

Množenje matrica - Rust i Python poređenje implementacija

Branislav Stojković E2 64/2025

1. Uvod

Množenje matrica predstavlja jedan od osnovnih problema numeričke matematike i računarskih nauka, sa širokom primenom u oblastima kao što su naučne simulacije, obrada slike, mašinsko učenje i visoko-performantno računanje (HPC). Efikasnost implementacije ovog algoritma ima direktan uticaj na performanse složenih sistema.

Cilj ovog seminarskog rada je uporedna analiza programskih jezika Rust i Python kroz implementaciju i evaluaciju sledećih algoritama za množenje matrica:

- Standardnog iterativnog algoritma,
- Zavadi pa vladaj (Divide-and-conquer) algoritma,
- Strassenovog algoritma.

Za zavadi-pa-vladaj i Strassenov algoritam razmatrane su i paralelne implementacije. Poređenje se vrši na osnovu vremena izvršavanja, potrošnje memorije, skalabilnosti, lakoće razvoja i modela paralelizacije.

2. Opis algoritama

2.1 Iterativni algoritam

Standardni iterativni algoritam koristi tri ugnježdene petlje i ima vremensku složenost $O(n^3)$. Predstavlja osnovni i referentni algoritam za poređenje ostalih pristupa, jer je jednostavan za implementaciju i ne uvodi dodatne alokacije podmatrica.

2.2 Divide-and-Conquer algoritam

Divide-and-conquer pristup deli matricu na četiri podmatrice dimenzije $\frac{n}{2} * \frac{n}{2}$ i rekurzivno računa rezultat. Iako teorijska složenost ostaje $O(n^3)$, ovaj pristup može imati bolju lokalnost podataka (u zavisnosti od reprezentacije matrice) i prirodno omogućava paralelizaciju nezavisnih rekurzivnih poziva.

2.3 Strassenov algoritam

Strassenov algoritam smanjuje broj rekurzivnih množenja sa 8 na 7 po nivou rekurzije, čime se postiže složenost $O(n^{2.81})$. Efikasniji je za veće dimenzije matrica, ali uvodi dodatne operacije sabiranja/oduzimanja i povećava potrošnju memorije zbog kreiranja većeg broja privremenih matrica.

3. Implementacija u programskom jeziku Rust

Rust je sistemski programski jezik fokusiran na visoke performanse i bezbedno upravljanje memorijom bez garbage collector. Implementacija koristi tipičnu reprezentaciju matrice kao `Vec<Vec<f64>>`, zbog čitljivosti i direktnog mapiranja algoritamske strukture na kod.

Za paralelizaciju je korišćena biblioteka `rayon`, pri čemu se rekurzivne operacije množenja izvršavaju paralelno korišćenjem `rayon::join`. Time se obezbeđuje efikasna upotreba više jezgara bez eksplicitnog upravljanja nitima. Kod je kompajliran u `--release` režimu, čime su omogućene optimizacije kompajlera.

Potrošnja memorije u Rust implementaciji merena je korišćenjem biblioteke `sysinfo` (zauzeće memorije sistema pre i nakon izvršavanja), pri čemu dobijena vrednost predstavlja aproksimaciju dodatne potrošnje memorije tokom rada algoritma.

4. Implementacija u programskom jeziku Python

Python je interpretirani jezik visokog nivoa poznat po čitljivosti i brzini razvoja. Matrice su predstavljene kao liste listi (`list[list[float]]`), što odgovara pristupu korišćenom u Rust implementaciji i olakšava fer poređenje algoritamske strukture.

Zbog Global Interpreter Lock (GIL) ograničenja, paralelizacija je realizovana korišćenjem multiprocessing modula (više procesa umesto niti). Ovaj pristup uvodi dodatni overhead, prvenstveno kroz kopiranje podataka između procesa i trošak kreiranja/koordinacije procesa.

Potrošnja memorije u Python implementaciji merena je korišćenjem biblioteke `psutil`, praćenjem zauzeća memorije procesa pre i nakon izvršavanja algoritma. Kao i u Rust slučaju, razlika predstavlja aproksimaciju dodatne potrošnje memorije, uz napomenu da uključuje i overhead Python interpretera i multiprocessing infrastrukture.

5. Sličnosti i razlike implementacija u Rustu i Pythonu

Iako su algoritmi implementirani na isti način u oba programska jezika, razlike između Rusta i Pythona značajno utiču na način implementacije, performanse i ponašanje programa.

U obe implementacije korišćena je ista struktura algoritama, isti pragovi za prelazak na iterativni algoritam i ista strategija podele matrica, čime je obezbeđeno fer poređenje performansi. Razlike se prvenstveno ogledaju u modelu izvršavanja, upravljanju memorijom i paralelizaciji.

Rust omogućava statičku proveru bezbednosti memorije kroz ownership i borrowing mehanizam, što eliminiše čitavu klasu runtime grešaka. Paralelizacija u Rustu je realizovana korišćenjem biblioteke *rayon*, koja omogućava efikasno korišćenje više jezgara bez eksplicitnog upravljanja nitima.

Python implementacija koristi *multiprocessing* modul zbog Global Interpreter Lock (GIL) ograničenja, što zahteva korišćenje više procesa umesto niti. Ovakav

pristup uvodi dodatni overhead u vidu kopiranja podataka između procesa i povećane potrošnje memorije.

Dok Rust omogućava visoke performanse i preciznu kontrolu nad resursima, Python se ističe čitljivošću i jednostavnošću implementacije, što ga čini pogodnijim za brze prototipe i edukativne svrhe.

6. Eksperimentalno okruženje

Testiranja su izvršena na istom računaru, nad matricama dimenzija 128×128 , 256×256 i 512×512 . Rust implementacija je izvršavana u release režimu, dok je Python izvršavan standardnim interpreterom.

7. Rezultati

Za matricu dimenzije 512×512 dobijeni su sledeći rezultati:

Algoritam	Rust – vreme (s)	Rust – memorija (MB)	Python – vreme (s)	Python – memorija (MB)
Iterativni	0.361	~7	7.737	~7
Divide & Conquer (sekv.)	0.355	~0	6.711	~17
Divide & Conquer (paral.)	0.049	~2	1.125	~12
Strassen (sekv.)	0.242	~2	5.035	~24
Strassen (paral.)	0.043	~5	1.030	~13

Rust pokazuje višestruko bolje performanse u svim slučajevima, a najveća razlika se vidi kod paralelnih algoritama. Primećuje se i da Strassen u sekvencijalnoj varijanti daje bolje rezultate od iterativnog i *divide-and-conquer* pristupa za dimenziju 512×512 .

Rezultati merenja memorije pokazuju da Strassenov algoritam ima veće zahteve za memorijom u odnosu na ostale pristupe, zbog većeg broja privremenih matrica. U Python implementaciji ovaj efekat je dodatno izražen zbog *multiprocessing* modela i kopiranja podataka između procesa.

8. Diskusija

Rezultati jasno pokazuju da paralelizacija ima veći uticaj na performanse od samog izbora algoritma, naročito kada se porede sekvencijalne i paralelne verzije istog pristupa. U Rustu, paralelizacija preko *rayon* biblioteke omogućava efikasnu upotrebu više jezgara uz relativno mali *overhead*, što objašnjava značajno skraćanje vremena izvršavanja za paralelne varijante.

U Pythonu, prelazak na paralelno izvršavanje donosi ubrzanje, ali ne u meri kao u Rustu, jer *multiprocessing* uvodi dodatne troškove: kreiranje procesa, serijalizaciju i kopiranje podataka, kao i međuprocesnu koordinaciju. Ovo se vidi i kroz povećanu potrošnju memorije, posebno kod rekurzivnih algoritama koji stvaraju veći broj privremenih struktura.

Sa aspekta lakoće razvoja, Python omogućava bržu implementaciju zbog jednostavne sintakse, dinamičkog tipiziranja i manjeg broja infrastrukturnih odluka. Sa druge strane, Rust zahteva detaljnije planiranje struktura podataka i eksplicitno upravljanje vlasništvom nad podacima. Iako to produžava vreme razvoja, rezultuje robusnijim i bezbednijim kodom, posebno u kontekstu paralelnog izvršavanja.

Ograničenja eksperimenta. Dobijeni rezultati zavise od hardverske arhitekture, hijerarhije keš memorije i broja dostupnih procesorskih jezgara. Takođe, performanse Strassenovog i *divide-and-conquer* pristupa zavise od izabranog praga (*cutoff*) za prelazak na iterativni algoritam, pa se rezultati ne mogu posmatrati kao apsolutni, već kao indikativni za upoređivanje jezika i modela paralelizacije.

9. Zaključak

Na osnovu dobijenih rezultata može se zaključiti da je Rust znatno pogodniji za visoko-performantno i paralelno numeričko računanje. Kombinacija kompajliranog koda, efikasnog modela niti i niskog *overhead*-a paralelizacije omogućava višestruko kraće vreme izvršavanja u odnosu na Python.

Python je pogodniji za brze prototipe i edukativne svrhe, ali pri paralelnom izvršavanju trpi ograničenja zbog GIL-a i potrebe za *multiprocessing* pristupom, što povećava *overhead* i potrošnju memorije. Izbor jezika zavisi od zahteva sistema i prioriteta između performansi i brzine razvoja.

10. Moguća proširenja rada

- SIMD optimizacije u Rustu
- Poređenje sa NumPy/BLAS bibliotekama
- GPU ubrzanje