**Documentatie – Laborator 1**

**Testare:**

Run option:

0 – secvential:

* Se parcurge matricea secvential linie cu linie, coloana cu coloana.

1 – impartire pe linii

* Fiecare thread primeste initial un numar de linii consecutive pe care le va actualiza egal cu n / p. Daca aceasta impartire are un rest, atunci primele n % p threaduri vor primi o linie in plus.

2 – impartire pe coloane

* Fiecare thread primeste initial un numar de linii consecutive pe care le va actualiza egal cu m / p. Daca aceasta impartire are un rest, atunci primele m % p threaduri vor primi o linie in plus.

3 – impartire liniara

* Impartirea liniara cauta sa aloce fiecarui thread un numar cat mai egal de elemente. Se va imparti numarul total de elemente (n \* m) la numarul de threaduri p. Fiecare thread primeste un index de start si un index final in intervalul 0 – n \*m. Se va actualiza fiecare element din acel range calculand index-ul elementului din matrice astfel: row = index / m, column = index % m.

4 – impartire circulara

* Impartirea liniara cauta sa aloce fiecarui thread un numar cat mai egal de elemente. Fiecare thread primeste un index de start impreuna cu un pas (p) si va prelucra toate elementele incepand de la acel index din p in p pana la numarul total de elemente n \* m. Calculul indecsilor elementului care trebuie actualizat este similar cu cel de la impartirea liniara

**Testare Java:**

|  |  |  |  |
| --- | --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie | Run option |
| 10x10 | secvential | 0.07 | 0 |
| 4 | 0.72 | 1 |
| 0.76 | 2 |
| 0.75 | 3 |
| 0.75 | 4 |
| 1000x1000 | secvential | 32.41 | 0 |
| 2 | 29.37 | 1 |
| 31.00 | 2 |
| 27.86 | 3 |
| 28.33 | 4 |
| 4 | 32.34 | 1 |
| 32.74 | 2 |
| 30.80 | 3 |
| 29.93 | 4 |
| 8 | 32.48 | 1 |
| 33.54 | 2 |
| 29.92 | 3 |
| 29.15 | 4 |
| 16 | 33.94 | 1 |
| 36.94 | 2 |
| 31.27 | 3 |
| 33.26 | 4 |
| 10x10000 | secvential | 7.51 | 0 |
| 2 | 11.72 | 1 |
| 11.34 | 2 |
| 11.15 | 3 |
| 10.92 | 4 |
| 4 | 13.32 | 1 |
| 13.89 | 2 |
| 14.01 | 3 |
| 13.93 | 4 |
| 8 | 14.48 | 1 |
| 14.58 | 2 |
| 14.40 | 3 |
| 14.09 | 4 |
| 16 | 14.65 | 1 |
| 14.58 | 2 |
| 13.96 | 3 |
| 15.77 | 4 |
| 10000x10 | secvential | 8.18 | 0 |
| 2 | 10.89 | 1 |
| 12.65 | 2 |
| 11.83 | 3 |
| 10.69 | 4 |
| 4 | 14.13 | 1 |
| 15.24 | 2 |
| 14.10 | 3 |
| 13.92 | 4 |
| 8 | 14.64 | 1 |
| 15.23 | 2 |
| 14.29 | 3 |
| 14.50 | 4 |
| 16 | 14.72 | 1 |
| 14.73 | 2 |
| 15.49 | 3 |
| 16.04 | 4 |
| 7000x7000 | secvential | 1727.08 | 0 |
| 2 | 941.91 | 1 |
| 2125.51 | 2 |
| 925.02 | 3 |
| 956.57 | 4 |
| 4 | 572.61 | 1 |
| 1335.67 | 2 |
| 590.12 | 3 |
| 598.41 | 4 |
| 8 | 490.11 | 1 |
| 1063.81 | 2 |
| 476.99 | 3 |
| 490.12 | 4 |
| 16 | 493.59 | 1 |
| 1148.33 | 2 |
| 495.35 | 3 |
| 509.30 | 4 |

**Testare C++ dinamic:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie | Run option |
| 10x10 | dinamic | secvential | 0.00 | 0 |
| 2 | 0.47 | 1 |
| 0.47 | 2 |
| 0.47 | 3 |
| 0.47 | 4 |
| 4 | 0.57 | 1 |
| 0.56 | 2 |
| 0.56 | 3 |
| 0.53 | 4 |
| 8 | 0.67 | 1 |
| 0.64 | 2 |
| 0.65 | 3 |
| 0.67 | 4 |
| 16 | 0.84 | 1 |
| 0.81 | 2 |
| 0.83 | 3 |
| 0.80 | 4 |
| 1000x1000 | secvential | 18.13 | 0 |
| 2 | 10.66 | 1 |
| 14.07 | 2 |
| 10.96 | 3 |
| 11.28 | 4 |
| 4 | 6.12 | 1 |
| 7.88 | 2 |
| 7.03 | 3 |
| 7.10 | 4 |
| 8 | 5.95 | 1 |
| 7.30 | 2 |
| 6.12 | 3 |
| 6.14 | 4 |
| 16 | 5.98 | 1 |
| 7.58 | 2 |
| 6.54 | 3 |
| 6.33 | 4 |
| 10x10000 | secvential | 1.93 | 0 |
| 2 | 1.63 | 1 |
| 1.65 | 2 |
| 1.79 | 3 |
| 1.82 | 4 |
| 4 | 1.26 | 1 |
| 1.30 | 2 |
| 1.46 | 3 |
| 1.29 | 4 |
| 8 | 1.37 | 1 |
| 1.26 | 2 |
| 1.30 | 3 |
| 1.39 | 4 |
| 16 | 1.32 | 1 |
| 1.31 | 2 |
| 1.35 | 3 |
| 1.37 | 4 |
| 10000x10 | secvential | 1.96 | 0 |
| 2 | 1.67 | 1 |
| 1.71 | 2 |
| 1.74 | 3 |
| 1.75 | 4 |
| 4 | 1.34 | 1 |
| 1.38 | 2 |
| 1.31 | 3 |
| 1.33 | 4 |
| 8 | 1.32 | 1 |
| 1.38 | 2 |
| 1.37 | 3 |
| 1.34 | 4 |
| 16 | 1.41 | 1 |
| 1.40 | 2 |
| 1.37 | 3 |
| 1.36 | 4 |
| 7000x7000 | secvential | 1208.35 | 0 |
| 2 | 645.57 | 1 |
| 1173.25 | 2 |
| 697.84 | 3 |
| 718.86 | 4 |
| 4 | 418.54 | 1 |
| 1016.25 | 2 |
| 484.20 | 3 |
| 481.08 | 4 |
| 8 | 316.71 | 1 |
| 974.65 | 2 |
| 329.21 | 3 |
| 354.01 | 4 |
| 16 | 323.09 | 1 |
| 1093.92 | 2 |
| 335.54 | 3 |
| 351.90 | 4 |

**Testare C++ static:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie | Run option |
| 10x10 | static |  | 0.01 | 0 |
| 4 | 0.56 | 1 |
| 0.53 | 2 |
| 0.53 | 3 |
| 0.54 | 4 |
| 1000x1000 |  | 20.07 | 0 |
| 2 | 11.93 | 1 |
| 16.26 | 2 |
| 12.86 | 3 |
| 13.72 | 4 |
| 4 | 8.09 | 1 |
| 10.58 | 2 |
| 8.11 | 3 |
| 8.97 | 4 |
| 8 | 6.34 | 1 |
| 9.41 | 2 |
| 6.57 | 3 |
| 8.06 | 4 |
| 16 | 6.62 | 1 |
| 9.50 | 2 |
| 6.85 | 3 |
| 8.80 | 4 |
| 10x10000 |  | 1.97 | 0 |
| 2 | 1.75 | 1 |
| 1.74 | 2 |
| 1.82 | 3 |
| 1.82 | 4 |
| 4 | 1.42 | 1 |
| 1.33 | 2 |
| 1.45 | 3 |
| 1.42 | 4 |
| 8 | 1.45 | 1 |
| 1.30 | 2 |
| 1.40 | 3 |
| 1.31 | 4 |
| 16 | 1.41 | 1 |
| 1.32 | 2 |
| 1.37 | 3 |
| 1.36 | 4 |
| 10000x10 |  | 12.33 | 0 |
| 2 | 8.05 | 1 |
| 15.08 | 2 |
| 7.79 | 3 |
| 14.08 | 4 |
| 4 | 5.50 | 1 |
| 15.07 | 2 |
| 5.31 | 3 |
| 14.06 | 4 |
| 8 | 4.14 | 1 |
| 21.77 | 2 |
| 4.29 | 3 |
| 20.81 | 4 |
| 16 | 4.13 | 1 |
| 22.01 | 2 |
| 4.18 | 3 |
| 17.97 | 4 |
| 7000x7000 |  | 1434.50 | 0 |
| 2 | 763.15 | 1 |
| 1321.44 | 2 |
| 815.29 | 3 |
| 828.00 | 4 |
| 4 | 491.17 | 1 |
| 1107.12 | 2 |
| 514.20 | 3 |
| 508.84 | 4 |
| 8 | 357.40 | 1 |
| 1131.92 | 2 |
| 366.16 | 3 |
| 397.84 | 4 |
| 16 | 352.86 | 1 |
| 1230.17 | 2 |
| 368.49 | 3 |
| 385.68 | 4 |

**Analiza:**

*Testarile s-au realizat pe un CPU Intel I3 12100F 4 cores 8 threads.*

* **Java -** secvential versus paralel

In Java se poate observa ca rularea secventiala este aproape la fel de buna ca timp (daca nu chiar mai buna uneori) comparativ cu rularile paralele pentru cazurile in care dimensiunile matricii sunt reduse. O cauza principala ar putea fi modul in care Java creaza threadurile, fiind nevoie de o comunicare intre masina virtuala Java si OS ceea ce poate adauga niste delay.

* **Java** – paralel versus paralel

In cazurile paralele in Java timpii sunt destul de apropiati, insa se poate observa o crestere in timp odata cu cresterea numarului de threaduri, fapt ce se poate datora din nou alocarii acestora. In cazurile cu un numar mai mare de elemente se poate observa totusi ca impartirea liniara scoate niste timpi mai buni datorita impartirii mai eficienta a elementelor pentru fiecare thread.

* **C++** – secvential versus paralel

In C++ cazurile secventiale vor avea mereu cea mai mica performanta cu exceptia cazului de 10x10 unde matricea este foarte mica si implicit foarte rapid de parcurs.

* **C++** – paralel versus paralel

Pentru rularile paralele, cresterea numarului de threaduri de la 2 la 4 aduce o imbunatatire, insa cresterea de la 4 la 8 si 16 ruleaza in timpi extrem de apropiati. In cazurile cu multe elemente rularea cu impartirea pe linii si cea liniara aduce cei mai buni timpi pentru 8 threaduri per total.

* **Java** versus **C++**

Timpii obtinuti de implementarea in C++ indiferent de modul de alocare sunt mult mai buni comparativ cu cei obtinuti de Java in toate cazurile. Aceasta s-ar putea datora modului in care C++ lucreaza mult mai aproape cu sistemul de operare si gestioneaza mult mai bine memoria.

* **C++** – dinamic versus static

In general timpii obtinuti de alocarea statica si cea dinamica par sa fie aproape identici, insa alocarea dinamica are o performanta mai buna in cazul matricelor cu foarte multe elemente (7000x7000) si una cu mult mai buna in cazul matricei de 10000x10.

**Diagrama de clase Java:**

