Documentatie lab 2

Hodosi Attila - g234/1

Analiza cerintelor:

Se considera o imagine reprezentata printr-o matrice de dimensiune (MxN).

Aceasta se va transforma aplicandui o filtrare cu o matrice mai mica, numita kernel.

Kernelul va avea numarul liniilor si coloanelor, numar impar pentru a avea un element central.

Pe fiecare element din matricea initiala asezam elementul central din kernel, iar rezultatul filtrari este suma produsul elementelor dintre elementele kernelului si elementele matricei intiale, incepand de la elementul central.

Idee generala:

Pentru impartirea sarcinilor am alocat sarcini in mod echilibrat pentru fiecare thread.

Fiecare thread primeste un numar de linii (daca matricea are mai multe sau egale linii decat coloane) sau coloane (daca matricea are mai multe coloane decat linii) cu cel mult o linie sau coloana diferenta.

Ficare thread copieaza initial frontierele, liniile/coloanele care vor fi folosite de mai multe threaduri in matrici auxiliare. Dupa ce toate threadurile si-au copiat frontierele, se elibereaza bariera de sincronizare si fiecare thread va calcula individual partile aferente lor, folosind vectorii auxiliari care se actualizeaza corespunzator.

Bordare:

```
int linConvolution = linMatrix + linKernel - n / 2;
int colConvolution = colMatrix + colKernel - m / 2;
```

Pe linii:

```
if (colConvolution < 0) {
    colConvolution = 0;
}
if (colConvolution >= M) {
    colConvolution = M - 1;
}

if (linConvolution >= linEnd - 1) {
    convolutionResult += tempEnd[linConvolution - linEnd + 1][colConvolution] * kernel[linKernel][colKernel];
} else {
    convolutionResult += matrix[linConvolution][colConvolution] * kernel[linKernel][colKernel];
}
```

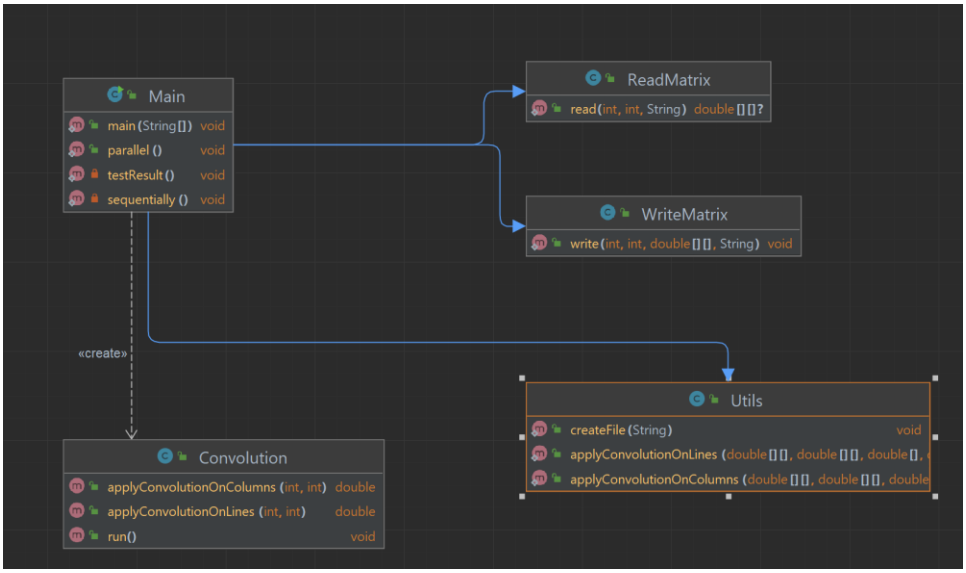
Pe coloane:

```java
if (linConvolution < 0) {
    linConvolution = 0;
}

if (linConvolution >= N) {
    linConvolution = N - 1;
}

if (colConvolution >= colEnd - 1) {
    convolutionResult += tempEnd[linConvolution][colConvolution - colEnd + 1] * kernel[linKernel][colKernel];
} else {
    convolutionResult += matrix[linConvolution][colConvolution] * kernel[linKernel][colKernel];
}
```

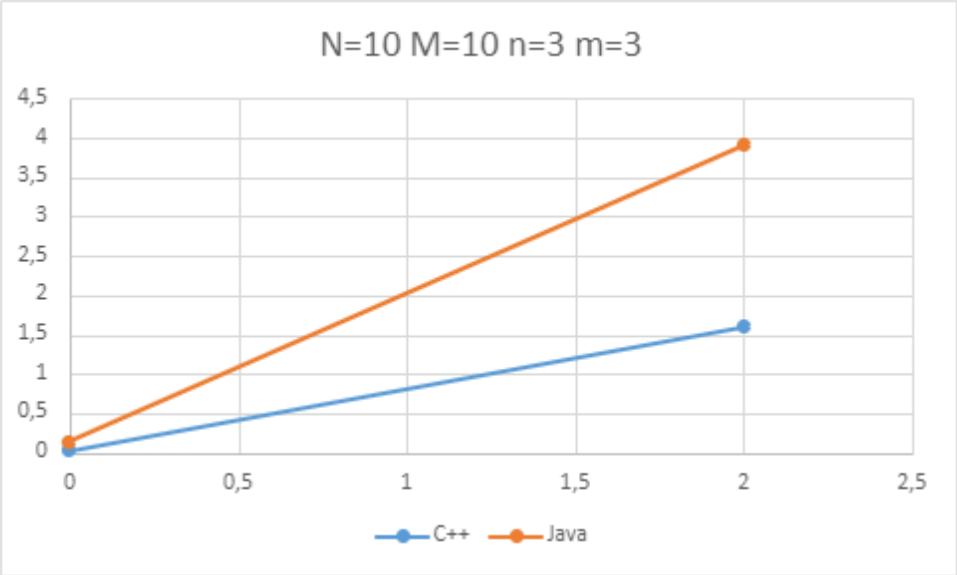
Diagrama de clase:

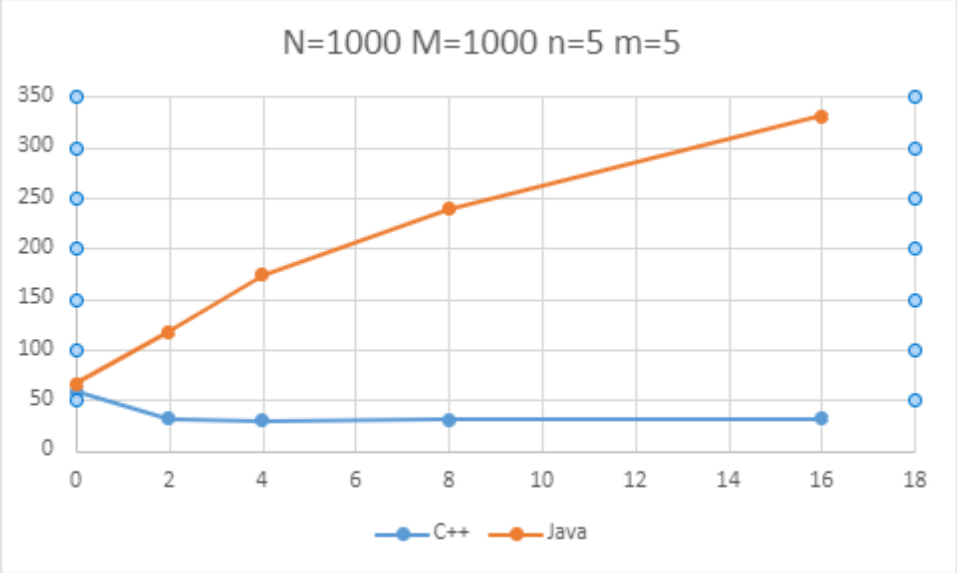


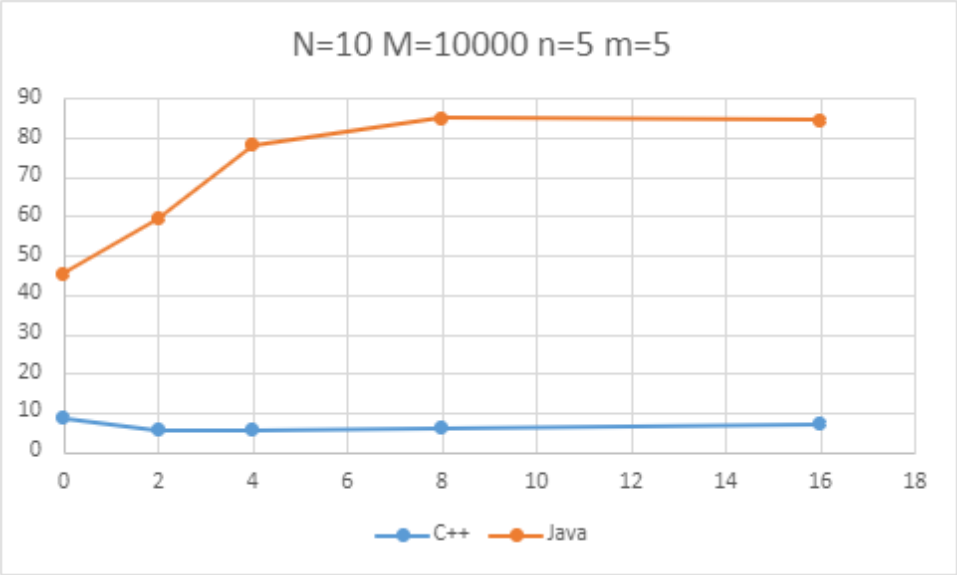
Comparatia generala in ms:

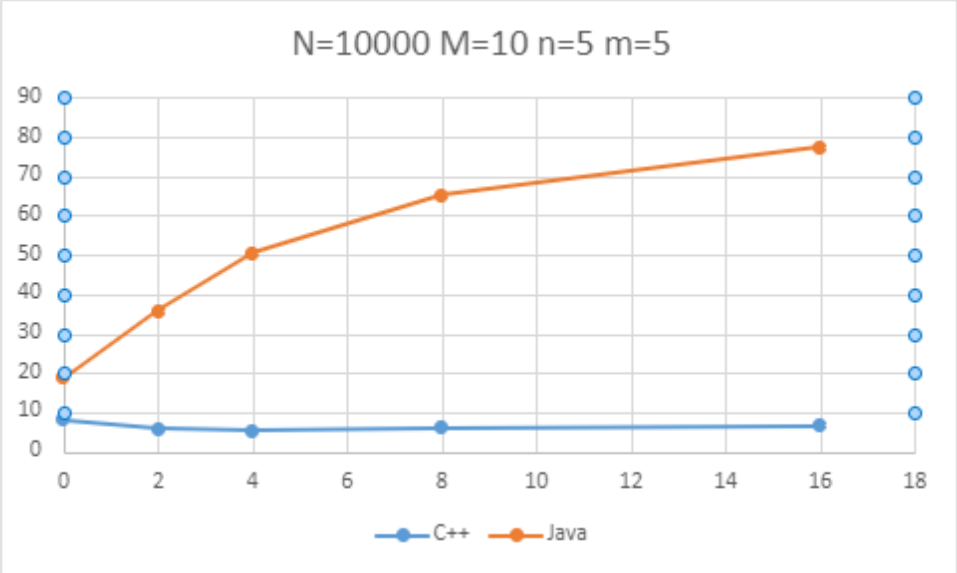
N	M	n2	m2	p	Time C++ static	Time Java
10	10	3	3	0	0,025653333	0,133926667
10	10	3	3	2	1,594546667	3,904173333
1000	1000	5	5	0	58,39510667	65,81461333
1000	1000	5	5	2	31,29344667	117,06524
1000	1000	5	5	4	28,90558667	173,3414067
1000	1000	5	5	8	30,65000667	238,59784
1000	1000	5	5	16	31,39212	330,7991067
10	10000	5	5	0	8,459926667	45,491
10	10000	5	5	2	5,520093333	59,30256
10	10000	5	5	4	5,455893333	78,1022
10	10000	5	5	8	6,104086667	84,90548
10	10000	5	5	16	6,931953333	84,45648
10000	10	5	5	0	7,925846667	18,68887333
10000	10	5	5	2	5,644826667	35,78901333
10000	10	5	5	4	5,194933333	50,39106667
10000	10	5	5	8	5,96242	65,19790667
10000	10	5	5	16	6,549333333	77,45006667

Diagrame:


N=10 M=10 n=3 m=3
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
0,5
1
1,5
2
2,5
C++
Java


N=1000 M=1000 n=5 m=5
350
300
250
200
150
100
50
0
0
2
4
6
8
10
12
14
16
18
C++
Java





Concluzie generala:

- folosirea threadurilor pe matrici mici incetineaza calculele

- folosirea threadurilor pe matrici mari aduce imbunatatiri semnificative, dar exista o limita la numarul threadurilor de unde performanta nu creste semnificativ sau chiar scade, deci trebuie avut grija sa alegem numarul corect de threaduri, nici prea putin nici, dar nici prea mult .

Analiza complexitatii de spatiu:

-Pe linii:

O(M x N + 2 x (n / 2 + 1) x M x p + M + n x m)

-Pe coloane:

O(M x N + 2 x N x (m / 2 + 1) x p + N + n x m)

Comparatie cu alte metode:

Metoda 2:

1. Fiecare thread copiaza intai vecinii (frontiera) pe care trebuie sa ii foloseasca
2. Seteaza (flag = true)
3. Verifica (vecin.flag=true pentru toti vecinii) si daca da atunci aplica 4.
4. Actualizeaza elementele folosind vectorii auxiliari care se actualizeaza corespunzatori.

Complexitate: O(M x N + 2 x (n / 2 + 1) x M x p + M + n x m + p)

Metoda 3:

5. Fiecare thread calculeaza valorile noi dar pentru celulele aflate pe frontiera (vecinatate cu alte threaduri) valorile noi se salveaza in vectori separate
6. Fiecare thread calculeaza valorile noi dar pentru celulele care nu sunt pe frontiera iar valorile noi se salveaza direct in matrice.
7. Bariera de sincronizare
8. Actualizeaza elementele de pe frontiera in matrice prin copierea vectorilor calculati la 1.

Complexitate: la fel

Metoda 3:

9. Fiecare thread calculeaza valorile noi pentru celulele aflate pe frontiera (vecinatate cu alte threaduri) si le salveaza in vectori separati
10. Seteaza (flag = true)–
11. Fiecare thread calculeaza valorile noi dar pentru celulele care nu sunt pe frontiera iar valorile noi se salveaza direct in matrice.
12. Verifica (vecin.flag=true, pentru toti vecinii) si daca da atunci aplica 5.
13. Actualizeaza elementele de pe frontiera prin copierea vectorilor calculati la 1.

Complexitate: Complexitate: O(M x N + 2 x (n / 2 + 1) x M x p + M + n x m + p)