FAKULTET TEHNIČKIH NAUKA Novi Sad

Master Računarski sistemi visokih performansi

SEMINARKSI RAD iz predmeta:

Paralelne i distribuirane arhitekture i jezici

Na temu:

POREĐENJE RUST I JAVA PROGRAMSKIH JEZIKA NA PRIMERU IMPLEMENTACIJE ALGORITAMA ZA MNOŽENJE MATRICA

Ognjen Zeković

E241/2025

Novi Sad, Januar 2026.

1. UVOD

Izbor programskog jezika za numerički intenzivne aplikacije predstavlja kritičnu odluku koja utiče na performanse, sigurnost, ali i lakoću i brzinu razvoja softvera. Trenutno se u softverskom svetu svega par jezika izdvajaju kao reprezentativni primeri različitih filozofija, od kojih je *Rust* predstavljen kao moderni jezik koji omogućava *zero-cost* apstrakcije (pisanje ekspresivnog koda visokog nivoa bez žrtvovanja performansi) i *memory safety* (zabrana pristupa memoriji na nedozvoljen način) bez *garbage collector*-a (automatsko oslobađanje memorije), a *Java* kao zreli, provereni jezik u industriji sa JIT kompajlerom i automatskim upravljanjem memorijom.

Rust, lansiran 2010. godine, dizajniran je da reši fundamentalne probleme C/C++ jezika, pre svega one u vezi sa sigurnošću memorije, bez žrtvovanja performansi. Njegov sistem vlasništva (*ownership*) i pozajmljivanja (*borrowing*) omogućava kompajleru da detektuje probleme poput trke za resursima(*data races)*, korišćenje resursa nakon oslobađanja (*use-after-free*) i ,,curenja'' bafera (*buffer overflow*) tokom kompajliranja, eliminišući čitavu klasu *runtime* grešaka.

Java, sa druge strane, predstavlja stabilnu platformu koja je preko dve decenije dominantna u *enterprise* okruženjima. Njena ,,Napiši jednom, pokreni svuda" filozofija, automatsko upravljanje memorijom i bogat ekosistem biblioteka čine je prirodnim izborom za brz razvoj poslovnih aplikacija. *Just-In-Time* (JIT) kompajler implementiran u Java Virtuelnoj Mašini (JVM) kontinuirano analizira i optimizuje kod tokom izvršavanja, što često rezultuje performansama bliskim *native compiled* jezicima (direktno se kompajliraju u mašinski kod bez potrebe za VM).

MOTIVACIJA

Množenje matrica predstavlja fundamentalnu operaciju u linearnoj algebri sa širokom primenom u naučnim proračunima, mašinskom učenju i kompjuterskoj grafici. Različiti algoritmi za množenje matrica različito podržavaju paralelizaciju, što ih čini idealnim kandidatima za procenu paralelnih mogućnosti programskih jezika.

CILJ RADA

Cilj ovog rada je empirijsko poređenje Rust i Java jezika kroz implementaciju tri algoritma za množenje matrica:

1. Standardni iterativni algoritam – Bazni pristup sa tri ugneždene petlje i složenošću $O(n^3)$
2. *Divide-and-Conquer* pristup – Rekurzivna dekompozicija problema na manje podprobleme
3. *Strassen* algoritam – Brži pristup sa složenošću $O(n^{2.807})$

Poređenje jezika je pošten za oba jezika jer: obe implementacije koriste identične strukture podataka (ravan 1D array), imaju iste optimizacije (za *cache* u *for* petljama), i ekvivalentne strategije paralelizacije (*task-based*).

Rad je rađen u dve faze:
U prvoj fazi sam implementirao algoritme koristeći niz nizova za predstavljanje matrice, gde je u *Rust*-u to Vec<Vec<f64>> i u *Java* je double[][]. Ova faza je otkrila značajne razlike u performansama, jer je *Java* bila brža 2-3 puta, što je motivisalo detaljniju analizu.
U drugoj fazi sam promenio strukture podataka da budu identične u oba jezika: ravan 1D vektor sa eksplicitnim indeksiranjem. Dodatno sam implpementirao *zero-copy view* abstrakcije koje omogućavaju pristup podmatricama bez kopiranja podataka, kao i *cache*-optimizovane algoritme sa i-k-j redosledom prolaska kroz petlju.

Rad je organizovan tako da drugo poglavlje opisuje teorijsku osnovu algoritama, treće analizira ključne razlike između Rust i Java jezika, zatim je detaljno opisana implementacija svih algoritama u četvrtom poglavlju, u petom su predstavljeni eksperimentalni rezultati nakon čega slede detaljna analiza i zaključak.

2. TEORIJSKA OSNOVA

Ovo poglavlje opisuje fundamentalne algoritme za množenje matrica i koncepte paralelizacije koji su neophodni za razumevanje implementacija i rezultata predstavljenih u ovom radu.

Algoritmi za množenje matrica

Množenje dve kvadratne matrice **A** i **B** dimenzija n×n proizvodi rezultujuću matricu **C** gde je svaki element definisan kao: $C[i][j] = \Sigma(k=0 \text{ to } n-1)\ A[i][k] \times B[k][j]$. Za izračunavanje svakog elementa potrebno je n množenja i n-1 sabiranja, pa je složenost $O(n^3)$

2.1. Standardni iterativni algoritam

Najjednostavniji i najintuitivniji pristup implementira direktnu definiciju množenja matrica

```
for i ← 0 to n-1 do
  for j ← 0 to n-1 do
    C[i][j] ← 0
    for k ← 0 to n-1 do
      C[i][j] ← C[i][j] + A[i][k] × B[k][j]
    end for
  end for
end for
return C
```

Standardna implementacija sa i-j-k redosledom petlji ima lošu cache lokalnost jer pristup elementima `B[k][j]` "skače" kroz kolone matrice B. Premeštanjem petlji u i-k-j redosled poboljšavamo performanse

2.2. Divide-and-Conquer algoritam

Zasniva se na particiji matrica na manje blokove i rekurzivnoj primeni množenja:

Particionisanje matrice C:

$C11 = A11 \times B11 + A12 \times B21$

$C12 = A11 \times B12 + A12 \times B22$

$C21 = A21 \times B11 + A22 \times B21$

$C22 = A21 \times B12 + A22 \times B22$

,a zatim kombinovanje dobijenih vrednosti nazad u matricu.

Sama složenost se ne menja ali se omogućava nezavisna paralelizacija blokova.

2.3. Strassen algoritam

Umesto direktnog računanja C11, C12, C21, C22 (što zahteva 8 množenja), Strassen definiše 7 pomoćnih proizvoda M1-M7:

M1 = (A11 + A22) × (B11 + B22)

M2 = (A21 + A22) × B11

M3 = A11 × (B12 - B22)

M4 = A22 × (B21 - B11)

M5 = (A11 + A12) × B22

M6 = (A21 - A11) × (B11 + B12)

M7 = (A12 - A22) × (B21 + B22)

, a zatim kombinovanjem dobijamo matricu C:
C11 = M1 + M4 - M5 + M7

C12 = M3 + M5

C21 = M2 + M4

C22 = M1 - M2 + M3 + M6

3. RUST I JAVA - KLJUČNE RAZLIKE

Ovo poglavlje fokusira se na dva ključna aspekta: upravljanje memorijom i modele konkurentnosti.

3.1. Upravljanje memorijom

RUST

Rust uvodi jedinstveni sistem vlasništva koji se zasniva na pravilima:

1. Može postojati tačno jedan vlasnik u datom trenutku

2. Kada vlasnik izađe iz *scope*-a, vrednost se automatski dealocira

Umesto prenošenja vlasništva, Rust omogućava pozajmljivanje (*borrowing)* čime obezbeđuje privremeni pristup podacima kroz reference, poštujući sledeća pravila:

1. Višestruko nepromenljivo (*immutable*) pozajmljivanje je dozvoljeno
2. Može postojati samo jedno promenljivo (*mutable*) pozajmljivanje u trenutku
3. Ne mogu da koegzistiraju promenljivo i nepromenljivo pozajmljivanje

Rust koristi i anotacije životnog ciklusa (*lifetime*) čime garantuje da reference ne žive duže od podataka na koje pokazuju, u kodu je korišćen na sledeći način:

```
pub struct MatrixView<'a> {
data: &'a [f64], // lifetime 'a: view živi koliko i originalni podatak matrice
rows: usize,
cols: usize,
//...
}
pub fn subview(&self, row_start: usize, row_end: usize, col_start: usize, col_end: usize) -> MatrixView<'a> {
MatrixView { data: self.data, // Isti data slice, nema kopiranja
rows: row_end - row_start,
row_offset: self.row_offset + row_start,
// ... }
 }
```

JAVA

Java koristi automatsko upravljanje memorijom kroz *garbage collector* (GC) koji periodično identifikuje i oslobađa nekorišćene objekte.

3.2. Konkurentnost i paralelizam

RUST

Rust pristup konkurentnosti zasniva se na principu neustrašivoj (*fearless)* konkurentosti, odnosno trka za podacima je garantovano onemogućena samim kompajlerom.

U rustu je korišćena Rayon biblioteka koja nudi jednostavan paralelizam:

```
use rayon::prelude::*;
```

```
// Sekvencijalno:
let results: Vec<_> = (0..8)
.map(|i| expensive_computation(i))
.collect();
// Paralelno, samo se zameni .iter() sa .par_iter():
let results: Vec<_> = (0..8)
.par_iter() // ← Paralelni iterator
.map(|i| expensive_computation(i))
.collect();
```

Primer iz projekat je paralelizacija 8 nezavisnih množenja *Divide and conquer* algoritma:

```
let products: Vec<Matrix> = vec![
(&a11, &b11), (&a12, &b21), (&a11, &b12),
(&a12, &b22), (&a21, &b11), (&a22, &b21),
(&a21, &b12), (&a22, &b22), ]
.par_iter() // Work-stealing thread pool
.map(|(x, y)| multiply(x, y)) // Svaki par se množi paralelno
.collect();
```

, gde *work-stealing pool* znači da prazne niti ,,kradu" zadatke iz tuđih redova zadataka.

JAVA

Java koristi *ForkJoinPool* za paralelizam zasnovan na zadacima, koji je deo *Java Concurrency framework*-a od verzije 7. Paralelni zadatak se definiše nasleđivanjem *RecursiveTask* klase:

```
class MultiplyTask extends RecursiveTask<Matrix> {
private final MatrixView a, b;
public MultiplyTask(MatrixView a, MatrixView b)
 this.a = a; this.b = b;
}
@Override protected Matrix compute() {
if (a.getRows() <= THRESHOLD) {
return multiplyStandard(a, b); // Bazni slučaj
}
// Kreiranje podzadataka
MultiplyTask task1 = new MultiplyTask(a11, b11);
MultiplyTask task2 = new MultiplyTask(a12, b21);
// ... još 6 zadataka
// Fork (asinhrono pokretanje)
task2.fork();
task3.fork();
// ...
// Compute (sinhrono izvršavanje)
```

```
Matrix p1 = task1.compute();

// Join (čekanje rezultata)

Matrix p2 = task2.join();

Matrix p3 = task3.join();

// ...

return combine(p1, p2, ...);

}}
```

Ključna razlika je što *Java* ne garantuje *thread safety* za vreme kompajliranja. Naravno ovim pristupom kod Jave dajemo više kontrole nad paralelizacijom ali i proizvodimo više koda.

3.3. Performanse

Krajnja razlika je i način kompajliranja koda gde Rust koristi AOT – Ahead of time, kod se optimizuje za prosečan slučaj i sve optimizacije su statične, nastaju prilikom kompajliranja i ostaju iste tokom rada programa.

Dok s druge strane, Java koristi JIT – Just in time kompajliranje koje izvorni kod prvo pretvara u *bytecode* i optimizuje specifično za svaku situaciju, recimo nakon određenog broja korišćenja neke funkcije pokušava da nađe šablon koji će da koristi kako bi smanjio korišćenu memoriju za dati zadatak. Dakle Java ima ,,vreme zagrevanja" nakon čega postaje brža u izvršavanju određenih zadataka.

Dakle Rust plaća unapred vremenom kompajliranja za maksimalne performanse dok Java optimizuje tokom izrvšavanja.

4. IMPLEMENTACIJA

4.1. Početni pristup

U prvoj iteraciji koristio sam 2D matrice:

Rust - Vec<Vec<f64>>:

```
type Matrix = Vec<Vec<f64>>;
```

Java - double[][]:

```
public class Matrix {
    private double[][] data;
```

Međutim ovako je svaki red je odvojena alokacija.

Dobio sam iznenađujuće rezultate:

Paralelizovani *Strassen,* za matricu 2048x2048, se u Javi izvršio za 1.763 sekundi dok je u rastu trebalo više nego dva puta više, čak 3.760 sekundi.

4.2. Optimizovani pristup

Zatim sam redizajnirao strukture podataka na 1D:

```
pub struct Matrix {
data: Vec<f64>,
rows: usize, cols: usize,
}
```

I dodao indeksiranje:

```
fn index(&self, row: usize, col: usize) -> usize {
row * self.cols + col
}
pub fn get(&self, row: usize, col: usize) -> f64 {
self.data[self.index(row, col)]
}
```

Zatim sam u Javi radi poštenog poređenja uradio isto:

```
public class Matrix {
private final double[] data;
private final int rows;
```

```
private final int cols;

private int index(int row, int col) {

return row * cols + col;

}

public double get(int row, int col) {

return data[index(row, col)];

}
```

Dodata je i MatrixView struktura za eliminaciju kopiranja pri kreiranju podmatrica:

RUST

```rust
pub struct MatrixView<'a> {
    data: &'a [f64],            // pozajmljivanje
    rows: usize,
    cols: usize,
    row_offset: usize,
    col_offset: usize,
    parent_cols: usize,         // za indeksiranje
}

impl<'a> MatrixView<'a> { // ne može da preživi originalnu matricu

    pub fn from_matrix(matrix: &'a Matrix) -> Self {
        MatrixView {
            data: &matrix.data,
            rows: matrix.rows,
            cols: matrix.cols,
            row_offset: 0,
            col_offset: 0,
            parent_cols: matrix.cols,
        }
```

```
    }

// Kreiranje subview-a bez kopiranja
    pub fn subview(&self, row_start: usize, row_end: usize,
                   col_start: usize, col_end: usize) -> MatrixView<'a>
{
        MatrixView {
            data: self.data,        // Isti data slice
            rows: row_end - row_start,
            cols: col_end - col_start,
            row_offset: self.row_offset + row_start,
            col_offset: self.col_offset + col_start,
            parent_cols: self.parent_cols,
        }
    }
}
```

Java ekvivalentna:

```
public class MatrixView {
    private final double[] data;  // referenca
    private final int rows;
    private final int cols;
    private final int rowOffset;
    private final int colOffset;
    private final int parentCols;

    public static MatrixView fromMatrix(Matrix matrix) {
        return new MatrixView(
            matrix.getData(),
            matrix.getRows(),
            matrix.getCols(),
            0, 0,
            matrix.getCols()
        );
    }

    // Subview bez kopiranja
    public MatrixView subview(int rowStart, int rowEnd,
```

```
                                int colStart, int colEnd) {
        return new MatrixView(
            this.data,
            rowEnd - rowStart,
            colEnd - colStart,
            this.rowOffset + rowStart,
            this.colOffset + colStart,
            this.parentCols
        );
    }
}
```

U Javi, *GC* automatski prati sve reference i osigurava da niz ne bude dealociran dok god postoji bilo koja *MatrixView* referenca.

Ovo nije znatno ubrzalo Javu, na svega 1.352 sekundi, ali Rust jeste, pao je na 0.744 sekundi.

I zatim sam još optimizovao *cache* sa drugim redosledom prolaska kroz matricu sekvencijalno:

```
for i in 0..n {
    for k in 0..p {
        let a_ik = a.get(i, k);     // Učitaj jednom
        for j in 0..m {
            let current = result.get(i, j);
            result.set(i, j, current + a_ik * b.get(k, j));
            // ↑ b.get(k, j) ide po redovima (uzastopni elementi)
            // smanjen cache miss, odnosno broj preskakanja praznih
kolona
        }
    }
}
```

4.3. Implementacija algoritama

Rust i Java implementacije su identične po logici:

4.3.1. Standardni algoritam

RUST

```rust
pub fn multiply_standard(a: &Matrix, b: &Matrix) -> Matrix {
    let n = a.rows();
    let m = b.cols();
    let p = a.cols();
    let mut result = Matrix::new(n, m);

    for i in 0..n {
        for k in 0..p {
            let a_ik = a.get(i, k);
            for j in 0..m {
                let current = result.get(i, j);
                result.set(i, j, current + a_ik * b.get(k, j));
            }
        }
    }
    result
}
```

JAVA

```java
public static Matrix multiply(Matrix a, Matrix b) {
    int n = a.getRows();
    int m = b.getCols();
    int p = a.getCols();
    Matrix result = new Matrix(n, m);

    for (int i = 0; i < n; i++) {
        for (int k = 0; k < p; k++) {
            double a_ik = a.get(i, k);
            for (int j = 0; j < m; j++) {
                result.set(i, j, result.get(i, j) + a_ik * b.get(k,
j));
            }
        }
    }
    return result;
}
```

4.3.2. Divide and conquer algoritam

```rust
use rayon::prelude::*;

pub fn multiply_dc_parallel(a: &MatrixView, b: &MatrixView) -> Matrix
{
    let n = a.rows();

    if n <= THRESHOLD {
        return multiply_standard_view(a, b);
    }

    let half = n / 2;
    let a11 = a.subview(0, half, 0, half);
    let a12 = a.subview(0, half, half, n);
    let a21 = a.subview(half, n, 0, half);
    let a22 = a.subview(half, n, half, n);

    let b11 = b.subview(0, half, 0, half);
    let b12 = b.subview(0, half, half, n);
    let b21 = b.subview(half, n, 0, half);
    let b22 = b.subview(half, n, half, n);

    // paralelno računanje svih 8 proizvoda
    let products: Vec<Matrix> = vec![
        (&a11, &b11), (&a12, &b21),  // Za C11
        (&a11, &b12), (&a12, &b22),  // Za C12
        (&a21, &b11), (&a22, &b21),  // Za C21
        (&a21, &b12), (&a22, &b22),  // Za C22
    ]
    .par_iter()
    .map(|(x, y)| multiply_dc(x, y))
    .collect();

    // Kombinovanje rezultata
    let c11 = products[0].add(&products[1]);
    let c12 = products[2].add(&products[3]);
    let c21 = products[4].add(&products[5]);
    let c22 = products[6].add(&products[7]);
```

```
    combine_blocks(&c11, &c12, &c21, &c22)
}
```

Optimalna vrednost TRESHOLD konstante zavisi od CPU, najbolje performanse sam dobio sa vrednošću 128, u tom momentu prelazi na sekvencijalno jer već postaje previše jednostavno da bismo alocirali novu memoriju.

JAVA

```
public static Matrix multiplyParallel(MatrixView a, MatrixView b)
{   ForkJoinPool pool = ForkJoinPool.commonPool();
    return pool.invoke(new MultiplyTask(a, b));
}

private static class MultiplyTask extends RecursiveTask<Matrix> {
    private final MatrixView a, b;

    public MultiplyTask(MatrixView a, MatrixView b) {
        this.a = a;
        this.b = b;
    }

    @Override
    protected Matrix compute() {
        int n = a.getRows();

        if (n <= THRESHOLD) {
            return MatrixOps.multiplyStandardView(a, b);
        }

        int half = n / 2;
        MatrixView a11 = a.subview(0, half, 0, half);
        // ... ostali view-ovi

        // Kreiranje zadataka
        MultiplyTask task1 = new MultiplyTask(a11, b11);
        MultiplyTask task2 = new MultiplyTask(a12, b21);
        // ... još 6

        // Fork (asinhrono pokretanje)
```

```
        task2.fork();
        task3.fork();
        task4.fork();
        task5.fork();
        task6.fork();
        task7.fork();
        task8.fork();

        // Compute prvi zadatak sinhrono
        Matrix p1 = task1.compute();

        // Join (čekanje rezultata)
        Matrix p2 = task2.join();
        Matrix p3 = task3.join();
        // ... ostali join-ovi

        // Kombinovanje
        Matrix c11 = p1.add(p2);
        Matrix c12 = p3.add(p4);
        Matrix c21 = p5.add(p6);
        Matrix c22 = p7.add(p8);

        return MatrixOps.combineBlocks(c11, c12, c21, c22);
    }
}
```

4.3.3. Strassen algoritam

RUST

```
pub fn multiply_strassen_parallel(a: &MatrixView, b: &MatrixView) ->
Matrix {
let n = a.rows();
    if n <= THRESHOLD {
        return multiply_standard_view(a, b);
    }

    let half = n / 2;

    let a11 = a.subview(0, half, 0, half);
```

```rust
let a12 = a.subview(0, half, half, n);
let a21 = a.subview(half, n, 0, half);
let a22 = a.subview(half, n, half, n);

let b11 = b.subview(0, half, 0, half);
let b12 = b.subview(0, half, half, n);
let b21 = b.subview(half, n, 0, half);
let b22 = b.subview(half, n, half, n);

// Priprema operanada
let temp_a1 = add_views(&a11, &a22);
let temp_b1 = add_views(&b11, &b22);
let temp_a2 = add_views(&a21, &a22);
let temp_b3 = subtract_views(&b12, &b22);
let temp_b4 = subtract_views(&b21, &b11);
let temp_a5 = add_views(&a11, &a12);
let temp_a6 = subtract_views(&a21, &a11);
let temp_b6 = add_views(&b11, &b12);
let temp_a7 = subtract_views(&a12, &a22);
let temp_b7 = add_views(&b21, &b22);

// Kreiranje view-ova od temp matrica
let view_a1 = MatrixView::from_matrix(&temp_a1);
let view_b1 = MatrixView::from_matrix(&temp_b1);
let view_a2 = MatrixView::from_matrix(&temp_a2);
let view_b3 = MatrixView::from_matrix(&temp_b3);
let view_b4 = MatrixView::from_matrix(&temp_b4);
let view_a5 = MatrixView::from_matrix(&temp_a5);
let view_a6 = MatrixView::from_matrix(&temp_a6);
let view_b6 = MatrixView::from_matrix(&temp_b6);
let view_a7 = MatrixView::from_matrix(&temp_a7);
let view_b7 = MatrixView::from_matrix(&temp_b7);

// paralelno računanje 7 proizvoda
let tasks = vec![
    (&view_a1, &view_b1),  // M1
    (&view_a2, &b11),      // M2
    (&a11, &view_b3),      // M3
    (&a22, &view_b4),      // M4
```

```
        (&view_a5, &b22),       // M5
        (&view_a6, &view_b6),   // M6
        (&view_a7, &view_b7),   // M7
    ];

    let products: Vec<Matrix> = tasks
        .par_iter()
        .map(|(x, y)| multiply_strassen_seq(x, y))
        .collect();

    // Raspakovati rezultate
    let m1 = &products[0];
    let m2 = &products[1];
    let m3 = &products[2];
    let m4 = &products[3];
    let m5 = &products[4];
    let m6 = &products[5];
    let m7 = &products[6];

    // Kombinovanje
    let c11 = m1.add(m4).subtract(m5).add(m7);
    let c12 = m3.add(m5);
    let c21 = m2.add(m4);
    let c22 = m1.subtract(m2).add(m3).add(m6);

    combine_blocks(&c11, &c12, &c21, &c22)
}
```

Java verzija je analogna (koristi RecursiveTask umesto .par_iter()).

5. EKSPERIMENTALNI REZULTATI

Sva merenja izvršena su na sledećem hardverskom i softverskom okruženju:

Hardver:

1. CPU: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz (2.70 GHz)

2. Broj jezgara: 8
3. RAM: 16GB
4. OS: Windows 11

Softver:

1. Rust verzija: 1.92
2. Compiler flags: `cargo build --release` (opt-level=3, LTO enabled)
3. Java verzija: JDK 17
4. JVM flags: Default
5. Warm-up: Svaki test pokrenut 3 puta, uzet prosek poslednjih 2 pokretanja

5.1. Poređenje

rust

Veličina	Standardni	D&C par	Strassen par
128x128	0.001s	0.001s	0.001s
256x256	0.006s	0.003s	0.004s
512x512	0.039s	0.020s	0.020s
1024x1024	-	0.145s	0.114s
2048x2048	-	1.476s	0.779s
4096x4096	-	10.889s	4.880s

java

Veličina	Standardni	D&C par	Strassen par
256x256	0.012s	0.012s	0.038s
512x512	0.086s	0.060s	0.035s
1024x1024	-	0.284s	0.206s
2048x2048	-	2.210s	1.311s
4096x4096	-	23.711s	11.244s

Rust je brži 2.3 puta.

5.2. Memorijska ograničenja

Bilo je memorijskih ograničenja:

OutOfMemoryError na 8192×8192 (Java)

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space at matrix.Matrix.<init>(Matrix.java:12) at matrix.Strassen$StrassenTask.compute(Strassen.java:87)

Analiza:

Jedna matrica: $8192^2 \times 8$ bytes = 512 MB

Tri matrice (A, B, C): 3 × 512 MB = 1.5 GB

Strassen privremene alokacije po nivou: - 7 množenja → 7 × 512 MB = 3.5 GB

(rezultati M1-M7) - 14 sabiranja/oduzimanja → 14 × 512 MB = 7 GB

Ukupna memorija: ~12-15 GB

Default Java heap: Obično 1/4 RAM-a = ~4 GB (nedovoljno)

Rust nije imao ovaj problem jer:

1. Nema *GC*
2. dealokacija odmah oslobađa memoriju
3. Bolja kontrola *lifetime*-a privremenih matrica

Rešenje za javu bi moglo da bude da povećamo heap na 16GB.

rust

Veličina	Matrice (MB)
512x512	4.00
1024x1024	16.00
2048x2048	64.00
4096x4096	256.00

java

Veličina	Matrice (MB)
512x512	8.01
1024x1024	20.00
2048x2048	68.00
4096x4096	260.00

Java definitvno koristi nešto više memorije iako je to manji procenat sa porastom veličine matrice odnosno kompleksnosti množenja. Rust odmah dealocira matrice pri izlasku iz scope-a i nema GC.

6. ZAKLUČAK

Ovaj rad predstavio je sveobuhvatno empirijsko poređenje Rust i Java programskih jezika kroz implementaciju tri algoritma za množenje matrica: standardni iterativni, Divide-and-Conquer i Strassen. Kroz proces dvofazne implementacije, od prirodnih idioma jezika do optimizovanih verzija sa identičnim strukturama podataka, došli smo do nekoliko značajnih zaključaka.

1. Struktura podataka

Dizajn struktura podataka ima značajniji uticaj na performanse nego sam izbor programskog jezika. Rezultati nedvosmisleno demonstriraju da programeri koji razumeju alociranje memorije mogu postići bolje performanse nego oni koji se slepo oslanjaju na "brži jezik".

2. Predvidive i konzistentne performanse Rust-a

U svim scenarijima Rust je pokazao konzistetnu prednost, bolje skaliranje paralelizacije i performanse su predvidive bez obzira na runtime uslove. Java s druge strane pokazuje odlične JIT optimizacije gde bilo koja optimizacija jedva i da pomaže performansama, dobija se sasvim doba paralelizacija odmah.

3. Paralelizacija

Možemo da biramo između brzine razvoja i kontrole u Rust-u, na datom primeru su rezultati gotovo isti koristeći deklarativan pristup u jednoj liniji: `.par_iter().map(|x| compute(x)).collect()`

ForkJoinPool u Javi zahteva eksplicitan pristup.

4. Memorijska efikasnost

Java objekti nose header koji uzima par bajtova i ima garbage collector, pa i minimalan i maksimalan utrošak mogu da variraju. Rust oslobađa memoriju što je omogućilo da radi i sa većim matricama bez problem dok smo u Javi imali OutOfMemoryError koji treba eksplicitno debugovati.

Rust definitvno pobeđuje kada je u pitanju ograničena memorija, thread safety garantovan prilikom kompajliranja, dok Java ima prednost u brzini i lakoći razvoja.

Dalji pravci istraživanja bi uključivali SIMD optimizacije u Rust-u, heterogene sisteme CPU i GPU, ozbiljnije algoritme i prave probleme.

Rust nudi performanse i memory safety bez kompromisa, ali zahteva investiciju u učenje i disciplinovan pristup vlasništvu podataka. Java nudi produktivnost i ekosistem, ali plaća cenu runtime-a i nedeterminističke GC-a.

Za kraj je svakako važno pomenuti da dobar dizajn struktura podataka u "sporijem" jeziku pobija loš dizajn u "bržem" jeziku. Naravno da je neophodno investiranje vremena u razumevanje memorije i algoritamske optimizacije pre nego što krivimo jezik za loše performanse.