Laborator 2

*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:

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

unde *F* 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>

**Constrângere:** NU se alocă o matrice rezultat (V) temporară!

*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.

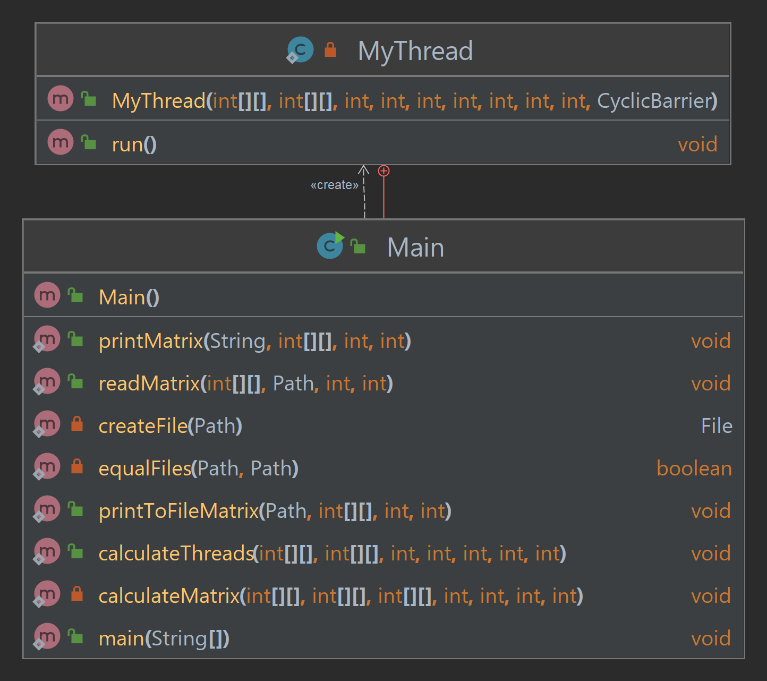
Java

Diagrama de clase

*\*Împărțirea workloadului pe threaduri*

Pentru calculul pe threaduri am folosit clasa *MyThread*, ce extinde clasa Threads, și funcția *calculateThreads*. Distribuția indecșilor matricei *F* fiecărui thread în parte se face ciclic. Fiecare thread va începe cu elementul F[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*.

Pentru a se respecta constrângerea dată se va folosi un buffer (de tip Vector<Integer>) în care se vor stoca valorile calculate.

O imagine care conține text

Descriere generată automat

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

La finalul parcurgerii matricei *F* de către un thread, acesta va apela funcția *await()* a barierei cyclicBarrier, de tip *CyclicBarrier*, dată ca parametru în momentul creării threadului respectiv.

Astfel, valorile calculate de un thread nu vor afecta calculul celorlalte threaduri.

O imagine care conține text

Descriere generată automatDupă ce toate threadurile apelează funcția *await()*, fiecare thread va stoca în matricea inițială *F* valorile stocate în bufferul propriu.

*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 *F* descris mai sus.

Funcția *calculateThreads* creează o barieră de tip *CyclicBarrier (cyclicBarrier)*, de dimensiune *no\_threads*, construiește un array de *MyThreads* de dimensiunea dorită (*no\_threads*) și inițializează fiecare obiect din array cu datele necesare (matricile *F*, *w*, dimensiunile, *iStart*, *jStart, cyclicBarrier*), 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 este scrisă într-un fișier. Pentru verificarea corectitudinii, fiecare fișier rezultat în urma folosirii threadurilor este comparat cu fișiere-test generate secvențial.

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

La final se afișează în consolă timpul de execuție măsurat anterior. Timpii obținuți în funcție de *no­\_threads* sunt:

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | 4 | 2,3107299 |
| N=M=1000  n=m=5 | 1 | 158,2756703 |
| 2 | 109,2340499 |
| 4 | 84,61667 |
| 8 | 122,2033497 |
| 16 | 143,6893702 |
| N=10  M=10000  n=m=5 | 1 | 47,5755404 |
| 2 | 72,4381499 |
| 4 | 87,76024 |
| 8 | 97,8193302 |
| 16 | 111,2936102 |
| N=10000  M=10  n=m=5 | 1 | 31,59169 |
| 2 | 33,5182399 |
| 4 | 35,71984 |
| 8 | 46,3571601 |
| 16 | 79,91821 |

Timpii obținuți la laboratorul anterior

Timpii obținuți la laboratorul actual

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.

O imagine care conține text

Descriere generată automatÎn loc de clasa nativă CyclicBarrier am creat o clasă *my\_barrier­* cu funția *wait()*, care imită funcția *await()*.

Creând un obiect global de acest tip se poate gestiona sincronizarea threadurilor create asemănător situației din Java. Am folosit și în acest caz un buffer propriu fiecărui thread de tip *vector<int>*.

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

Timpii obținuți în funcție de alocarea statică vs. dinamică, respectiv în funcție de *no\_threads* sunt următorii:

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | Static | 4 | 0,7996 |
| dinamic | 4 | 0,80736 |
| N=M=1000  n=m=5 | static | 1 | 183,8126 |
| 2 | 83,87645 |
| 4 | 47,71086 |
| 8 | 35,69344 |
| 16 | 33,68564 |
| dinamic | 1 | 174,432 |
| 2 | 94,23423 |
| 4 | 57,2035 |
| 8 | 39,13952 |
| 16 | 36,06526 |
| N=10  M=10000  n=m=5 | static | 1 | 18,0456 |
| 2 | 9,17646 |
| 4 | 6,11867 |
| 8 | 4,63757 |
| 16 | 4,58508 |
| dinamic | 1 | 19,01522 |
| 2 | 10,02183 |
| 4 | 5,70674 |
| 8 | 5,10866 |
| 16 | 5,25928 |
| N=10000  M=10  n=m=5 | static | 1 | 17,4601 |
| 2 | 9,70727 |
| 4 | 5,46493 |
| 8 | 5,51477 |
| 16 | 5,03304 |
| dinamic | 1 | 19,16103 |
| 2 | 9,70449 |
| 4 | 6,11034 |
| 8 | 5,58407 |
| 16 | 5,28984 |

*Timpii obținuți la laboratorul 1*

*Timpii obținuți la laboratorul actual*

*Observații*

Corectitudinea valorilor obținute a fost asigurată comparând rezultatele cu cele obținute în cadrul laboratorului anterior.

Analizând timpii obținuți în cele două laboratoare se poate observa că performanța are de suferit, atât în Java, cât și în C++. Cu toate acestea, creșterea numărului de threaduri rămâne o metodă de îmbunătățire a acesteia, în special în C++. În Java, creșterea numărului de threaduri este benefică până la un punct (1-2-4 threaduri), după care performanța începe să scadă.

În continuare, timpii obținuți de programul C++ sunt sensibili mai buni decât cei din Java.

Dacă în cazul implementării din laboratorul 1, în memoria fiecărui thread se creează o întreagă matrice-finală *V*, în acest laborator fiecare thread își creează exact atât spațiu de memorie cât are nevoie pentru memorarea valorilor calculate de sine. Astfel, fiecare thread va aloca un spațiu de memorie ce însumat va fi egal cu matricea inițială, în loc de o nouă matrice pentru fiecare thread, cazul primului laborator. Complexitatea-spațiu rămâne constantă indiferent de numărul de threaduri folosite.