**Documentație**

1. **Analiza cerințelor**

Se considera problema de la laboratorul 1 cu urmatoarea modificare:

Se cere un program care sa asigure urmatoarea postconditie:

Postconditie: Matricea initiala contine imaginea filtrata.

Constrangere: NU se aloca o alta matrice rezultat si nici o matrice temporara!

Se pot folosi/aloca doar vectori temporari pentru care complexitatea spatiu se incadreaza in O(n).

Atentie si implementarea secventiala trebuie adaptata la constrangerea specificata. Va fi nevoie sa se foloseasca vectori auxiliari temporari.

Obiectiv: optimizarea complexitatii-spatiu in conditiile obtinerii unei performante ridicate.

Datele de intrare se citesc dintr-un fisier de intrare “date.txt”.

(Fisierul trebuie creat anterior prin adaugare de numere generate aleator.)

Rezultatul se salveaza intr-un fisier output.txt

1. **C++**
   1. **Proiectare**

**Structuri de Date**

A. **Vectori Bidimensionali** (vector<vector<int>>)

* Descriere: Folosim std::vector pentru a reprezenta matricea de intrare (mat) și matricea de convoluție (convMat).
* Motive pentru alegere:
  + Dimensiune Dinamică: std::vector permite crearea de matrice de dimensiuni variabile, ceea ce este util pentru lucrul cu matrice de dimensiuni diferite.
  + Acces Rapid: Oferă acces rapid la elemente prin indexare.
  + Gestionare Automată a Memoriei: Evită gestionarea manuală a memoriei, reducând riscurile de scurgeri de memorie.

B. **Mutex (std::mutex)**

* Descriere: Utilizăm un mutex pentru a controla accesul la resursele partajate în mediul de execuție multithreading.
* Motive pentru alegere:
  + Securitate: Asigură că doar un singur thread poate accesa datele partajate simultan, prevenind conflictele de date.

Partiționare pe Thread-uri

* Modul de Împărțire a Sarcinilor:
  + Sarcinile sunt împărțite pe linii ale matricei (mat). Fiecare thread preia un interval de linii pe care efectuează convoluția.
  + De exemplu, dacă avem 4 thread-uri și o matrice cu 8 linii, fiecare thread va prelua câte 2 linii.
* Beneficii:
  + Eficiență: Împărțirea sarcinilor pe thread-uri permite utilizarea eficientă a procesorului, reducând timpul total de execuție.
  + Scalabilitate: Programul poate scala ușor pe mașini cu mai multe procesoare, maximizând performanța în funcție de numărul de thread-uri disponibile.

1. **Clase și Funcții**
   1. **Clase**

1. ThreadLinii

* Atribute:
  + int start: Indicele de început al liniei pe care thread-ul o va procesa.
  + int stop: Indicele final al liniei.
* Metode:
  + ThreadLinii(int start, int stop): Constructor care inițializează atributele.
  + void operator()(): Metodă care implementează operația de convoluție pe intervalul specificat de linii.
  1. **Funcții**

1. void readInput()

* Parametri: N/A
* Rezultate: Umple mat cu date din fișierul de intrare și inițializează convMat.
* Limite: Se bazează pe formatul corect al fișierului de intrare.

2. void writeOutput()

* Parametri: N/A
* Rezultate: Scrie matricea procesată în fișierul "output.txt".
* Limite: Se așteaptă ca matricea să fie validă și să fie scrisă corect.

3. void secvential()

* Parametri: N/A
* Rezultate: Aplică convoluția secvențial pe întreaga matrice.
* Limite: Performanța este limitată de numărul de CPU cores.

4. void linii()

* Parametri: N/A
* Rezultate: Aplică convoluția în mod paralel pe întreaga matrice folosind OpenMP.
* Limite: Necesită un compilator care suportă OpenMP.

5. int main(int argc, char \*argv[])

* Parametri:
  + int argc: Numărul de argumente din linia de comandă.
  + char \*argv[]: Lista argumentelor din linia de comandă.
* Rezultate: Rulează aplicația, prelucrează matricea și scrie rezultatul.
* Limite: Se așteaptă argumente corecte pentru a funcționa corect.
  1. **Relații**

**Descrierea Relațiilor**

* main apelează funcțiile readInput(), secvential(), și linii() în funcție de alegerea utilizatorului.
* ThreadLinii este utilizat în cadrul funcției linii() pentru a gestiona sarcinile pe thread-uri.
* Funcțiile writeOutput() și readInput() sunt independente, dar sunt legate prin fluxul de date (matricea citită este procesată și apoi scrisă).
  1. **Specificația Funcțiilor**

**void readInput()**

* Parametri: N/A
* Rezultate:
  + Populează mat cu datele din fișier.
  + Inițializează convMat cu matricea de convoluție.
* Limite: Trebuie să existe un fișier "date.txt" formatat corect.

**void writeOutput()**

* Parametri: N/A
* Rezultate: Scrie matricea procesată în "output.txt".
* Limite: Trebuie să fie apelată după prelucrarea matricei.

**void secvential()**

* Parametri: N/A
* Rezultate:
  + Procesarea matricei mat prin convoluție secvențială.
* Limite: Poate fi ineficient pentru matrice mari.

**void linii()**

* Parametri: N/A
* Rezultate:
  + Aplică convoluția în mod paralel pe matricea mat.
* Limite: Necesită OpenMP; eficiența depinde de numărul de thread-uri.

**int main(int argc, char \*argv[])**

* Parametri:
  1. int argc: Numărul de argumente.
  2. char \*argv[]: Lista argumentelor.
* Rezultate:
  1. Rularea programului și gestionarea prelucrării.
* Limite: Se așteaptă argumente corecte; eroare la parametri nevalizi.

1. **Detalii de Implementare**

Implementarea soluției pentru programul de convoluție a matricei implică mai mulți pași esențiali, de la citirea datelor din fișier până la executarea operațiilor de convoluție și scrierea rezultatelor într-un fișier de ieșire. În continuare, voi explica modul de implementare, subliniind exemple relevante de cod pentru fiecare componentă.

* 1. **Citirea Datelor din Fișier**

Funcția readInput() este responsabilă pentru citirea matricei de intrare dintr-un fișier. Acesta verifică dacă fișierul este deschis cu succes și apoi populază vectorul bidimensional mat.

void readInput() {  
 ifstream inputFile("date.txt");  
 if (!inputFile) {  
 cerr << "Error opening file." << endl;  
 return;  
 }  
 mat.resize(N, vector<int>(M));  
 for (int i = 0; i < N; ++i) {  
 for (int j = 0; j < M; ++j) {  
 inputFile >> mat[i][j];  
 }  
 }  
 // 1 0 1  
 // 0 2 0  
 // -1 0 -1  
 convMat.resize(n, vector<int>(n));  
 convMat.assign({{1, 0, 1}, {0, 2, 0}, {-1, 0, -1}});  
  
 inputFile.close();  
}

* 1. **Scrierea Rezultatelor în Fișier**

Funcția writeOutput() scrie matricea procesată în fișierul "output.txt". Se asigură că matricea este formatată corect înainte de a fi scrisă.

void writeOutput() {  
 ofstream outputFile("output.txt");  
 if (!outputFile) {  
 cerr << "Error opening file." << endl;  
 return;  
 }  
  
 for (int i = 0; i < N; ++i) {  
 for (int j = 0; j < M; ++j) {  
 outputFile << mat[i][j];  
 if (j < M - 1) {  
 outputFile << " ";  
 }  
 }  
 if (i < N - 1) {  
 outputFile << endl;  
 }  
 }  
 outputFile.close();  
}

* 1. **Convoluția Secvențială**

Funcția secvential() aplică convoluția secvențial pe matricea mat, iterând prin fiecare element și aplicând kernel-ul de convoluție.

void secvential() {  
 vector<int> auxRow(M + n - 1);  
  
 for (int i = 0; i < N; ++i) {  
 for (int j = -n / 2; j < M + n / 2; ++j) {  
 int x = max(0, min(i, N - 1));  
 int y = max(0, min(j, M - 1));  
 auxRow[j + n / 2] = mat[x][y];  
 }  
  
 for (int j = 0; j < M; ++j) {  
 int sum = 0;  
 for (int i1 = 0; i1 < n; ++i1) {  
 for (int j1 = 0; j1 < n; ++j1) {  
 int x = i - n / 2 + i1;  
 int y = j - n / 2 + j1;  
 if (x >= 0 && y >= 0 && x < N && y < M) {  
 sum += auxRow[y + n / 2] \* convMat[i1][j1];  
 }}}   
 mat[i][j] = sum;  
 }}}

* 1. Convoluția Paralelă

Funcția linii() utilizează OpenMP pentru a executa convoluția pe linii ale matricei. Aceasta permite procesarea simultană a mai multor linii

void linii() {  
 int numThreads = p;  
  
 if (numThreads == 0) {  
 cerr << "Error: Number of threads is zero." << endl;  
 return;  
 }  
  
 if (N == 0) {  
 cerr << "Error: Number of rows (N) is zero." << endl;  
 return;  
 }  
  
 // Create a barrier for synchronization  
 barrier syncBarrier(numThreads);  
  
 vector<thread> threads;  
 int rowsPerThread = N / numThreads;  
  
 auto threadFunction = [&](int startRow, int endRow) {  
 vector<vector<int>> borderValues(endRow - startRow + 1, vector<int>(M));  
 vector<int> auxRow(M + n - 1);  
  
 // Step 1: Calculate new values for border cells and save them in separate vectors  
 for (int i = startRow; i <= endRow; ++i) {  
 for (int j = -n / 2; j < M + n / 2; ++j) {  
 int x = max(0, min(i, N - 1));  
 int y = max(0, min(j, M - 1));  
 auxRow[j + n / 2] = mat[x][y];  
 }  
  
 for (int j = 0; j < M; ++j) {  
 int sum = 0;  
 for (int i1 = 0; i1 < n; ++i1) {  
 for (int j1 = 0; j1 < n; ++j1) {  
 int x = i - n / 2 + i1;  
 int y = j - n / 2 + j1;  
 if (x >= 0 && y >= 0 && x < N && y < M) {  
 sum += auxRow[y + n / 2] \* convMat[i1][j1];  
 }  
 }  
 }  
 borderValues[i - startRow][j] = sum;  
 }  
 }  
  
 syncBarrier.arrive\_and\_wait();

// Step 3: Calculate new values for non-border cells and save them directly in the matrix  
 for (int i = startRow; i <= endRow; ++i) {  
 for (int j = -n / 2; j < M + n / 2; ++j) {  
 int x = max(0, min(i, N - 1));  
 int y = max(0, min(j, M - 1));  
 auxRow[j + n / 2] = mat[x][y];  
 }  
  
 for (int j = 0; j < M; ++j) {  
 int sum = 0;  
 for (int i1 = 0; i1 < n; ++i1) {  
 for (int j1 = 0; j1 < n; ++j1) {  
 int x = i - n / 2 + i1;  
 int y = j - n / 2 + j1;  
 if (x >= 0 && y >= 0 && x < N && y < M) {  
 sum += auxRow[y + n / 2] \* convMat[i1][j1];  
 }  
 }  
 }  
 mat[i][j] = sum;  
 }  
 }  
  
 syncBarrier.arrive\_and\_wait();  
  
 // Step 5: Update border elements by copying values from the vectors calculated in step 1  
 int i = startRow;  
 for (const auto& borderValue : borderValues) {  
 copy(borderValue.begin(), borderValue.end(), mat[i].begin());  
 i++;  
 }  
 };  
  
 for (int i = 0; i < numThreads; ++i) {  
 int startRow = i \* rowsPerThread;  
 int endRow = (i == numThreads - 1) ? N - 1 : (startRow + rowsPerThread - 1);  
 threads.emplace\_back(threadFunction, startRow, endRow);  
 }  
  
 for (auto& t : threads) {  
 t.join();  
 }  
}

* 1. **Utilizarea Clasei ThreadLinii**

Clasa ThreadLinii este folosită pentru a gestiona sarcinile fiecărui thread. Constructorul primește indicii de început și sfârșit, iar metoda operator() efectuează convoluția pe intervalul specificat.

class ThreadLinii {  
private:  
 int startRow, endRow, n, p;  
 vector<vector<int>>& mat;  
 vector<vector<int>>& convMat;  
 vector<vector<int>> borderValues;  
 bool flag;  
 barrier<>\* syncBarrier;  
  
public:  
 ThreadLinii(int startRow, int endRow, int n, int p, vector<vector<int>>& mat, vector<vector<int>>& convMat, barrier<>\* syncBarrier)  
 : startRow(startRow), endRow(endRow), n(n), p(p), mat(mat), convMat(convMat), syncBarrier(syncBarrier) {  
 borderValues.resize(endRow - startRow + 1, vector<int>(mat[0].size()));  
 flag = false;  
 }  
  
 void operator()() {  
 // Step 1: Calculate new values for border cells and save them in separate vectors  
 calculateBorderValues();  
  
 // Step 2: Set flag to true  
 flag = true;  
  
 syncBarrier->arrive\_and\_wait();  
  
 // Step 3: Calculate new values for non-border cells and save them directly in the matrix  
 calculateNonBorderValues();  
  
 // Step 4: Check if all neighboring threads have their flags set to true  
 syncBarrier->arrive\_and\_wait();  
  
 // Step 5: Update border elements by copying values from the vectors calculated in step 1  
 updateBorderValues();  
 }  
  
private:  
 void calculateBorderValues() {  
 vector<int> auxRow(mat[0].size() + n - 1);  
 for (int i = startRow; i <= endRow; ++i) {  
 for (int j = -n / 2; j < mat[0].size() + n / 2; ++j) {  
 int x = max(0, min(i, static\_cast<int>(mat.size()) - 1));  
 int y = max(0, min(j, static\_cast<int>(mat[0].size()) - 1));  
 auxRow[j + n / 2] = mat[x][y];  
 }  
  
 for (int j = 0; j < mat[0].size(); ++j) {  
 int sum = 0;  
 for (int i1 = 0; i1 < n; ++i1) {  
 for (int j1 = 0; j1 < n; ++j1) {  
 int x = i - n / 2 + i1;  
 int y = j - n / 2 + j1;  
 if (x >= 0 && y >= 0 && x < mat.size() && y < mat[0].size()) {  
 sum += auxRow[y + n / 2] \* convMat[i1][j1];  
 }  
 }  
 }  
 borderValues[i - startRow][j] = sum;  
 }  
 }  
 }  
  
 void calculateNonBorderValues() {  
 vector<int> auxRow(mat[0].size() + n - 1);  
 for (int i = startRow; i <= endRow; ++i) {  
 for (int j = -n / 2; j < mat[0].size() + n / 2; ++j) {  
 int x = max(0, min(i, static\_cast<int>(mat.size()) - 1));  
 int y = max(0, min(j, static\_cast<int>(mat[0].size()) - 1));  
 auxRow[j + n / 2] = mat[x][y];  
 }  
  
 for (int j = 0; j < mat[0].size(); ++j) {  
 int sum = 0;  
 for (int i1 = 0; i1 < n; ++i1) {  
 for (int j1 = 0; j1 < n; ++j1) {  
 int x = i - n / 2 + i1;  
 int y = j - n / 2 + j1;  
 if (x >= 0 && y >= 0 && x < mat.size() && y < mat[0].size()) {  
 sum += auxRow[y + n / 2] \* convMat[i1][j1];  
 }  
 }  
 }  
 mat[i][j] = sum;  
 }  
 }  
 }  
  
 void updateBorderValues() {  
 int i = startRow;  
 for (const auto& borderValue : borderValues) {  
 copy(borderValue.begin(), borderValue.end(), mat[i].begin());  
 i++;  
 }  
 }  
};

* 1. **Funcția Principală**

Funcția main() gestionează fluxul programului, inclusiv citirea inputului, alegerea metodei de procesare (secvențial sau paralel) și scrierea rezultatelor.

int main(int argc, char \*argv[]) {  
 if (argc < 7) {  
 cerr << "Usage: " << argv[0] << " <number\_of\_threads> <matrix\_rows> <matrix\_cols> <function\_name> <conv\_matrix\_size> <compare-1/0>" << endl;  
 return 1;  
 }  
  
 p = stoi(argv[1]);  
 N = stoi(argv[2]);  
 M = stoi(argv[3]);  
 string functionName = argv[4];  
 n = stoi(argv[5]);  
 compare = stoi(argv[6]);  
  
 // block folosit pentru testare  
 // p = 8;  
 // N = 1000;  
 // M = 1000;  
 // string functionName = "linii";  
 // n = 3;  
 // compare = 1;  
  
 auto start = chrono::high\_resolution\_clock::*now*();  
 readInput();  
  
 if (functionName == "secvential") {  
 secvential();  
 } else if (functionName == "linii") {  
 linii();  
 } else {  
 cerr << "Invalid function name. Use one of: secvential, linii" << endl;  
 return 1;  
 }  
  
 auto stop = chrono::high\_resolution\_clock::*now*();  
 chrono::duration<double, std::micro> duration = stop - start;  
 double microseconds = duration.count();  
 double milliseconds = microseconds / 1000;  
  
 cout << milliseconds << " milliseconds" << endl;  
  
 writeOutput();  
 if (compare == 1) {  
 //compare the output with the expected output  
 // am cate un fisier expected pt fiecare N, M, adica expected10x10.txt,  
 // expected1000x1000.txt si expected10000x10000.txt  
 if (N != 10 && N != 1000 && N != 10000) {  
 cout << "Invalid matrix size" << endl;  
 return 1;  
 }  
  
 ifstream outputFile("output.txt");  
 ifstream expectedFile = ifstream("expected" + to\_string(N) + "x" + to\_string(N) + ".txt");  
  
 string outputLine, expectedLine;  
 while (getline(outputFile, outputLine) && getline(expectedFile, expectedLine)) {  
 //strip both lines of trailing whitespaces  
 outputLine.erase(outputLine.find\_last\_not\_of(" \n\r\t") + 1);  
 expectedLine.erase(expectedLine.find\_last\_not\_of(" \n\r\t") + 1);  
 // cout<<outputLine<<endl;  
 // cout<<expectedLine<<endl;  
 if (outputLine != expectedLine) {  
 cout << "Output is not correct" << endl;  
 return 1;  
 }  
 }  
 cout<<"Output is correct"<<endl;  
 }  
  
 return 0;  
}

1. **Java**

**2. Proiectare**

1. **Structuri de Date**
   * **Matrice (2D array)**:
     + **Tip**: int[][]
     + **Utilizare**: Este utilizată pentru a stoca matricea de intrare și matricea de convoluție. Fiecare element din matrice reprezintă un pixel sau o valoare care va fi procesată.
     + **Motive**:
       1. Dimensiunile fixe permit acces rapid la elemente.
       2. Implementarea unui algoritm de convoluție necesită accesul direct la vecinii unui pixel, ceea ce este eficient în structurile de date 2D.
   * **CyclicBarrier**:
     + **Tip**: java.util.concurrent.CyclicBarrier
     + **Utilizare**: Este utilizat pentru a sincroniza multiple fire de execuție (threads) la un punct comun, asigurându-se că toate thread-urile au terminat de procesat porțiunea asignată înainte de a continua.
     + **Motive**:
       1. Asigură că toate thread-urile finalizează calculele înainte de a începe actualizarea matricei finale, prevenind problemele de concurență.
2. **Partiționare pe Thread-uri**

Sarcinile sunt împărțite pe thread-uri în funcție de rândurile matricei de intrare. Fiecare thread preia un subset de rânduri pentru a procesa convoluția, utilizând matricea de convoluție.

* **Mod de împărțire**:
  + Numărul de rânduri din matrice este împărțit între thread-uri, astfel încât fiecare thread să preia un număr aproximativ egal de rânduri de procesat.
  + Se calculează numărul de rânduri pe care fiecare thread ar trebui să le proceseze, iar rândurile rămase sunt distribuite între thread-uri.
* **Beneficii**:
  + Îmbunătățirea performanței prin utilizarea paralelismului, reducând timpul total de procesare.
  + Distribuirea echitabilă a sarcinii minimizează risipa de resurse și timpul de așteptare.

1. **Clase și Funcții**
   1. **Main**

* **Atribute**:
  + N, M: Dimensiunile matricei.
  + p: Numărul de thread-uri.
  + n: Dimensiunea matricei de convoluție.
  + mat: Matricea de intrare.
  + convMat: Matricea de convoluție.
* **Funcții**:
  + main(String[] args): Funcția principală care inițiază citirea datelor, execuția algoritmului de procesare și compararea rezultatelor.
  + readInput(): Citește matricea de intrare dintr-un fișier.
  + saveOutput(): Salvează rezultatul în fișier.

**2. Secvential**

* **Atribute**:
  + N, M, n, mat, convMat: Dimensiuni și matricele necesare.
* **Funcții**:
  + run(): Procesează matricea de intrare folosind metoda de convoluție în mod secvențial.

**3. ThreadLinii**

* **Atribute**:
  + start, stop: Indici pentru rândurile de procesat.
  + mat, convMat, barrier: Referințe la matricea de intrare, matricea de convoluție și barrier pentru sincronizare.
* **Funcții**:
  + run(): Procesează rândurile specificate ale matricei, calculând valorile convoluției.

**4. Linii**

* **Atribute**:
  + N, M, n, p, mat, convMat, barrier: Dimensiuni, matricele de intrare și de convoluție, și barrier pentru sincronizare.
* **Funcții**:
  + run(): Creează și pornește thread-urile pentru procesarea matricei folosind clasa ThreadLinii.

**Relații**

1. **Main**:
   * Creează instanțe ale claselor Secvential și Linii, delegând execuția funcțiilor de procesare.
   * Apelează metodele readInput() și saveOutput() pentru gestionarea fișierelor.
2. **Linii**:
   * Folosește ThreadLinii pentru a distribui sarcinile de procesare pe multiple thread-uri.
3. **ThreadLinii**:
   * Este utilizată de clasa Linii pentru a efectua calculul de convoluție pe rândurile asignate.

**Specificația Funcțiilor**

**Main**

* **main(String[] args)**:
  + **Parametri**: Argumente din linia de comandă (numărul de thread-uri, dimensiunile matricei, numele funcției, etc.).
  + **Rezultate**: Începe execuția programului, citind input-ul, procesându-l, și scriind output-ul.
  + **Limite**: Necesită cel puțin 6 argumente.

**Secvential**

* **run()**:
  + **Parametri**: N/A (folosește atributele clasei).
  + **Rezultate**: Modifică matricea mat cu rezultatul convoluției.
  + **Limite**: Trebuie să se asigure că matricea mat este corect inițializată și are dimensiunile așteptate.

**ThreadLinii**

* **run()**:
  + **Parametri**: N/A (folosește atributele clasei).
  + **Rezultate**: Procesează rândurile specificate și actualizează matricea mat.
  + **Limite**: Trebuie să respecte limitele matricei pentru a evita ArrayIndexOutOfBoundsException.

**Linii**

* **run()**:
  + **Parametri**: N/A (folosește atributele clasei).
  + **Rezultate**: Creează și pornește thread-urile pentru procesarea matricei.
  + **Limite**: Trebuie să se asigure că numărul de thread-uri nu depășește dimensiunea matricei.

**3. Detalii de Implementare**

**Descrierea Implementării**

Implementarea soluției se concentrează pe aplicarea unui algoritm de convoluție asupra unei matrice folosind un număr specificat de thread-uri pentru a îmbunătăți performanța procesării. În cadrul aplicației, utilizatorul poate alege între două metode de procesare: secvențială și pe baza de fire de execuție (multithreading).

**Structura Proiectului**

Proiectul este structurat în patru clase principale:

1. **Main**: Controlerul principal care inițiază execuția aplicației.
2. **Secvential**: Clasa care implementează procesarea secvențială a matricei.
3. **Linii**: Clasa care coordonează crearea și execuția thread-urilor pentru procesarea pe linii.
4. **ThreadLinii**: Clasa care reprezintă fiecare thread responsabil pentru procesarea unei porțiuni a matricei.

**Detalii de Implementare pentru Fiecare Clasă**

* 1. **Clasa Main**

*public class* Main {  
  
 *static int N*, *M*, *p*, *compare*;  
 *static int n* = 3;  
 *static int*[][] *mat*;  
 *static int*[][] *convMat* = {  
 {1, 0, 1},  
 {0, 2, 0},  
 {-1, 0, -1}  
 };  
 *static* CyclicBarrier *barrier*;  
  
 *public static void* main(String[] args) *throws* FileNotFoundException, InterruptedException {  
 *if* (args.length < 6) {  
 System.err.println("Usage: java .jar <number\_of\_threads> <matrix\_rows> <matrix\_cols> <function\_name> <conv\_matrix\_size> <compare=1/0>");  
 *return*;  
 }  
  
 *p* = Integer.*parseInt*(args[0]);  
 *N* = Integer.*parseInt*(args[1]);  
 *M* = Integer.*parseInt*(args[2]);  
 String functionName = args[3];  
 *n* = Integer.*parseInt*(args[4]);  
 *compare* = Integer.*parseInt*(args[5]);  
  
 *readInput*();  
 *barrier* = *new* CyclicBarrier(*p*);  
  
 *long* startTime = System.*nanoTime*();  
  
 *switch* (functionName) {  
 *case* "secvential":  
 Secvential secvential = *new* Secvential(*N*, *M*, *n*, *mat*, *convMat*);  
 secvential.run();  
 *break*;  
 *case* "linii":  
 Linii linii = *new* Linii(*N*, *M*, *n*, *p*, *mat*, *convMat*);  
 linii.run();  
 *break*;  
 *default*:  
 System.err.println("Invalid function name. Use one of: secvential, linii");  
 *return*;  
 }  
  
 *long* endTime = System.*nanoTime*();  
 *double* milliseconds = (endTime - startTime) / 1\_000\_000.0;  
  
 System.out.println(milliseconds + " milliseconds");  
  
 *saveOutput*();

*if*(*compare* == 1){  
 *//compare the output with the expected output  
 // am cate un fisier expected pt fiecare N, M, adica expected10x10.txt,  
 // expected1000x1000.txt si expected10000x10000.txt* Scanner scanner = *new* Scanner(*new* File("expected" + *N* + "x" + *M* + ".txt"));  
 Scanner scanner2 = *new* Scanner(*new* File("output.txt"));  
 *boolean* ok = *true*;  
 *for* (*int* i = 0; i < *N*; ++i) {  
 *for* (*int* j = 0; j < *M*; ++j) {  
 *if*(scanner.nextInt() != scanner2.nextInt()){  
 ok = *false*;  
 System.out.println("Different values at " + i + " " + j);  
 *break*;  
 }  
 }  
 *if*(!ok)  
 *break*;  
 }  
 *if*(ok)  
 System.out.println("The output is correct");  
 *else* System.out.println("The output is incorrect");  
 }  
 }

*private static void* readInput() *throws* FileNotFoundException {  
 Scanner scanner = *new* Scanner(*new* File("date.txt"));  
 *mat* = *new int*[*N*][*M*];  
 *for* (*int* i = 0; i < *N*; ++i)  
 *for* (*int* j = 0; j < *M*; ++j)  
 *mat*[i][j] = scanner.nextInt();  
}

*private static void* saveOutput() *throws* FileNotFoundException {  
 PrintWriter writer = *new* PrintWriter(*new* File("output.txt"));  
 *for* (*int* i = 0; i < *N*; ++i) {  
 *for* (*int* j = 0; j < *M*; ++j) {  
 writer.print(*mat*[i][j] + " ");  
 }  
 writer.println();  
 }  
 writer.close();  
}

* 1. **Clasa Secvential**

1. *public class* Secvential {  
    *private final int* N, M, n;  
    *private final int*[][] mat;  
    *private final int*[][] convMat;  
     
    *public* Secvential(*int* N, *int* M, *int* n, *int*[][] mat, *int*[][] convMat) {  
    *this*.N = N;  
    *this*.M = M;  
    *this*.n = n;  
    *this*.mat = mat;  
    *this*.convMat = convMat;  
    }  
     
    *public void* run() {  
    *int*[] auxRow = *new int*[M + n - 1]; *//randul auxiliar pentru a nu iesi din matrice  
    for* (*int* i = 0; i < N; ++i) {  
    *for* (*int* j = -n / 2; j < M + n / 2; ++j) {  
    *int* x = Math.*max*(0, Math.*min*(i, N - 1));  
    *int* y = Math.*max*(0, Math.*min*(j, M - 1));  
    auxRow[j + n / 2] = mat[x][y];  
    }  
     
    *for* (*int* j = 0; j < M; ++j) {  
    *int* sum = 0;  
    *for* (*int* i1 = 0; i1 < n; ++i1) {  
    *for* (*int* j1 = 0; j1 < n; ++j1) {  
    *int* x = i - n / 2 + i1;  
    *int* y = j - n / 2 + j1;  
    *if* (x >= 0 && y >= 0 && x < N && y < M) {  
    sum += auxRow[y + n / 2] \* convMat[i1][j1];  
    }  
    }  
    }  
    mat[i][j] = sum;  
    }  
    }  
    }  
   }
   1. **Clasa ThreadLinii**
2. *public class* ThreadLinii *implements Runnable* {  
    *private int* startRow, endRow, n, p;  
    *private int*[][] mat, convMat;  
    *private int*[][] borderValues;  
    *private boolean* flag;  
    *private* CyclicBarrier barrier;  
     
    *public* ThreadLinii(*int* startRow, *int* endRow, *int* n, *int* p, *int*[][] mat, *int*[][] convMat, CyclicBarrier barrier) {  
    *this*.startRow = startRow;  
    *this*.endRow = endRow;  
    *this*.n = n;  
    *this*.p = p;  
    *this*.mat = mat;  
    *this*.convMat = convMat;  
    *this*.barrier = barrier;  
    *this*.borderValues = *new int*[endRow - startRow + 1][mat[0].length];  
    *this*.flag = *false*;  
    }  
     
    @Override  
    *public void* run() {  
    *// Step 1: Calculate new values for border cells and save them in separate vectors* calculateBorderValues();  
     
    *// Step 2: Set flag to true* flag = *true*;  
     
    *try* {  
    barrier.await();  
    } *catch* (InterruptedException | BrokenBarrierException e) {  
    e.printStackTrace();  
    }  
     
    *// Step 3: Calculate new values for non-border cells and save them directly in the matrix* calculateNonBorderValues();  
     
    *// Step 4: Check if all neighboring threads have their flags set to true  
    try* {  
    barrier.await();  
    } *catch* (InterruptedException | BrokenBarrierException e) {  
    e.printStackTrace();  
    }  
     
    *// Step 5: Update border elements by copying values from the vectors calculated in step 1* updateBorderValues();  
    }  
     
    *private void* calculateBorderValues() {  
    *int* auxRow[] = *new int*[mat[0].length + n - 1];  
    *for* (*int* i = startRow; i <= endRow; ++i) {  
    *for* (*int* j = -n / 2; j < mat[0].length + n / 2; ++j) {  
    *int* x = Math.*max*(0, Math.*min*(i, mat.length - 1));  
    *int* y = Math.*max*(0, Math.*min*(j, mat[0].length - 1));  
    auxRow[j + n / 2] = mat[x][y];  
    }  
     
    *for* (*int* j = 0; j < mat[0].length; ++j) {  
    *int* sum = 0;  
    *for* (*int* i1 = 0; i1 < n; ++i1) {  
    *for* (*int* j1 = 0; j1 < n; ++j1) {  
    *int* x = i - n / 2 + i1;  
    *int* y = j - n / 2 + j1;  
    *if* (x >= 0 && y >= 0 && x < mat.length && y < mat[0].length) {  
    sum += auxRow[y + n / 2] \* convMat[i1][j1];  
    }  
    }  
    }  
    borderValues[i - startRow][j] = sum;  
    }  
    }  
    }  
     
    *private void* calculateNonBorderValues() {  
    *int* auxRow[] = *new int*[mat[0].length + n - 1];  
    *for* (*int* i = startRow; i <= endRow; ++i) {  
    *for* (*int* j = -n / 2; j < mat[0].length + n / 2; ++j) {  
    *int* x = Math.*max*(0, Math.*min*(i, mat.length - 1));  
    *int* y = Math.*max*(0, Math.*min*(j, mat[0].length - 1));  
    auxRow[j + n / 2] = mat[x][y];  
    }  
     
    *for* (*int* j = 0; j < mat[0].length; ++j) {  
    *int* sum = 0;  
    *for* (*int* i1 = 0; i1 < n; ++i1) {  
    *for* (*int* j1 = 0; j1 < n; ++j1) {  
    *int* x = i - n / 2 + i1;  
    *int* y = j - n / 2 + j1;  
    *if* (x >= 0 && y >= 0 && x < mat.length && y < mat[0].length) {  
    sum += auxRow[y + n / 2] \* convMat[i1][j1];  
    }  
    }  
    }  
    mat[i][j] = sum;  
    }  
    }  
    }  
     
    *private void* updateBorderValues() {  
    *int* i = startRow;  
    *for* (*int*[] borderValue : borderValues) {  
    System.*arraycopy*(borderValue, 0, mat[i], 0, borderValue.length);  
    i++;  
    }  
    }  
   }
   1. **Clasa Linii**
3. *public class* Linii {  
    *private int* N, M, n, p;  
    *private int*[][] mat, convMat;  
     
    *public* Linii(*int* N, *int* M, *int* n, *int* p, *int*[][] mat, *int*[][] convMat) {  
    *this*.N = N;  
    *this*.M = M;  
    *this*.n = n;  
    *this*.p = p;  
    *this*.mat = mat;  
    *this*.convMat = convMat;  
    }  
     
    *public void* run() *throws* InterruptedException {  
    CyclicBarrier barrier = *new* CyclicBarrier(p);  
    Thread[] threads = *new* Thread[p];  
    *int* rowsPerThread = N / p;  
     
    *for* (*int* i = 0; i < p; i++) {  
    *int* startRow = i \* rowsPerThread;  
    *int* endRow = (i == p - 1) ? N - 1 : (startRow + rowsPerThread - 1);  
    threads[i] = *new* Thread(*new* ThreadLinii(startRow, endRow, n, p, mat, convMat, barrier));  
    threads[i].start();  
    }  
     
    *for* (Thread thread : threads) {  
    thread.join();  
    }  
    }  
   }

**Exemplu de Utilizare**

* Compilarea

javac -d bin src/ro/ppd/lab1/\*.java

* Executarea aplicatiei

java -cp bin ro.ppd.lab1.Main 4 10 10 secvential 3 1

Aici, se utilizează 4 thread-uri pentru a procesa o matrice de 10x10 cu o matrice de convoluție de 3x3.

* Rezultatul: Rezultatul procesării va fi salvat în fișierul output.txt, iar timpul de execuție va fi afișat în consolă.

**Cazuri de Testare C++**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tip Matrice** | **Matrice Convoluție** | **Nume Funcție** | **Nr. Threads** | **Timp Execuție (ms)** |
| 10x10 | 3x3 | secvențial | 1 | 0.2459 |
| 10x10 | 3x3 | linii | 2 | 0.2456 |
| 10x10 | 3x3 | linii | 4 | 0.2367 |
| 10x10 | 3x3 | linii | 8 | 0.2314 |
| 10x10 | 3x3 | linii | 16 | 0.2347 |
| 1000x1000 | 3x3 | secvențial | 1 | 206.4098 |
| 1000x1000 | 3x3 | linii | 2 | 203.9199 |
| 1000x1000 | 3x3 | linii | 4 | 202.6071 |
| 1000x1000 | 3x3 | linii | 8 | 200.5643 |
| 1000x1000 | 3x3 | linii | 16 | 203.3448 |
| 10000x10000 | 3x3 | secvențial | 1 | 20538.34 |
| 10000x10000 | 3x3 | linii | 2 | 18749.24 |
| 10000x10000 | 3x3 | linii | 4 | 18184.48 |
| 10000x10000 | 3x3 | linii | 8 | 15873.32 |
| 10000x10000 | 3x3 | linii | 16 | 15977.83 |

**Explicații:**

* **Tip Matrice**: Dimensiunea matricei de intrare.
* **Matrice Convoluție**: Dimensiunea matricei de convoluție utilizate.
* **Nume Funcție**: Numele funcției utilizate pentru procesare (secvențial sau linii).
* **Nr. Threads**: Numărul de fire (threads) utilizate pentru execuție.
* **Timp Execuție (ms)**: Timpul de execuție în milisecunde.

**Cazuri de Testare Java**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tip Matrice** | **Matrice Convoluție** | **Nume Funcție** | **Nr. Threads** | **Timp Execuție (ms)** |
| 10x10 | 3x3 | secvențial | 1 | 0.51561 |
| 10x10 | 3x3 | linii | 2 | 2.72349 |
| 10x10 | 3x3 | linii | 4 | 2.75956 |
| 10x10 | 3x3 | linii | 8 | 3.04168 |
| 10x10 | 3x3 | linii | 16 | 3.20058 |
| 1000x1000 | 3x3 | secvențial | 1 | 42.88476 |
| 1000x1000 | 3x3 | linii | 2 | 44.81351 |
| 1000x1000 | 3x3 | linii | 4 | 32.56567 |
| 1000x1000 | 3x3 | linii | 8 | 40.57604 |
| 1000x1000 | 3x3 | linii | 16 | 81.50314 |
| 10000x10000 | 3x3 | secvențial | 1 | 3560.55077 |
| 10000x10000 | 3x3 | linii | 2 | 2326.75071 |
| 10000x10000 | 3x3 | linii | 4 | 1315.47774 |
| 10000x10000 | 3x3 | linii | 8 | 817.60592 |
| 10000x10000 | 3x3 | linii | 16 | 605.21188 |

**Explicații:**

* **Tip Matrice**: Dimensiunea matricei de intrare.
* **Matrice Convoluție**: Dimensiunea matricei de convoluție utilizate.
* **Nume Funcție**: Numele funcției utilizate pentru procesare (secvențial sau linii).
* **Nr. Threads**: Numărul de fire (threads) utilizate pentru execuție.
* **Timp Execuție (ms)**: Timpul de execuție în milisecunde.

**Observații:**

* Timpul de execuție pentru metoda secvențială este semnificativ mai mic în cazul matricei 10x10, dar devine mult mai mare pentru dimensiuni mai mari.
* Utilizarea firelor în execuție în cazul Java nu pare să ofere îmbunătățiri semnificative, iar unele execuții cu mai multe fire sunt mai lente comparativ cu execuția secvențială, în special pentru matricele mari.

**Evaluarea Performanței**

Performanța soluției a fost evaluată prin compararea timpilor de execuție pentru diferite dimensiuni ale matricei, atât pentru implementarea secvențială, cât și pentru cea bazată pe fire (multi-threading).

* **Soluția Secvențială**:
  + La dimensiuni mici (10x10), timpul de execuție este de ordinul milisecundelor, ceea ce este acceptabil.
  + Pe măsură ce dimensiunea matricei crește, cum ar fi 1000x1000 și 10000x10000, timpul de execuție devine semnificativ mai mare, depășind 20 de secunde în cazul matricei de 10000x10000.
* **Soluția Thread-uri împărțite pe linii**:
  + Timpul de execuție pentru 10x10 este mai mare decât în cazul soluției secvențiale, indicând o posibilă problemă de overhead generat de crearea și gestionarea firelor.
  + La dimensiuni medii (1000x1000), s-au observat fluctuații în timp, iar pentru dimensiuni mari (10000x10000), s-au obținut îmbunătățiri semnificative, unde timpul a scăzut semnificativ (de exemplu, 605.21188 ms cu 16 fire).
  + Este important de menționat că numărul optim de fire nu a fost constant pentru toate dimensiunile, ceea ce sugerează că optimizarea poate fi sensibilă la variabilele de mediu și la caracteristicile hardware-ului.

**Rezultate Finale și Concluzii**

* **Realizări Principale**:
  + Implementarea cu succes a ambelor versiuni, secvențială și multi-threading, a matricei de convoluție.
  + Crearea și utilizarea unei metode de testare pentru a compara rezultatele obținute cu cele așteptate, asigurând corectitudinea soluției.
* **Provocări Întâmpinate**:
  + O provocare majoră a fost gestionarea sincronizării între fire, mai ales atunci când scriam rezultatele finale în matricea de ieșire. Aici, utilizarea CyclicBarrier a fost esențială, dar a introdus complexitate suplimentară.
  + Observația că optimizarea prin utilizarea firelor nu a fost constantă pe toată gama de dimensiuni a matricei a fost un alt punct de îngrijorare. Acest lucru subliniază necesitatea unor teste detaliate și adaptarea algoritmului în funcție de dimensiunea și structura datelor.