## Java 17 SDK

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie (ms) |
| N=M=10; n=m=3 | Secvential | 0.13073 |
| 4 | 0.99585 |
| N=M=1000; n=m=5 | Secvential | 56.94892 |
| 2 | 49.33796 |
| 4 | 45.7855 |
| 8 | 151.07461 |
| 16 | 273.88357 |
| N=10, M=10000; n=m=5 | Secvential | 42.44545 |
| 2 | 56.80476 |
| 4 | 61.69143 |
| 8 | 70.6964 |
| 16 | 90.47949 |
| N=10000, M=10; n=m=5 | Secvential | 15.52742 |
| 2 | 18.97037 |
| 4 | 29.94551 |
| 8 | 20.35047 |
| 16 | 22.99065 |

## C++

|  |  |  |  |
| --- | --- | --- | --- |
| Tip alocare | Tip Matrice | Nr threads | Timp executie |
| Static | N=M=10; n=m=3 | Secvențial | 0.00809 |
| 4 | 0.97261 |
| N=M=1000; n=m=5 | Secvențial | 107.3113 |
| 2 | 57.73171 |
| 4 | 30.60383 |
| 8 | 21.82698 |
| 16 | 24.29044 |
| N=10, M=10000; n=m=5 | Secvențial | 10.52597 |
| 2 | 7.42699 |
| 4 | 5.30948 |
| 8 | 4.49363 |
| 16 | 4.50966 |
| N=10000, M=10; n=m=5 | Secvențial | 23.11555 |
| 2 | 14.16178 |
| 4 | 9.72477 |
| 8 | 7.45843 |
| 16 | 7.67634 |
| Dinamic | N=M=10; n=m=3 | Secvențial | 0.024 |
| 4 | 1.0912 |
| N=M=1000; n=m=5 | Secvențial | 490.9599 |
| 2 | 254.366 |
| 4 | 133.3661 |
| 8 | 80.93825 |
| 16 | 88.31287 |
| N=10, M=10000; n=m=5 | Secvențial | 50.09932 |
| 2 | 26.37867 |
| 4 | 15.69062 |
| 8 | 15.21811 |
| 16 | 11.48163 |
| N=10000, M=10; n=m=5 | Secvențial | 52.20556 |
| 2 | 27.89024 |
| 4 | 16.69741 |
| 8 | 14.82615 |
| 16 | 12.30753 |

De completat…

# Distribuția datelor

Pentru a rezolva problema am ales sa facem o distribuire pe linii sau pe coloane in functie de dimensiunea datelor, mai precis, din moment ce threadurile merg pe linii sau pe coloane am decis sa merg fie pe linii, fie pe coloane in functie de cele care sunt mai putine.

# 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.

# Analiza rezultatelor

## C++

Din punctul de vedere al alocării statice vs. dinamică, alocarea statică se dovedește clar mai eficientă din punct de vedere al timpului de execuție cauzele principale fiind: în cazul alocării statice, alocarea se face compile time într-un spațiu cunoscut dinainte pe heap, în timp ce în cazul alocării dinamice, alocarea se face la runtime alocand memorie în memoria RAM, astfel interogarea elementelor alocate static va fi mult mai eficientă decât cea a elementelor alocate static.

În cazul matricei de 10x10 și matricea kernel de 3x3 varianta secvențială se dovedește a fi clar mai bună decât cea paralele, în primul rând datorită faptului că pornirea celor 3 threaduri consumă time, iar operația în sine nu este una prea costisitoare.

În cadrul matricei de 1000x1000 și matricea kernel de 5x5 varianta secvențială se dovedește a fi mai ineficientă, aici putând observa clar avantajele paraleleismului , cel mai bun timp mediu putând fi obținut cu 8 threaduri, faptul ca varianta cu 16 threaduri e putin mai neficientă ne demonstreză faptul că threadurile în exces pot duce la ridicarea timpului de execuție.

În cadrul matricei de 10x10000 și matricea kernel de 5x5 variantele cu 2-4 threaduri sunt cele mai eficiente, asta se datorează în primul rând faptului că facem o împărțire pe linii, iar un număr mai mare de threaduri nu ar aduce o mai mare paralelizare a calcului, doar ar consuma timp pentru a porni și a se opri. Această operație dovedește că în anumite cazuri execuția paralelă poate aduce îmbunătățiri seminficative.

În cadrul matricei de 10000x10 și matricea kernel de 5x5 variantelele paralele sunt mai eficiente, în acest caz, numărul maxim de threaduri aducând eificența maximă.

## Java

În timp ce în majoritatea cazurilor varianta secvențială este cea mai rapidă (datorita paralelizării realizate la nivelul compilatorului și mașinii virtuale java) există totuși un caz (10x10000) unde paralelismul duce la o eficiență crescută.

## Java vs. C++

În general threadurile în C++ sunt mai rapide decât cele în Java, însă uneori execuți secvențială în Java este la nivelul unor execuții paralele în C++.