Contents

[Problemă 1](#_Toc149085177)

[Condiții rulare 1](#_Toc149085178)

[Testarea efectiva a timpului de executie 1](#_Toc149085179)

[C++ 1](#_Toc149085180)

[Java 2](#_Toc149085181)

[Analiza rezultatelor 2](#_Toc149085182)

[C++ 2](#_Toc149085183)

[Java 3](#_Toc149085184)

[Java vs. C++ 3](#_Toc149085185)

# Problemă

Am studiat eficienta execuției paralele în contextul problemei calculării matricei de convoluție intre 2 matrici, fara a folosi o matrice auxiliara in care stocam datele

# Condiții rulare

* Java version: 17
* C++ version: 17
* C++ compilator: Microsoft Compiler

# Testarea efectiva a timpului de executie

## C++

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie |
| 10x10 | Secvential | 0.27432 |
| 2 | 1.44138 |
| 1000x1000 | Secvential | 3047.081 |
| 2 | 3789.808 |
| 4 | 4692.501 |
| 8 | 5598.343 |
| 16 | 6869.465 |
| 10000x10000 | Secvential | 306185.753 |
| 2 | 204179.302 |
| 4 | 200684.654 |
| 8 | 215648.564 |
| 16 | 269853.147 |

## Java

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie |
| 10x10 | Secvential | 0.05692 |
| 2 | 1.57651 |
| 1000x1000 | Secvential | 23.24734 |
| 2 | 34.4453 |
| 4 | 39.28688 |
| 8 | 53.53687 |
| 16 | 52.76445 |
| 10000x10000 | Secvential | 542.05565 |
| 2 | 301.38138 |
| 4 | 203.26237 |
| 8 | 164.68256 |
| 16 | 225.7121 |

# Distribuția datelor

Pentru a rezolva problema am ales sa facem o distribuire pe linii astfel incat sa avem o impartire echitabila a numarului de linii pe care fiecare thread trebuie sa calcueze matricea de convolutie

# Metoda de rezolvare

Pentru a rezolva problema am ales sa folosesc un algortim care imparte cantitatea de lucru la un numar *p* de threaduri, citit din linia de comanda. De fiecare data, incerc sa impart cantitea de calcul la numarul de threaduri intr-un mod cat mai echitabil in asa fel incat distribuirea sa fie uniforma, mai precis de fiecare data, impart numarul de linii/coloane la cate threaduri trebuie sa faca impartirea si retin si restul, apoi calculez 2 pozitii de start si end pe care le updatatez la fiecare pas al initializarii threadurilor. Deoarece nu putem folosi o matrice auxiliara pentru a stoca rezultatele, am ales sa copiez tot timpul intr-un buffer, rezultatele de la prima si de la ultima linie din sub-matricea alocata fiecarui thread (deoarece aceste linii sunt implicate in convolutia celulelor alocate altor threaduri) in timp, celelalte elemente le modificam destructiv, acest lucru este posibil deoarece retinem in permanenta linia actuala si linia precedenta (complexitate spatiu Teta(M)). Actualizam distructiv si valorile celor doua linii salvate in buffer dupa ce toate threadurile au terminat de lucrat cu acele valori (folosim o bariera de sincronizare)

# Analiza rezultatelor

## C++

In C++ se poate observa ca executiile secventiale functioneaza mai rapid in cazul matricei de 10x10 si celei de 1000x1000, dar la cea de 10000x10000 deja putem observa ca paralelizarea aduce un plus de eficienta din punct de vedere temporal

## Java

Varianta secventiala, si aici este mai rapida in cazul matricelor de 10x10 si celei de 1000x1000 dar in cazul matricei de 10000x10000 paralelizarea se dovedeste mai eficienta.

## Java vs. C++

Per total Java este mai eficienta decat C++ la realizarea acestei operatii, in cadrul matricelor mai mari C++ avand probleme mari care pot intinde executia un timp exagerat de mare.

# Complexitatea spatiu

Pentru a rezolva aceasta problema am folosi doi vectori dinamici auxiliari, ceea ce incadreaza complexitatea solutiei in Teta(M))