Laborator 1

*Documentație*

*Enunț*

Se consideră o imagine reprezentată printr-o matrice de pixeli *F* de dimensiune N\*M.

Se cere transformarea ei aplicând o filtrare cu o fereastră reprezentată printr-o altă matrice *w* de dimensiune n\*m, n, m impare, n < N, m < M, după formula:

V[i, j] = { | -n/2 <= k <= n/2, -m/2 <= l <=m/2}

unde *V* este matricea-rezultat.

În cazul frontierelor se consideră că un element este egal cu cel mai apropiat element vecin din matricea *F* ( 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], etc.).

Exemplificare -> <https://de.wikipedia.org/wiki/Datei:2D_Convolution_Animation.gif>

*Abordare*

Datele de intrare se citesc din fișiere ce conțin matrici de tip *F* de dimensiune N\*M cu valori întregi generate randomizat, respectiv matrici de tip *w* de dimensiune n\*m cu valori întregi generate randomizat.

Java

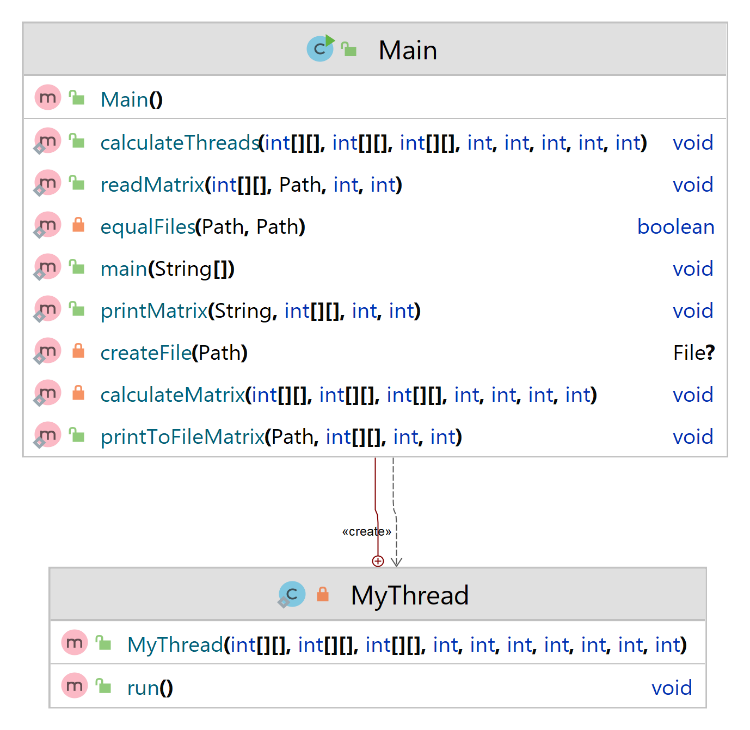


Diagrama de clase

Pentru calculul secvențial am folosit funcția *calculateMatrix*, care parcurge fiecare element al matricei *F* și, cu ajutorul matricei *w* și al operațiilor definite, calculează matricea *V*.

*\*Împărțirea workloadului pe threaduri*

Pentru calculul pe threaduri am folosit clasa *MyThreads*, ce extinde clasa Threads, și funcția *calculateThreads*. Distribuția indecșilor matricei *F* fiecărui thread în parte se face ciclic. Fiecare thread va începe cu elementul V[iStart][jStart] (pentru threadul 1 iStart = 0, jStart = 0, pentru threadul 2 iStart = 0, jStart = 1 etc.) și, din *pas* în *pas* elemente, unde *pas* = numărul de threaduri alocate (*no\_threads)*, va parcurge matricea *F*. În cazul în care parcurgerea iese din frontiera matricei, se calculează următorul element din interiorul matricei în funcție de dimensiunea M, *matrix\_columns*.

O imagine care conține text

Descriere generată automat

*Folosind această distribuție a elementelor matricei se asigură atât împărțirea unui număr egal de elemente între threaduri și deci de calcule de efectuat, precum și faptul că toate threadurile asignate vor fi utilizate și niciunul dintre ele nu va efectua un calcul redundant, efectuat deja de alt thread.*

Clasa *MyThreads* implementează în metoda *run()* calculul matricei *V* descris mai sus.

Funcția *calculateThreads* construiește un array de *MyThreads* de dimensiunea dorită (*no\_threads*) și inițializează fiecare obiect din array cu datele necesare (matricile *F*, *w*, *V*, dimensiunile, *iStart* și *jStart*), actualizând pentru fiecare nou thread valorile *iStart* și *jStart*. De asemenea, pornește și unește threadurile cu threadul clasei *Main*, după finalizare.

Durata de execuție după apel ale funcțiilor *calculateMatrix*, respectiv *calculateThreads* este măsurată. Matricea-rezultat *V* este scrisă într-un fișier. Pentru verificarea corectitudinii, fiecare fișier rezultat în urma folosirii threadurilor este comparat cu fișiere-test generate secvențial.

La final se afișează în consolă timpul de execuție măsurat anterior. Timpii obținuți în funcție de rularea secvențială vs. pe threaduri, respectiv în funcție de *no­\_threads* sunt următorii:

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | secvențial | 0,05431 |
| 4 | 1,60328 |
| N=M=1000  n=m=5 | secvențial | 88,4006 |
| 1 | 81,62894 |
| 2 | 58,50882 |
| 4 | 42,94364 |
| 8 | 45,62096 |
| 16 | 76,91551 |
| N=10  M=10000  n=m=5 | secvențial | 32,93158 |
| 1 | 37,66228 |
| 2 | 49,49241 |
| 4 | 52,96885 |
| 8 | 66,25325 |
| 16 | 94,67041 |
| N=10000  M=10  n=m=5 | secvențial | 22,57198 |
| 1 | 18,6589 |
| 2 | 18,63336 |
| 4 | 22,75454 |
| 8 | 35,32658 |
| 16 | 60,21252 |

C++

Abordarea C++ este identică ca logică cu cea Java.

Diferența constă în utilizarea unei funcții *calculateMatrixByThreads* în locul clasei *MyThreads*, echivalentă ca logică cu funcția *run* din clasa *MyThreads*, care este dată ca parametru obiectelor native de tip *thread* inițializate în funcția *calculateThreads*.

De asemenea, inițializarea matricilor, precum și signatura funcțiilor, diferă în cazul alocării statice vs. dinamice.

Citirea, scrierea, măsurarea și verificarea corectitudinii sunt echivalente cu implementarea Java.

Timpii obținuți în funcție de rularea secvențială vs. pe threaduri, alocării statice vs. dinamice, respectiv în funcție de *no\_threads* sunt următorii:

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | Static | secvențial | 0,00691 |
| 4 | 0,70453 |
| dinamic | secvențial | 0,00731 |
| 4 | 0,53611 |
| N=M=1000  n=m=5 | static | secvențial | 131,0763 |
| 1 | 136,687 |
| 2 | 76,14148 |
| 4 | 40,16231 |
| 8 | 37,19738 |
| 16 | 32,32051 |
| dinamic | secvențial | 161,0611 |
| 1 | 162,9107 |
| 2 | 70,109 |
| 4 | 40,23968 |
| 8 | 32,06299 |
| 16 | 30,67815 |
| N=10  M=10000  n=m=5 | static | secvențial | 12,66436 |
| 1 | 13,58294 |
| 2 | 6,72848 |
| 4 | 4,19436 |
| 8 | 3,80257 |
| 16 | 3,42035 |
| dinamic | secvențial | 13,79221 |
| 1 | 14,70523 |
| 2 | 7,56057 |
| 4 | 4,65615 |
| 8 | 3,79861 |
| 16 | 4,12402 |
| N=10000  M=10  n=m=5 | static | secvențial | 13,34775 |
| 1 | 13,84281 |
| 2 | 7,3485 |
| 4 | 4,74231 |
| 8 | 3,94617 |
| 16 | 3,71737 |
| dinamic | secvențial | 14,1095 |
| 1 | 14,87931 |
| 2 | 7,72651 |
| 4 | 5,10003 |
| 8 | 4,18721 |
| 16 | 3,94695 |

*Observații*

În afara cazului în care dimensiunile N și M ale matricilor sunt mici ( N, M ~< 100), performanța obținută utilizând threaduri este în general mai bună decât în cazul rulării secvențiale.

Dacă în cazul variantei C++ sporirea performanței este direct proporțională cu creșterea numărului de threaduri folosite, la Java timpul obținut este cel mai bun când este folosit un număr de threaduri mediu (în general, 2-4 threaduri).

În cazul rulării secvențiale, timpii obținuți sunt asemănători în Java și C++, fiind ușor mai buni în Java pentru dimensiuni mari. Totuși, threadurile din C++ sunt consistent mai rapide decât cele din Java.

În cazul C++, abordarea alocării statice generează timpi mai buni decât alocarea dinamică doar la rularea secvențială a programului. Dacă se folosesc threaduri, alocarea dinamică scoate timpi asemănători sau ușor mai buni decât alocarea statică a matricilor.