# ***Timpi de executie***

## C++

|  |  |  |  |
| --- | --- | --- | --- |
| Tip alocare | Tip Matrice | Nr threads | Timp executie |
| Static | N=M=10; n=m=3 | Secvențial | 0.01521 |
| 4 | 1.42275 |
| N=M=1000; n=m=5 | Secvențial | 109.762 |
| 2 | 55.621 |
| 4 | 34.737 |
| 8 | 26.773 |
| 16 | 26.661 |
| N=10, M=10000; n=m=5 | Secvențial | 10.316 |
| 2 | 1.3239 |
| 4 | 1.6608 |
| 8 | 2.3141 |
| 16 | 3.6392 |
| N=10000, M=10; n=m=5 | Secvențial | 25.372 |
| 2 | 15.200 |
| 4 | 9.377 |
| 8 | 7.572 |
| 16 | 7.297 |
| Dinamic | N=M=10; n=m=3 | Secvențial | 0.022 |
| 4 | 1.466 |
| N=M=1000; n=m=5 | Secvențial | 596.56 |
| 2 | 314.21 |
| 4 | 174.6054 |
| 8 | 139.7726 |
| 16 | 132.3709 |
| N=10, M=10000; n=m=5 | Secvențial | 58.02855 |
| 2 | 1.24702 |
| 4 | 1.68074 |
| 8 | 2.34085 |
| 16 | 3.54365 |
| N=10000, M=10; n=m=5 | Secvențial | 59.1973 |
| 2 | 32.29432 |
| 4 | 17.1015 |
| 8 | 16.71759 |
| 16 | 14.79081 |

## Java

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie (ms) |
| N=M=10; n=m=3 | Secvential | 0.11148 |
| 4 | 0.89488 |
| N=M=1000; n=m=5 | Secvential | 46.59235 |
| 2 | 38.25662 |
| 4 | 36.80918 |
| 8 | 119.20943 |
| 16 | 130.97654 |
| N=10, M=10000; n=m=5 | Secvential | 20.30934 |
| 2 | 0.71171 |
| 4 | 0.88387 |
| 8 | 1.61725 |
| 16 | 2.72081 |
| N=10000, M=10; n=m=5 | Secvential | 12.32987 |
| 2 | 18.72173 |
| 4 | 19.28836 |
| 8 | 19.38669 |
| 16 | 17.27772 |

***Distribuția datelor***

Pentru a rezolva problema, am adoptat o strategie de distribuire a lucrului pe linii sau pe coloane, în funcție de dimensiunea datelor. Mai precis, am ales să lucrăm fie pe linii, fie pe coloane, în funcție de dimensiunea mai mică dintre cele două. Metoda de rezolvare implică împărțirea cantității de lucru în funcție de numărul de fire de execuție (thread-uri) specificat prin linia de comandă. Am încercat să împărțim echitabil cantitatea de calcul la numărul de fire, astfel încât distribuția să fie uniformă. Am împărțit numărul de linii sau coloane la numărul de thread-uri și am calculat restul. Apoi, am determinat două poziții de start și de final, pe care le-am actualizat la fiecare pas al inițializării thread-urilor.

***Analiza rezultatelor***

C++

În ceea ce privește alocarea statică versus alocarea dinamică, alocarea statică se dovedește a fi mai eficientă din punct de vedere al timpului de execuție. Alocarea statică se realizează în timpul compilării într-un spațiu cunoscut dinainte în heap, în timp ce alocarea dinamică se face la runtime, adăugând o suprasarcină suplimentară.

Pentru matricea de 10x10 și matricea kernel de 3x3, varianta secvențială se dovedește a fi mai eficientă, în principal datorită faptului că inițierea thread-urilor și gestionarea acestora consumă timp, iar operațiile în sine nu sunt suficient de costisitoare pentru a justifica paralelizarea.

Pentru matricea de 1000x1000 și matricea kernel de 5x5, varianta secvențială devine mai ineficientă, evidențiind avantajele paralelismului. Cele mai bune rezultate se obțin cu 8 thread-uri, iar creșterea la 16 thread-uri poate duce la creșterea timpului de execuție.

Pentru matricea de 10x10000 și matricea kernel de 5x5, variantele cu 2-4 thread-uri se dovedesc cele mai eficiente. Aici, împărțirea pe linii și numărul mai mic de thread-uri aduc îmbunătățiri semnificative.

Pentru matricea de 10000x10 și matricea kernel de 5x5, variantele paralele devin mai eficiente, iar numărul maxim de thread-uri aduce eficiența maximă. Este evident că în acest caz execuția paralelă poate aduce îmbunătățiri semnificative.

Aceste observații sugerează că alegerea între alocarea statică și cea dinamică, precum și optimizarea numărului de thread-uri, depind în mare măsură de dimensiunea și structura datelor, precum și de algoritmii folosiți.