Documentație Tema 2

**Laborator 2**

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**

1. Java
2. 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:**

* Fiecare test trebuie repetat de 10 ori si evaluarea timpul de executie se considera media aritmetica a celor 10 rulari.
* Pentru fiecare varianta a cazului de testare 1) folositi acelasi fisier “date.txt”;
* similar pentru cazurile 2), 3), 4)

**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.

**Deadline:**

* saptamana 5

**Detalii de implementare valabile pentru ambele limbaje de programare:**

Table

Description automatically generated with low confidence

Ex cu impartirea matricii.

Pentru impartirea sarcinilor, am folosit formulele de la laboratorul trecut. Cu un for de la 0 la numarul de thread-uri am calculate intervalele de procesat pentru fiecare thread. Am folosit un singur for pentru ca dintr-un singur indice pot sa ii calculez pe cei doi, ai matricii.

Avem matricea initiala de dimensiune MxN, Indexul din for-ul de la 0 la numarul de thread-uri k, Indexul matricii i, j

k = i \* m + j

i= k / N

j= k % N

Functia de run a thread-urilor salveaza intr-un buffer o parte din matricea initiala, cat are nevoie fiecare thread pentru a putea aplica kernelul. Dimensiunea bufferului poate varia in functie de dimensiunea matricii initiale si a kernelului

Graphical user interface, application

Description automatically generated

**Detalii Java:**

Proiectul este cel de la laboratorul 1, cu clasele ParallelMemory si ThreadMemory, in plus.

Am clasa abstracta Worker, care continue matricile: matrix, kernel, functiile de citire a matricii din fisier, functia de scriere in fisier, si functia abstracta doOperation, implementata de clasele Sequential, Parallel, ParallelMemory.

Clasa Sequential implementeaza doOperation secvential, cu for-uri pana la N si M, pentru a parcurge matricea si for-uri de – n / 2 pana la n / 2 si - m / 2 pana la m / 2 pentru a trece prin matricea de kernel.

Clasa Parallel, de asemenea implementeaza doOperation, care foloseste clasa MyThread. MyThread contine functia run care va fi rulata de fiecare thread pentru obtinerea rezultatului dorit. Calculul pe thread-uri este la fel in ambele limbaje si este scris mai sus.

Clasa MemoryParallel foloseste functia saveInBuffer, unde salvez intr-un buffer elementele matricei initiale pe care trebuie sa le procesam(portiunea care trebuie calculate + una sau mai multe linii deasupra si de mai jos, in functie de dimensiunea kernelului). Functia doOperation foloseste o bariera pentru a ne asigura ca toate thread-urile reusesc sa stocheze bufferul, apoi parcurg bufferul si calculez in matricea initiala rezultatul.

Am verificat de asemenea in functia rulata de fiecare thread, cazul in care se iasa din matrice, caz in care am folosit valoarea vecina, valorii din afara matricii.

Application, table

Description automatically generated

**Detalii C++:**

Metoda de lucru din C++ este la fel cu cea din Java.

Functia run\_memory este functia pe care thread-urile o vor rula-o.

Functia do\_do\_parallel\_memory este functia care va crea thread-urile si va imparti sarcinile.

**Testare:**

Java:

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 | 2 | 43.15309 |
|  |  |
| N=M=1000  n=m=5 | secvential |  |
| 1 | 793.99255 |
| 2 | 730.71398 |
| 4 | 704.85478 |
| 8 | 721.05586 |
| 16 | 873.75153 |
| N= 10 M=10000  n=m=5 | secvential |  |
| 1 | 180.83812 |
| 2 | 184.16711 |
| 4 | 183.49265 |
| 8 | 193.6367 |
| 16 | 241.41086 |

|  |  |  |
| --- | --- | --- |
| N= 10000 M=10  n=m=5 | secvential |  |
| 1 | 196.4669 |
| 2 | 203.8818 |
| 4 | 228.67817 |
| 8 | 234.58155 |
| 16 | 253.06795 |

**C++**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr threads** | **Timp executie** |
| N=M=10  n=m=3 |  | 2 | 1.50798 |
|  |  |  |
| N=M=1000  n=m=5 | static | 1 | 814.5385 |
| 2 | 810.847 |
| 4 | 847.8757 |
| 8 | 783.9679 |
| 16 | 835.3162 |
| dinamic | 1 |  |
| 2 |  |
| 4 |  |
| 8 |  |
| 16 |  |
| N= 10 M=10000  n=m=5 | static | 1 | 148.7922 |
| 2 | 140.1802 |
| 4 | 133.0311 |
| 8 | 126.7109 |
| 16 | 122.8529 |
| dinamic | 1 |  |
| 2 |  |
| 4 |  |
| 8 |  |
| 16 |  |
| N= 10000 M=10  n=m=5 | static | 1 | 86.08605 |
| 2 | 83.58251 |
| 4 | 82.69351 |
| 8 | 83.33145 |
| 16 | 83.75453 |
| dinamic | 1 |  |
| 2 |  |
| 4 |  |
| 8 |  |
| 16 |  |

Analiza testarii:

* se oberva ca pe cazul cu matricea de 1000 pe 1000, programul Java este mai rapid decat cel C++, in restul cazurilor programul C++ este mai rapid
* programele Java au functionat cel mai bine cand au fost rulate pe 4 thread-uri
* la fel si cele C++, au functionat cel mai bine cand au fost rulate pe 4 thread-uri
* cele cu 8 si 16 thread-uri se apropie de performanta celor cu 4, dar cand rulam cu 1 thread sau 2 e prea costisitor.
* Se observa ca timpul de procesare din acest laborator este cu putin mai mare decat cel din laboratorul trecut (cu aproximativ 10 milisecunde)
* De asemenea in laboratul trecut cele mai bune rulari au fost cu 8 sau 16 thread-uri, aici cum am zis cu 4 thread-uri