**Documentatie Lab2 PPD**

**Cerinta:**

**Text

Description automatically generated**

**Proiectare**

A screenshot of a computer

Description automatically generated with medium confidence

**Implementare:**

Impartire pe thread-uri:

* dupa lini cand nr de lini >= nr coloane
* dupa coloane cand nr de coloane > nr de lini

Padding:

* in java – se adauga padding-ul la matricea initiala
* In c++ - se creaza o noua matrice in care se scrie padding-ul si continutul matricii initiale

Expansionarea matrici este diferita la java si c++ insa apelarea thread-urilor si executia lor este identica.

Am implementat Varianta 2b.

Graphical user interface, text, application

Description automatically generated

1. Calculez frontierele, care pot fi 4 linii/coloane prelucrate daca folosim kernel5 sau 2 linie/coloana daca folosim kernel3, dintre „chunk-urile” asociate thread-urilor
2. Aici folosesc o matrice aux care memoreaza 2 sau 3 linii/coloane in functie de kernel3 sau kernel5. Folosind aceasta matrice parcurg „chunk-ul” asociat thread-ului curent fara frontiere si adaug liniile sau coloanele prelucrate in aceasta matrice. Cand matricea se umple atunci stiu ca pot schimba matricea originiala si schimb folosind valorile din aux, urmand sa sterg linia/coloana din aux cu care am schimbat matricea.
3. Folosesc o bariera ca sa astept ca thread-urile sa incheie modificarea matricei initiale(fara frontiere)
4. Cand toate thread-urile au trecut de bariera schimb valorile din matrice unde ar fi frontierele cu valorile salvate in matricile de frontiera

**Java:**

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr. Threads | Timp Executie(milisecunde) |
| N=M=10  n=m=3 | secvential | 0.7 |
| 2 | 2.7 |
| N=M=1000  n=m=5 | secvential | 147.8 |
| 1 | 211.3 |
| 2 | 121.6 |
| 4 | 93.4 |
| 8 | 110.3 |
| 16 | 166.7 |
| N=10  M=10000  n=m=5 | secvential | 39.2 |
| 1 | 68.7 |
| 2 | 59.6 |
| 4 | 81.3 |
| 8 | 141.7 |
| 16 | 153.2 |
| N=10000  M=10  n=m=5 | secvential | 42.5 |
| 1 | 55.6 |
| 2 | 55.2 |
| 4 | 58.9 |
| 8 | 80.2 |
| 16 | 125.7 |

**C++:**

|  |  |  |  |
| --- | --- | --- | --- |
| Tip Matrice | Tip Alocare | Nr. Threads | Timp Executie(milisecunde) |
| N=M=10  n=m=3 | static | 2 | 2.1 |
| dinamic | 2 | 1.5 |
| N=M=1000  n=m=5 | static | secvential | 67.7 |
| 1 | 82.4 |
| 2 | 42 |
| 4 | 25 |
| 8 | 22.2 |
| 16 | 21.9 |
| dinamic | secvential | 73.3 |
| 1 | 82.6 |
| 2 | 42.3 |
| 4 | 25.2 |
| 8 | 22.3 |
| 16 | 21.7 |
| N=10  M=10000  n=m=5 | static | secvential | 6.8 |
| 1 | 8.8 |
| 2 | 5.5 |
| 4 | 4.1 |
| 8 | 5.2 |
| 16 | 7.6 |
| dinamic | secvential | 6.9 |
| 1 | 9.3 |
| 2 | 5.4 |
| 4 | 4.9 |
| 8 | 5.3 |
| 16 | 7.8 |
| N=10000  M=10  n=m=5 | static | secvential | 7.3 |
| 1 | 8.3 |
| 2 | 5.3 |
| 4 | 4.5 |
| 8 | 5.4 |
| 16 | 6.7 |
| dinamic | secvential | 7 |
| 1 | 9.1 |
| 2 | 5.4 |
| 4 | 4.4 |
| 8 | 4.9 |
| 16 | 7.3 |

**Grafuri:**

* **N=10 M=10**

**Chart, line chart

Description automatically generated**

* **N=1000 M=1000**

Chart, line chart

Description automatically generated

* **N=10000 M=10**

**Chart, line chart

Description automatically generated**

* **N=10 M=10000**

**Chart, line chart

Description automatically generated**

C++ vs Java:

* Se poate observa ca c++ este mult mai eficient din punct de vedere al timpului
* Thread-urile in java pare ca consuma multe resurse deoarece pentru inputuri mici cu cat crestea nr de thread-uri cu atat crestea si timpul, pe cand in c++ thread-urile imbunatateau timpi chiar si la inputuri mici

Paralelism in:

1. Java

* Se observa ca numarul 4 de thread-uri are cele mai bune rezultate in cazurile cu input mare
* In cazul inputurilor mici thread-urile in java pierd mult timp in crearea lor care nu este salvat din impartirea timpului de lucru pe o operatie scurta, iar numarul 2 de thread-uri da cel mai bun timp
* Deci de obicei la inputuri mici cu cresterea thread-urilor crestea si timpul

1. C++

* Se observa ca numerele 4,8 de thread-uri au cele mai bune rezultate in majoritatea cazurilor
* Timpul de executie scade de obicei pana la 8, iar apoi creste incet indiferent de dimensiunea inputuilor din ex.
* In comparatie cu java, in aceasta problema, in C++ numarul de thread-uri ajuta mult la scaderea timpului

Paralelism vs Secv. :

1. Java

* In java paralelismul a avut un timp mai bun ca var. Secventiala doar in cazul cand numarul operatiilor a crescut la 10^6(n=m=1000)
* in celelalte cazuri algoritmul secvential a avut un timp mai bun
* Asta are loc deoarece in cazurile input <= 10^5 nu erau destul de multe operati cat sa justifice crearea de thread-uri noi.

1. C++

* Metoda secventiala para sa fi fost mai putin eficienta din punct de vedere a timpului cand se foloseau 4 sau 8 thread-uri
* In unele cazuri folosirea a 2 thread-uri nu imbunatatea timpul

C++ static vs dinamic:

* In general urmeaza cam acelasi model de urcare/scadere a timpilor in functie de nr thread-uri
* Au avut timpi asemanatori

Complexitatea-Spatiu:

* in tema de la lab1 foloseam 2 N X M matrici
* in aceasta tema folosesc pentru fiecare thread doua matrici de frontiera(int(K/2)XM) si o matrice in care tin minte ultimele int(K/2)+1 vectori(M) de elemente prelucrate din parcurgerea chunk-ului curent
* deci in lab2 folosesc (N-((K/2)\*2+1))XM ori mai putina memorie