Picus Oskar Anton gr 236

PPD – Laborator 2

Analiza cerintelor

Se considera o imagine reprezentata printr-o matrice de pixeli, F, cu N linii si M coloane. Se cere transformarea ei, aplicand o filtrare cu o fereastra definita de multimea de indici W cu coeficienti w_{kl} (reprezentati prin matricea W[k,l], unde $-\frac{n}{2} \leq k \leq \frac{n}{2}, -\frac{m}{2} \leq l \leq \frac{m}{2}, n < N, m < M$, n,m impare.

Transformarea unui pixel:

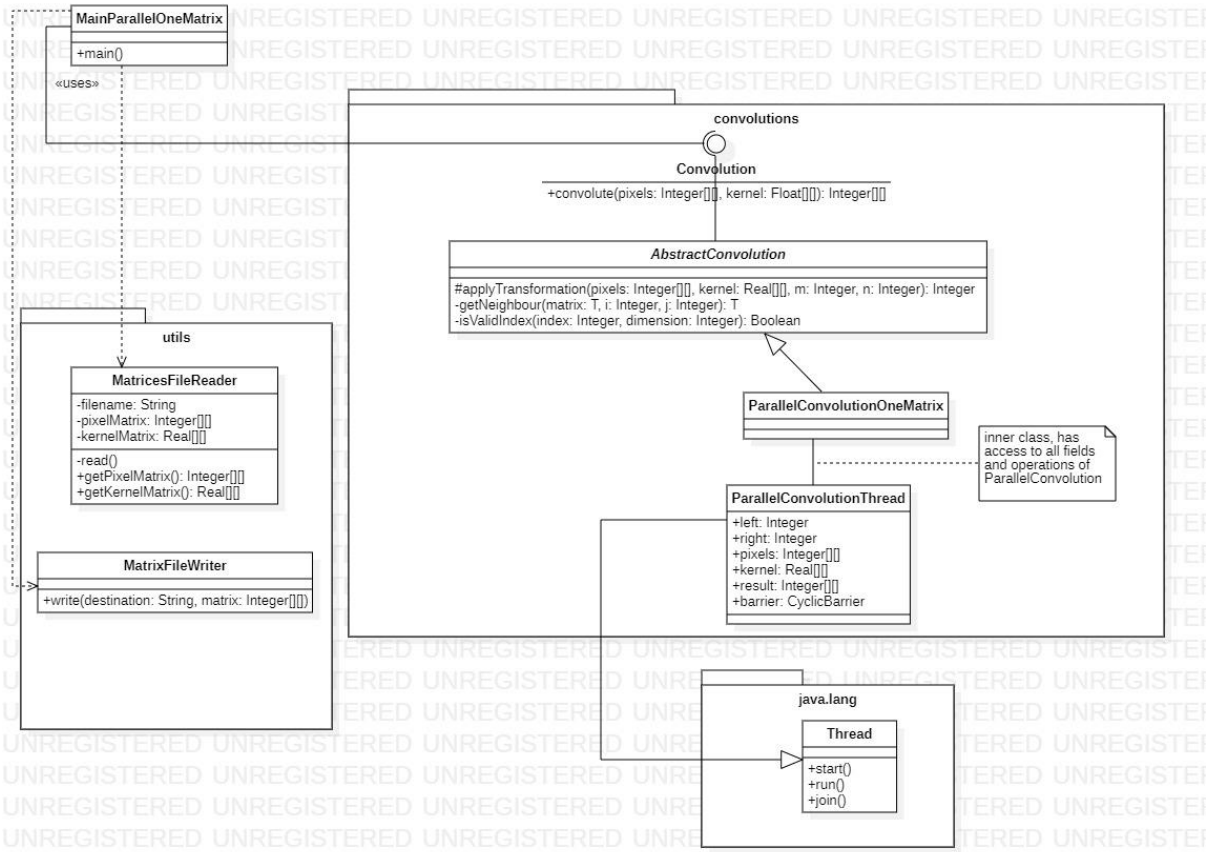
$$v(m,n) = \sum_{(k,l) \in W} w_{kl} f(m-k, n-l)$$

Constrangeri:

- Pentru frontiere se considera ca un element este egal cu elementul din celula vecina din matrice
- Datele de intrare se citesc dintr-un fisier de intrare `date.txt`

Postconditie: matricea rezultat V contine imaginea filtrata a imaginii initiale F, $V \neq F$.

Proiectare



Pachetul **`utils`** este responsabil de citirea si scrierea datelor. Clasa **`MatricesFileReader`**, prin metoda read, citeste si construieste din fisierul dat atat matricea de pixeli, cat si cea a kernel-ului, in timp ce **`MatrixFileWriter`**, prin metoda write, scrie intr-un fisier rezultatul aplicarii filtrului, constituit din datele citite anterior.

Pachetul **`convolutions`** contine algoritmul necesar aplicarii filtrarii. **`AbstractConvolution`**, desi abstracta, implementeaza metoda **`applyTransformation`**; ea efectueaza o singura transformare pe un singur pixel, identificat prin pozitia sa in matrice.

`ParallelConvolutionOneMatrix` extinde aceasta clasa si implementeaza metoda **`convolute`** si se foloseste de clasa **`ParallelConvolutionThread`**, o abstractizare a conceptului de thread, in care, prin metoda run, defineste responsabilitatea unui singur fir de executie.

Detalii de implementare

Mai intai se imparte echitabil numarul de pixeli pe care un fir de executie trebuie sa aplice filtrul, impartind astfel matricea initiala in diferite blocuri de dimensiune aproximativ egala. Fiecare thread va primi pozitia de start si de final,

corespunzatoare formei liniarizate a matricei si cu ajutorul acestor indici, fiecare thread va parcurge blocul asignat lui.

Inainte de a aplica transformarile, fiecare thread va copia elementele necesare intr-o zona de memorie proprie (elementele propriu-zise si vecinii necesari, in functie de dimensiunea kernel-ului). Dupa aceea, thread-ul notifica celelalte thread-uri, cu ajutorul unei bariere, ca a terminat copierea si poate incepe prelucrarea efectiva. Calculele se vor face pe copia pe care a facut-o thread-ul, dar rezultat in sine se va salva in matricea originala.

Cazuri de testare

$$F = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} ==> V = \begin{pmatrix} 21 & 27 & 33 \\ 39 & 45 & 51 \\ 57 & 63 & 69 \end{pmatrix}$$

Rezultate

Java:

Tip matrice	**Numar thread-uri**	**Timp executie (ms)**
N=M=10; n=m=3; (matrixA.txt)	4	95
N=M=1000; n=m=5; (matrixB.txt)	2	1823
	4	1688
	8	1869
	16	1810
N=10,M=10000; n=m=5; (matrixC.txt)	2	735
	4	694
	8	683
	16	690
N=10000,M=10; n=m=5; (matrixD.txt)	2	491
	4	525
	8	565
	16	533

C++:

Tip matrice	**Tip alocare**	**Numar thread-uri**	**Timp executie (ms)**
N=M=10; n=m=3; (matrixA.txt)	Static	4	4.90405
	Dinamic	4	4.95653
N=M=1000; n=m=5;	Static	2	8165.338000000001

(matrixB.txt)		4	7811.125
		8	7776.712000000001
		16	7746.791000000002
	Dinamic	2	8562.385999999999
		4	8183.258
		8	8113.6039999999975
		16	8074.946000000001
N=10,M=10000; n=m=5; (matrixC.txt)	Static	2	824.5966000000001
		4	772.8140999999999
		8	783.2991
		16	859.9744999999999
	Dinamic	2	856.7089
		4	816.4488
		8	817.124
		16	829.5635
N=10000,M=10; n=m=5; (matrixD.txt)	Static	2	873.7447999999998
		4	789.5374
		8	804.5767000000001
		16	813.1384999999999
	Dinamic	2	875.1140000000001
		4	834.6226999999999
		8	835.6587
		16	823.1434999999999

Rezultatul fiecarui test a fost comparat cu rezultatul obtinut din variantele implementate pentru Laborator 1, pentru a asigura corectitudinea algoritmului si a implementarii.

Nota: fiecare test a fost rulat de 10 ori si pentru evaluarea timpului de executie s-a considerat media aritmetica a acestor 10 rulari. Fiecare test a fost rulat pe o masina cu Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 MHz 2 cores 4 Logical Processors cu sistem de operare Windows 10 versiunea 21H1.

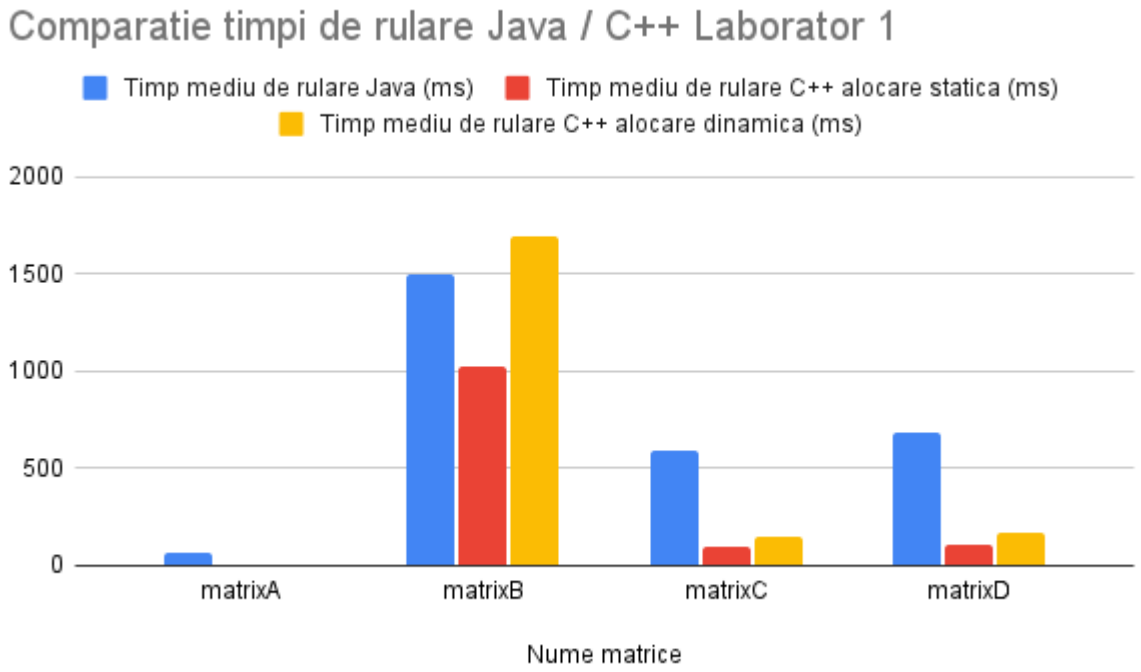
Analiza rezultatelor

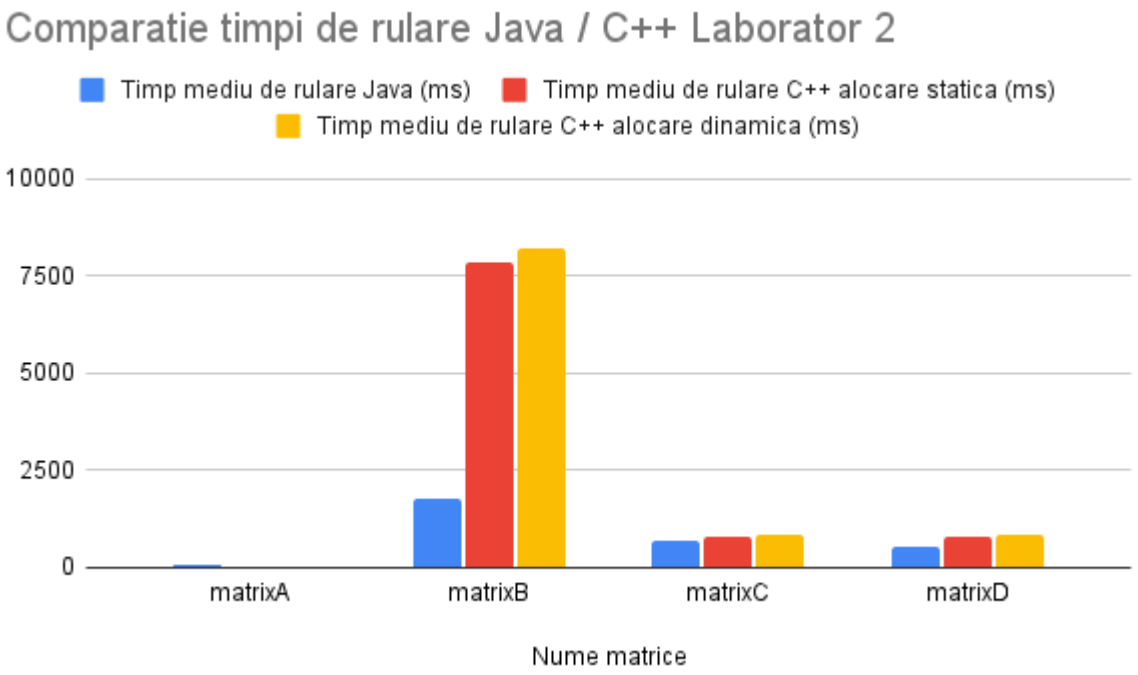
Pe masina pe care au fost efectuate testele, s-au obtinut timpi mai buni de executie la implementarile Java, decat la cele scrise in C++. Un posibil motiv ar fi faptul ca programele C++ au fost compilate cu un compilator diferit de cel din Laboratorul 1, din cauza unor erori neasteptate cu cel precedent.

Din punct de vedere al complexitatii timp, variantele scrise pentru Laboratorul 1 sunt mai rapide decat cele curente. Un posibil motiv este faptul ca fiecare thread trebuie sa isi copieze in zona proprie de memorie elementele

necesare aplicarii transformarii, un lucru neexistent in variantele precedente, unde thread-ul principal se ocupa de alocarea memoriei pentru matricea rezultat, iar thread-urile doar faceau calculele necesare. Aceasta copiere, de altfel, necesita folosirea unui mecanism de sincronizare, bariera, crescand astfel timpul de executie.

Urmatoarele 2 figuri prezinta o comparatie intre timpii de rularea obtinuti la Laborator 1 si Laborator 2. Se poate observa ca pentru Laboratorul 2 se obtin timpi de executie mult mai mari. Ca si exemplu concret, pentru matricea D, in urma Laboratorului 1, s-a obtinut un timp mediu de executie pentru implementarea C++ cu alocare statica de 110.158ms, un timp mult mai mic fata de cel 820.24935ms obtinut pentru Laboratorul 2.





Desi se obtin timpi mai mari de executie, variantele din Laboratorul 2 sunt mai eficiente din punct de vedere al memoriei, deoarece fiecare thread isi va copia doar un subloc din matricea data, iar schimbarile se vor face direct pe aceasta. In schimb, in implementarile din Laboratorul 1, se aloca o matrice de aceleasi dimensiuni cu cea initiala.