Documentatie-PPD1

ANALIZA CERINTELOR

Tema implică dezvoltarea unui program care să aplice o operație de convoluție pe o matrice F de dimensiuni n×mn , utilizând o matrice de convoluție C de dimensiuni k×k. Programul trebuie implementat atât secvențial, cât și paralel, cu utilizarea de threaduri.

**Cerinte de implementare:**

1. **Program secvențial:**
   * Scrie un program care aplică convoluția în mod secvențial pe matricea F folosind matricea de convoluție C.
2. **Program paralel:**
   * Folosirea a p threaduri pentru calculul convoluției.
   * Împărțirea sarcinilor de calcul pe threaduri folosind două metode:
     + **Împărțire pe linii:** Mai multe linii din matricea F sunt alocate fiecărui thread.
     + **Împărțire pe coloane:** Mai multe coloane din matricea F sunt alocate fiecărui thread.
3. **Metode avansate (opțional, pentru puncte suplimentare):**
   * **Bloc:** Împărțirea matricii în submatrici, care sunt alocate fiecărui thread.
   * **Distribuție bazată pe o funcție de distribuție delta:** Atribuirea indecșilor fiecărui thread pe baza unei funcții de distribuție (liniară sau ciclică).

**Date de intrare și ieșire:**

* **Fișier de intrare :** Matricea F și matricea de convoluție C trebuie generate și stocate anterior într-un fișier text.
* **Fișier de ieșire :** Programul trebuie să scrie timpul rezultat in urma aplicarii convoluției in consola(fprintf, respective System.out)

**Constrângeri:**

* **Tipuri de matrici:**
  + Pentru C++:
    - **Alocare statică:** Matricile sunt definite cu o dimensiune maximă.
    - **Alocare dinamică:** Matricile sunt alocate dinamic .
  + În Java, nu sunt impuse restricții specifice, dar trebuie folosită metoda standard de alocare a matricilor.
* **Threading:**
  + Trebuie folosite threaduri în mod explicit, fără utilizarea executorilor sau bibliotecilor de threading de nivel înalt.
* **Testare:**
  + Timpul de execuție trebuie măsurat pentru diverse dimensiuni ale matricilor și număr de threaduri, conform următoarelor seturi de date:
    - N=M=10N = M = 10N=M=10, n=m=3n = m = 3n=m=3, p=4
    - N=M=1000N = M = 1000N=M=1000, n=m=5n = m = 5n=m=5, p=2,4,8,16
    - N=10N = 10N=10, M=10000M = 10000M=10000, n=m=5n = m = 5n=m=5, p=2,4,8,16
    - N=10000N = 10000N=10000, M=10M = 10M=10, n=m=5n = m = 5n=m=5, p=2,4,8,
* N=10000N = 10000N=10000, M=10000M = 10000M=10000, n=m=5n = m = 5n=m=5, p=2,4,8,
* **Performanță:**
  + Fiecare test trebuie executat de 10 ori, iar timpul de execuție raportat va fi media aritmetică a celor 10 rulari.

**Cerințe de documentare și analiză:**

* **Analiza performanței:**
  + Compară performanța între implementările secvențiale și paralele.
  + Compară diferitele variante de împărțire pe threaduri.
  + Compară performanța implementării în Java cu cea în C++.
  + Analiza performanței implementării cu alocare statică vs alocare dinamică în C++.

**Constrângeri adiționale:**

* **Folosirea aceluiași fișier de intrare pentru toate testele:** Fișierul „input.txt” trebuie să fie identic pentru toate rularile .

Java

**input10\_10\_3.txt**

**Timpul de execuție pentru metoda secvențială este cel mai mic, ceea ce sugerează că pentru matrice mici, Java nu simte prea mult impactul legat de gestionarea firelor de execuție. Acest lucru sugerează că pentru dimensiuni mici, overhead-ul generat de crearea și gestionarea thread-urilor este semnificativ, iar pentru matrici cu dimensiuni mici, metodele paralele nu oferă un avantaj.**

**Columns da mai rau decat lines de exemplu pt ca la lines elementele sunt mai aproape unele de altele in memorie**

**input10\_10000\_5.txt**

**La dimensiuni mai mari, se observă o diferențiere vizibila. Metodele bazate pe linii (lines cycle și lines batch) sunt cele mai eficiente, cred eu ca datorită modului în care sunt gestionate datele în memorie+paralelizare.**

**input1000\_1000\_5.txt**

**Metoda secvențială este extrem de ineficientă, iar cele mai rapide metode sunt cele de linearizare, arătând o performanță bună pentru matrice mari. Observ că utilizarea unui număr mic de thread-uri (2 și 4) oferă cele mai bune rezultate, în timp ce utilizarea unui număr mai mare de thread-uri (8) duce la o performanță inferioară, probabil din cauza overhead-ului suplimentar în gestionarea acestora(la matrici mici).**

**input10000\_10\_5.txt**

Deși metoda secvențială rămâne mai lentă decât metodele paralele, performanțele sunt comparabile. Timpul de execuție pentru metodele cu linii rămâne superior celor cu coloane, pt ca avem o eficiență mai bună a gestionării memoriei pentru matricele cu nr de coloane mai mic.

Rezultatele arată o performanță mai bună pentru 2 thread-uri în comparație cu 8 în unele cazuri=>degeaba paralelizez daca imi da overhead-ul mare

Daca impartim timpul la secvential : 22.4352/0.0098=2.289,306, deci face procesorul ceva modificari

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie | Metoda |  |
| input10\_10\_3.txt | 1 | 0.00981 | sequential | |
| input10\_10\_3.txt | 4 | 0.56233 | lines cycle | |
| input10\_10\_3.txt | 4 | 0.65127 | lines batch | |
| input10\_10\_3.txt | 4 | 0.52755 | columns batch | |
| input10\_10\_3.txt | 4 | 0.7116 | columns cycle | |
| input10\_10\_3.txt | 4 | 0.64214 | linearization batch | |
| input10\_10\_3.txt | 4 | 0.54108 | linearization cycle | |
|  |  |  |  | |
| input10\_10000\_5.txt | 1 | 2.43708 | sequential | |
| input10\_10000\_5.txt | 4 | 1.42698 | lines cycle | |
| input10\_10000\_5.txt | 4 | 1.45267 | lines batch | |
| input10\_10000\_5.txt | 4 | 3.55268 | columns batch | |
| input10\_10000\_5.txt | 4 | 1.59031 | linearization batch | |
| input10\_10000\_5.txt | 4 | 1.63225 | linearization cycle | |
| input10\_10000\_5.txt | 4 | 1.61954 | columns cycle | |
|  |  |  |  | |
| input1000\_1000\_5.txt | 1 | 22.43525 | sequential | |
| input1000\_1000\_5.txt | 4 | 13.88335 | lines cycle | |
| input1000\_1000\_5.txt | 4 | 13.14088 | columns cycle | |
| input1000\_1000\_5.txt | 4 | 12.00404 | linearization cycle | |
| input1000\_1000\_5.txt | 4 | 12.90631 | linearization batch | |
| input1000\_1000\_5.txt | 4 | 12.44664 | lines batch | |
| input1000\_1000\_5.txt | 4 | 14.47051 | columns batch | |
|  |  |  |  | |
| input10000\_10\_5.txt | 1 | 2.63463 | sequential | |
| input10000\_10\_5.txt | 4 | 1.76308 | lines cycle | |
| input10000\_10\_5.txt | 4 | 1.84568 | columns cycle | |
| input10000\_10\_5.txt | 4 | 1.93251 | linearization cycle | |
| input10000\_10\_5.txt | 4 | 1.80505 | linearization batch | |
| input10000\_10\_5.txt | 4 | 1.8028 | columns batch | |
| input10000\_10\_5.txt | 4 | 1.80475 | lines batch | |
|  |  |  |  | |
| input10\_10000\_5.txt | 1 | 2.35432 | sequential | |
| input10\_10000\_5.txt | 2 | 1.93168 | lines batch | |
| input10\_10000\_5.txt | 2 | 1.91336 | columns batch | |
| input10\_10000\_5.txt | 2 | 1.99186 | linearization batch | |
| input10\_10000\_5.txt | 2 | 1.9957 | linearization cycle | |
| input10\_10000\_5.txt | 2 | 1.94274 | columns cycle | |
| input10\_10000\_5.txt | 2 | 1.87209 | lines cycle | |
|  |  |  |  | |
| input1000\_1000\_5.txt | 2 | 13.50605 | lines cycle | |
| input1000\_1000\_5.txt | 2 | 12.51728 | lines batch | |
| input1000\_1000\_5.txt | 2 | 15.92779 | columns batch | |
| input1000\_1000\_5.txt | 2 | 16.05985 | columns cycle | |
| input1000\_1000\_5.txt | 2 | 15.13205 | linearization cycle | |
| input1000\_1000\_5.txt | 2 | 14.59182 | linearization batch | |
|  |  |  |  | |
| input10000\_10\_5.txt | 2 | 2.05593 | columns batch | |
| input10000\_10\_5.txt | 2 | 2.22659 | columns cycle | |
| input10000\_10\_5.txt | 2 | 1.99245 | lines cycle | |
| input10000\_10\_5.txt | 2 | 1.85145 | lines batch | |
| input10000\_10\_5.txt | 2 | 2.08493 | linearization batch | |
| input10000\_10\_5.txt | 2 | 2.06429 | linearization cycle | |
|  |  |  |  | |
| input10\_10000\_5.txt | 8 | 1.93559 | lines batch | |
| input10\_10000\_5.txt | 8 | 2.00985 | lines cycle | |
| input10\_10000\_5.txt | 8 | 2.05142 | columns cycle | |
| input10\_10000\_5.txt | 8 | 2.22198 | linearization cycle | |
| input10\_10000\_5.txt | 8 | 2.14091 | linearization batch | |
| input10\_10000\_5.txt | 8 | 2.02929 | columns batch | |
|  |  |  |  | |
| input1000\_1000\_5.txt | 8 | 10.18008 | columns batch | |
| input1000\_1000\_5.txt | 8 | 10.97088 | columns cycle | |
| input1000\_1000\_5.txt | 8 | 8.18209 | lines cycle | |
| input1000\_1000\_5.txt | 8 | 8.52887 | lines batch | |
| input1000\_1000\_5.txt | 8 | 8.93537 | linearization batch | |
| input1000\_1000\_5.txt | 8 | 10.37285 | linearization cycle | |
|  |  |  |  | |
| input10000\_10\_5.txt | 8 | 1.76357 | columns cycle | |
| input10000\_10\_5.txt | 8 | 2.01717 | columns batch | |
| input10000\_10\_5.txt | 8 | 1.75674 | lines batch | |
| input10000\_10\_5.txt | 8 | 1.99465 | lines cycle | |
| input10000\_10\_5.txt | 8 | 2.02557 | linearization cycle | |
| input10000\_10\_5.txt | 8 | 1.89447 | linearization batch | |

C++ -static

**input10\_10\_3.txt, 4 threads**

**Timpul de execuție pentru metoda secvențială este mult mai mic decât pentru cele paralele. Acest lucru sugerează că pentru matrice mici, alocarea thread-urilor de la sistemul de operare este costisitoare, iar overhead-ul generat de aceasta inrautateste execuția metodelor paralele. De asemenea, metoda secvențială este mult mai rapidă în acest caz, deoarece datele sunt gestionate în memorie fără a necesita coordonare între thread-uri.**

**input10\_10000\_5.txt, 4 threads**

**Aici, metoda secvențială devine mult mai lentă, dar, în comparație cu metodele paralele, este totuși sub nivelul a ceea ce oferă C++. Este de remarcat că metodele de batch, atât pentru linii cât și pentru coloane, obțin cele mai bune rezultate. Acest lucru sugerează că, pentru matrice mari, strategia de alocare a sarcinilor pe fire se dovedește a fi eficientă, cu o distribuție echilibrată a sarcinilor între thread-uri.**

**input1000\_1000\_5.txt, 4 threads**

**Metoda secvențială este extrem de ineficientă, iar cele mai rapide metode sunt cele de tip batch pentru linii. Acest lucru sugerează o gestionare mai eficientă a memoriei și a sarcinilor de către metoda de alocare pe linii.**

**input10000\_10\_5.txt, 4 threads**

**La fel ca în cazul anterior, metoda secvențială este mai lentă, iar metodele paralele se comportă mult mai bine. Metoda de alocare pe linii (în special batch) a fost din nou cea mai eficientă, sugerând că atunci când există multe linii, sarcinile sunt distribuite mai eficient între thread-uri.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie | Metoda |
| input10\_10\_3.txt | static | 1 | 0.02353 | sequential |
| input10\_10\_3.txt | static | 4 | 1.59528 | lines cyclic |
| input10\_10\_3.txt | static | 4 | 1.75984 | columns cyclic |
| input10\_10\_3.txt | static | 4 | 1.56036 | linearization cyclic |
| input10\_10\_3.txt | static | 4 | 1.7788 | linearization batch |
| input10\_10\_3.txt | static | 4 | 1.6741 | lines batch |
| input10\_10\_3.txt | static | 4 | 1.53558 | columns batch |
|  |  |  |  |  |
| input10\_10000\_5.txt | static | 1 | 8.44673 | sequential |
| input10\_10000\_5.txt | static | 4 | 4.5705 | lines batch |
| input10\_10000\_5.txt | static | 4 | 4.37206 | columns batch |
| input10\_10000\_5.txt | static | 4 | 4.40269 | linearization batch |
| input10\_10000\_5.txt | static | 4 | 4.61639 | linearization cyclic |
| input10\_10000\_5.txt | static | 4 | 4.28866 | lines cyclic |
| input10\_10000\_5.txt | static | 4 | 4.49651 | columns cyclic |
|  |  |  |  |  |
| input1000\_1000\_5.txt | static | 1 | 82.5589 | sequential |
| input1000\_1000\_5.txt | static | 4 | 28.0719 | lines cyclic |
| input1000\_1000\_5.txt | static | 4 | 33.1262 | columns cyclic |
| input1000\_1000\_5.txt | static | 4 | 29.9827 | linearization cyclic |
| input1000\_1000\_5.txt | static | 4 | 28.2978 | linearization batch |
| input1000\_1000\_5.txt | static | 4 | 27.1586 | lines batch |
| input1000\_1000\_5.txt | static | 4 | 32.7086 | columns batch |
|  |  |  |  |  |
| input10000\_10\_5.txt | static | 1 | 20.2086 | sequential |
| input10000\_10\_5.txt | static | 4 | 9.30469 | lines cyclic |
| input10000\_10\_5.txt | static | 4 | 19.7564 | columns cyclic |
| input10000\_10\_5.txt | static | 4 | 17.1726 | linearization cyclic |
| input10000\_10\_5.txt | static | 4 | 12.58537 | linearization batch |
| input10000\_10\_5.txt | static | 4 | 8.42452 | lines batch |
| input10000\_10\_5.txt | static | 4 | 19.0138 | columns batch |

C++-dinamic

**Rezultatele sugerează că metodele secvențiale sunt mai rapide pentru matricile mici, dar devin ineficiente pe măsură ce dimensiunea matricii crește.**

**Metodele paralele, în special cele bazate pe batch,au timpi mai buni, mai ales pentru matrice mari, unde trebuie s gestionam efficient resusrsele.**

**Distribuția pe linii tinde să ofere cele mai bune rezultate în cazul matricilor cu multe linii, în timp ce metodele de alocare pe coloane sunt mai eficiente în cazul matricilor cu multe coloane.**

**=> trebuie sa adaptam metoda aleasa in functie de dimensiunea și structura matricii pentru a avea o eficiența a execuției mai buna .**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie | Metoda |
| input10\_10\_3.txt | dinamic | 1 | 0.00394 | sequential |
| input10\_10\_3.txt | dinamic | 4 | 0.62318 | lines batch |
| input10\_10\_3.txt | dinamic | 4 | 0.58045 | columns batch |
| input10\_10\_3.txt | dinamic | 4 | 0.57006 | linearization batch |
| input10\_10\_3.txt | dinamic | 4 | 0.59341 | linearization cyclic |
| input10\_10\_3.txt | dinamic | 4 | 0.57025 | lines cyclic |
| input10\_10\_3.txt | dinamic | 4 | 0.53908 | columns cyclic |
|  |  |  |  |  |
| input10\_10000\_5.txt | dinamic | 1 | 8.96173 | sequential |
| input10\_10000\_5.txt | dinamic | 4 | 3.4035 | lines batch |
| input10\_10000\_5.txt | dinamic | 4 | 3.26198 | columns batch |
| input10\_10000\_5.txt | dinamic | 4 | 3.35533 | linearization batch |
| input10\_10000\_5.txt | dinamic | 4 | 3.65072 | linearization cyclic |
| input10\_10000\_5.txt | dinamic | 4 | 3.97704 | lines cyclic |
| input10\_10000\_5.txt | dinamic | 4 | 3.39121 | columns cyclic |
|  |  |  |  |  |
| input1000\_1000\_5.txt | dinamic | 4 | 94.0414 | sequential |
| input1000\_1000\_5.txt | dinamic | 4 | 33.1649 | lines batch |
| input1000\_1000\_5.txt | dinamic | 4 | 34.0298 | columns batch |
| input1000\_1000\_5.txt | dinamic | 4 | 30.749 | linearization batch |
| input1000\_1000\_5.txt | dinamic | 4 | 32.6461 | linearization cyclic |
| input1000\_1000\_5.txt | dinamic | 4 | 33.4045 | lines cyclic |
| input1000\_1000\_5.txt | dinamic | 4 | 38.8184 | columns cyclic |
|  |  |  |  |  |
| input10000\_10\_5.txt | dinamic | 1 | 9.22506 | sequential |
| input10000\_10\_5.txt | dinamic | 4 | 3.24303 | lines batch |
| input10000\_10\_5.txt | dinamic | 4 | 3.58513 | linearization batch |
| input10000\_10\_5.txt | dinamic | 4 | 3.90266 | columns batch |
| input10000\_10\_5.txt | dinamic | 4 | 3.81567 | columns cyclic |
| input10000\_10\_5.txt | dinamic | 4 | 3.7146 | lines cyclic |
| input10000\_10\_5.txt | dinamic | 4 | 3.51992 | linearization cyclic |

Analiza performantei

Observam ca in general, columns batch este cel mai rau dintre metodele paralele din cauza accesului la coloane diferite(zone indepartate de memorie, nu accesam o memorie continua) .

Dacă un thread trebuie să acceseze datele din coloane diferite(columns batch), este mai puțin probabil să beneficieze de cache, comparativ cu **columns cycle**, unde fiecare thread accesează datele din aceeași coloană într-un mod secvențial.

Liniarizarea permite un acces rapid la date, deoarece permite procesarea continuă a memoriei, reducând cache misses.

**1. Dimensiuni mici ale matricii:**

* **C++ static > Java > C++ dynamic:**
  + **C++ static** alocă matricea în **stack**, ceea ce înseamnă acces rapid la memorie pentru matrici mici, deoarece datele sunt aproape de procesor și accesul la stack e rapid.
  + **C++ dinamic** alocă memoria în **heap**, deci la dimensiuni relative mici e ok
  + **Java** folosește **heap-ul** pentru toate alocările de obiecte, inclusiv pentru matrici. Sarcina in plus de administrare a heap-ului, colectarea deșeurilor (garbage collection) și verificările de siguranță (de exemplu, verificarea limitelor) contribuie la un timp de execuție mai lent în Java comparativ cu C++ la dimensiuni mici.

**2. Dimensiuni mari ale matricii:**

* **Java <C++ static < C++ dinamic:**
  + **C++ static** suferă din cauza utilizării **stack-ului** pentru alocarea memoriei. Stack-ul are o capacitate limitată și, pe măsură ce matricele devin mari, alocările statice pe stack pot cauza depășiri ale limitei stack-ului si au access mai lent la memorie.
  + **C++ dinamic** folosește **heap-ul** pentru alocarea memoriei => la matrici mari avem fragmentarea memoriei și in plus mai avem sarcini de gestionare a heap-ului care mananca timp
  + **Java**, în schimb, are un **management automatizat al memoriei** și un **garbage collector** foarte optimizat pentru obiecte mari și alocări pe termen lung, ceea ce îi permite să gestioneze matrici mari eficient. JVM-ul poate optimiza accesul la date, iar Java poate compensa prin optimizări interne (de exemplu, JIT - Just In Time compilation), care devin mai eficiente pe măsură ce codul rulează mai mult.

**Cod**

**Abordarea Secvențială**  
Aici aplicăm pur și simplu operația de convoluție pentru fiecare element secvențial, fără a folosi paralelism.

**Abordarea pe Linii**  
Aici rulăm un thread pentru anumite linii. Atunci când distribuția nu este uniformă, unele thread-uri vor prelua cel mult 1 linie suplimentară, ceea ce poate fi dezavantajos pentru matricele cu un număr mare de linii. În acest caz, putem utiliza abordările pe loturi și cele ciclice; un avantaj ar fi că thread-urile pot salva în cache unele linii, ceea ce le face mai rapide.

Pentru abordarea pe loturi, împărțim numărul de linii la P (numărul de thread-uri) și oferim fiecărui thread N / P linii, plus 1 dacă mai sunt linii rămase (distribuim N % P pentru primele linii). Pentru abordarea ciclică, trecem de la P la P linii.

**Abordarea pe Coloane**  
Aici rulăm un thread pentru anumite coloane; avem același dezavantaj ca în abordarea pe linii, dar pentru matricele cu un număr mare de coloane, putem folosi și abordările pe loturi și cele ciclice. Această abordare nu beneficiază de avantajul cache-ului, deoarece ar necesita accesarea unei pagini pentru fiecare element din coloană.

Pentru abordarea pe loturi, împărțim similar cu abordarea anterioară, dar pentru coloane, fiecare thread preluând M / P coloane, plus 1 dacă mai sunt coloane rămase (distribuim M % P ca și data trecută). Pentru abordarea ciclică, trecem de la P la P coloane.

**Abordarea de Linearizare**  
Aici folosim o funcție de mapare care linearizează matricea (i,j)→i∗M+j(i, j) \rightarrow i \* M + j(i,j)→i∗M+j, x→(x/M,x%M)x \rightarrow (x / M, x \% M)x→(x/M,x%M). Spre deosebire de celelalte abordări, unele thread-uri vor prelua cel mult un element, deoarece spațiul este contiguu, iar acest lucru le-ar putea permite să salveze în cache unele pagini cu elementele, dar doar în abordarea pe loturi; în abordarea ciclică ar trebui să performeze mai rău.

Având în vedere maparea, pentru abordarea pe loturi, oferim fiecărui thread N \* M / P elemente, plus 1 dacă mai sunt elemente rămase (distribuim N \* M % P pentru primele linii). Pentru abordarea ciclică, trecem de la P la P elemente în linearizare.

**Abordarea pe Blocuri**  
Similar cu abordarea de linearizare, dar calculăm indicele de început și indicele de sfârșit înainte ca thread-urile să ruleze, astfel încât fiecare thread să nu efectueze multe operații la fiecare pas. Avantajul este foarte nesemnificativ în comparație cu abordarea de linearizare.

Similar cu abordarea anterioară, oferim fiecărui thread N \* M / P elemente, calculând punctul de start (startI,startJ)(startI, startJ)(startI,startJ) și punctul de sfârșit (endI,endJ)(endI, endJ)(endI,endJ) și parcurgând aceste elemente.