Picus Oskar Anton gr 236

PPD – Laborator 1

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:

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

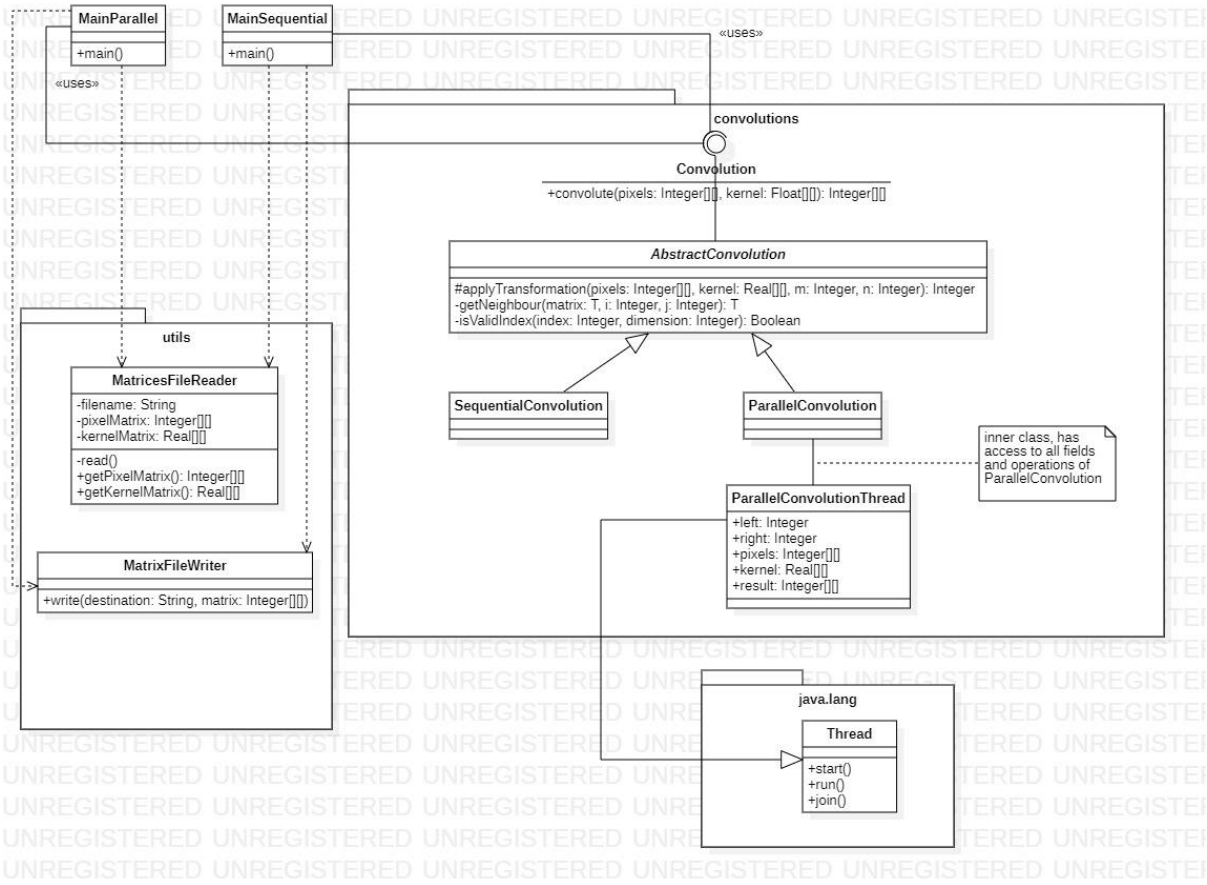
Constrangeri:

- 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

Diagrama ce prezinta clasele care au fost folosite in implementare, impreuna cu metodele lor si relatiile dintre ele. Sunt prezentate clasele folosite atat in varianta secventiala a programului, cat si in cea paralela.



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`**, care este comuna ambelor variante ale programului; ea efectueaza o singura transformare pe un singur pixel, identificat prin pozitia sa in matrice.

`SequentialConvolution` si **`ParallelConvolution`** extind aceasta clasa si implementeaza metoda **`convolute`**, aici surprinzandu-se diferenta dintre cele 2 variante de implementare. In plus, varianta paralela se foloseste si de clasa **`ParallelConvolutionThread`**, o abstractizare a conceptului de thread, in care, prin metoda run, defineste responsabilitatea unui singur fir de executie.

Diagrama de secventa corespunzatoare rularii programului – varianta secventiala:

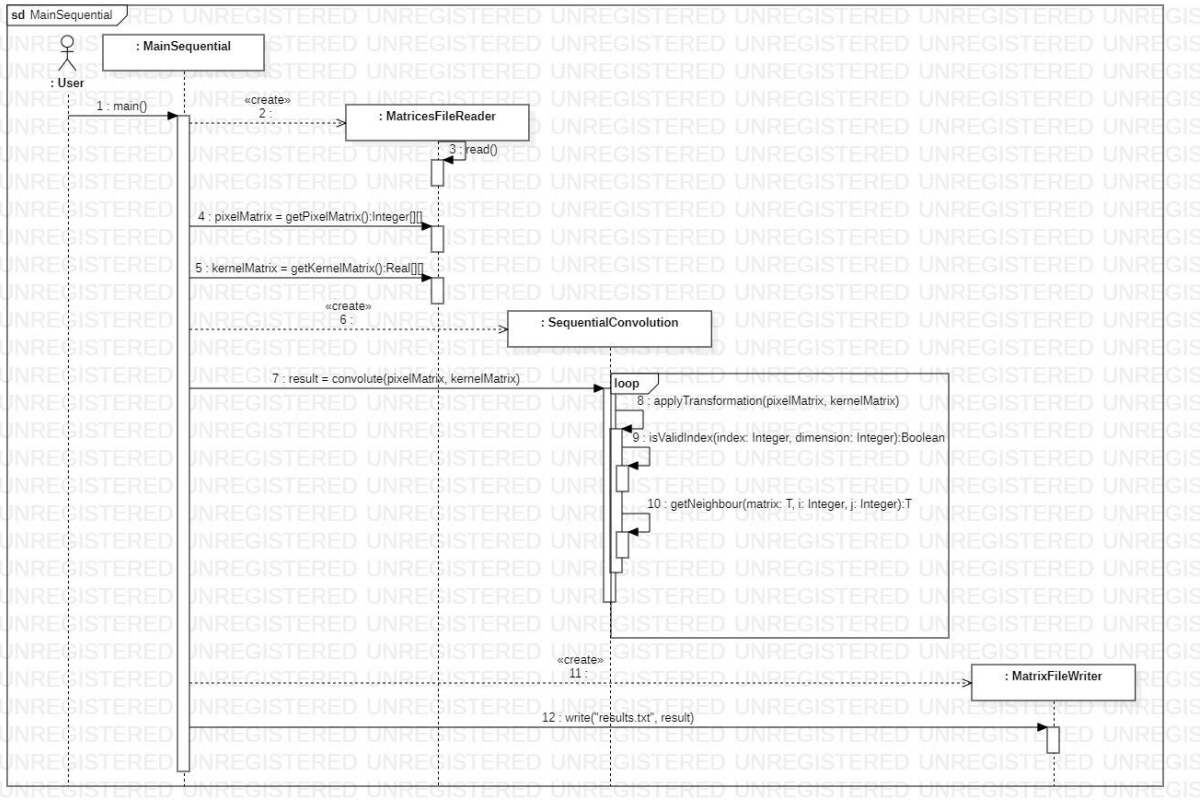
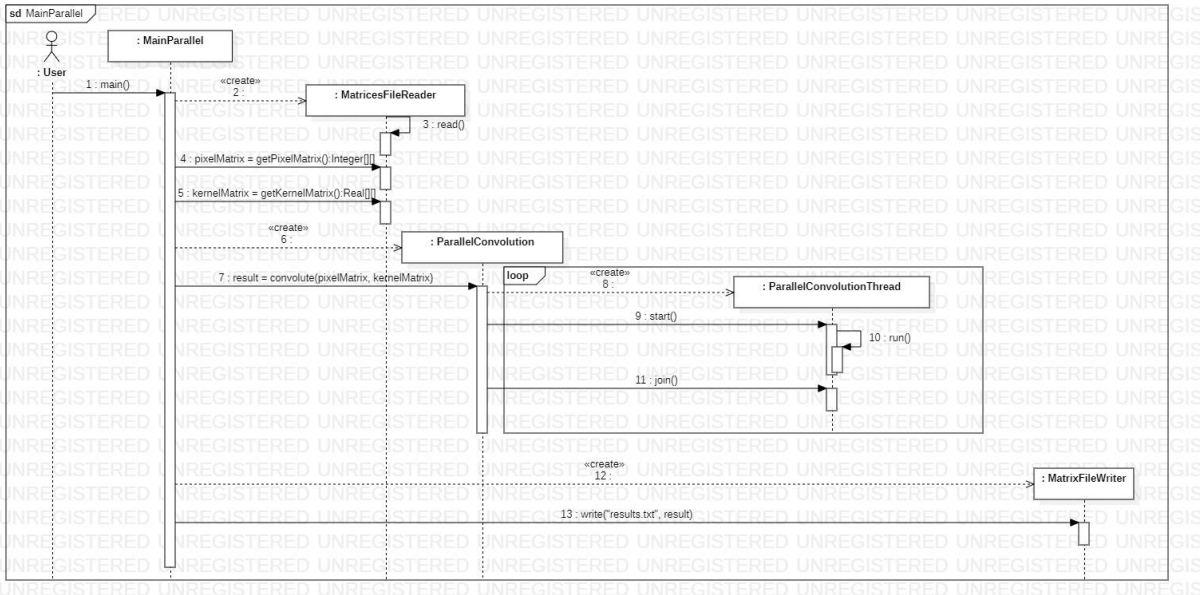


Diagrama de secventa corespunzatoare rularii programului – varianta secventiala:



Detalii de implementare

Daca varianta secventiala parcurge intreaga matrice si aplica pe fiecare pixel filtrul, varianta paralela 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.

Pentru generare de numere aleatoare pentru matrici, s-a folosit scriptul Shell **`generateMatrix.sh`**.

Fisierul de intrare contine pe prima linie numarul de linii si coloane a matricei de pixeli, urmata de continutul efectiv al matricei, iar apoi, de numarul de linii si coloane a kernel-ului si continutul efectiv. Fisierul de iesire contine doar matricea rezultat.

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)	secvential	66
	4	70
N=M=1000; n=m=5; (matrixB.txt)	secvential	1448
	2	1471
	4	1482
	8	1451
	16	1635
N=10,M=10000; n=m=5; (matrixC.txt)	secvential	608
	2	576
	4	575
	8	608
	16	605
N=10000,M=10; n=m=5; (matrixD.txt)	secvential	670
	2	698
	4	675
	8	661
	16	718

C++:

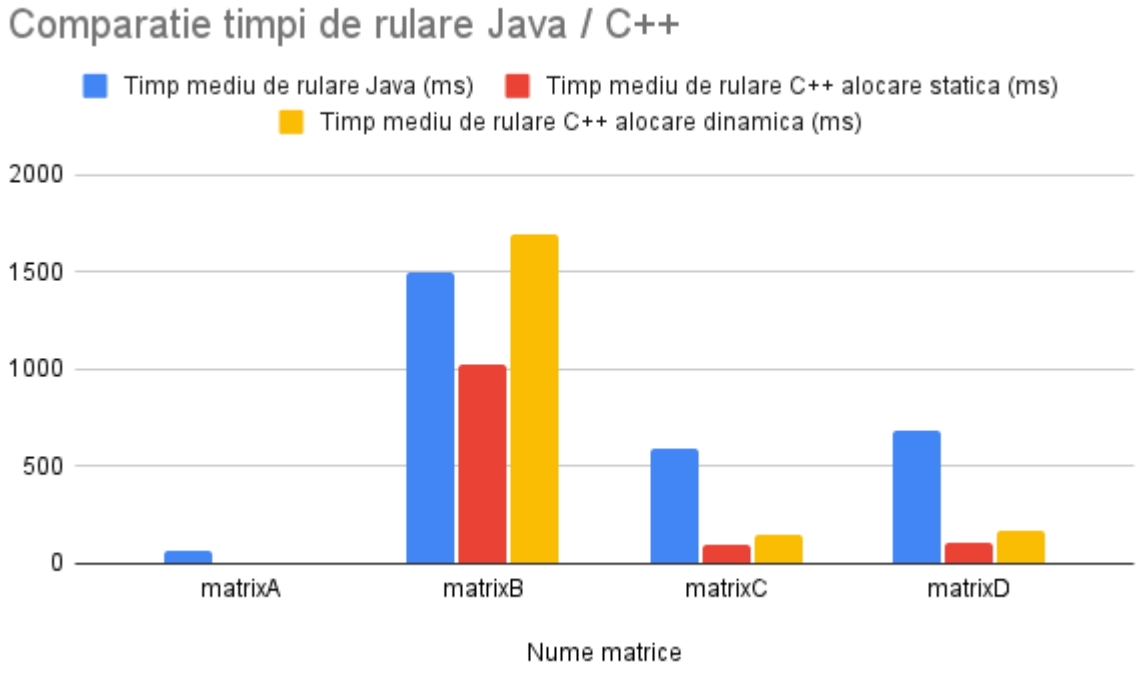
Tip matrice	**Tip alocare**	**Numar thread-uri**	**Timp executie (ms)**
N=M=10; n=m=3; (matrixA.txt)	Static	secvential	1.4414099999999999
		4	2.46266
	Dinamic	secvential	1.82179
		4	2.50951
N=M=1000; n=m=5; (matrixB.txt)	Static	secvential	1559.1709999999998
		2	1070.6949000000002
		4	819.2545
		8	829.1298999999999
		16	837.4440000000001
	Dinamic	secvential	2815.943
		2	1806.9450000000002
		4	1281.892
		8	1280.7990000000002
		16	1287.723
N=10,M=10000; n=m=5; (matrixC.txt)	Static	secvential	135.4925
		2	93.37429999999999
		4	75.26429999999999
		8	84.90047999999999
		16	83.979
	Dinamic	secvential	243.3602
		2	153.68
		4	119.7195
		8	114.98410000000001
		16	130.15439999999998
N=10000,M=10; n=m=5; (matrixD.txt)	Static	secvential	152.5906
		2	103.90881999999999
		4	83.3502
		8	88.64769
		16	122.29553000000001
	Dinamic	secvential	249.09709999999995
		2	177.8325
		4	127.78039999999999
		8	145.7406
		16	142.70890000000003

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

Se observa faptul ca implementarile scrise in Java sunt mai putin performante decat cele in C++, un posibil motiv fiind faptul ca limbajul C++ este unul mai low-level, mai apropiat de masina, in timp ce Java ofera abstractizari care pot mari timpul de executie. Graficul urmator prezinta o comparatie intre timpii de rulare medii in aceste limbaje pentru fiecare matrice testata.

Nume matrice	Timp mediu de rulare Java (ms)	Timp mediu de rulare C++ alocare statica (ms)	Timp mediu de rulare C++ alocare dinamica (ms)
matrixA	68	1.952035	2.16565
matrixB	1497.6	1023.13886	1694.6604
matrixC	594.4	94.602116	152.37964
matrixD	684.4	110.158568	168.6319



De asemenea, se poate observa ca variantele din C++ in care alocarea memoriei se face dinamic sunt mai lente decat cele cu alocare statica. Desi sunt mai eficiente din punct de vedere al memoriei utilizate, un posibil motiv pentru timpii de executie mai mari ar putea fi necesitatea dealocarii zonelor de memorie folosite la finalul programului, un pas pe care varianta secventiala il omite complet.

Desi matricile B, C si D au acelasi numar de elemente (1000000 de elemente), pentru acestea, timpii de executie sunt foarte variati. Se constata ca daca numarul de linii a matricei este mai mare, timpul de executie creste. O posibila explicatie ar fi legata de memoria cache, aducandu-se in avans memorie contigua. Matricile, fiind memorate in tablouri bidimensionale, beneficiaza de

acest mecanism, deoarece, pentru un anume element, se aduc in avans elementele de pe coloanele vecine. Astfel, sunt avantajate matricile cu un numar mai mic de linii, dar cu numar mai mare de coloane.

Timpii de executie pentru variantele paralele sunt dependente de masina pe care ruleaza testele, fiind influentate de aspecte precum numarul de core-uri ale masinii. Pentru masina pe care s-au rulat testele, se observa ca cei mai buni timpi de executie se obtin la rularea cu 4 thread-uri. De la un numar mai mare de thread-uri, se constata ca devine mult mai costisitor crearea si intretinerea lor.