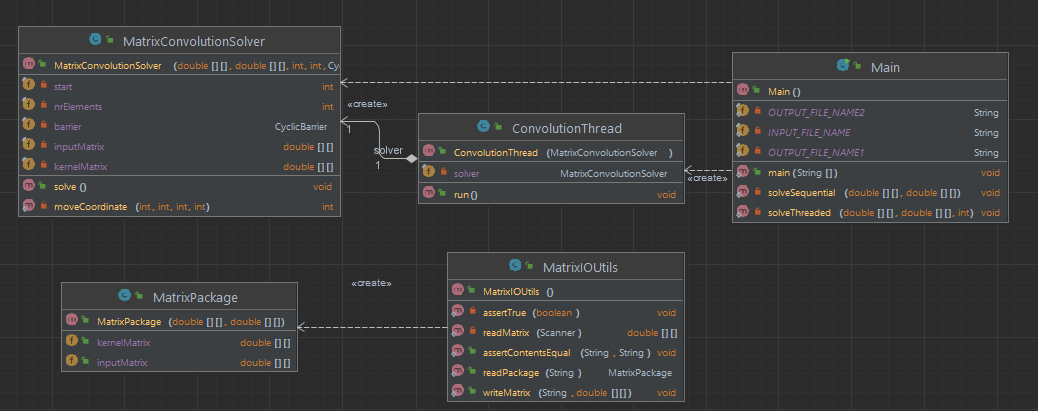
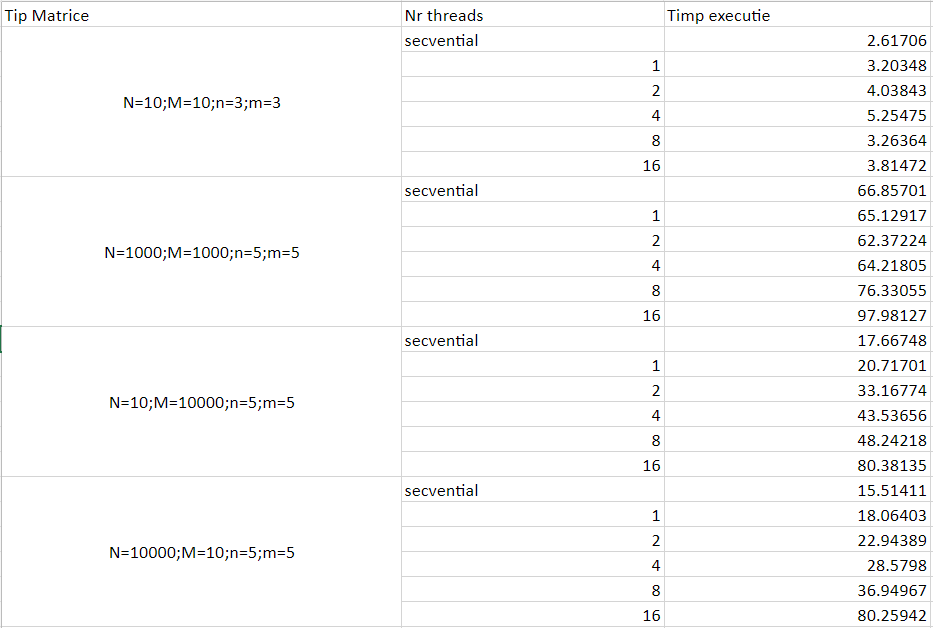
Documentatie PPD Lab 2

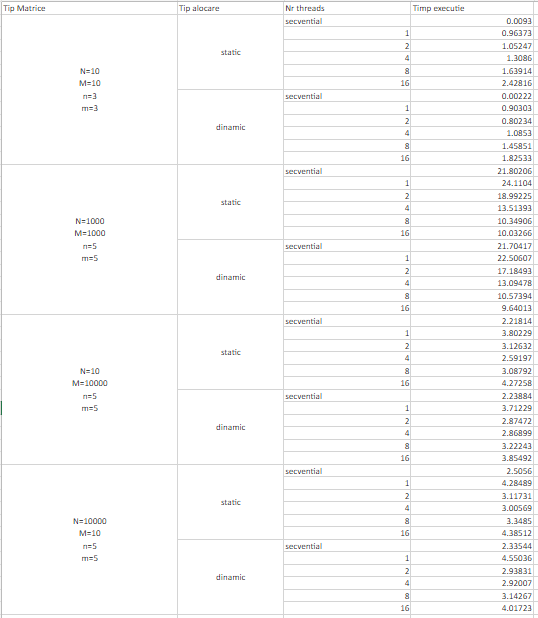
* Info:
  + OS: Windows 10
  + Processor: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz 2.30 GHz
  + Cores: 8
  + Logical Processors: 16
  + RAM: 16.0 GB (15.7 GB usable)
* Idee:
  + Avand o matrice de intrare de N pe M, vom avea N\*M elemente. Vom distribui in mod egal elementele la P thread-uri (N\*M/P elemente per thread), ceea ce ramane fiind distribuit de asemeni in mod egal (N\*M%P). Astfel, workload-ul dintre oricare doua thread-uri difera maxim printr-o unitate
  + Consideram urmatoarea indexare: coordonata (i,j) din matricea normala corespunde coordonatei j+M\*i in matricea reprezentata liniarizat; k in matricea reprezentata liniarizat corespunde coordonatelor (k/M,k%M) in matricea reprezentata normal
  + Fiecare thread va operea pe elementele din matrice indexate de la k la k+nrElemente (considerand reprezentarea liniarizata)
  + Pentru fiecare punct, se va aplica formula calcului convolutiei, obtinand astfel in matricea de output valoarea corespunzatoare
  + Pentru memorarea matricelor voi folosi chiar array-uri (double[][] a in Java, double\*\* si double[N][M] in C++ - pentru fiecare matrice in C++ am alocat doar atata memorie cat e necesara)
  + Rezultatele vor fi memorate intr-un buffer local pentru fiecare thread in parte. Utilizand o bariera, atunci cand toate thread-urile isi termina calculele, voi da drumul copierii buffer-lor in matricea initiala
  + Aceasta logica va fi aplicata atat in C++ at si in Java
* Specificatii:
  + Specifcatiile sunt puse pe codul sursa, la fiecare metoda, impreuna cu detaliile extra de implementare
* In Java:
  + Diagrama:



* + Rezultate:



* + Pentru N=M=10 si n=m=3:
    - Varianta secventiala este cea mai eficienta
    - De la 1 la 4 thread-uri, timpul creste, la 8 thread-uri scazand brusc, iar la 16 thread-uri incepand iar sa urce; un comportament ciudat – posibil ca pe fundal sa se fi oprit brusc un alt proces, eliberand resurse? Posibil sa fie si din cauza modului in care java gestioneaza thread-urile
    - Pentru o matrice atat de mica, era de asteptat ca varianta secventiala sa ruleze mai rapid, majoritatea timpului din variantele paralele probabil fiind alocat crearii si gestionarii thread-urilor
  + Pentru N=M=1000 si n=m=5:
    - Cea mai eficienta solutie este cea paralelizata cu 2 thread-uri – timpul scade de la secvential si 1 thread la 2 thread-uri, apoi urcand
  + Pentru cazurile (N,M,n,m)=(10,10000,5,5) si (10000,10,5,5):
    - Cea mai eficienta solutie este cea secventiala
    - Se observa ca o data cu cresterea numarului de thread-uri, contraintuitiv, scade performanta
  + Concluzii:
    - DISCLAIMER: Logica aplicata in Java este aceiasi cu cea din C/C++; Am verificat sa nu existe inadvertene, dar totul este in regula, output-urile fiind bune;
    - Este posibil ca paralelizarea convolutiei in Java (sau cel putin solutia mea) sa nu fie o solutie eficienta – presupunerea mea este ca nevoia constanta de a face kernel[i][j]\*input[i][j] necesita zone diferite de memorie, de fiecare data fiind invalidat cache-ul – o data pentru a aduce kernel, o data input – neputand utiliza cache-ul pentru a refolosi valori aflate alaturat (cum ar fi kernel[0][0] si kernel[0][1])
    - Un lucru de luat in considerare este ca, rularea initiala a solutiei secventiale, iar apoi a celei paralele in cadrul aceleiasi rulari genereaza timpi mult mai buni, probabil datorita unui mecanism de caching
* In C++:
  + Rezultate:



* + Pentru N=M=10 si n=m=3:
    - Varianta secventiala este cea mai eficienta, detasat
    - Cresterea numarului de thread-uri duce la cresterea duratei de executie
    - Pentru o matrice atat de mica, era de asteptat ca varianta secventiala sa ruleze mai rapid, majoritatea timpului din variantele paralele probabil fiind alocat crearii si gestionarii thread-urilor
  + Pentru N=M=1000 si n=m=5:
    - Cea mai eficienta solutie este cea paralelizata cu 16 thread-uri, timpul scazand cu numarul de thread-uri
  + Pentru cazurile (N,M,n,m)=(10,10000,5,5) si (10000,10,5,5):
    - Cea mai eficienta solutie este cea secventiala – ma gandesc ca este mai usor ca thread-urile sa-si invalideze cache-urile unui altuia, necesitand mai multe accesuri la memorie
    - Pentru variantele paralelizate, timpul in principiu scade cu cresterea numarului de thread-uri pana la atingerea unui spot (deobicei 4 thread-uri), dupa care iar urca
  + Static vs dinamic
    - Static este overall mai eficient – cand o matrice este alocata static, acesteia i se aloca o zona continua de memorie, deci e ca si cand ar fi liniarziat, probabil constribuind la eficienta (int a[10][10], la adresa a se va aloca spatiu pentru 10\*10 ints //vs// int\*\* a=new int\*[10], urmat de un for, cu a[i]=new int[10], caz in care pentru a se aloca spatiu pentru 10 pointeri, iar la fiecare pointer fiind alocat spatiu pentru 10 intregi – in cazul asta, vom avea 10 arrays de intregi alocate in zone diferite de memorie, caching-ul fiind mai ineficient)
    - Cazurile in care static pierde sunt N=M=10, n=m=3 – nu sunt complet sigur de ce, dar posibil ca alocarea sa ia putin mai mult timp fiind observabila?
* Java vs C++:
  + Chiar daca logica este aceiasi, C++ are timpi mult mai buni, lucrand mai low-level si fiind mai direct – probabil si compilatorul C++ realizeaza optimizari in spate
* Output Matrix vs Using Input Matrix As Output:
  + Atat in Java cat si in C++:
    - Output Matrix castiga doar pentru (N,M,n,m)=(10,10,3,3) , pentru secvential si numar de thread-uri intre 1 si 8
    - Pentru restul variantelor, castiga aproape detasat varianta de Input as Output
* Overall Conclusions:
  + Pare ca secvential merge mai bine in multe cazuri, probabil deoarece thread-urile isi invalideaza cache-urile mai rapid deoarece matricile sunt bidimensionale?
  + Pare ca variante de Input Matrix as Output Matrix este mai scalabila ca cea in care se foloseste o matrice separata de Output, dand rezultate mai bune atunci cand N,M devin numere foarte mari.