Programare paralela si distribuita – Laborator 2

Munteanu Bianca-Stefania

Grupa 235/1

**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.
3. NU se aloca o matrice rezultat (V) temporara!

* **Postconditie:** Matricea initiala contine imaginea filtrata.
* **Implementare:**

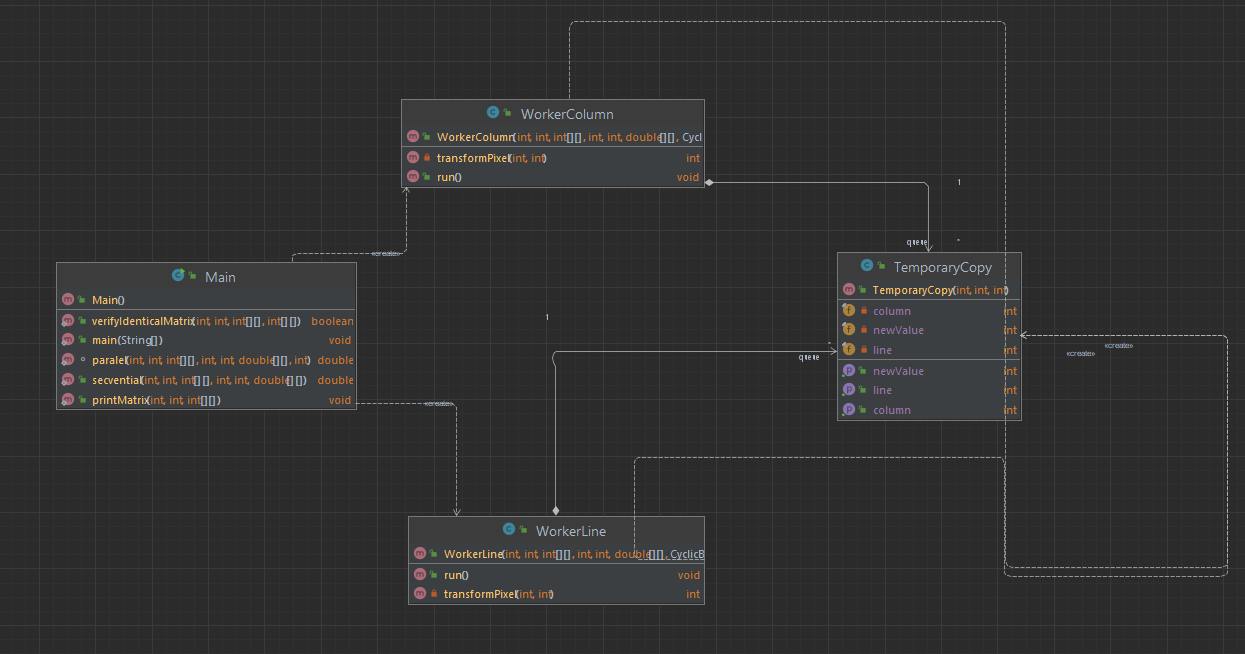
1. Java
2. C++ ( cel putin C++11 )
   1. matricile sunt alocate static
   2. matricile sunt alocate dinamic

* **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**

**JAVA:**



In functia main primesc ca argument numarul de threaduri care doresc sa se execute in varianta paralela a programului. Apoi, citesc din fisier dimensiunile si elementele matricii de pixeli (voi avea 2 matrici F1[ ][ ] si F2[ ][ ] cu exact aceleasi elemente, pentru a folosi F1 la calculul secvential si F2 la calculul paralel), cat si cele pentru matricea kernel (acestea au fost generate aleator intr-un alt program). Apoi, tratez 2 cazuri:

* **Varianta secventiala**: apelez functia secvential() care imi calculeaza timpul de executie a calculului serial. In functia secvential():
  1. Pornesc timerul
  2. Initializez un vector solPartiala[], in care voi retine elementele filtrate
  3. Iau fiecare element din matricea F, calculez coordonatele centrului kernelului (kernelCenterL si kernelCenterC) si alte 2 coordonate (line, col) in functie de indicele curent si dimensiunea kernelulului. Apoi, pentru fiecare element, parcurg matricea kernel, calculez indicii de pe care trebuie sa iau elementele din matricea F (daca acestia depasesc bordura matricii tratez punctual cazurile) si realizez inmultirile corespunzatoare
  4. (solPartiala[i] += F[poz1][poz2] \* kernel[k][l])
  5. Inlocuiesc elementele initiale ale matricii F cu elementele vectorului solPartiala
  6. Opresc timerul
  7. Calculez si returnez timpul de executie in milisecunde
* **Varianta paralela**: apelez functia paralel() care imi calculeaza timpul de executie a calculului paralel. In functia paralel():

1. Initializez bariera
2. Creez p threaduri si le aloc proportional in functie de N si M (daca N>=M fac impartirea pe linii, iar daca N<M fac impartire pe coloane)
3. Pornesc threadul curent, care poate fi un WorkerLine(N>=M) sau WorkerColumn(N<M) in functie de caz
4. Fiecare thread calculeaza valorile noi dar pentru celulele aflate pe frontiera (vecinatate cu alte threaduri) valorile noi se salveaza in vectori separati
5. Fiecare thread calculeaza valorile noi dar pentru celulele care nu sunt pe frontiera iar valorile noi se salveaza direct in matrice. (initial le adaug intr-o coada si atunci cand de elementul [i][j] nu mai depinde niciun alt element, il modific definitiv)
6. Bordarea se face cu if-uri
7. Bariera de sincronizare
8. Actualizeaza elementele de pe frontiera in matrice prin copierea vectorilor calculati la 4.
9. Astept ca toate threadurile sa isi incheie executia
10. Opresc timerul
11. Calculez si returnez timpul de executie in milisecunde

La final, compar matricile rezultate F1[][] si F2[][] 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

kernel – matricea kernel

Return: double time – timpul de executie a calcululiui

\*/

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

/\* 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

kernel – matricea kernel

p – nr de threaduri care trebuie create

Return: double time – timpul de executie a calcululiui

\*/

double paralel(int N, int M, int[][] F, int n, int m, double[][] kernel, 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)

**C++:**

Diferenta fata de Java: Nu exista implementare pentru bariera in limbaj, asa ca am creat clasa MyBarrier, care simuleaza comportamentul unei bariere.

* Alocare statica:

Global:

# define MAX 10001

# define MAXK 100

# define MAXP 16

int F1[MAX][MAX];

int F2[MAX][MAX];

double kernel[MAXK][MAXK];

* Alocare dinamica:

Global:

int\*\* F1;

int\*\* F2;

double\*\* kernel;

In functia Main():

F1 = new int\* [N];

F2 = new int\* [N];

for (int i = 0; i < N; i++) {

F1[i] = new int[M];

F2[i] = new int[M];

}

kernel = new double\* [n];

for (int i = 0; i < n; i++) {

kernel[i] = new double[m];

}

**Rezultate Java**

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr threads** | **Timp executie** (ms) |
| N=M=10  n=m=3 | secvential | 0.06473 |
| 4 | 2.10482 |
|  |  |  |
| N=M=1000  n=m=5 | secvential | 124.46504 |
| 2 | 134.48149 |
| 4 | 118.03324 |
| 8 | 113.40916 |
| 16 | 144.73426 |
|  |  |  |
| N=10  M=10000  n=m=5 | secvential | 27.06049 |
| 2 | 42.38369 |
| 4 | 46.12677 |
| 8 | 44.28593 |
| 16 | 44.78265 |
|  |  |  |
| N=10000  M=10  n=m=5 | secvential | 23.38529 |
| 2 | 25.21617 |
| 4 | 27.78326 |
| 8 | 24.33019 |
| 16 | 26.01325 |

**Rezultate C++**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | static | secvential | 0.01792 |
| 4 | 2.05567 |
| dinamic | secvential | 0.00824 |
|  | 4 | 0.89887 |
|  |  |  |  |
| N=M=1000  n=m=5 | static | secvential | 284.7512 |
| 2 | 613.4512 |
| 4 | 375.0555 |
| 8 | 346.3112 |
| 16 | 518.5625 |
| dinamic | secvential | 124.0959 |
| 2 | 240.1721 |
| 4 | 125.9339 |
| 8 | 173.7489 |
| 16 | 453.4564 |
|  |  |  |  |
| N=10  M=10000  n=m=5 |  | secvential | 29.1154 |
|  | 2 | 70.95118 |
| static | 4 | 45.24848 |
|  | 8 | 44.49846 |
|  | 16 | 47.3193 |
| dinamic | secvential | 29.66728 |
| 2 | 69.90545 |
| 4 | 43.81696 |
| 8 | 45.29726 |
| 16 | 42.9724 |
|  |  |  |  |
| N=10000  M=10  n=m=5 |  | secvential | 29.9126 |
| static | 2 | 74.23774 |
|  | 4 | 45.95143 |
|  | 8 | 43.68521 |
|  | 16 | 47.46884 |
|  | dinamic | secvential | 12.98574 |
|  |  | 2 | 26.07162 |
|  |  | 4 | 14.86152 |
|  |  | 8 | 19.07992 |
|  |  | 16 | 48.10454 |

**Analiza rezultatelor**

Observatii:

* Executia concurenta este eficienta doar in cazul volumelor mari de date, si doar daca folosim un numar mai mic de threaduri(4-8). Deja in cazul in care folosim 16 threaduri timpul incepe sa creasca.
* C++ este mai eficient decat Java, in majoritatea situatiilor evaluate
* Varianta cu alocare dinamica din C++ este mai rapida decat cea cu alocare statica
* Evaluare complexitatea-spatiu: am redus complexitatea spatiu, pentru ca nu mai utilizez un tablou bidimensional pentru a stoca o matrice rezultat, ci utilizez doar 2 vectori pentru a retine rezultatele de pe frontiere si o coada in care retin elementele modificate, din care scot primul element imediat ce nu mai am nevoie de el
* Impartire pe linii: O(N \* M + n \* m + M \* n \* p + 4\* n/2 \* M + n/2 \* M \* n \* m)
* Impartire pe coloane: O(N \* M + n \* m + N \* m\* p + 4\* n/2 \* N + n/2 \* N \* n \* m)
* Comparare performanta pentru fiecare caz:

Din punct de vedere al spatiului, variantele care folosesc bariere sunt mai eficiente decat cele cu flaguri deoarece bariera stocheaza doar un intreg, in loc de cate o variabila separata pentru fiecare thread, ca in cazul flagurilor. Varianta cu prelucrarea elementelor inainte de bariera este mai eficienta decat cea cu prelucrarea dupa bariera, deoarece nu mai trebuie sa pastreze frontiera unui element, ci calculeaza valorile de pe frontiera direct in niste vectori rezultat.

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

* Java

* C++