# Rezultate lab1

Enunț

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

Ind[n,m] = {(k,l) | -n/2<=k<=n/2, -m/2<=l<=m/2}

si de coeficientii wkl

n,m impare si n<N, m<M.

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] }

unde

De exemplu:

multimea de indici este

Ind [3,3] ={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (-0,1), (1,-1), (1,0), (1,1)}

Si ponderile asociate

Actualizarea unui pixel de pe pozitia (i,j)

v[i,j] =f[i,j] \*1/9+f[i-1,j]\* 1/9+f[i,j-1]\* 1/9+f[i-1,j-1]\* 1/9+f[i+1,j]\* 1/9+f[i,j+1]\* 1/9+f[i+1,j+1]\* 1/9

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

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

Se cere asigurarea urmatoarei postconditii:

Postconditie: Matricea rezultat V contine imaginea filtrata a imaginii initiale F (unde V != F)

A) Program secvential

B) Program paralel: folositi p threaduri pentru calcul.

Obiectiv: Impartire cat mai echilibrata si eficienta a calculul pe threaduri!

Pentru impartirea sarcinilor de calcul (taskuri) se foloseste descompunere geometrica care poate fi (puteti alege o varianta sau sa incercati mai multe si sa o identificati pe cea mai buna):

- Pe orizontala (mai multe linii alocate unui thread)

- Pe verticala (mai multe coloane alocate unui thread)

- Bloc – submatrici alocate unui thread

- bazat pe o functie de distributie prin care unui index al unui thread i se distribuie o submultime de indecsi din matrice;

distributia se poate face prin:

- distributie liniara (indici alaturati la acelasi thread) sau

- distributie ciclica( cu pas egal cu p).

Datele de intrare se citesc dintr-un fisier de intrare “date.txt”.

(Fisierul trebuie creat anterior prin adaugare de numere generate aleator. Toate rularile trebuie executate cu acelasi fisier.)

Implementare:

a) Java

b) C++ ( cel putin C++11 )

i. matricile sunt alocate static (int f[MAX][MAX] )

ii. matricile sunt alocate dynamic (new…)

Folosire directa a threadurilor (creare explicita) => nu se permite folosirea executorilor.

Testare: masurati timpul de executie pentru

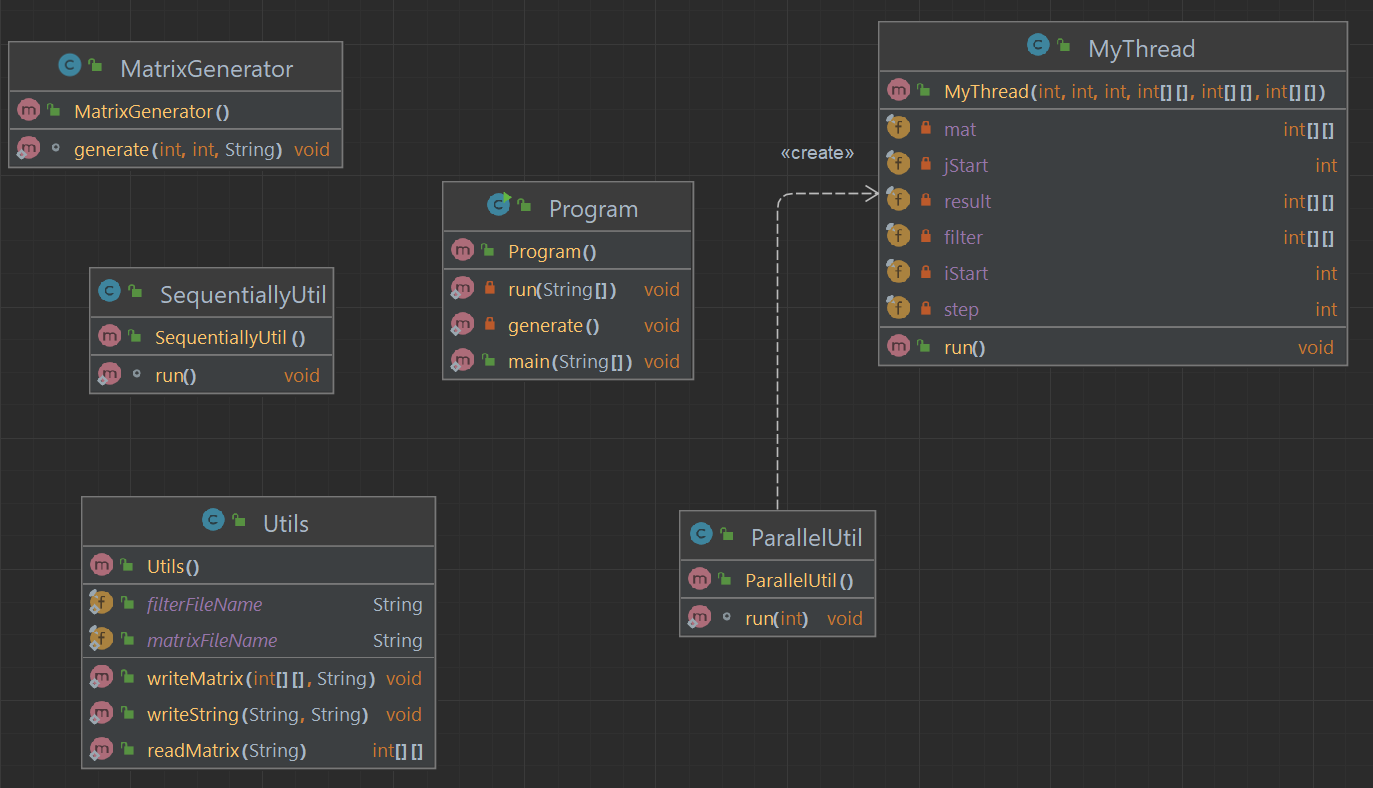
1) N=M=10 si n=m=3; p=4;

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

Java



C++

|  |
| --- |
| Program |
| N: Int  M: Int  n: Int  p: Int  mat: Int[][]  filter: Int[][]  result: Int[][]  mat2: Int\*\*  filter2: Int\*\*  result2: Int\*\* |
| runSequentiallyStatic(): void  runParallelStatic(): void  runSequentiallParallel(): void  runParallelParallel(): void  worker(int id, int iStart, int jStart, int step): void |

Partiționarea datelor

* Matricea rămâne cum este citită din fișier si fiecare thread primește un iStart, jStart și pasul (nr de thread-uri)
* Thread-ul calculează valoarea elementului de pe poziția iStart jStart și
* După calcularea noii valori, thread-ul isi crește jStart-ul cu p; dacă jStart depășește numărul de coloane, iStart crește cu o unitate iar jStart scade cu M(numărul de coloane)

j = j + step;

while (j >= M) {

j = j - M;

i++;

}

* Astfel, fiecare thread parcurge circular matricea
* Implementările în Java și C++ folosesc aceeași idee, structurată diferit

Java

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr threads** | **Timp executie** |
| N = M = 10  n = m = 3 | secvential | 41.21031 |
| 4 | 41.45196 |
| N = M =1000  n = m = 5 | secvential | 634.41466 |
| 2 | 608.22596 |
| 4 | 601.38145 |
| 8 | 607.16303 |
| 16 | 628.85544 |
| N = 10  M = 10000  n = m = 5 | secvential | 167.66607 |
| 2 | 184.13589 |
| 4 | 182.60336 |
| 8 | 189.19932 |
| 16 | 218.3829 |
| N = 10000  M = 10  n = m = 5 | secvential | 159.43449 |
| 2 | 149.63527 |
| 4 | 159.56199 |
| 8 | 174.82202 |
| 16 | 199.98314 |

C++

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr threads** | **Timp executie** |
| N = M = 10  n = m = 3 | static | 1 | 0.12328 |
| 4 | 1.00093 |
| dinamic | 1 | 0.12579 |
| 4 | 1.06824 |
| N = M =1000  n = m = 5 | static | 1 | 469.4386 |
| 2 | 492.4699 |
| 4 | 479.7567 |
| 8 | 478.1051 |
| 16 | 489.0608 |
| dinamic | 1 | 516.7777 |
| 2 | 500.2446 |
| 4 | 493.6726 |
| 8 | 485.7096 |
| 16 | 511.6221 |
| N = 10  M = 10000  n = m = 5 | static | 1 | 46.45639 |
| 2 | 50.76811 |
| 4 | 50.21274 |
| 8 | 50.06021 |
| 16 | 53.64017 |
| dinamic | 1 | 56.25396 |
| 2 | 52.53376 |
| 4 | 50.75353 |
| 8 | 50.4967 |
| 16 | 50.64713 |
| N = 10000  M = 10  n = m = 5 | static | 1 | 48.11447 |
| 2 | 54.90315 |
| 4 | 50.80695 |
| 8 | 50.56927 |
| 16 | 50.56041 |
| dinamic | 1 | 54.54655 |
| 2 | 53.22703 |
| 4 | 51.66837 |
| 8 | 53.00789 |
| 16 | 51.85359 |

Concluzii

* Se poate observa că implementarea în C++ este mult mai rapidă
* La Java nu putem construi o opinie solidă: așteptările mele au fost ca în fiecare caz de testare, cel mai bun timp să fie la 8 thread-uri, dar se observă ca în fiecare caz, tot alt număr este ideal
* La C++, total contrar așteptărilor, în abordarea secvențială și cu alocare statică a fost cel mai bun timp, indiferent de dimensiunea datelor
* De menționat, la C++ cazul cu alocare dinamică pe 8 thread-uri se apropie de minumul obținut de mine