatul obtinut in urma aplicarii kernelului.
* Varianta paralela: initializez matricea rezultat vParal[][] si apelez functia paralel() care imi calculeaza timpul de executie a calculului paralel. In functia paralel() creez p threaduri si le aloc proportional un numar de linii din matricea de pixeli pe care trebuie sa le calculeze.

La final, compar matricile rezultate vSecv[][] si vParal[][] pentru a ma asigura ca au aceleasi elemente.

Specificatia functiilor:

/\* Date de intrare: N- nr de linii din matricea de pixeli F

M - nr de coloane din matricea de pixeli F

F – matricea de pixeli

n- nr de linii din matricea kernel

m - nr de coloane din matricea kernel

ind – vectorul de elemente corespunzator matricii kernel

Date de iesire: v – matricea rezultat care va contine imaginea filtrata prin kernel

Return: double time – timpul de executie a calcululiui

\*/

double secvential(int N, int M, int[][] F, int n, int m, double[] ind, int[][] v)

/\* Date de intrare: N- nr de linii din matricea de pixeli F

M - nr de coloane din matricea de pixeli F

F – matricea de pixeli

n- nr de linii din matricea kernel

m - nr de coloane din matricea kernel

ind – vectorul de elemente corespunzator matricii kernel

p – nr de threaduri care trebuie create

Date de iesire: v – matricea rezultat care va contine imaginea filtrata prin kernel

Return: double time – timpul de executie a calcululiui

\*/

double paralel(int N, int M, int[][] F, int n, int m, double[] ind, int[][] v, int p) throws InterruptedException

/\* Date de intrare: N- nr de linii a matricilor

M - nr de coloane a matricilor

F1 – prima matrice

F2 – a 2 a matrice

Return: true, daca matricile contin exact aceleasi elemente,

false, altfel

\*/

public static boolean verifyIdenticalMatrix(int N, int M, int[][] F1, int[][] F2)

**Rezultate Java**

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | secvential | 0.06221 |
| 4 | 0.41635 |
|  |  |  |
| N=M=1000  n=m=5 | secvential | 90.50014 |
| 2 | 57.95898 |
| 4 | 43.42045 |
| 8 | 35.89648 |
| 16 | 38.57109 |
|  |  |  |
| N=10  M=10000  n=m=5 | secvential | 26.50235 |
| 2 | 24.85086 |
| 4 | 27.68795 |
| 8 | 38.62873 |
| 16 | 36.22594 |
|  |  |  |
| N=10000  M=10  n=m=5 | secvential | 20.85018 |
| 2 | 15.85539 |
| 4 | 17.45608 |
| 8 | 32.83038 |
| 16 | 56.73476 |

**Rezultate C++**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | static | secvential | 0.01138 |
| 4 | 2.06835 |
| dinamic | secvential | 0.00334 |
|  | 4 | 1.91973 |
|  |  |  |  |
| N=M=1000  n=m=5 | static | secvential | 274.5422 |
| 2 | 145.067 |
| 4 | 84.31838 |
| 8 | 83.16876 |
| 16 | 77.69937 |
| dinamic | secvential | 4.10992 |
| 2 | 4.20894 |
| 4 | 4.41987 |
| 8 | 4.31034 |
| 16 | 4.54793 |
|  |  |  |  |
| N=10  M=10000  n=m=5 |  | secvential | 27.56713 |
|  | 2 | 16.82855 |
| static | 4 | 11.59792 |
|  | 8 | 11.8093 |
|  | 16 | 11.40786 |
| dinamic | secvential | 0.40078 |
| 2 | 2.04607 |
| 4 | 2.19197 |
| 8 | 2.39158 |
| 16 | 2.3963 |
|  |  |  |  |
| N=10000  M=10  n=m=5 |  | secvential | 28.03122 |
| static | 2 | 16.25578 |
|  | 4 | 10.59116 |
|  | 8 | 11.03845 |
|  | 16 | 11.21826 |
|  | dinamic | secvential | 1.30801 |
|  |  | 2 | 2.62011 |
|  |  | 4 | 2.49732 |
|  |  | 8 | 2.89679 |
|  |  | 16 | 3.45365 |

**Analiza rezultatelor**

Observatii:

* Executia concurenta este eficienta doar in cazul volumelor mari de date, si doar daca folosim un numar mai mic de threaduri(2-4). Deja in cazul in care folosim 8-16 threaduri timpul incepe sa creasca.
* C++ este mult mai eficient decat Java, in majoritatea situatiilor evaluate (in afara de cazul 2 in care varianta statica din C++ este vizibil mai lenta)
* Varianta cu alocare dinamica din C++ este mult mai rapida decat cea cu alocare statica, in cazul volumelor mari de date

In continuare, voi reprezenta graphic rezultatele obtinute pentru fiecare limbaj:

* Java

* C++