Laborator 2 PPD – Salsigan Razvan-Dan 236/2

## Cerinta

TLDR: Se da o matrice de pixeli si un kernel, se cere sa se aplice o convolutie cu acel kernel pe matricea intiala si sa se calculeze timpul de executie, folosind un numar diferit de threaduri. Se cere o solutie eficienta din pdv al memoriei utilizate.

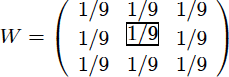
Se considera o imagine reprezentata printr-o matrice de pixeli, F , de dimensiune (MxN).

Se cere transformarea ei aplicand o filtrare cu o fereastra definita de multimea de indici W cu coeficientii *wkl (*reprezentati prin matricea W[k,l], unde -n/2<=k<=n/2, -m/2<=l<=m/2; si n<N, m<M, n,m impare).

Transformarea unui pixel:



De exemplu:



v[m,n] = f[m,n] \*1/9+ f[m-1,n]\* 1/9+

f[m,n-1]\* 1/9+

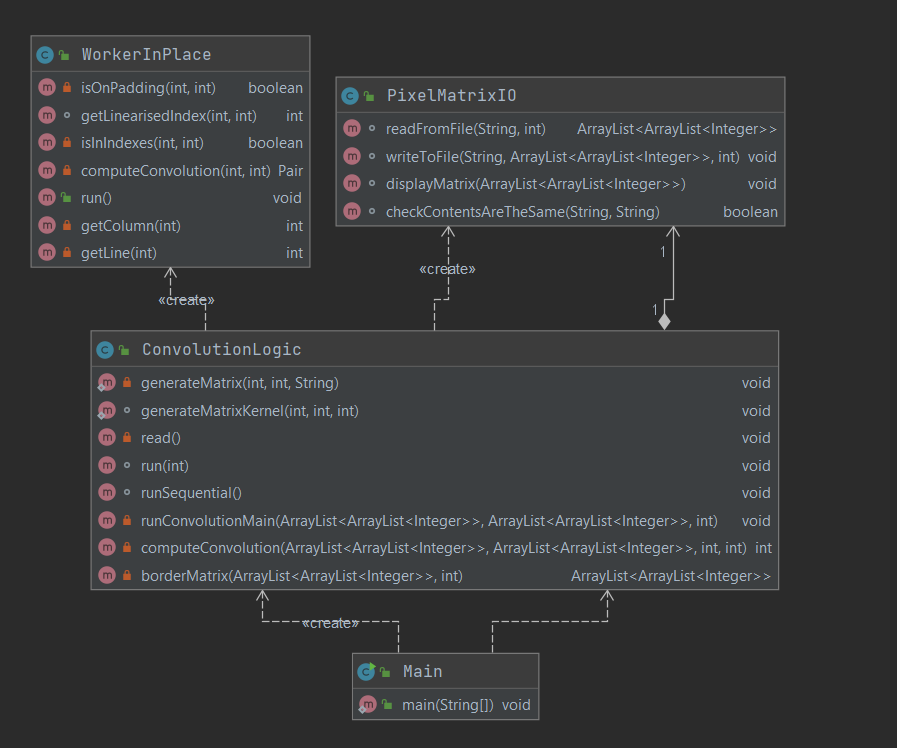
f[m-1,n-1]\* 1/9+ f[m+1,n]\* 1/9+ f[m,n+1]\* 1/9+ f[m+1,n+1]\* 1/9

Pentru frontiere se considera ca un element este egal cu elemental din celula vecina din matrice f[-1,j]= f[0,j]; f[i,-1]=f[i,0]; f[M,j]= f[M-1,j]; f[i,N]=f[i,N-1];

Se cere ca solutia sa nu foloseasca o matrice rezultat, ci sa se lucreze numai cu matricea de input.

## Exemplificare -> https://de.wikipedia.org/wiki/Datei:2D\_Convolution\_Animation.gif

## Arhitectura



PixelMatrixIO se ocupa cu scrierea/citirea din fisere.

Pentru C++ structura este similara, dar am creat o clasa pentru alocarea dinamica si cea statica, iar in loc de worker am creat doar o functie doWork pe care o pasez threadului.

## Impartirea elementelor

Elementele au fost distribuite threadurilor liniar, pe chunkuri de elemente consecutive, fiecare thread executand operatia pe numar\_elemente/numar\_threaduri [+1] pixeli. (Threadul primeste un start si un end, corespunzand indicilor din matricea liniarizata)

Determin cate elemente am (n\*m), aflu numarul de chunkuri impartind numarul de elemente la numarul de threaduri si distribui restul, cate un element pe fiecare thread pana il epuizez.

Deoarece am facut paddingul in-place, am grija sa pasez workerului inclusiv indicii din padding, doar ca acestia nu vor fi prelucrati, deci distributia uniforma se pastreaza.

La fiecare iteratie, finalul ia valoarea start\_precedent+chunk\_size+[1]

## Sincronizare

Sincronizarea threadului se face folosind o bariera (abordarea 2b). Fiecare thread determina elementele aflate pe frontiera, si le salveaza noile valori local. Pentru elementele ce nu sunt pe frontiera, valorile se stocheaza intr-o ***coada*** de dimensiunea kernelului, folosind euristica pentru care, pentru distributia liniara, nu va mai fi nevoie de valoarea initiala a elementului de pe pozitia curenta-kernelSize^2-1.

Dupa ce toate threadurile ajung la bariera, elementele de pe frontierele fiecarui thread sunt si ele stocate in matricea initiala.

**Complexitatea** **spatiu** ajunge sa fie O(m\*n+k\*noOfThreads+border\_size), unde m si n sunt dimensiunile matricei bordate, k este dimensiunea kernelului si noOfThreads este numarul de threaduri, iar border\_size este numarul de elemente de pe frontier, care e 2\*m\*(k/2), ceea ce este semnificativ mai putin fata de varianta cu matrice rezultat.

## Padding

Pentru padding am o metoda care primeste matricea initiala si dimensiunea marginii (@border = 1 pentru kernel de 3x3, 2 pentru kernel de 5x5 etc) si o bordeaza corespunzator (citirea s-a facut in interiorul marginilor, punand 0 unde trebuie sa vina paddingul).

Parcurg pe rand primele @border randuri si le completez cu elementele de pe prima linie din matricea initiala (si cele de pe colt in colt – primele/ultimele @border elemente de pe primele/ultimele @border linii vor primi valoarea din coltul matricii initiale), analog pentru ultimele @border randuri cu elementele de pe ultima linie din matricea initiala, dupa care fac acelasi procedeu pentru bordarea pe coloane.

//top and bottom rows  
for (int i = 0; i < border; i++) {  
 int col = 0;  
 //first @border elements (corners)  
 for (int j = 0; j < border; j++) {  
 pixelMatrix.get(i).set(col, pixelMatrix.get(border).get(border));  
 pixelMatrix.get(lines - border + i).set(col, pixelMatrix.get(lines - border - 1).get(border));  
 col++;  
 }  
 for (int j = border; j < columns - border; j++) {  
 pixelMatrix.get(i).set(col, pixelMatrix.get(border).get(j));  
 pixelMatrix.get(lines - border + i).set(col, pixelMatrix.get(lines - border - 1).get(j));  
 col++;  
 }  
 //last @border elements (corners)  
 for (int j = 0; j < border; j++) {  
 pixelMatrix.get(i).set(col, pixelMatrix.get(border).get(columns - border - 1));  
 pixelMatrix.get(lines - border + i).set(col, pixelMatrix.get(lines - border - 1).get(columns - border - 1));  
 col++;  
 }  
}  
//fill rest of matrix  
for (int i = border; i < lines; i++) {  
 int col=0;  
 for (int j = 0; j < border; j++) {  
 pixelMatrix.get(i).set(col, pixelMatrix.get(i).get(border));  
 col++;  
 }  
 col = columns - border;  
 for (int j = 0; j < border; j++) {  
 pixelMatrix.get(i).set(col, pixelMatrix.get(i).get(columns - border - 1));  
 col++;  
 }  
}

## Analiza timpilor de executie

Testele au fost efectuate pe un procesor i5-9300H (4C, 8T) , in Windows

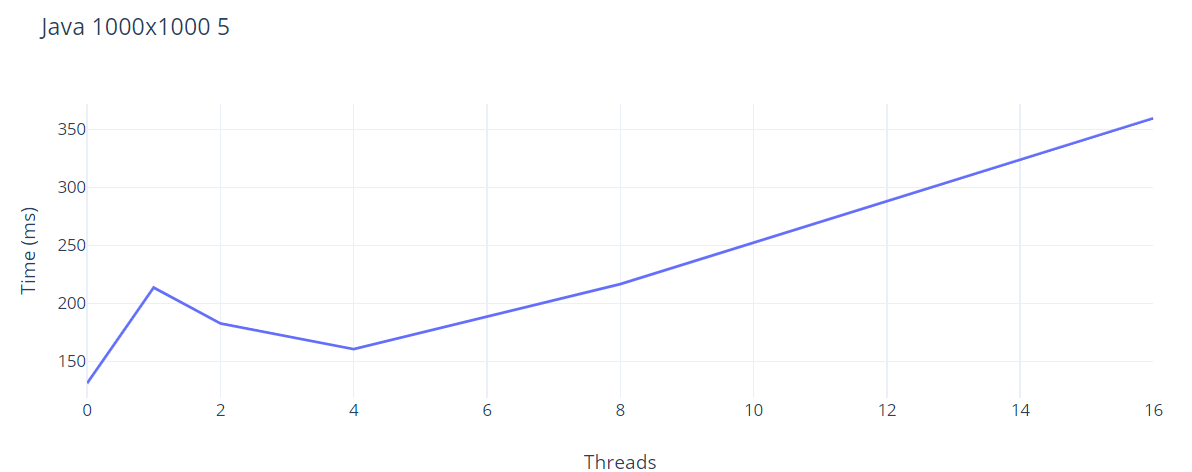
Timpul afisat este in ms

### Java

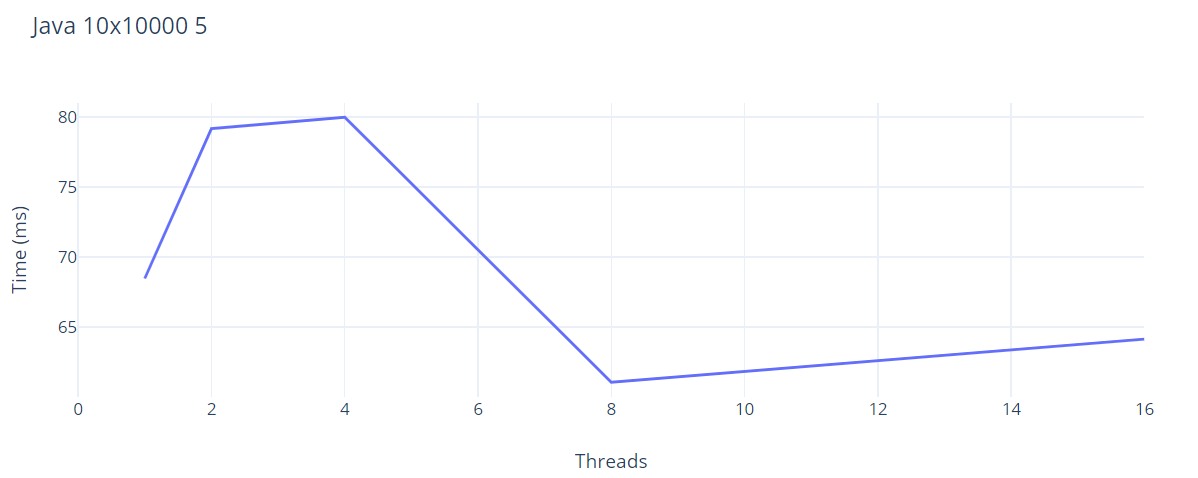
|  |  |  |
| --- | --- | --- |
| Tip matrice | Nr threads | Timp executie |
| N=M=10  n=m=3 | secvential | 0.44522 |
| 1 | 2.03792 |
| 4 | 3.62969 |
| N=M=1000  n=m=5 | secvential | 131.3603 |
| 1 | 213.756 |
| 2 | 182.8721 |
| 4 | 160.8399 |
| 8 | 216.67 |
| 16 | 359.5412 |
| N=10 M=10000  n=m=5 | secvential | 31.50909 |
| 1 | 68.47085 |
| 2 | 79.17357 |
| 4 | 79.98156 |
| 8 | 61.0678 |
| 16 | 64.14842 |
| N=10000 M=10  n=m=5 | secvential | 37.36046 |
| 1 | 48.42546 |
| 2 | 67.36394 |
| 4 | 78.24558 |
| 8 | 71.26753 |
| 16 | 81.56358 |

#### Grafice

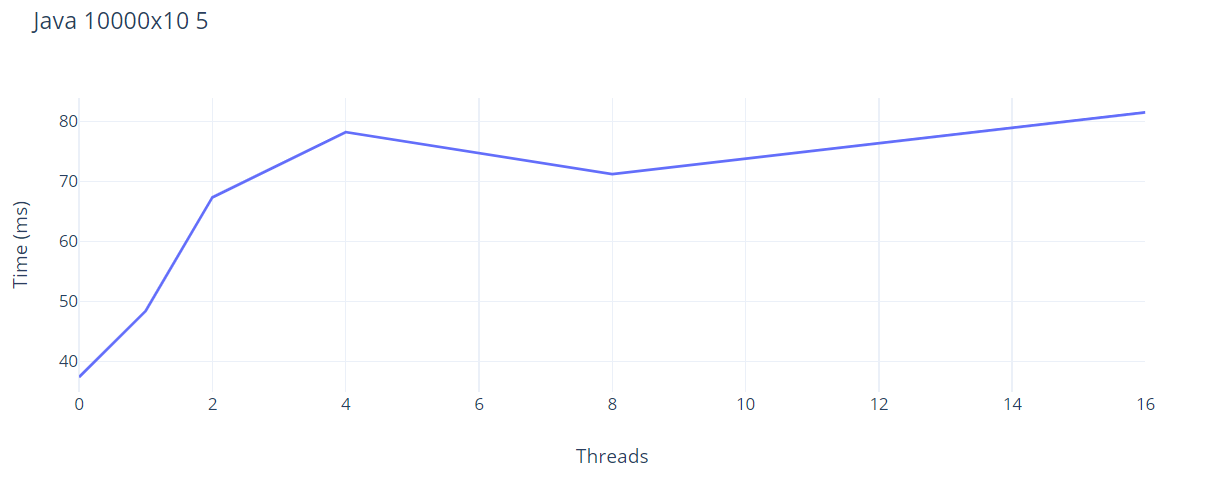
1. N=M=1000 n=m=5



1. N=10 M=10000 n=m=5



1. N=10000 M=10 n=m=5

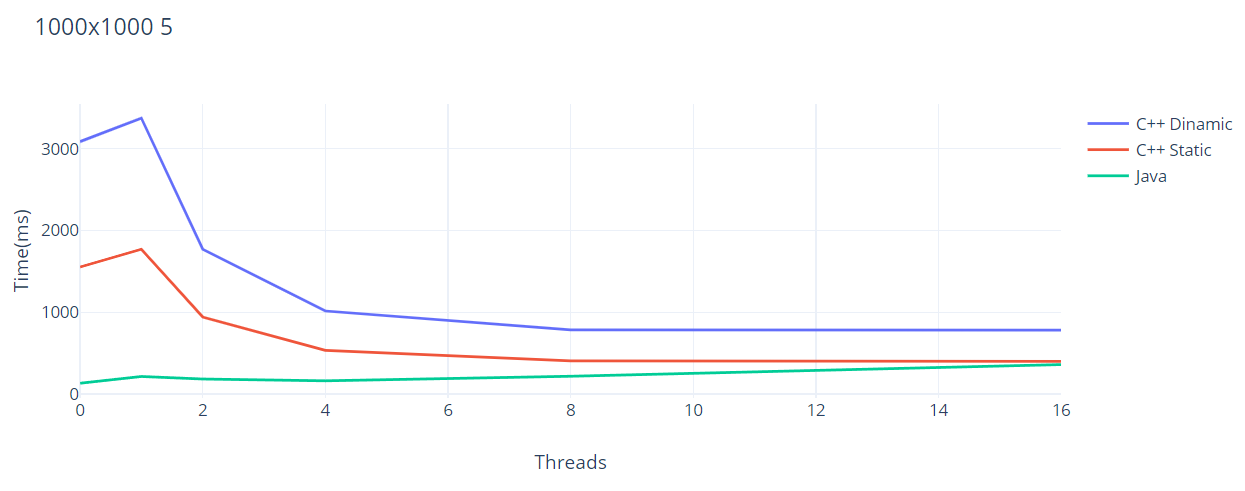


### C++

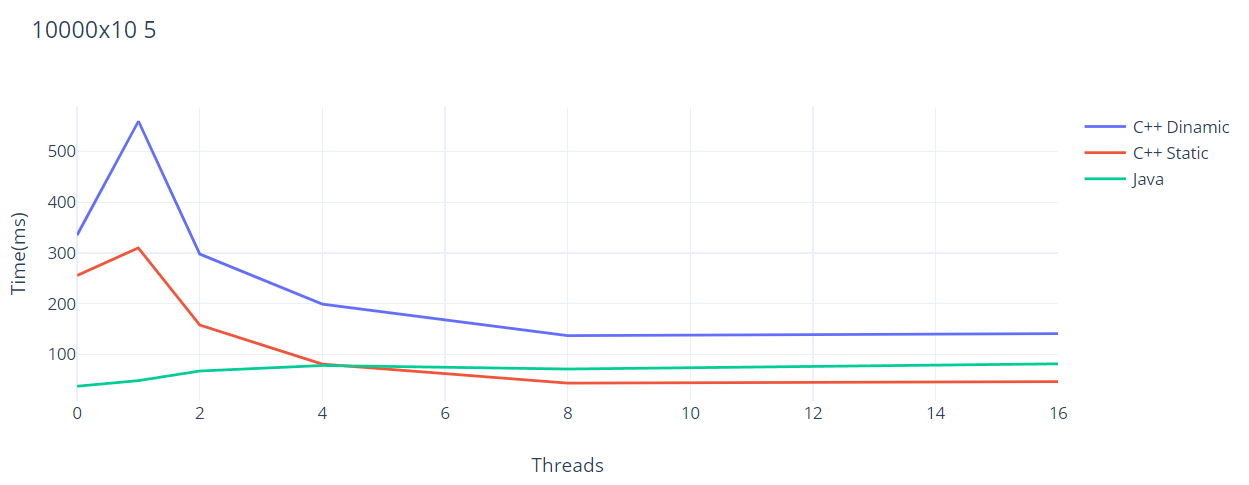
|  |  |  |  |
| --- | --- | --- | --- |
| Tip matrice | Tip alocare | Nr threads | Timp executie |
| N=M=10  n=m=3 | Static | secvential | 0.06882 |
| 1 | 1.6313 |
| 4 | 2.58076 |
| Dinamic | secvential | 0.04674 |
| 1 | 2.51174 |
| 4 | 4.11433 |
| N=M=1000  n=m=5 | Static | secvential | 1553.678 |
| 1 | 1771.208 |
| 2 | 941.2443 |
| 4 | 532.6621 |
| 8 | 405.2687 |
| 16 | 397.5411 |
| Dinamic | secvential | 3086.716 |
| 1 | 3372.677 |
| 2 | 1770.713 |
| 4 | 1014.646 |
| 8 | 783.1367 |
| 16 | 779.7244 |
| N=10 M=10000  n=m=5 | Static | secvential | 154.9778 |
| 1 | 172.2177 |
| 2 | 83.0475 |
| 4 | 39.20912 |
| 8 | 22.31084 |
| 16 | 21.05494 |
| Dinamic | secvential | 186.8114 |
| 1 | 334.2703 |
| 2 | 167.7691 |
| 4 | 90.35751 |
| 8 | 60.56828 |
| 16 | 64.41744 |
| N=10000 M=10  n=m=5 | Static | secvential | 255.8794 |
| 1 | 310.3328 |
| 2 | 158.2049 |
| 4 | 80.86108 |
| 8 | 43.3087 |
| 16 | 46.33823 |
| Dinamic | secvential | 335.148 |
| 1 | 559.9844 |
| 2 | 298.0087 |
| 4 | 199.3794 |
| 8 | 137.0906 |
| 16 | 141.2368 |

#### Grafice

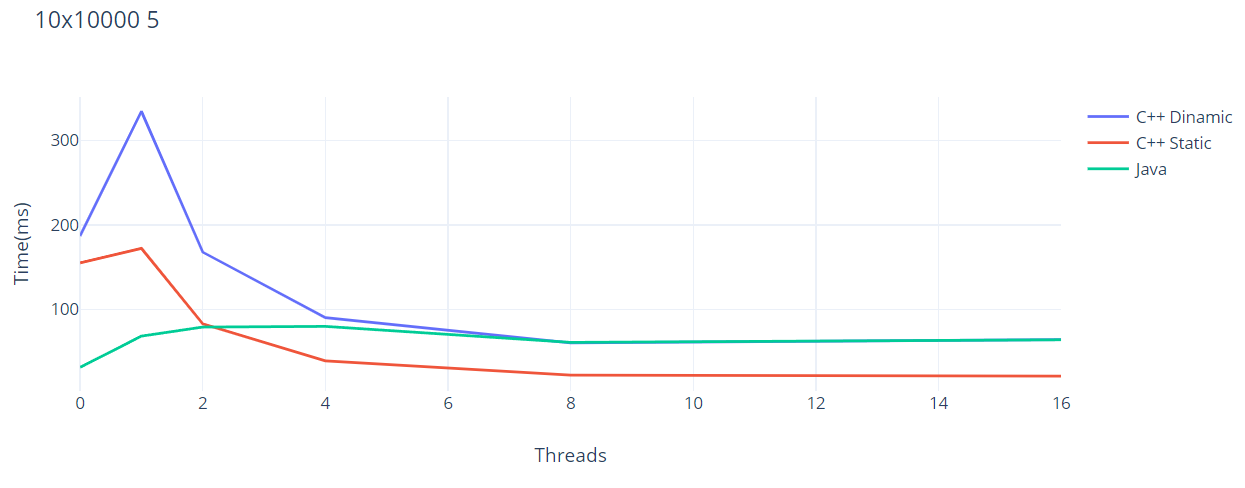
1. N=M=1000 n=m=5



1. N=10 M=10000 n=m=5



1. N=10000 M=10 n=m=5



## Concluzie

Varianta de C++ statica a rulat net superior variantei Java, mai putin in varianta cu matrice de 1000x1000, unde cele doua au avut rezultate similare, iar varianta de C++ dinamica a rulat cel mai incet dintre cele 3.

In Java varianta secventiala a rulat mai bine decat orice varianta paralela. In C++ variantele paralele au demonstrat imbunatatiri considerabile a timpului de executie.

Comparativ cu laboratorul precedent, timpii de executie au crescut semnificativ, in unele cazuri pana la de 10x mai mult, media situandu-se undeva la 2x, deoarece complexitatea calculelor si a instructiunilor if a crescut considerabil.

Comparativ cu celelalte 3 variante de sincronizare, algoritmul s-a comportat astfel:

1b. Complexitate spatiu egala, deoarece se salveaza frontierele, dar si intern se foloseste o coada, practic acelasi numar de elemente vor fi stocate, doar ca au semnificatii diferite

1f. Complexitate spatiu mai mare, deoarece pe langa coada interna si frontiere, mai e nevoie si de falguri pentru fiecare element in parte

2f. Complexitate spatiu sensibil mai buna decat in cazul 2f, dar tot mai slaba decat la cele cu bariere, deoarece este nevoie de stocarea flagurilor, pe langa spatiul folosit de stocarea noilor valori de pe frontiere si a cozii interne.