Documentatie Laborator 2

**Analiza cerintelor:**

Se considera problema de la laboratorul 1 cu urmatoarea modificare:

Se cere un program care sa asigure urmatoarea postconditie:

**Postconditie**: Matricea initiala contine imaginea filtrata.

**Constrangere**: NU se aloca o matrice rezultat (V) temporara!

**Obiectiv**: optimizarea complexitatii-spatiu in conditiile obtinerii unei performante ridicate.

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

(Fisierul trebuie creat anterior prin adaugare de numere generate aleator.)

**Implementare**

a) Java

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

**Testare:** masurati timpul de executie pentru

1) N=M=10 si n=m=3; p=2;

2) N=M=1000 si n=m=5; p=1,2,4,8,16

3) N=10 M=10000 si n=m=5; p=1,2,4,8,16

4) N=10000 M=10 si n=m=5; p=1,2,4,8,16

**Observatii** (valabile pentru fiecare caz de testare 1-4):

- Fiecare test trebuie repetat de 10 ori si evaluarea timpul de executie se considera media aritmetica a celor 10 rulari.

- Pentru fiecare varianta de testare (dintre cele 10) folositi acelasi fisier “date.txt”;

- Pentru fiecare varianta de testare (dintre cele 10) verificati corectitudinea prin comparatia rezultatului cu fisierul rezultat prin executia secventiala.

**Analiza:**

Verificati corectitudinea prin comparatie cu rezultatul de la laboratorul 1.

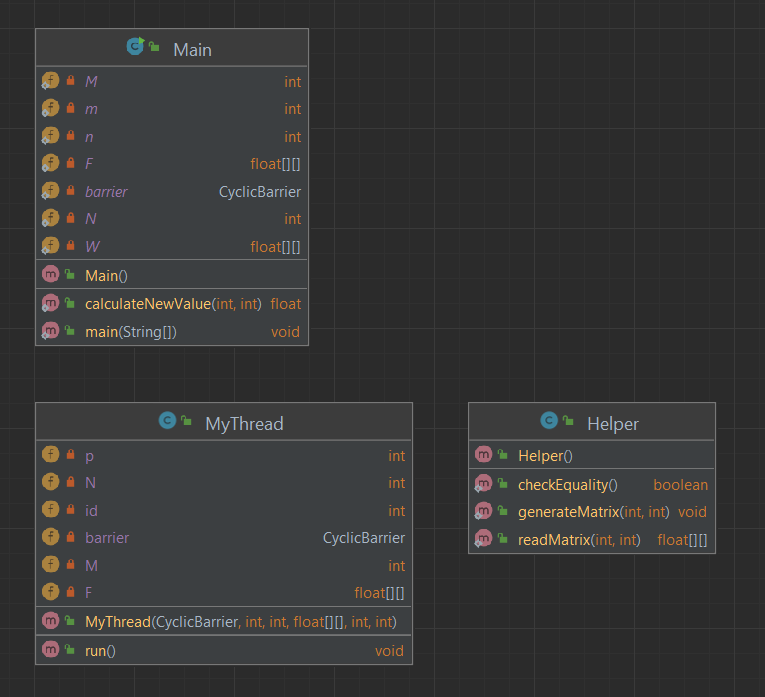
Comparati performanta pentru fiecare caz !

Comparati timpii obtinuti cu implementarea Java versus implementarea C++.

Evaluati complexitatea-spatiu.

**Proiectare:**

**Java:**

****

**Main() -** clasa de intrare in program

**main(String[]):void** – functia principala a programului

**calculateNewValue(int, int) :float** – functie care calculeaza noua valoare din matrice, folosind matricea de filtrare. Se inmultesc pozitiile din kernel cu cele din matrice corespunzatoare.

**MyThread(CyclicBarrier, int,int,float[][],int,int)** – constructorul pentru clasa MyThread, care primeste un id, un numar p de thread-uri, matricea initiala F, si dimensiunile matricei initiale, M si N

**run()** – functie suprascrisa a clasei Thread, in care fiecare thread calculeaza noile valori din matricea F si modifica matricea initiala cu valorile noi, cu ajutorul unei bariere. Fiecare thread calculeaza linii intregi din matrice, din p in p linii.

**generateMatrix(int,int): void** – functie pentru generarea unei matrici random, de dimensiune MxN

**checkEquality():Boolean** - functie pentru verificarea egalitatii fisierelor: secvential si parallel

**readMatrix(int,int): float[][]** – functie pentru citirea matricii initiale din fisier.

**C++:**

**Dynamic:**

|  |
| --- |
| +main(int, char\*): int |
| +readMatrix(vector<vector<float>>): void |
| +calculateNewValue(vector<vector<float>>, vector<vector<float>>, int, int): float |
| +calculateThread(int, int, vector<vector<float>>, vector<vector<float>>, int, int): void |
| +checkEquality(): bool |

**Static:**

|  |
| --- |
| +main(int, char\*): int |
| +readMatrix(float matrix[][] void |
| +calculateNewValue(float W[][], float F[][], int, int): float |
| +calculateThread(int, int, float W[][], float F[][], int, int): void |
| +checkEquality(): bool |

|  |
| --- |
| My\_barrier |
| - m : mutex |
| - cv : condition\_variable |
| - counter : int |
| - waiting : int |
| - thread\_count : int |
| +my\_barrier(int) |
| +wait() : void |

**my\_barrier:** constructor pentru clasa my\_barrier

**wait(): void –** functie pentru thread-urile sa astepte la bariera, count-nr de thread-uri care asteapta la bariera.

**Detalii de implementare:**

Codul din Java si C++ este similar. Atat in Java cat si in C++, modul de lucru este la fel. Se citeste matricea din fisierul “inputMatrix.txt”, dupa care formam matricea kernel, W, in functie de dimensiunea acesteia. Am considerat ca fiecare element al matricii kernel sa fie 1/9. Matricea initiala este declarata din start mai mare cu 4 unitati iar bordarea acesteia are loc prin asignarea de valori in functie de valorile celulelor vecine.

Astfel, formule de calcul pentru calcularea colturilor bordurii, unde N=nr coloane, M=nr linii:

*F*[0][0] = *F*[0][1] = *F*[1][0] = *F*[1][1] = *F*[2][2];  
*F*[*M* + 3][1] = *F*[*M* + 2][1] = *F*[*M* + 3][0] = *F*[*M* + 2][0] = *F*[*M* + 1][2];  
*F*[0][*N* + 3] = *F*[0][*N* + 2] = *F*[1][*N* + 3] = *F*[1][*N* + 2] = *F*[2][*N* + 1];  
*F*[*M* + 3][*N* + 3] = *F*[*M* + 3][*N* + 2] = *F*[*M* + 2][*N* + 3] = *F*[*M* + 2][*N* + 2] = *F*[*M* + 1][*N* + 1];

Si pentru spatiul dintre colturi:

for (int i = 2; i < *M* + 2; i++) {  
 *F*[i][0] = *F*[i][1] = *F*[i][2];  
 *F*[i][*N* + 2] = *F*[i][*N* + 3] = *F*[i][*N* + 1];  
}

for (int j = 2; j < *N* + 2; j++) {  
 *F*[0][j] = *F*[1][j] = *F*[2][j];  
 *F*[*M* + 2][j] = *F*[*M* + 3][j] = *F*[*M* + 1][j];  
}

Mai departe, in functie de optiunea utilizatorului, se poate face un calcul secvential sau parallel care utilizeaza aceeasi functie de calcul a noii valori: calculateNewValue, care face suma inmultirii matricii kernel, W, cu pozitiile corespunzatoare din matricea initiala, F. In cazul in care se opteaza pentru varianta paralela, se initializeaza bariera cu un numar p de thread-uri + un vector de p thread-uri, in care fiecare thread calculeaza noile valori ale unei linii, din p in p linii, dupa care bariera asteapta toate thread-urile si apoi se actualizeaza valorile pe matricea F initiala. Astfel, thread-urile sunt distribuite liniar.

Rezultatele obtinute in urma calculelor sunt scrise in fisierul de output: “outputMatrix.txt”. De asemenea, o functie de checkEquality verifica daca rezultatele obtinute in mod paralel si secvential sunt aceleasi.

**Cazuri de testare:**

**Java:**

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie |
| N=M=10 n=m=3 | secvential | 8.66068 |
|  | 2 | 1.86578 |
| N=M=1000 n=m=5 | secvential | 290.5746 |
|  | 1 | 69.41236 |
|  | 2 | 67.26408 |
|  | 4 | 71.21667 |
|  | 8 | 81.69236 |
|  | 16 | 110.2661 |
| N=10 M=10000 n=m=5 | secvential | 94.4581 |
|  | 1 | 21.8333 |
|  | 2 | 27.49035 |
|  | 4 | 30.98547 |
|  | 8 | 28.901 |
|  | 16 | 29.62919 |
| N=10000 M=10 n=m=5 | secvential | 88.88795 |
|  | 1 | 14.35966 |
|  | 2 | 16.78595 |
|  | 4 | 17.74201 |
|  | 8 | 17.33412 |
|  | 16 | 22.16185 |

**C++ dinamic:**

|  |  |  |  |
| --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie |
| N=M=10 n=m=3 | dinamic | secvential | 0.1892 |
|  |  | 2 | 6.7146 |
| N=M=1000 n=m=5 | dinamic | secvential | 1374.38 |
|  |  | 1 | 1332.67 |
|  |  | 2 | 1298.33 |
|  |  | 4 | 1358.21 |
|  |  | 8 | 1343.52 |
|  |  | 16 | 1275.18 |
| N=10 M=10000 n=m=5 | dinamic | secvential | 135.053 |
|  |  | 1 | 123.368 |
|  |  | 2 | 121.129 |
|  |  | 4 | 122.486 |
|  |  | 8 | 131.258 |
|  |  | 16 | 143.67 |
| N=10000 M=10 n=m=5 | dinamic | secvential | 114.663 |
|  |  | 1 | 115.278 |
|  |  | 2 | 120.4 |
|  |  | 4 | 135.341 |
|  |  | 8 | 122.287 |
|  |  | 16 | 128.62 |

**C++ static:**

|  |  |  |  |
| --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie |
| N=M=10 n=m=3 | static | secvential | 0.2659 |
|  |  | 2 | 6.504 |
| N=M=1000 n=m=5 | static | secvential | 1326.63 |
|  |  | 1 | 1456.95 |
|  |  | 2 | 1352.33 |
|  |  | 4 | 1332.5 |
|  |  | 8 | 1449.08 |
|  |  | 16 | 1284.16 |
| N=10 M=10000 n=m=5 | static | secvential | 120.159 |
|  |  | 1 | 125.792 |
|  |  | 2 | 117.204 |
|  |  | 4 | 120.924 |
|  |  | 8 | 124.948 |
|  |  | 16 | 130.696 |
| N=10000 M=10 n=m=5 | static | secvential | 125.196 |
|  |  | 1 | 120.684 |
|  |  | 2 | 116.801 |
|  |  | 4 | 123.362 |
|  |  | 8 | 129.592 |
|  |  | 16 | 128.748 |

**Analiza rezultatelor:**

**Java: secvential-paralel:** se poate observa ca performanta este mai scazuta in cazul calculului secvential, iar in cazul calculului parallel, la mine, cu cat creste numarul de thread-uri, cu atat scade putin si performanta. La matricile mai mari, absolute ca timpul va fi putin mai ridicat.

**C++: dynamic:** secvential si parallel se obtin aproximativ aceleasi rezultate. In cazul matricei de mici dimensiuni, calculul este rapid. Cu cat dimensiunea creste, timpul creste. Nu putem spune ca avem imbunatatire foarte mare.

**C++: static:** secvential si parallel se obtin aproximativ aceleasi rezultate. In cazul matricei de mici dimensiuni, calculul este rapid. Cu cat dimensiunea creste, timpul creste. Nu putem spune ca avem imbunatatire foarte mare.

**C++: dynamic vs static:** nu exista diferente mari sesizabile intre variant statica si dinamica, rezultatele obtinute sunt aproximativ egale.

**Java comparat cu C++:** in Java, dupa cate putem observa, performanta ramane mai ridicata, comparata cu rezultatele obtinute din C++.