Documentatie lab1

Oniga Andreea-Simona – 235/1

Java:

|  |  |  |
| --- | --- | --- |
| **Tip matrice** | **Nr. threads** | **Timp executie** |
| N=M=10  n=m=3 | Secvential | 0.7 |
| 4 | 3.9  1.1 |
| N=M=1000  n=m=5 | Secvential | 42.2 |
| 1 | 37.4  36.3 |
| 2 | 21.6  22.2 |
| 4 | 13.9  13.8 |
| 8 | 17.5  17.3 |
| 16 | 15.7  19.2 |
| N=10000  M=10  n=m=5 | Secvential | 10 |
| 1 | 6.2  7.8 |
| 2 | 6.2  3.6 |
| 4 | 6.2  2.9 |
| 8 | 5.9  3.2 |
| 16 | 5.1  3 |
| N=10  M=10000  n=m=5 | Secvential | 9.6 |
| 1 | 7.1  5.6 |
| 2 | 5.3  4.4 |
| 4 | 2.2  4.3 |
| 8 | 6.1  3.3 |
| 16 | 4.3  2.9 |

C++

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip matrice** | **Tip alocare** | **Nr. threads** | **Timp executie** |
| N=M=10  n=m=3 | Static | 4 | 0.03457  4.5013  1.73917 |
| dinamic | 4 | 0.02809  4.22257  2.03814 |
| N=M=1000  n=m=5 | static | secvential | 419.3793 |
| 1 | 424.4311  425.715 |
| 2 | 220.6964  221.973 |
| 4 | 119.4256  118.7622 |
| 8 | 80.0649  82.3959 |
| 16 | 68.5173  67.5933 |
| dinamic | secvential | 423.7502 |
| 1 | 431.6525  432.1045 |
| 2 | 222.1824  220.8415 |
| 4 | 115. 9345  113.285 |
| 8 | 86.73688  80.51278 |
| 16 | 69.72681  70.79503 |
| N=10000  M=10  n=m=5 | Static | secvential | 45.1166 |
| 1 | 53.4733  54.8979 |
| 2 | 34.6156  33.8527 |
| 4 | 25.9739  18.7144 |
| 8 | 22.4906  16.9921 |
| 16 | 27.9455  22.9614 |
| dinamic | secvential | 46.089 |
| 1 | 54.63874  53.50139 |
| 2 | 59.40702  57.771329 |
| 4 | 45.55238  33.56648 |
| 8 | 39.67353  26,01162 |
| 16 | 47.82534  36.31281 |
| N=10  M=10000  n=m=5 | Static | secvential | 60.6185 |
| 1 | 73.9368  79.0752 |
| 2 | 43.3929  44.8044 |
| 4 | 28.8184  32.0464 |
| 8 | 24.80468  36.1256 |
| 16 | 27.04393  39.44264 |
| dinamic | secvential | 108.86181 |
| 1 | 104.94714  90.41952 |
| 2 | 70.49784  53.48071 |
| 4 | 42.95711  43.83855 |
| 8 | 45.14097  35.82545 |
| 16 | 47.48421  34.70412 |

Observatii:

* testarea a fost facuta manual, motiv pentru care timpi pot sa nu fie cei mai apropiati de realitate.
* Pentru implementarile propuse de mine, java obtine timpi mai buni, pentru datele mai voluminoase(cazurile 2,3,4)
* Cei mai eficienti timpi se obtin de obicei pentru 4 thread uri si de cele mai multe ori pentru varinata paralela care lucreaza pe coloane
* Se mai observa ca de cele mai multe ori varianta secventiala este mai peerformanta decat utilizarea unui thread, in C++, in Java pare a fi diferit, thread-urile, indiferent de numar sa fie mai performante

Metoda pe care am abordat- o atat in java cat si in c++ a fost sa calculez fiecare element in parte in cadrul unei functii calculate, folosind o bordare virtuala.

Secvential, pur si simplu parcurg matricea si calculez pe rand fiecare element.

Pentru metoda paralela am folosit 2 indici start si end, calculati in functie de linii/coloane si numarul de thread-uri, care imparteau linile/coloanele la thread-uri. In cadrul thread-ului se prelucreaza elementele, tinand cont de start si end. Astfel, un thread va prelucra linile de la start la end sau coloanele de la start la end, in functie de cazul utilizat.

int[][] result3 = new int[N][M];  
threads = new Thread[p];  
cat = M/p;  
R = M%p;  
start = 0;  
  
start\_t = System.*currentTimeMillis*();  
for(int i = 0; i<p; i++){  
 int end = start+cat;  
 if(R>0){  
 end++;  
 R--;  
 }  
 threads[i] = new ThreadPtColoane(i, start, end, N, M, n, matrix, conv, result3);  
 threads[i].start();  
 start = end;  
}  
for(int i=0; i<p; i++){  
 try {  
 threads[i].join();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
}

Aceasta secventa de cod face urmatoarele:

* creaza matricea rezultat
* creaza thread-urile
* calculeaza start si end pentru fiecare thread, in cazul de fata catul si restul sunt folofite pentru a determina indicii pentru urmatorul thread

pentru linii pur si simplu catul si restul se calculeaza folosind N, adica numarul de linii.

public static int calculate(int startL, int startC, int N, int M, int K1, int K2, int[][] F, int[][] C){  
 int a = 0, b, s = 0, el;  
 int n = N;  
 int m = M;  
 int halfK1 = K1 / 2;  
 int halfK2 = K2 / 2;  
  
 for (int i = startL - halfK1; i <= startL + halfK1; i++) {  
 b = 0;  
 for (int j = startC - halfK2; j <= startC + halfK2; j++) {  
 int row = *min*(*max*(i, 0), n - 1);  
 int col = *min*(*max*(j, 0), m - 1);  
 el = F[row][col];  
 s += el \* C[a][b];  
 b++;  
 }  
 a++;  
 }  
 return s;  
}

acesta secventa demostreaza modul in care se calculeaza fiecare element.

Funcția calculează o sumă ponderată într-o submatrice de dimensiune K1 x K2 a matricei F, centrată în jurul poziției (startL, startC).

Submatricea F este centrată în jurul poziției (startL, startC) și, în cazul în care submatricea depășește limitele matricei F, se limitează valorile la marginea matricei.

Clase:

Convolution

Nu are parametrii, dar are metodele:

* main() – de tip int
* calculate(int startL, int startC, int N, int M, int K1, int K2, int[][] F, int[][] C) – de tip int
* convSecventila(int N, int M, int[][] exp, int n, int[][] conv) – de tip int[][]
* verifCorectitudinii(int[][] matrix, int[][] matrix1) – de tip int

ThreadPtLinii

Parametrii:

* id – int
* start - int
* end – int
* N – int
* M – int
* n -int
* exp – int[][]
* conv – int[][]
* res – int[][]

Metode:

* run()

ThreadPtColoane

Parametrii:

* id – int
* start - int
* end – int
* N – int
* M – int
* n -int
* exp – int[][]
* conv – int[][]
* res – int[][]

Metode:

* run()

Clasa generareMatrici este folosita pentru a genera date aleatori pentru matricile pentru test