Paralelizarea Buclelor: Tehnici, Provocări și Impact în Optimizarea Performanței

Ingineria sistemelor distribuite

Cod sursă: https://github.com/AndreeaDraghici/Parallel-Matrix-Multiplication

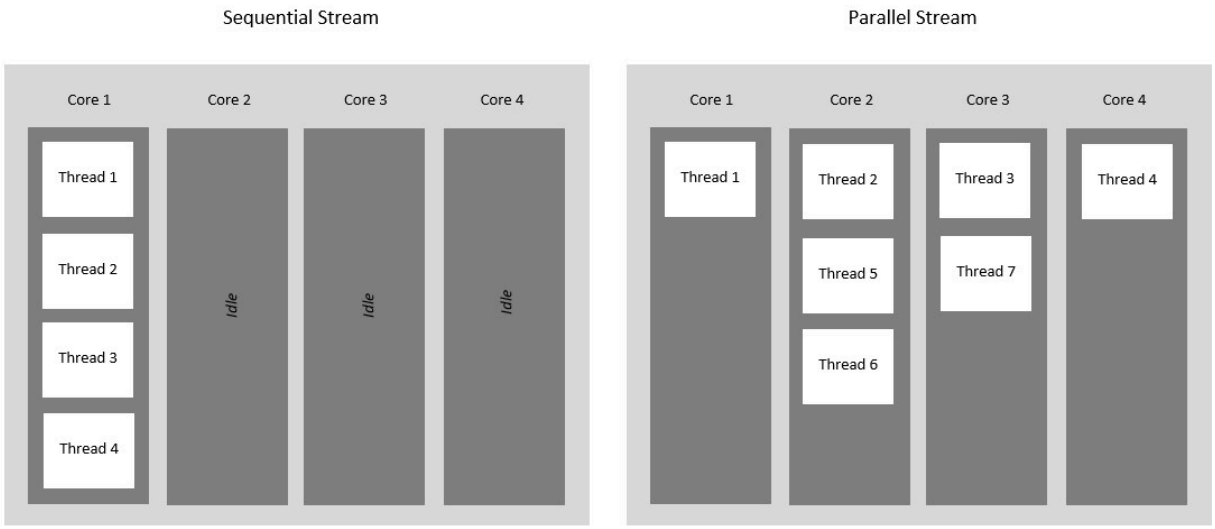
Specializarea: Inginerie Software
Student: Draghici Andreea-Maria
Grupa: IS 2.1 B

Cuprins

Cuprins 2
1. Introducere 3
2. Metodologii pentru Paralizarea Buclelor 3
DISTRIBUTED Loop Parallelism 3
DOACROSS Parallelism 4
DOPIPE Parallelism (Pipelined Parallelism) 4
3. Tehnici de Paralelizare a Buclelor 5
4. Provocări în Paralelizarea Buclelor 6
1. Dependențele de Date 6
2. Supraîncărcarea Resurselor 6
3. Sincronizarea și Blocajele 6
4. Granularitatea Sarcinilor 7
5. Probleme de Scalabilitate 7
5. Impactul Paralelizării asupra Performanței 7
1. Creșterea vitezei de execuție 7
2. Utilizarea eficientă a resurselor hardware 7
3. Probleme și limitări 8
4. Domenii de aplicare 8
6. Concluzie 8
7. Demo Aplicație Practică 9
Obiective 9
7.1 Algoritmi utilizați 9
7.1.1 Analiza algoritmilor utilizați pentru înmulțirea matricelor 11
1. Algoritmul clasic de multiplicare (Sequential) 11
2. Versiunea paralelă cu ExecutorService (Parallel) 11
3. Algoritmul lui Strassen 12
4. Versiunea paralelă cu Fork Join 13
5. Versiunea paralelă cu Parallel Stream 15
7.2 Rezultate obținute 16
7.2.2 Dataset de 1000 elemente 16
7.2.3 Dataset de 100 elemente 18
7.2.4 Dataset de 500 elemente 19
7.2.5 Dataset de 50 elemente 21
7.2.6 Dataset de 10 elemente 22
8. Structura aplicației 23
9. Rularea aplicației folosind interfața grafică 24
1. Secțiunea de Introducere a Datelor 24
2. Secțiunea de Alegere a Algoritmului 25
3. Butonul "Start Multiplication" 25
10. Bibliografie 27
11. Hardware 28

1. Introducere

Paralelizarea buclelor reprezintă una dintre cele mai eficiente tehnici pentru optimizarea performanței în aplicațiile software. Această metodă facilitează divizarea buclelor mari în părți independente, care pot fi executate în paralel, utilizând mai multe unități de procesare, reducând astfel timpul total de execuție.



2. Metodologii pentru Paralizarea Buclelor

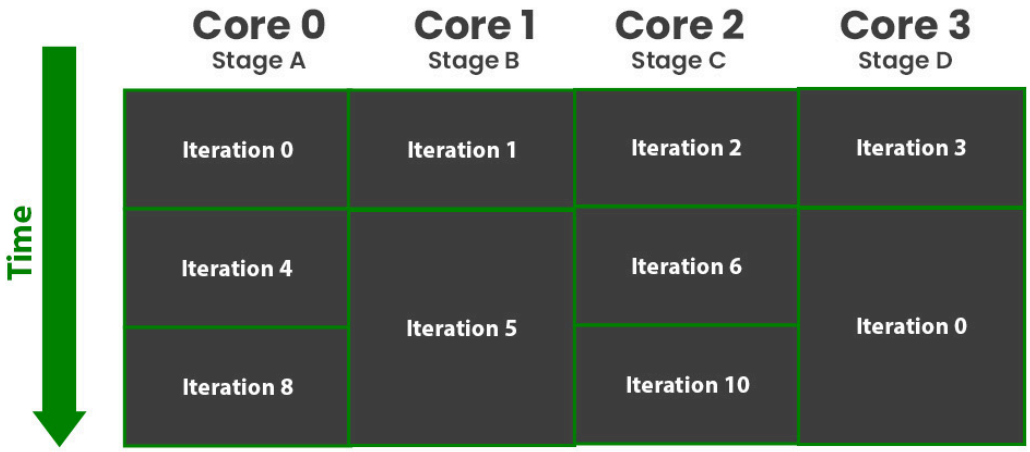
Metodologiile pentru paralelizarea buclelor includ mai multe tehnici bine definite, fiecare potrivită pentru diferite tipuri de dependențe între iterații și de resurse de calcul. Un rezumat al principalelor metode:

DISTRIBUTED Loop Parallelism

Această metodologie împarte o buclă cu dependențe în mai multe bucle separate, fiecare conținând doar instrucțiuni independente. Prin separarea instrucțiunilor independente în bucle distincte, acestea pot fi executate în paralel. Această metodă este eficientă pentru reducerea dependențelor în bucle și permite executarea concurentă a sarcinilor care nu interferează între ele.

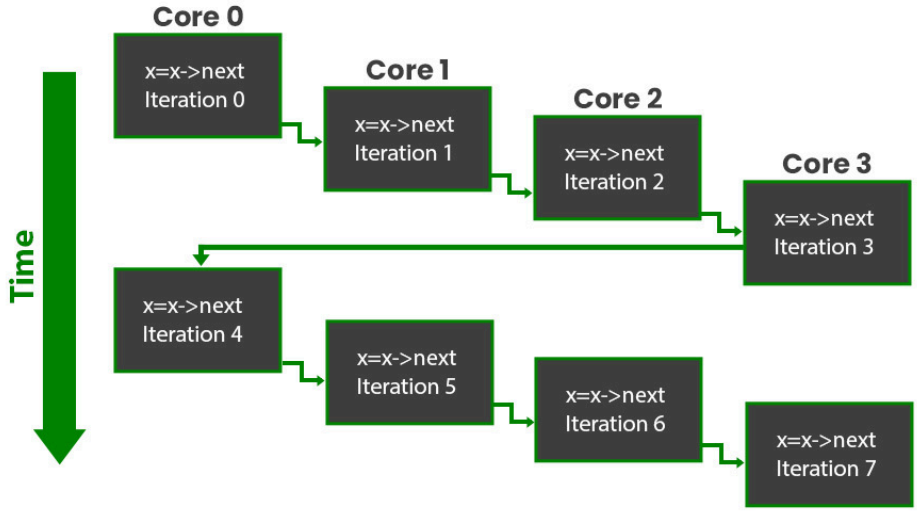
DOALL Parallelism

DOALL Parallelism presupune că toate iterațiile unei bucle pot fi executate independent, fără dependințe între ele. Astfel, fiecare iterație poate fi rulată simultan pe un procesor diferit. Aceasta este o metodă de paralelizare pură, care permite maximizarea vitezei de execuție, fiind aplicabilă în scenarii unde fiecare iterație este complet autonomă.



DOACROSS Parallelism

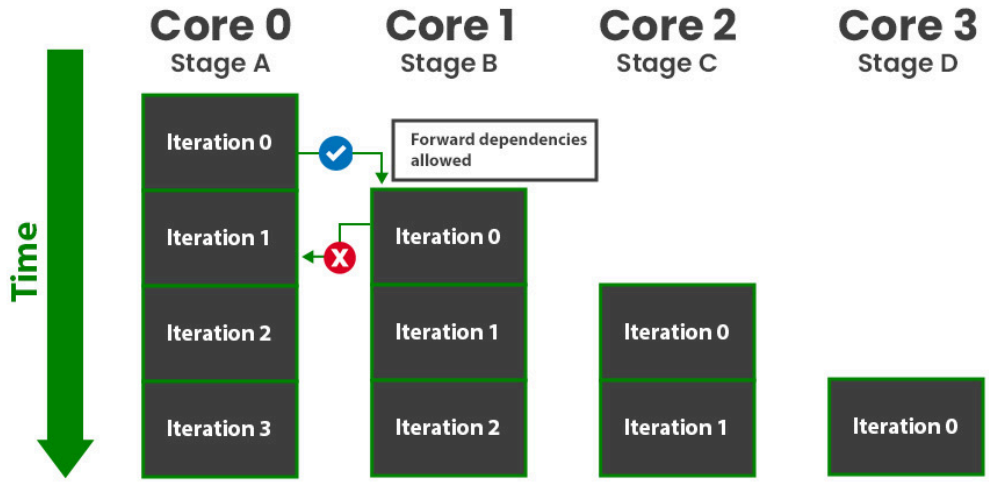
DOACROSS Parallelism permite paralelizarea parțială a buclelor care au dependințe între iterații, dar numai pentru anumite operații. Dependințele sunt sincronizate astfel încât secțiunile independente ale buclei să poată fi executate în paralel. Metoda implică sincronizări punctuale care coordonează iterațiile și evită conflictele de date, permițând totodată execuția unor calcule simultane.



DOPIPE Parallelism (Pipelined Parallelism)

DOPIPE Parallelism implementează paralelizarea în conducte (pipeline) pentru bucle cu dependințe. Fiecare iterație este împărțită în mai multe etape, iar fiecare etapă poate începe imediat ce datele necesare sunt disponibile de la etapa anterioară. Aceasta permite rularea

etapelor în paralel, în stil de „linie de asamblare”, unde procesele se succed sincronizat, optimizând timpul total de execuție.



3. Tehnici de Paralelizare a Buclelor

Paralelizarea buclelor este o tehnică de optimizare în programare, folosită pentru a distribui execuția unui cod iterativ (de obicei bucle) pe mai multe nuclee sau unități de procesare, astfel încât performanța să fie îmbunătățită.
Există o varietate de tehnici utilizate pentru paralelizarea buclelor, fiecare cu avantaje și limitări specifice. Printre cele mai utilizate tehnici se numără:

- Diviziunea Statică și Dinamică: În paralelizarea statică, fiecare fir de execuție (thread) primește un număr fix de iterații ale buclei la începutul execuției. Această abordare minimizează costurile de alocare, însă poate rezulta într-o încărcare neuniformă a resurselor. Paralelizarea dinamică, pe de altă parte, ajustează încărcarea pe parcursul execuției, în funcție de disponibilitatea resurselor, oferind un grad superior de echilibrare.

- OpenMP și MPI: OpenMP este o bibliotecă larg utilizată pentru paralelizarea buclelor în aplicații multi-core, prin simpla adăugare de directive care indică compilatorului care bucle să fie paralelizate. MPI (Message Passing Interface) este folosit pentru sisteme distribuite, în care buclele sunt paralelizate prin distribuirea calculelor între mai multe noduri dintr-o rețea.

- Vectorizare: Vectorizarea este o tehnică prin care se utilizează instrucțiuni SIMD (Single Instruction, Multiple Data) pentru procesarea simultană a mai multor elemente de date într-o singură iterație, oferind astfel un spor considerabil de performanță pentru buclele care operează pe structuri de date precum vectori sau matrice.

- Pipeline Parallelism (Paralelizare în Conducte): Această tehnică presupune împărțirea unei bucle complexe într-o serie de etape secvențiale, fiecare dintre acestea putând fi executată în paralel pe fire de execuție diferite.Este utilă în scenarii de prelucrare în flux (de exemplu, procesarea unui flux de date continuu), unde fiecare fir poate începe o nouă etapă înainte ca etapa anterioară să fie complet finalizată.

- Task Parallelism (Paralelizare pe Sarcini): Task parallelism se referă la împărțirea unei bucle în sarcini independente, unde fiecare sarcină este asignată unui fir de execuție diferit.Fiecare sarcină poate să conțină mai multe iterații ale buclei sau chiar bucle diferite, iar resursele sunt împărțite dinamic pentru a obține un echilibru optim în execuție.

4. Provocări în Paralelizarea Buclelor

Paralelizarea buclelor este o tehnică puternică pentru creșterea performanței aplicațiilor software, însă procesul de implementare aduce o serie de provocări care trebuie abordate pentru a obține rezultate optime. Aceste provocări se regăsesc la niveluri diferite, de la complexitatea algoritmilor și gestionarea resurselor, până la limitările hardware și software.
În continuare sunt detaliate cele mai comune provocări asociate cu paralelizarea buclelor:

1. Dependențele de Date

Una dintre cele mai mari provocări în paralelizarea buclelor este gestionarea dependențelor dintre iterații. Aceste dependențe pot fi de mai multe tipuri:

- RAW (Read After Write): O iterație citește o valoare care a fost scrisă de o altă iterație.
- WAR (Write After Read): O iterație scrie o valoare care a fost citită de o altă iterație.
- WAW (Write After Write): Două iterații scriu în aceeași locație de memorie.

Dependențele de date limitează nivelul de paralelizare posibil și necesită tehnici avansate, cum ar fi reorganizarea buclelor, utilizarea de tehnici DOACROSS sau sincronizarea strictă.

2. Supraîncărcarea Resurselor

Paralelizarea buclelor presupune distribuirea sarcinilor între mai multe fire de execuție sau unități de procesare. Dacă numărul de fire este prea mare sau alocarea resurselor nu este echilibrată, pot apărea probleme de supraîncărcare a procesorului sau memoriei, ceea ce reduce performanța globală.

3. Sincronizarea și Blocajele

Când mai multe fire de execuție accesează simultan resurse comune, devine esențială utilizarea mecanismelor de sincronizare pentru a evita conflictele. Totuși, sincronizarea excesivă poate introduce blocaje sau contention, reducând astfel avantajele paralelizării.

4. Granularitatea Sarcinilor

Stabilirea granularității corecte a sarcinilor este critică:

- Granularitate fină: Permite un nivel mai ridicat de paralelizare, dar introduce o încărcare ridicată pentru gestionarea firelor.
- Granularitate grosieră: Reduce supraîncărcarea gestionării firelor, dar limitează oportunitățile de paralelizare.

Alegerea unui echilibru optim între aceste două tipuri de granularitate depinde de specificul aplicației și al resurselor disponibile.

5. Probleme de Scalabilitate

Performanța unui algoritm paralel nu crește linear cu numărul de nuclee disponibile, din cauza constrângerilor impuse de Legea lui Amdahl. Proporția secvențială a codului și costurile asociate sincronizării limitează câștigurile de performanță.

Paralelizarea buclelor reprezintă o provocare majoră în optimizarea performanței aplicațiilor software, însă abordarea corectă a acestor provocări poate transforma un sistem lent într-unul extrem de performant. Strategiile adecvate și utilizarea unor instrumente precum OpenMP, MPI sau alte biblioteci avansate pot ajuta la depășirea obstacolelor și la obținerea unor beneficii semnificative.

5. Impactul Paralelizării asupra Performanței

Paralelizarea reprezintă o metodă de optimizare a proceselor prin divizarea sarcinilor în sub-sarcini care pot fi executate simultan, folosind mai multe unități de procesare. Această abordare are un impact semnificativ asupra performanței sistemelor de calcul, dar și asupra aplicațiilor software.

1. Creșterea vitezei de execuție

- Reducerea timpului de execuție: Prin împărțirea unui proces în mai multe fire de execuție sau procese paralele, timpul total de execuție poate fi redus considerabil. Acest lucru este reflectat în legea lui Amdahl, care explică cum partea serială a unui program limitează câștigul teoretic al paralelizării.
- Scalabilitate: Performanța se îmbunătățește odată cu creșterea numărului de nuclee de procesare, atâta timp cât sarcinile sunt suficient de bine împărțite.

2. Utilizarea eficientă a resurselor hardware

- Paralelizarea permite utilizarea mai eficientă a resurselor hardware disponibile, precum procesoarele multi-core, GPU-urile, sau clusterele de calcul.

- Tehnologii precum OpenMP, MPI sau CUDA sunt exemple de instrumente care ajută la implementarea proceselor paralele și la optimizarea resurselor hardware.

3. Probleme și limitări

- Dezavantajele sincronizării: Comunicarea și sincronizarea între firele de execuție sau procese pot introduce latențe suplimentare, care reduc eficiența paralelizării.
- Supraîncărcarea hardware: Dacă numărul de sarcini paralele depășește capacitatea hardware, performanța poate scădea din cauza competiției pentru resurse.
- Dificultăți în fragmentarea sarcinilor: Unele probleme nu sunt ușor de divizat în sub-probleme independente, ceea ce limitează beneficiile paralelizării.

4. Domenii de aplicare

- Aplicații de înaltă performanță: Modelarea fenomenelor fizice, analiza datelor mari și simulările științifice beneficiază enorm de pe urma paralelizării.
- Machine Learning și AI: Antrenarea modelelor de învățare automată este accelerată prin utilizarea paralelizării pe GPU-uri.
- Gaming și grafică: Randarea grafică în timp real și simulările fizice în jocuri utilizează intens paralelizarea.

6. Concluzie

Impactul paralelizării asupra performanței este evident în creșterea vitezei de procesare și utilizarea optimă a resurselor hardware. Totuși, maximizarea beneficiilor paralelizării depinde de factorii precum natura sarcinilor, arhitectura sistemului și tehnicile de implementare utilizate. O analiză atentă a raportului între costuri și beneficii este esențială pentru a determina dacă paralelizarea este soluția potrivită.

Paralelizarea buclelor este o tehnică esențială în optimizarea performanței aplicațiilor software, mai ales în contextul creșterii complexității algoritmilor și al volumului de date procesate.

Aceasta permite reducerea semnificativă a timpului de execuție prin utilizarea eficientă a resurselor hardware moderne, cum ar fi procesoarele multi-core sau sistemele distribuite.
Cu toate acestea, implementarea paralelizării vine cu un set de provocări notabile, incluzând gestionarea dependențelor de date, sincronizarea firelor de execuție, balansarea încărcării și evitarea erorilor precum deadlock-uri sau race conditions.

7. Demo Aplicație Practică

În domeniul calculului numeric și al programării aplicațiilor software, procesarea matricelor este o componentă esențială a multor algoritmi și sisteme complexe. O operație fundamentală în acest context este înmulțirea matricelor, care joacă un rol cheie în aplicații variate, de la procesarea imaginilor și învățarea automată, până la simulări fizice și calculul științific. Dimensiunile mari ale matricelor implică însă o complexitate computațională ridicată, ceea ce face necesară o abordare eficientă a acestei operații.

Acest proiect propune dezvoltarea unei aplicații în limbajul de programare Java care să implementeze înmulțirea matricelor de dimensiuni mari. În cadrul aplicației, dimensiunile matricelor vor fi generate aleator, permițând testarea în diverse scenarii. Scopul principal este de a evalua performanța în diferite cazuri de complexitate, comparând rezultatele în termeni de timp de execuție și resurse utilizate.

Obiective

1. Generare aleatorie a matricelor: Dimensiunile și valorile elementelor matricelor vor fi generate automat, oferind o gamă largă de cazuri de testare.
2. Implementare eficientă: Dezvoltarea unei metode pentru înmulțirea matricelor care să minimizeze timpul de procesare și să gestioneze optim resursele sistemului.
3. Testare și comparare: Evaluarea performanței aplicației în scenarii diverse, inclusiv înmulțirea matricelor de dimensiuni mici, medii și mari.
4. Analiză comparativă: Compararea timpilor de execuție pentru diferite dimensiuni ale matricelor și identificarea limitelor de performanță.

7.1 Algoritmi utilizați

Pentru a realiza o comparație detaliată și pentru a explora eficiența înmulțirii matricelor, au fost utilizați cinci algoritmi diferiți: algoritmul clasic de multiplicare, o versiune paralelă bazată pe ExecutorService din Java, algoritmul lui Strassen, o versiune paralelă bazată pe Parallel Stream și abordarea Fork-Join Parallel Matrix Multiplication.

1. Algoritmul clasic de multiplicare a matricelor

   Acest algoritm folosește metoda standard pentru înmulțirea matricelor, având o complexitate de $O(n^3)$. Fiecare element al matricei rezultat este calculat prin suma produselor elementelor corespunzătoare din liniile primei matrice și coloanele celei de-a doua matrice. Este ușor de implementat, dar devine ineficient pentru dimensiuni mari ale matricelor.

2. Algoritmul paralel bazat pe ExecutorService

Pentru a îmbunătăți performanța, înmulțirea matricelor poate fi optimizată prin paralelizare. Folosind ExecutorService din Java, calculele sunt distribuite pe mai multe fire de execuție (threads), ceea ce permite un timp de execuție redus pe sisteme multi-core. Această abordare menține complexitatea $O(n^3)$, dar exploatează puterea de procesare a mai multor nuclee pentru a reduce timpul de execuție.

3. Algoritmul lui Strassen

   Algoritmul Strassen este o tehnică de divizare și stăpânire (Divide and Conquer) pentru înmulțirea matricelor, care reduce numărul de operații necesare în comparație cu metoda clasică. În loc să efectueze n^3 operații, acesta le reduce la aproximativ O(n^{2.81}), utilizând o metodă recursivă de împărțire a matricelor în submatrici mai mici. Deși are avantaje teoretice, poate fi mai puțin eficient în practică pentru dimensiuni mici ale matricelor, din cauza costurilor suplimentare de recursivitate și gestionare a memoriei.

4. Fork-Join Parallel Matrix Multiplication

   Abordarea Fork-Join folosește ForkJoinPool din Java pentru a împărți calculul în sarcini mai mici care sunt executate în paralel. Această metodă se bazează pe recursivitate și divide-and-conquer.

   - Dacă dimensiunea matricei depășește un prag predefinit (ex. 64x64), calculul este împărțit în patru submatrici.
   - Aceste submatrici sunt procesate în paralel folosind ForkJoinPool, iar rezultatele sunt combinate pentru a obține matricea finală.
   - Se utilizează parallel streams pentru optimizarea calculelor, ceea ce reduce semnificativ timpul de execuție pe sisteme multi-core.

5. Stream Parallel Matrix Multiplication

   Această abordare utilizează Stream API din Java pentru a îmbunătăți performanța prin execuție paralelă. Se folosește metoda parallel() pentru a permite distribuirea calculelor pe mai multe fire de execuție.

   - Fiecare rând al matricei rezultat este calculat independent, utilizând un flux de date paralel.
   - Metoda permite o execuție mai rapidă fără a fi necesară recursivitatea sau gestionarea manuală a firelor de execuție.
   - Această abordare este eficientă, dar poate avea supraponderi dacă nu este utilizată corect, din cauza sincronizării necesare între firele de execuție.

Concluzie

Fiecare dintre aceste abordări are avantaje și dezavantaje:

- Metoda clasică este cea mai simplă, dar ineficientă pentru matrici mari.
- ExecutorService oferă o paralelizare ușor de implementat, dar nu reduce complexitatea algoritmică.
- Strassen este mai eficient teoretic, dar implică costuri suplimentare de recursivitate.
- Fork-Join îmbunătățește eficiența utilizând strategia divide-and-conquer, fiind potrivit pentru sisteme multi-core.
- Stream Parallel permite paralelizare fără gestionarea manuală a firelor, dar poate avea probleme de sincronizare.

7.1.1 Analiza algoritmilor utilizați pentru înmulțirea matricelor

1. Algoritmul clasic de multiplicare (Sequential)

Implementare: Este metoda tradițională, în care fiecare element din matricea rezultat este calculat iterând prin rândurile primei matrice și coloanele celei de-a doua.

Avantaje:

- Ușor de înțeles și implementat.
- Potrivit pentru matrici mici sau medii.

Dezavantaje:

- Ineficient pentru matrici mari datorită complexității O(n^3).

Cod relevant:

```
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        resultData[i][j] = 0;
        for (int k = 0; k < colsA; k++) {
            resultData[i][j] += matrixA.getData()[i][k] * matrixB.getData()[k][j];
        }
    }
}
```

2. Versiunea paralelă cu ExecutorService (Parallel)

Implementare: Împarte sarcina între mai multe fire de execuție, fiecare procesând un subset de rânduri din matricea rezultat.

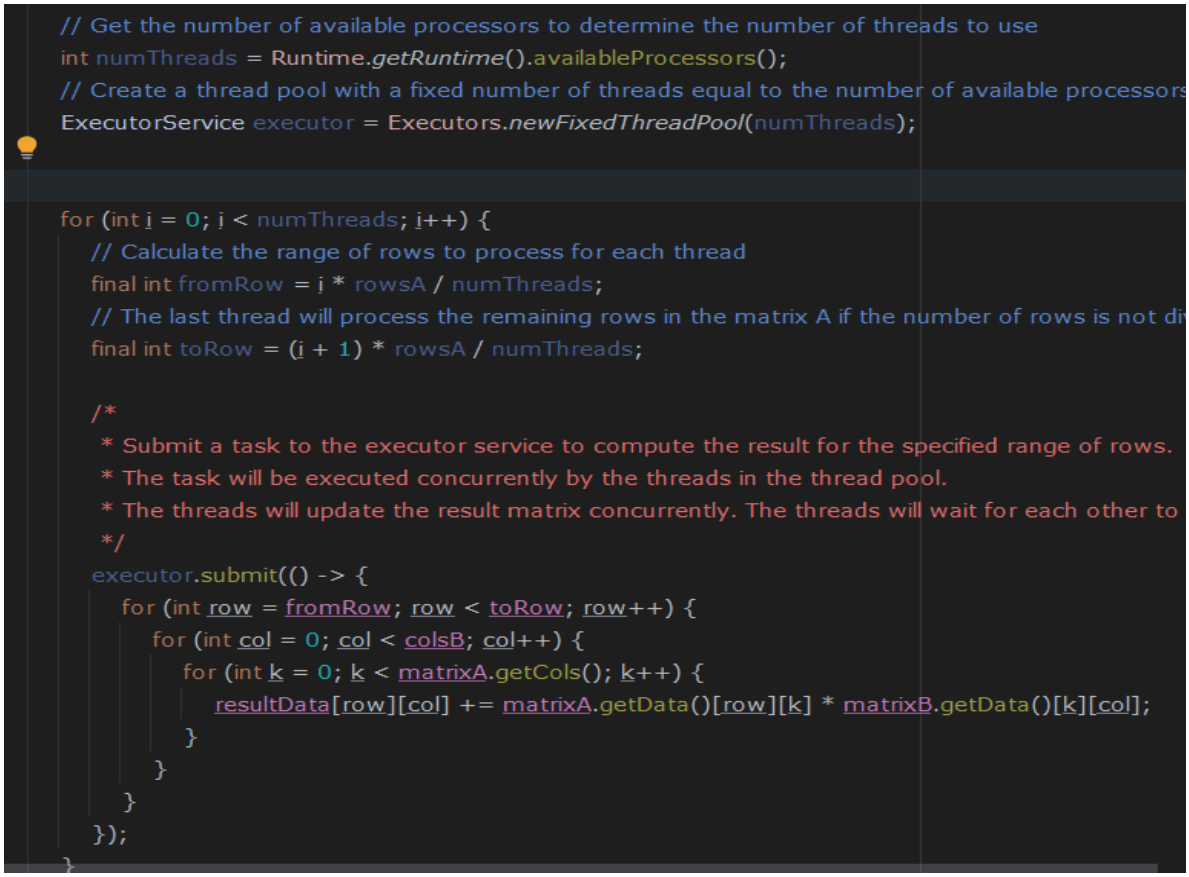
Avantaje:

- Utilizează mai eficient resursele unui procesor multicore.
- Scalabil pentru matrici mari.

Dezavantaje:

- Overhead-ul sincronizării firelor poate reduce eficiența pentru matrici mici.

Cod relevant:



```
// Get the number of available processors to determine the number of threads to use
int numThreads = Runtime.getRuntime().availableProcessors();
// Create a thread pool with a fixed number of threads equal to the number of available processors
ExecutorService executor = Executors.newFixedThreadPool(numThreads);


for (int i = 0; i < numThreads; i++) {
    // Calculate the range of rows to process for each thread
    final int fromRow = i * rowsA / numThreads;
    // The last thread will process the remaining rows in the matrix A if the number of rows is not di
    final int toRow = (i + 1) * rowsA / numThreads;

    /*
     * Submit a task to the executor service to compute the result for the specified range of rows.
     * The task will be executed concurrently by the threads in the thread pool.
     * The threads will update the result matrix concurrently. The threads will wait for each other to
     */
    executor.submit(() -> {
        for (int row = fromRow; row < toRow; row++) {
            for (int col = 0; col < colsB; col++) {
                for (int k = 0; k < matrixA.getCols(); k++) {
                    resultData[row][col] += matrixA.getData()[row][k] * matrixB.getData()[k][col];
                }
            }
        }
    });
}
```

3. Algoritmul lui Strassen

Implementare: Un algoritm de tip divide-et-impera care descompune matricile în submatrici mai mici și calculează doar 7 produse în loc de 8, reducând complexitatea teoretică la O(nlog7).

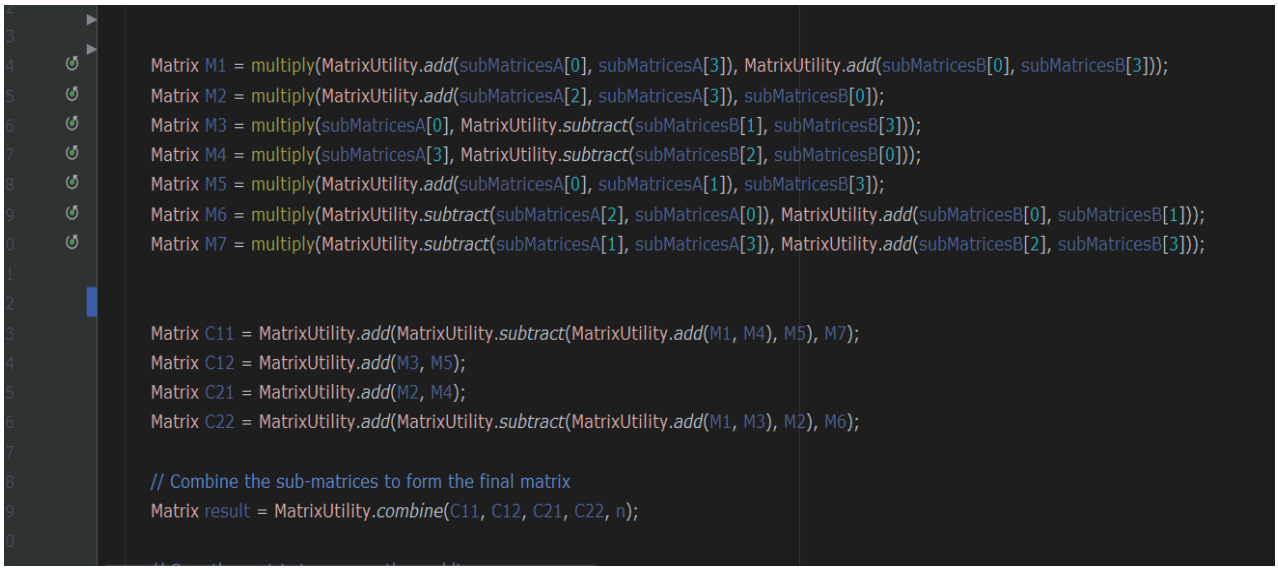
Avantaje:

- Mai rapid decât metoda clasică pentru matrici foarte mari.
- Reduce numărul total de operații.

Dezavantaje:

- Necesită ca dimensiunile matricelor să fie puteri ale lui 2 (se adaugă padding dacă nu este cazul).

Cod relevant:



```
Matrix M1 = multiply(MatrixUtility.add(subMatricesA[0], subMatricesA[3]), MatrixUtility.add(subMatricesB[0], subMatricesB[3]));
Matrix M2 = multiply(MatrixUtility.add(subMatricesA[2], subMatricesA[3]), subMatricesB[0]);
Matrix M3 = multiply(subMatricesA[0], MatrixUtility.subtract(subMatricesB[1], subMatricesB[3]));
Matrix M4 = multiply(subMatricesA[3], MatrixUtility.subtract(subMatricesB[2], subMatricesB[0]));
Matrix M5 = multiply(MatrixUtility.add(subMatricesA[0], subMatricesA[1]), subMatricesB[3]);
Matrix M6 = multiply(MatrixUtility.subtract(subMatricesA[2], subMatricesA[0]), MatrixUtility.add(subMatricesB[0], subMatricesB[1]));
Matrix M7 = multiply(MatrixUtility.subtract(subMatricesA[1], subMatricesA[3]), MatrixUtility.add(subMatricesB[2], subMatricesB[3]));


Matrix C11 = MatrixUtility.add(MatrixUtility.subtract(MatrixUtility.add(M1, M4), M5), M7);
Matrix C12 = MatrixUtility.add(M3, M5);
Matrix C21 = MatrixUtility.add(M2, M4);
Matrix C22 = MatrixUtility.add(MatrixUtility.subtract(MatrixUtility.add(M1, M3), M2), M6);

// Combine the sub-matrices to form the final matrix
Matrix result = MatrixUtility.combine(C11, C12, C21, C22, n);
```

4. Versiunea paralelă cu Fork Join

Implementare: Abordarea Fork-Join folosește ForkJoinPool din Java pentru a împărți calculul în sarcini mai mici care sunt executate în paralel. Această metodă se bazează pe recursivitate și divide-and-conquer.

- Dacă dimensiunea matricei depășește un prag predefinit (ex. 64x64), calculul este împărțit în patru submatrici.
- Aceste submatrici sunt procesate în paralel folosind ForkJoinPool, iar rezultatele sunt combinate pentru a obține matricea finală.
- Se utilizează parallel streams pentru optimizarea calculelor, ceea ce reduce semnificativ timpul de execuție pe sisteme multi-core.

Avantaje:

- Performanță îmbunătățită datorită execuției paralele.
- Scalabilitate bună pe procesoare multi-core.
- Reduce timpul de execuție în comparație cu metoda secvențială.

Dezavantaje:

- Crearea și gestionarea firelor de execuție poate introduce un overhead suplimentar.
- Ineficient pentru dimensiuni mici ale matricelor, unde timpul de divizare a sarcinilor depășește câștigul obținut prin paralelizare.
- Necesită mai multă memorie datorită subdivizării recursive a matricelor.

Cod relevant:

```java
@Override
protected Void compute() {  Complexity is 16 You must be kidding
    int rowCount = rowEnd - rowStart;
    int colCount = colEnd - colStart;

    if (rowCount * colCount <= THRESHOLD) {
        // Parallel processing using Java Streams
        IntStream.range(rowStart, rowEnd).parallel().forEach(i -> {
            IntStream.range(colStart, colEnd).parallel().forEach(j -> {
                int sum = 0;
                for (int k = 0; k < common; k++) {
                    sum += A.getValue(i, k) * B.getValue(k, j);
                }
                result.setValue(i, j, sum);
            });
        });
    } else {
        // Divide the task into four parallel tasks
        int midRow = (rowStart + rowEnd) / 2;
        int midCol = (colStart + colEnd) / 2;

        MatrixMultiplyTask topLeft = new MatrixMultiplyTask(A, B, result, rowStart, midRow, colStart, midCol, common);
        MatrixMultiplyTask topRight = new MatrixMultiplyTask(A, B, result, rowStart, midRow, midCol, colEnd, common);
        MatrixMultiplyTask bottomLeft = new MatrixMultiplyTask(A, B, result, midRow, rowEnd, colStart, midCol, common);
        MatrixMultiplyTask bottomRight = new MatrixMultiplyTask(A, B, result, midRow, rowEnd, midCol, colEnd, common);

        // Fork the tasks and wait for the results to be computed by
        // the threads in the pool (concurrently) using join() method to synchronize the tasks and get the results
        invokeAll(topLeft, topRight, bottomLeft, bottomRight);
    }
    return null;
}
```

5. Versiunea paralelă cu Parallel Stream

Implementare: Această abordare utilizează Stream API din Java pentru a îmbunătăți performanța prin execuție paralelă. Se folosește metoda parallel() pentru a permite distribuirea calculelor pe mai multe fire de execuție.

- Fiecare rând al matricei rezultat este calculat independent, utilizând un flux de date paralel.
- Metoda permite o execuție mai rapidă fără a fi necesară recursivitatea sau gestionarea manuală a firelor de execuție.
- Această abordare este eficientă, dar poate avea supraponderi dacă nu este utilizată corect, din cauza sincronizării necesare între firele de execuție.

Avantaje:

- Permite o paralelizare simplă și intuitivă folosind Stream API din Java.
- Execuție mai rapidă pe sisteme multi-core, fără necesitatea unei gestionări manuale a firelor de execuție.
- Cod mai curat și mai ușor de întreținut comparativ cu Fork-Join.
- Se integrează bine în pipeline-uri de procesare a datelor, oferind un mod eficient de a opera asupra stream-urilor mari de date.

Dezavantaje:

- Control redus asupra firelor de execuție și al managementului memoriei.
- Poate introduce supraponderi și probleme de sincronizare, în special pentru operații dependente între ele.
- Nu este întotdeauna mai rapid decât metodele secvențiale, mai ales pentru matrici de dimensiuni mici.
- Eficiența depinde de implementarea ForkJoinPool implicită, care poate să nu fie optimizată pentru toate cazurile.

Cod relevant:

```
public Matrix multiply(Matrix matrixA, Matrix matrixB) {  Complexity is 11 You must be kidding
    if (matrixA.getCols() != matrixB.getRows()) {
        throw new IllegalArgumentException("The number of columns in matrix A must equal the number of r
    }

    int rowsA = matrixA.getRows();
    int colsB = matrixB.getCols();
    int[][] resultData = new int[rowsA][colsB];

    /**
     * Use parallel streams to improve performance  by dividing the work into smaller tasks
     * that can be executed concurrently by multiple threads.
     * The parallel() method is used to create a parallel stream that will process the elements concurrently.
     */
    IntStream.range(0, rowsA).parallel().forEach(i -> {
        for (int j = 0; j < colsB; j++) {
            for (int k = 0; k < matrixA.getCols(); k++) {
                resultData[i][j] += matrixA.getData()[i][k] * matrixB.getData()[k][j];
            }
        }
    });

    return new Matrix(resultData, rowsA, colsB);
}
```

7.2 Rezultate obținute

7.2.2 Dataset de 1000 elemente

Graficul și datele afișează timpii obținuți pentru înmulțirea a două matrici de dimensiune 1000×1000).

Timpul pentru Algoritmul Strassen (191.0012 secunde):

- Pentru matrice foarte mari, cum ar fi 1000×1000, apare un cost suplimentar din cauza apelurilor recursive și a divizării/recombinării submatricilor. Acest lucru face ca algoritmul să fie semnificativ mai lent pe hardware care nu este optimizat pentru astfel de operații.

Timpul pentru Înmulțirea Clasică (2.564481 secunde):

- Această metodă folosește algoritmul standard O(n^3) pentru înmulțirea matricilor. Deși are o complexitate mai mare, funcționează bine pentru matrici dense și mari datorită optimizărilor hardware moderne (cum ar fi utilizarea eficientă a memoriei cache și execuția liniară a operațiilor).

Timpul pentru Înmulțirea Paralelă (0.689253 secunde):

- Înmulțirea paralelă utilizează mai multe nuclee ale procesorului pentru a realiza calculele simultan. Această metodă reduce semnificativ timpul de execuție, mai ales pentru matrice mari, datorită distribuției eficiente a sarcinii și utilizării arhitecturilor multi-core.

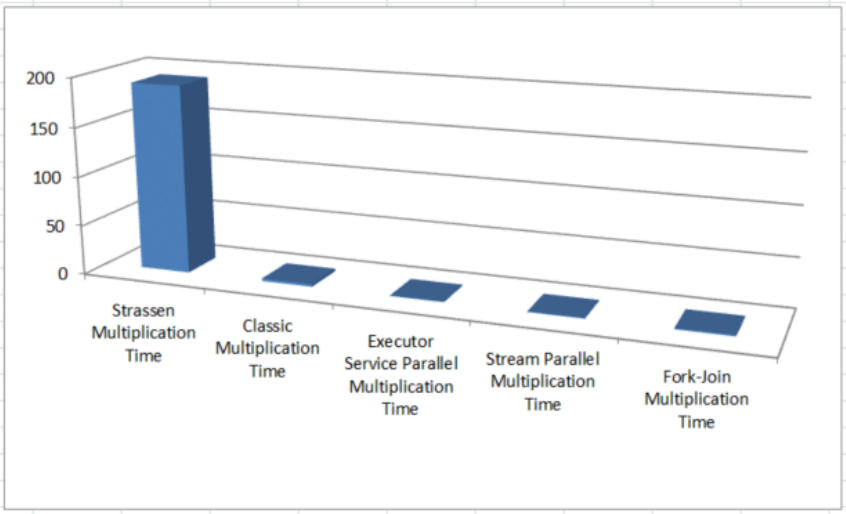
Timpul pentru metoda paralelă bazată pe Stream (0.737447 secunde):

- Utilizează Java Streams și .parallel() pentru a paraleliza calculele.
- Este puțin mai lentă decât metoda cu ExecutorService (0.737 secunde față de 0.689 secunde), deoarece:
- Paralelizarea la nivel de Stream-uri implică un cost de configurare mai mare.
- Concluzie: Performanța este bună, dar metodele dedicate de paralelizare (cum ar fi Fork-Join) sunt mai rapide pentru dimensiuni mari.

Timpul pentru metoda paralelă folosind Fork-Join (1.281482 secunde):

- Utilizează Fork-Join Framework pentru a împărți multiplicarea în sarcini mai mici și a le executa în paralel.
- Performanța este bună, dar mai lentă decât ExecutorService și Stream Parallel pentru matrice mari.
- Costurile de gestionare și sincronizare între subtârguri cresc la dimensiuni mari.
- Concluzie: Fork-Join este mai potrivit pentru matrice medii, dar mai puțin eficient pentru 1000x1000 în acest caz.

Strassen Multiplication Time	191.0012
Classic Multiplication Time	2.564481
Executor Service Parallel Multi	0.689253
Stream Parallel Multiplication T	0.737447
Fork-Join Multiplication Time	1.281482



7.2.3 Dataset de 100 elemente

Graficul și datele afișează timpii obținuți pentru înmulțirea a două matrici de dimensiune 100×100.

Timpul pentru Înmulțirea Paralelă (0.049439 secunde):

- Metoda paralelă folosește mai multe nuclee ale procesorului pentru a efectua operațiile simultan, ceea ce o face mai rapidă decât metodele tradiționale pentru dimensiuni mari.
- Totuși, pentru matrice mai mici, beneficiile paralelizării sunt mai reduse, deoarece costurile asociate inițializării și gestionării paralelismului devin semnificative.

Timpul pentru Înmulțirea Clasică (0.00804 secunde):

- Metoda clasică, deși are o complexitate mai mare O(n^3), este foarte rapidă pentru dimensiuni mici, cum este cazul unei matrici de 100×100.
- Simplitatea implementării și lipsa costurilor suplimentare (cum ar fi cele din algoritmii paraleli sau algoritmii recursivi) fac ca această metodă să fie cea mai eficientă pentru acest set de date.

Timpul pentru Algoritmul Strassen (0.610323 secunde):

- Algoritmul Strassen, deși are o complexitate mai mică decât O(n^3), implică operații suplimentare de divizare a matricei în submatrici și combinarea rezultatelor.
- Aceste costuri suplimentare sunt foarte semnificative pentru dimensiuni mici de matrice (100×100), ceea ce face ca Strassen să fie mai lent decât metodele clasice și paralele pentru acest caz.

Timpul pentru metoda paralelă bazată pe Stream-uri (0.025693 secunde):

Această metodă folosește Java Streams și .parallel() pentru a paraleliza înmulțirea.

Este mai rapidă decât ExecutorService deoarece:

- Stream-urile Java sunt optimizate pentru paralelizare pe mai multe fire.
- Costurile de gestionare a firelor sunt reduse.
- Totuși, este mai lentă decât metoda clasică, deoarece paralelizarea nu aduce beneficii semnificative pentru matrice de dimensiuni medii.

Timpul pentru metoda paralelă bazată pe Fork-Join (0.018241 secunde):

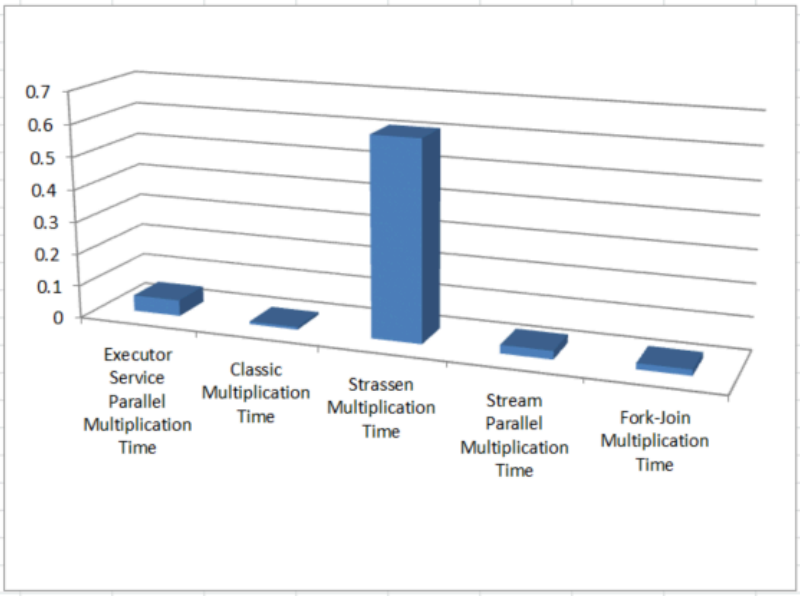
Metoda Fork-Join utilizează un mecanism mai eficient de împărțire a sarcinilor în subtârguri și executare paralelă.

Este mai rapidă decât Stream Parallel și ExecutorService, deoarece:

- Fork-Join minimizează costurile de sincronizare și utilizează eficient nucleele procesorului.
- Este ideală pentru matrice de dimensiuni medii-mari, cum sunt cele de 100x100.

Cu toate acestea, metoda clasică rămâne puțin mai rapidă pentru această dimensiune, datorită absenței costurilor de paralelizare.

Executor Service Parallel Multiplicati	0.049439
Classic Multiplication Time	0.00804
Strassen Multiplication Time	0.610323
Stream Parallel Multiplication Time	0.025693
Fork-Join Multiplication Time	0.018241



7.2.4 Dataset de 500 elemente

Graficul și datele afișează timpii obținuți pentru înmulțirea a două matrici de dimensiune 500×500.

Timpul pentru Înmulțirea Paralelă folosind Executor Service (0.094973 secunde):

- Metodă foarte eficientă pentru dimensiuni medii precum 500×500, deoarece poate distribui eficient calculele între multiplele nuclee ale procesorului.
- Timpul redus se datorează atât paralelizării, cât și eficienței arhitecturilor moderne multi-core, care sunt optimizate pentru calcule distribuite.

Timpul pentru Înmulțirea Clasică (0.225409 secunde):

- Mai lentă decât înmulțirea paralelă, dar încă eficientă pentru dimensiuni medii.
- Simplitatea metodei clasice și optimizările hardware (precum utilizarea memoriei cache) îi permit să funcționeze bine. Totuși, spre deosebire de înmulțirea paralelă, calculele sunt secvențiale, ceea ce explică timpul mai mare.

Timpul pentru Algoritmul Strassen (28.89773 secunde):

- Cel mai lent pentru acest caz, deoarece algoritmul Strassen necesită operații complexe de divizare și combinare a submatricilor.
- Dimensiunea de 500×500 nu este suficient de mare pentru ca beneficiile asimptotice ale algoritmului Strassen să depășească costurile suplimentare de procesare. Costurile de memorie și operațiile recursive afectează negativ performanța.

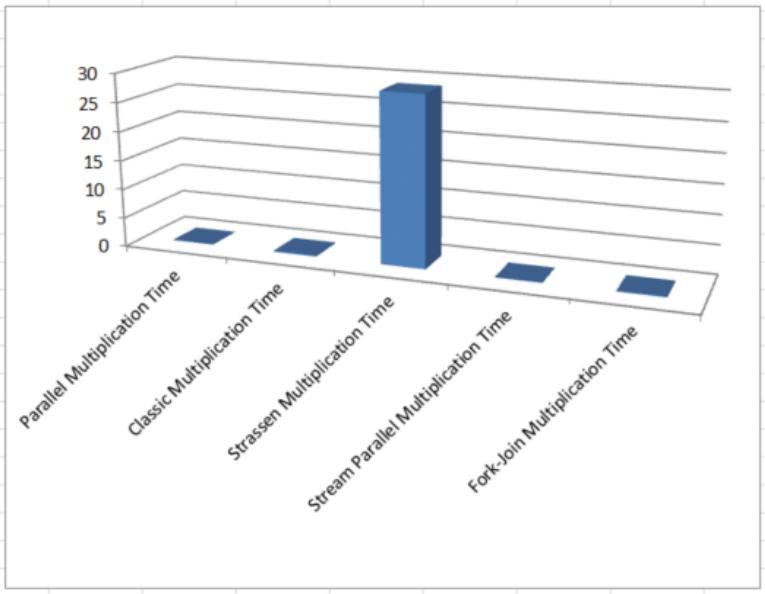
Timpul pentru Stream Parallel Multiplication (0.084535 secunde):

- Această metodă paralelă bazată pe fluxuri are cel mai mic timp de execuție, de 0.084535, fiind cea mai eficientă metodă pentru matrici de 500x500.
- Aceasta este o metodă modernă care utilizează fluxuri de date optimizate pentru execuție paralelă.
- Utilizează eficient paralelismul și minimizarea comunicării între procese.

Timpul pentru metoda paralelă Fork Join folosind fire de execuție (0.180898 secunde):

- Timpul de execuție este de 0.180898 secunde, fiind mai mare decât primele două metode.
- Deși utilizează o strategie de divizare și cucerire, overhead-ul datorat recursivității și subdivizării poate afecta performanța la această dimensiune a matricelor.

Parallel Multiplication Time	0.094973
Classic Multiplication Time	0.225409
Strassen Multiplication Time	28.89773
Stream Parallel Multiplication Time	0.084535
Fork-Join Multiplication Time	0.180898



7.2.5 Dataset de 50 elemente

Graficul și datele afișează timpii obținuți pentru înmulțirea a două matrici de dimensiune 50×50.

Timpul pentru Înmulțirea Paralelă folosind Executor Service (0.031101 secunde):

- Mai lentă decât metoda clasică, deși paralelizarea este utilă pentru dimensiuni mari de matrice.
- Pentru matrice mici, costurile asociate inițializării paralelizării și distribuirii sarcinilor între nuclee devin disproporționat de mari comparativ cu câștigurile obținute.
- De aceea, înmulțirea paralelă nu este ideală pentru matrice foarte mici.

Timpul pentru Înmulțirea Clasică (0.004004 secunde):

- Cea mai rapidă metodă pentru dimensiuni mici de matrice, datorită simplității algoritmului și lipsei costurilor suplimentare asociate cu gestionarea paralelizării sau divizarea matricei.
- În cazul matricilor mici, calculele secvențiale sunt rapide și bine optimizate de hardware-ul modern (prin utilizarea eficientă a memoriei cache).

Timpul pentru Algoritmul Strassen (0.095215 secunde):

- Cel mai lent pentru acest caz, deoarece algoritmul Strassen implică operații complexe de divizare și combinare a submatricilor.

- La dimensiuni foarte mici precum 50×50, aceste costuri suplimentare depășesc complet beneficiile teoretice ale algoritmului, făcându-l ineficient în comparație cu metoda clasică.

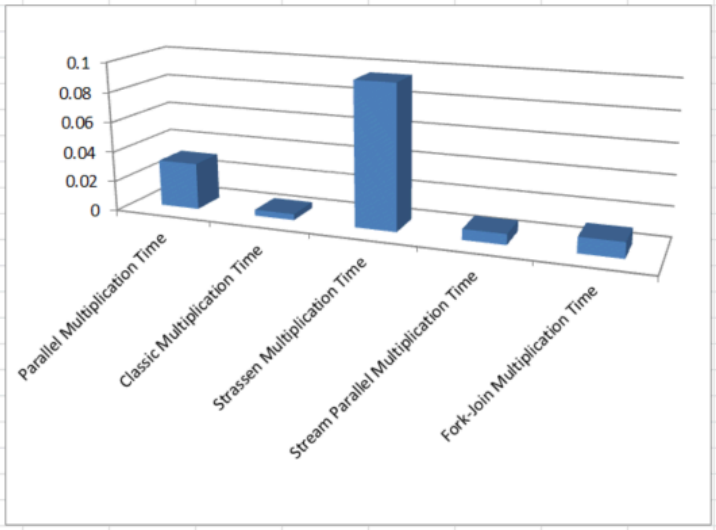
Timpul pentru pentru Înmulțirea Paralelă folosind Parallel Stream (0.007009 secunde):

- Este cea mai rapidă metodă, având un timp de 0.007009 secunde.
- Paralelizarea utilizând API-ul Stream funcționează foarte bine pe această dimensiune, deoarece overhead-ul este minim, iar execuția este eficientă.

Timpul pentru pentru Înmulțirea Paralelă folosind Fork-Join (0.010373 secunde) :

- Atinge un timp de 0.010373 secunde, fiind mai lent decât Stream Parallel, dar încă destul de rapid pentru dimensiuni mici.
- Overhead-ul creat de divizarea recursivă a sarcinilor afectează performanța pe matrici mici.

Parallel Multiplication Time	0.031101
Classic Multiplication Time	0.004004
Strassen Multiplication Time	0.095215
Stream Parallel Multiplication Time	0.007009
Fork-Join Multiplication Time	0.010373



7.2.6 Dataset de 10 elemente

Graficul și datele afișează timpii obținuți pentru înmulțirea a două matrici de dimensiune 10×10.

Timpul pentru Înmulțirea Paralelă (0.0289086 secunde):

- Mai lentă decât celelalte metode, deși este optimizată pentru dimensiuni mari. Pentru matrice de dimensiuni foarte mici, costurile de configurare a paralelizării (distribuirea sarcinilor între nuclee și sincronizarea) depășesc câștigurile.
- Procesarea paralelă nu este justificată pentru o sarcină atât de mică.

Timpul pentru Înmulțirea Clasică (0.0000714 secunde):

- Cea mai rapidă metodă în acest caz. Algoritmul clasic funcționează foarte bine pentru matrice mici, deoarece calculele sunt simple, secvențiale și foarte rapide.
- Hardware-ul modern optimizează execuția acestui tip de algoritm pentru dimensiuni reduse.

Timpul pentru Algoritmul Strassen (0.0052335 secunde):

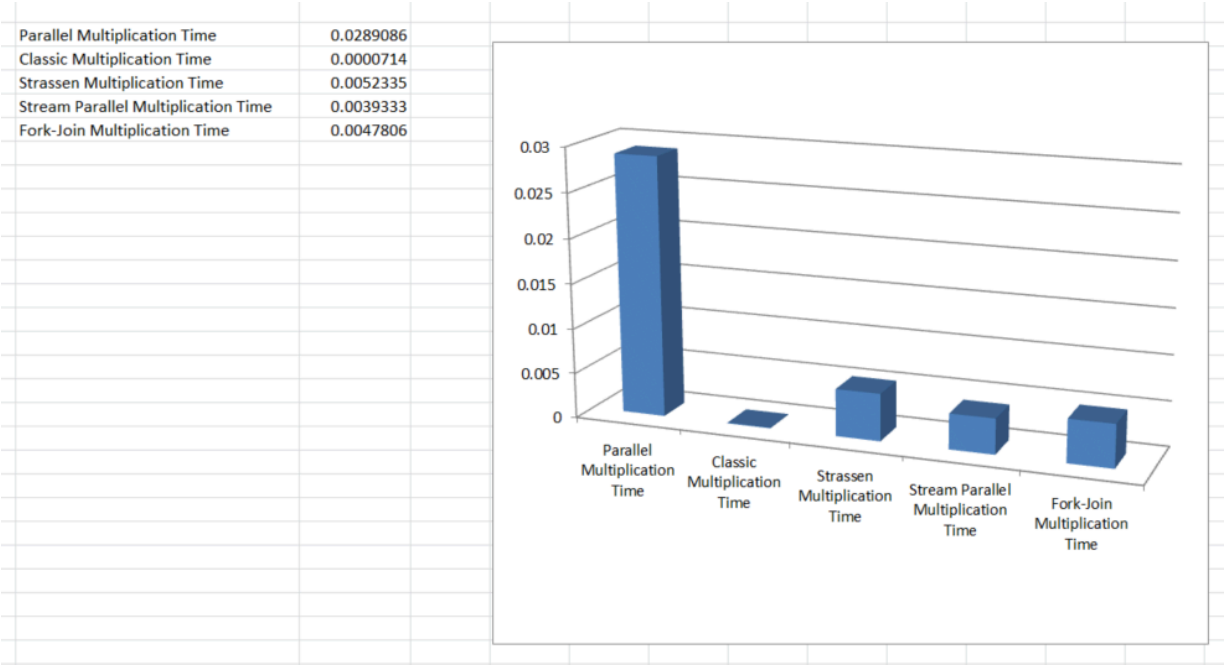
- Semnificativ mai lent decât metoda clasică, deoarece implică operații complexe de divizare și combinare a submatricilor.
- Pentru dimensiuni foarte mici (10×10), aceste operații suplimentare sunt disproporționat de costisitoare în comparație cu câștigurile teoretice.

Timpul pentru varianta Paralelă cu Stream-uri (0.0039333 secunde):

- Este mai rapidă decât ExecutorService și Strassen, deoarece costurile de paralelizare sunt mai mici, iar execuția pe mai multe fire este optimizată.
- Totuși, pentru matrice de dimensiuni mici, metoda clasică rămâne mai rapidă.

Timpul pentru varianta folosind Fork-Join (0.0047806 secunde):

- Această metodă folosește Fork-Join Framework pentru a împărți sarcina în sub-sarcini și a le executa în paralel.
- Performanța este comparabilă cu Stream Parallel, dar puțin mai lentă din cauza costurilor suplimentare pentru împărțirea și combinarea sarcinilor.



8. Structura aplicației

Structura aplicației este bine organizată, urmând o arhitectură modulară și separând diferitele responsabilități în pachete distincte.

1. Pachetul ace.ucv.approach

Acesta conține diferitele abordări utilizate pentru multiplicarea matricelor, fiecare într-un subpachet dedicat:

- fork_join: Implementări bazate pe Fork-Join Framework pentru paralelizare.
- parallel: Abordări bazate pe paralelizare clasică folosind ExecutorService.
- sequential: Implementări secvențiale pentru multiplicare simplă.
- strassen: Conține implementarea algoritmului Strassen pentru optimizarea multiplicării.
- stream: Folosește Java Streams pentru paralelizare.

2. Pachetul ace.ucv.model

Conține modelul de bază al aplicației:

- Matrix: Clasa care reprezintă structura matricii, incluzând datele și metodele de acces/manipulare.

3. Pachetul ace.ucv.service

Conține serviciile utilizate pentru generarea și manipularea matricelor:

- generator: Clase pentru generarea matricelor, cum ar fi RandomMatrixGenerator.
- parser: Manageri și parsere pentru dimensiuni și structuri de matrice (DimensionManager, MatrixParser).
- output: Clase pentru afișarea matricelor (MatrixPrinter) și înregistrarea metricilor de performanță (PerformanceMetricsRecorder).

4. Pachetul ace.ucv.utils

Include utilități și componente auxiliare:

- Constants: Conține valori statice pentru configurare.
- MatrixUtility: Metode ajutătoare pentru operații cu matrice.
- MatrixMultiplicationApp și MatrixMultiplicationView: Gestionarea interfeței și punctul principal de rulare a aplicației.

5. Clasa Principală (Main)

- Coordonează execuția aplicației, permițând utilizatorului să aleagă diferite metode de multiplicare și să analizeze performanța.

9. Rularea aplicației folosind interfața grafică

Interfața grafică (GUI) a aplicației **Matrix Multiplication App** este proiectată pentru a fi simplă și ușor de utilizat, oferind utilizatorilor o modalitate intuitivă de a efectua multiplicarea matricelor și de a analiza rezultatele.

1. Secțiunea de Introducere a Datelor

Aceasta este situată în partea de sus a aplicației și permite utilizatorilor să definească dimensiunile matricelor:

- **Rows Min/Max**: Câmpuri pentru introducerea numărului minim și maxim de rânduri ale matricelor.
- **Cols Min/Max**: Câmpuri pentru introducerea numărului minim și maxim de coloane ale matricelor.
- Aceste câmpuri oferă flexibilitate, permițând definirea matricelor de dimensiuni diferite, de la mici la mari.

2. Secțiunea de Alegere a Algoritmului

- Utilizatorii pot selecta algoritmul dorit pentru multiplicarea matricelor folosind **butoanele radio**. Opțiunile includ:
    1. **Sequential Approach**: Metoda tradițională secvențială, cu un singur fir de execuție.
    2. **Parallel Approach**: Calcul paralel utilizând fire multiple pentru a accelera procesul.
    3. **Strassen Approach**: Algoritm optimizat bazat pe metoda divide-et-impera.
    4. **Parallel Stream Approach**: Utilizează Java Streams pentru execuție paralelă.
    5. **Fork-Join Approach**: Folosește Fork-Join Framework pentru paralelism eficient.

3. Butonul "Start Multiplication"

- După completarea dimensiunilor matricelor și selectarea algoritmului, utilizatorii pot apăsa pe butonul **Start Multiplication** pentru a iniția calculul.
- După apăsarea butonului, rezultatele și timpul de execuție sunt afișate în secțiunea de output.

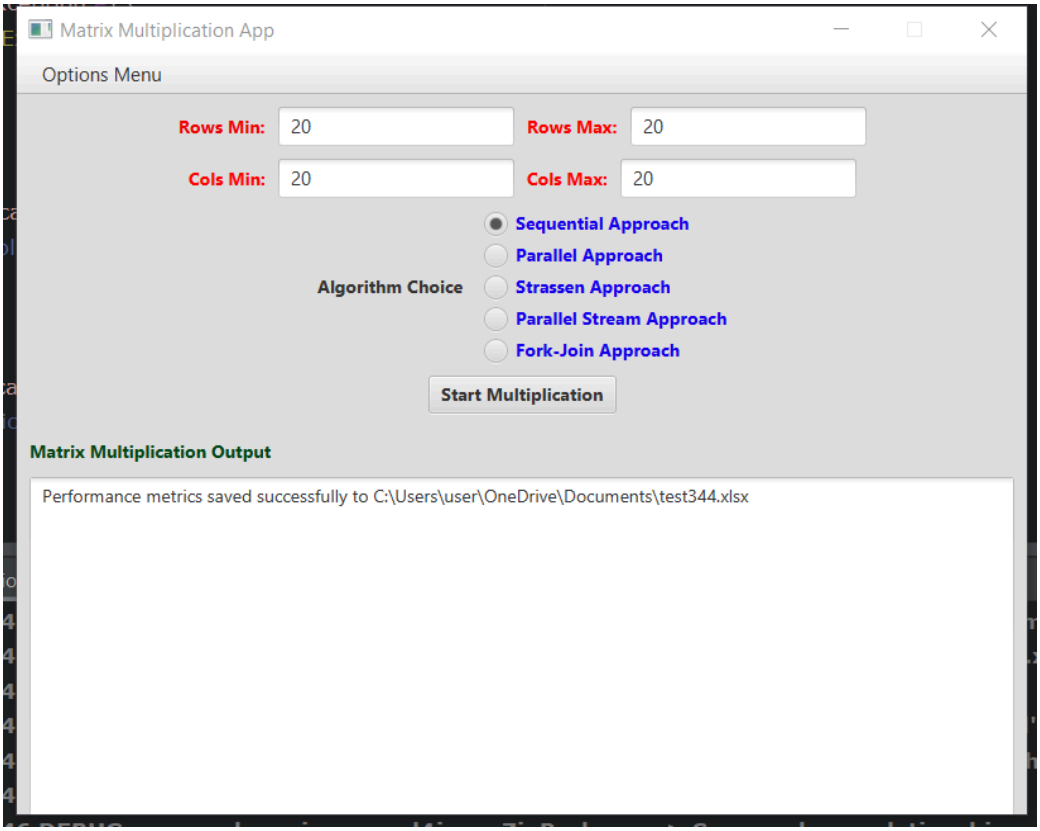
4. Secțiunea de Output

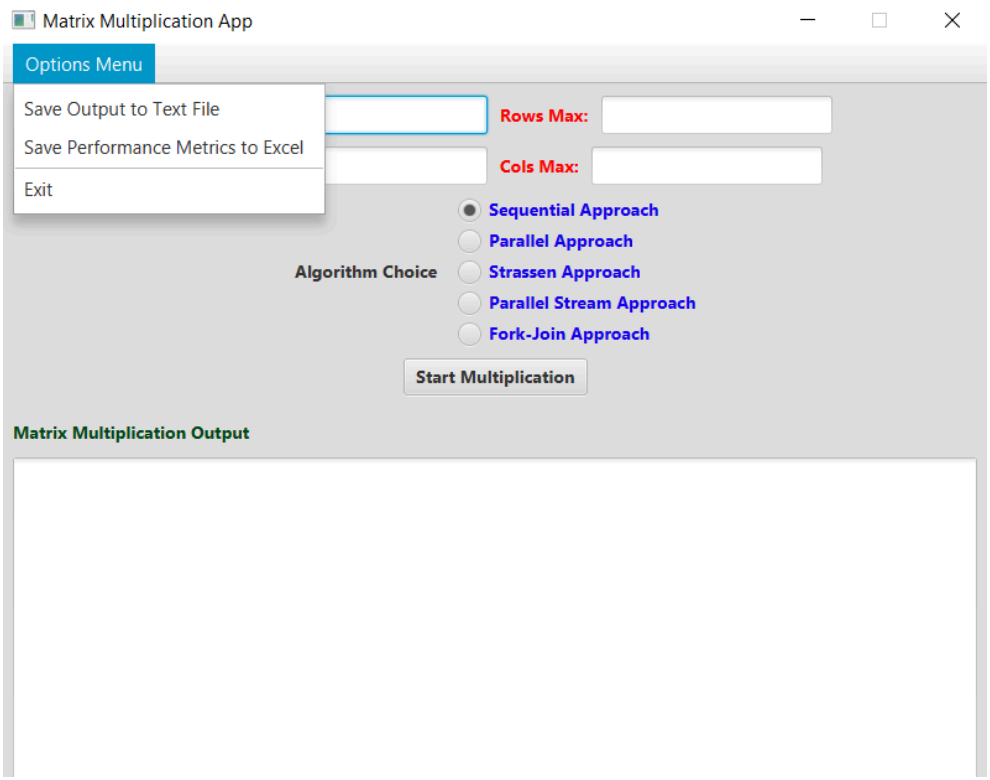
- În partea de jos a aplicației, această secțiune afișează:
    - Matricele generate.
    - Rezultatul multiplicării matricelor.
    - Timpul de execuție al algoritmului selectat.
- Este utilă pentru analizarea performanței fiecărui algoritm.

5. Meniul "Options Menu"

Situat în partea de sus, meniul permite utilizatorilor să acceseze funcții suplimentare:

- **Save Output to Text File**: Salvează rezultatele multiplicării într-un fișier .txt.
- **Save Performance Metrics to Excel**: Salvează timpii de execuție și alte date de performanță într-un fișier Excel.
- **Exit**: Închide aplicația.





10. Bibliografie

1. “Loop-Level Parallelism.” *Wikipedia*, 30 Jan. 2023, en.wikipedia.org/wiki/Loop-level_parallelism.

2. “Loop Parallelism | Our Pattern Language.” *Berkeley.edu*, 2024, patterns.eecs.berkeley.edu/?page_id=562. Accessed 9 Nov. 2024.

3. “Loop Level Parallelism in Computer Architecture.” *GeeksforGeeks*, 20 June 2022, www.geeksforgeeks.org/loop-level-parallelism-in-computer-architecture/.

4.“StrassenAlgorithm.”*Wikipedia*, 4 May 2022,en.wikipedia.org/wiki/Strassen_algorithm.

5. “Threads, ThreadPools and Executors - Multi Thread Processing in Java | SoftwareMill.”*SoftwareMill*,2024, softwaremill.com/threadpools-executors-and-java/. Accessed 6 Jan. 2025.

6. “Matrix Multiplication Algorithm.” *Wikipedia*, 27 Sept. 2021, en.wikipedia.org/wiki/Matrix_multiplication_algorithm.

7. “Parallelism (the Java™ Tutorials > Collections > Aggregate Operations).” *Docs.oracle.com*, docs.oracle.com/javase/tutorial/collections/streams/parallelism.html.

8. baeldung. “Guide to the Fork/Join Framework in Java | Baeldung.” *Www.baeldung.com*, 25 Apr. 2016, www.baeldung.com/java-fork-join.

9. “What Is Java Parallel Streams?” *GeeksforGeeks*, 26 Aug. 2020, www.geeksforgeeks.org/what-is-java-parallel-streams/.

10. “Java ExecutorService - Javatpoint.” *Www.javatpoint.com*, www.javatpoint.com/java-executorservice.

11. Hardware

- Windows 10 Pro
- 16 GB RAM
- Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz
- 1 TB SSD
- NVIDIA GeForce RTX 2060
- 16 processors
- 8 cores per processor