Kanonov algoritam množenja matrica

# O algoritmu:

Kanonov algoritam (*Cannon's algorithm*) je algoritam osmišljen za distribuirano množenje matrica. Ideja je da se dve matrice čiji se rezultat traži podele na blokove, tj. podmatrice fiksne veličine, zatim da se pomeranjem i deljenjem blokova između procesa dolazi do rešenja, korak po korak.

Prednost ovoga algoritma u odnosu na klasično paralelno izračunavanje je to što proces u jednom koraku čuva samo blokove/podmatrice matrica koje trenutno računa, pa je zbog toga zauzeće memorije malo i nemamo deljenu memoriju između procesa.

**Primer**:

|  |  |  |
| --- | --- | --- |
| A00 | A01 | A02 |
| A10 | A11 | A12 |
| A20 | A21 | A22 |

|  |  |  |
| --- | --- | --- |
| B00 | B01 | B02 |
| B10 | B11 | B12 |
| B20 | B21 | B22 |

Imamo dve matrice A i B koje trebamo pomnožiti i rezultat smestiti u matricu C. Da bismo izračunali rezultat potrebno je izračunati sledeće:  
  
 C00 = A00\*B00 + A01\*B10 + A02\*B20  
 C01 = A00\*B01 + A01\*B11 + A02\*B21 C02= A00\*B02 + A01\*B12 + A02\*B22 C10 = A10\*B00 + A11\*B10 + A12\*B20 C11 = A10\*B01 + A11\*B11 + A12\*B21 C12 = A10\*B02 + A11\*B12 + A12\*B22 C20 = A20\*B00 + A21\*B10 + A22\*B20 C21 = A20\*B01 + A21\*B11 + A22\*B21 C22 = A20\*B02 + A21\*B12 + A22\*B22  
U svakoj jednačini imamo tri sabiranja (pretpostavlja se da je svaka vrednost u matrici C pre početka računanja postavljena na nulu). U ovom algormu, imaćemo tri koraka, svaki će da pomnoži blokove i da sabere sa prethodnim rezultatom.  
Umesto da za svaku ćeliju matrice C prilikom računanja moramo da pristupamo različitim ćelijama matrica A i B, Kanonov algoritam zahteva da se u svakom koraku množe brojevi na korespodentnim ćelijama, a da se pomeraju blokovi podataka.  
Prilikom računanja jedne vrednosti ćelije matrice koristimo sve vrednosti iz jednog reda matrice A, kao i sve vrednosti iz iste kolone matrice B, koji se međusobno množe, a potom sabiraju.  
To znači da je potrebno samo pomerati blokove matrice A na levo (pri tom krajnja leva kolona dolazi na kraj), a blokove matrice B na gore (takodje, prvi red dolazi na kraj).  
Međutim, ako to krenemo da radimo, iako u svakoj ćeliji dobijamo blokove koji ulaze u rezultat, parovi podataka koji se množe nisu odgovarajući (osim za C00). Dakle, potrebno je uvesti još jedan, inicijalni korak koji će poravnati podatke.   
Ukoliko pre početka računanja matricu A „uvrnemo“ tako da blokove u prvom redu pomeramo za 0 koraka u levo, blokove u drugom redu za 1 korak u levo... blokove u n-tom redu za (n-1) koraka u levo, za matricu B ponovimo algoritam za kolone na gore, blokovi će se poravnati tako da će se u svakom narednom koraku se množiti odgovarajući blokovi podataka. Množenja ima onoliko, koliko ima blokova u jednom redu, u ovom slučaju 3.

Ukratko, Kanonov algoritam se sastoji iz sledećih koraka:  
 1. Korak:  
 - „uvrtanje“ matrice A na levo  
 - „uvrtanje“ matrice B na gore  
 2. Korak:  
 - izračunavanje proizvoda korespodentnih blokova i sabiranje sa prethodnim rezultatom  
 - ukoliko nije kraj pomeranja  
 - pomeranje svih kolona matrice A za jedno mesto u levo  
 - pomeranje svih redova matrice B za jedno mesto na gore  
 3.Korak:  
 - ponavljati korak 2 onoliko puta koliko ima blokova u jednom redu

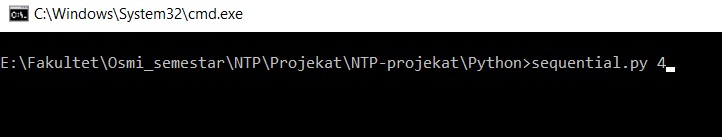
Napomena: Blokovi su matrice manjih dimenzija od originalne i množenje dva bloka je obično množenje matrica

# Python

Implementirana je sekvencijalna i paralelna verzija algoritma. Kodovi se nalaze na putanji: NTP-Projekat -> Python. U folderu input nalaze se fajlovi matrixA i matrixB, iz kojih programi čitaju ulazne matrice. Rezultati programa se upisuju u tekstualne fajlove u folderu output.  
Sekvencijalna verzija prati Kanonov algoritam, vrši podelu ulaznih matrica na blokove i pomera ih, iako se sva izračunavanja vrše sekvencijalno.  
Paralelna verzija je implementirana korišćenjem mpi4py, koji omogućava rad sa procesima i komunikaciju među procesima.

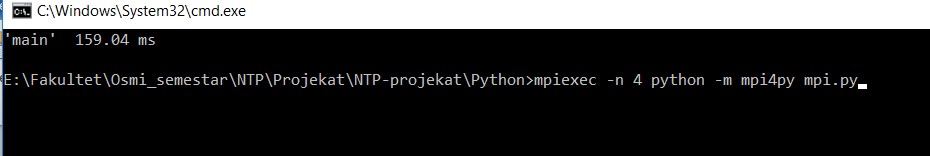
## Sekvencijalna verzija programa:

**Pokretanje programa**:  
 Otvoriti CommandPrompt u folderu: NTP-Projekat -> Python. Ukucati komandu sequential.py broj\_procesa



## Paralelna verzija

Pokretanje programa:  
 Otvoriti CommandPrompt u folderu: NTP-Projekat -> Python. Ukucati komandu mpiexec -n broj\_procesa python -m mpi4py mpi.py



U paralelnoj verziji programa, pomeranje matrica se svodi na slanje bloka podataka sledećem procesu i preuzimanje novog bloka od prethodnog procesa.

## Organizacija koda:

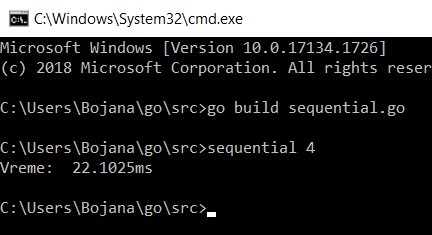
U folderu Python se nalaze sledeći fajlovi:  
 - matrix\_basic.py: modul koji sadrži operacije za osnovni rad sa matricama, kao što su učitavanje matrice, množenje..  
 - cannon.py: modul koji sadrži funkcije za rad sa matricama, koje prate Kanonov algoritam  
 - measure.py: modul koji sadrži funkciju koja meri vreme potrebno da se izvrši funkcija, na koju se primeni kao dekorator  
 - sequential.py: implementirana sekvencijalna verzija programa  
 - mpi.py: implementirana paralelna verzija programa

# Go

Implementirana je sekvencijalna i paralelna verzija algoritma. Kodovi se nalaze na putanji: NTP-Projekat -> Go. U folderu input nalaze se fajlovi matrixA i matrixB, iz kojih programi čitaju ulazne matrice. Rezultati programa se upisuju u tekstualne fajlove u folderu output.

## Sekvencijalna verzija programa:

**Pokretanje programa**:  
 Kod treba da se nalazi na putanji: C:\Users\NAME\go\src.Otvoriti CommandPrompt u folderu: NTP-Projekat -> Go. Ukucati sledeće komande: go build sequential.go, pa sequential broj\_procesa



## Paralelna verzija

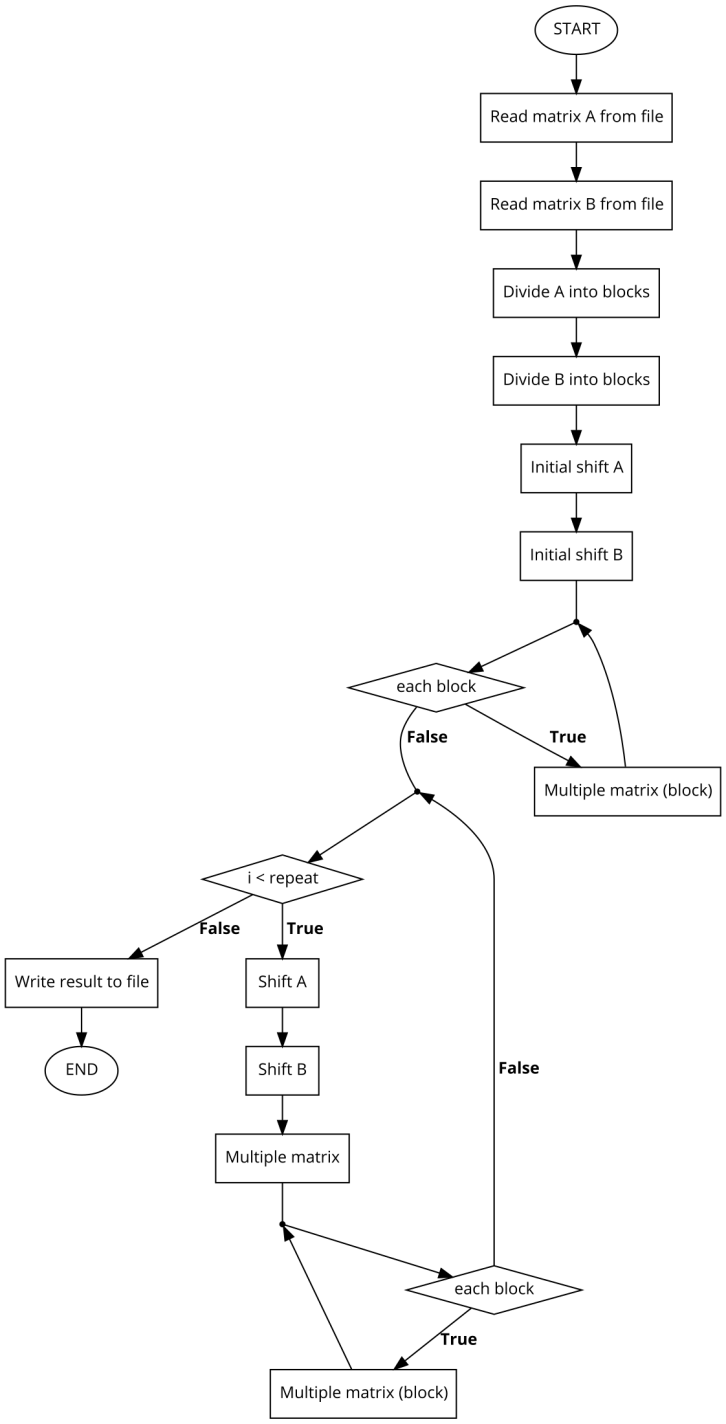
Pokretanje programa:  
 Otvoriti CommandPrompt u folderu: NTP-Projekat -> Go. Ukucati komande: go build mpi.go, pa mpi broj\_procesa  
  
Organizacija koda:  
Organizacija koda je slična kao u Python verziji.

## Razlike između paralelnih verzija Python i Go programa

Za rešenje implementiranou Pythonu korišćen je MPI, koji omogućava komunikaciju između procesa. Pomeranje blokova se svodi na slanje i prijem blokova podataka. Svaki proces ima svoju memoriju i međusobno ne dele memoriju.

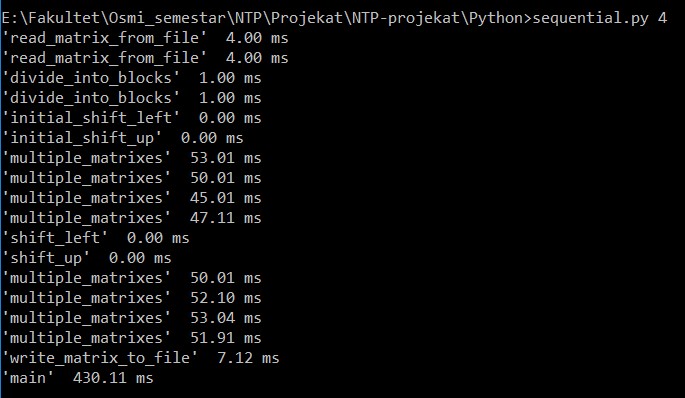
Go verzija korisni niti, tj. Go rutine, kojima upravlja Go izvršno okruženje. Sve niti mogu da pristupaju istom adresnom prostoru. Niti komuniciraju preko kanala. Da bi se sinhronizovao rad algoritma, potrebno je blokirati pisanje u kanal ako već postoji blok u kanalu, i blokirati čitanje ukoliko je kanal prazan.Analiza izvršavanja programa

Na sledećoj slici je prikazan graf toga programa za sekvencijalnu verziju:



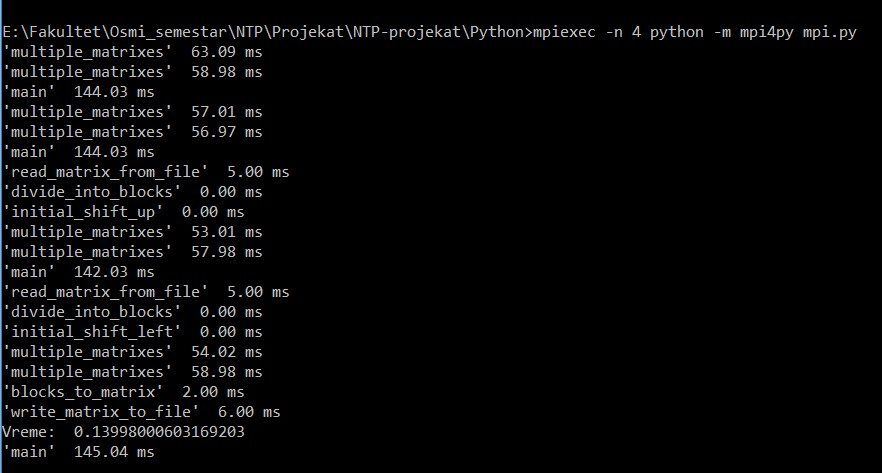
Graf toka programa

Pokrenućemo sekvencijalnu verziju programa u Python-u i ispisati vremena izvršavanja pojedinačnih funkcija, prikazanih na slici grafa toka programa



Program se izvršavao oko 430ms. Simuliran je rad algoritma sa 4 procesa. Vidimo da su vremena računanja proizvoda matrica slična. U ovom slučaju, vreme potrebno za pomeranje matrica je zanemarljivo.   
Kada bi rasporedili ovaj program na 4 procesa, u idealnom slučaju on bi se izvršio za oko 108ms.  
Međutim, ne može ceo program da se izvršava u paraleli i da koristi sva 4 procesa.   
U prvom delu programa, možemo koristiti 2 procesa, svaki će da učita matricu, podeli je na blokove i uradi inicijalno pomeranje blokova. Zatim, svaki proces vrši računanje za svoj blok, zatim čeka pomeranje, pa ponovo radi računanje, onoliko puta koliko ima pomeranja. Na kraju, potreban je jedan proces koji će rezultujuću matricu ispisati u fajl.   
Vreme izvršavanja bi bilo najmanje: 4ms + 1ms + 53.01ms + 53.04ms + 7.12ms = 118.17ms

Pokrenućemo program za iste matrice kao u analizi sekvencijalne verzije programa, sa istim brojem procesa:



Proces koji se najduže izvršavao je trajao 145.04ms. Znatno je skraćeno vreme u odnosu na sekvencijalno izvršavanje programa, ali je vreme veće od izračunatog, zbog komunikacije među procesima.

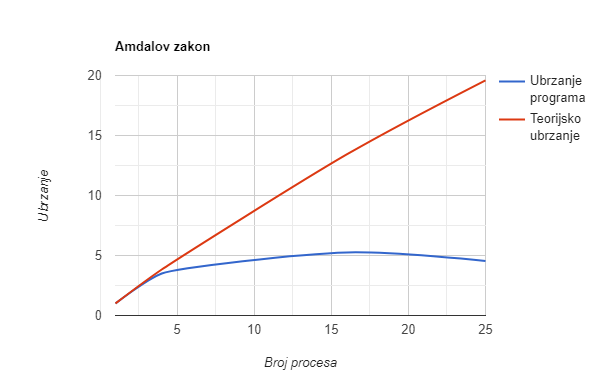
# Eksperimenti skaliranja

## Amdalov zakon – jako skaliranje

Amdalov zakon se formuliše kao:

speedup = 1 / (s + p / N)  
pri čemu je s deo vremena izvršavanja koji se troši na izvršavanje sekvencijalnog dela programa koji se ne može paralelizovati, p je deo vremena izvršavanja programa koji se troši na deo koji se može paralelizovati, a N je broj procesora.

Grafik jakog skaliranja za Python implementaciju:



Grafik jakog skaliranja za Go implementaciju:



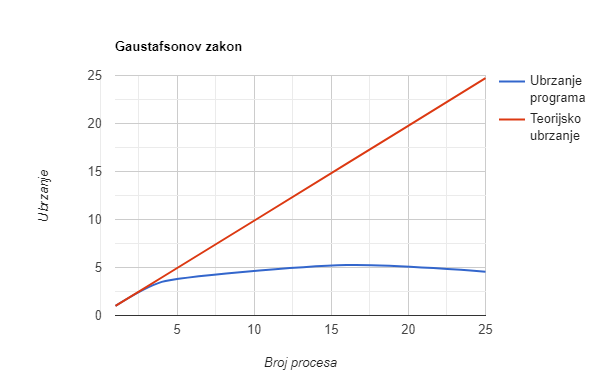
## Gaustafsonov zakon – slabo skaliranje

Gaustafsonov zakon se formuliše kao:

speedup = s + p × N

pri čemu s, p i N ima isto značenje kao u Amdalovom zakonu.

Grafik slabog skaliranja za Python implementaciju:



Grafik slabog skaliranja za Go implementaciju:

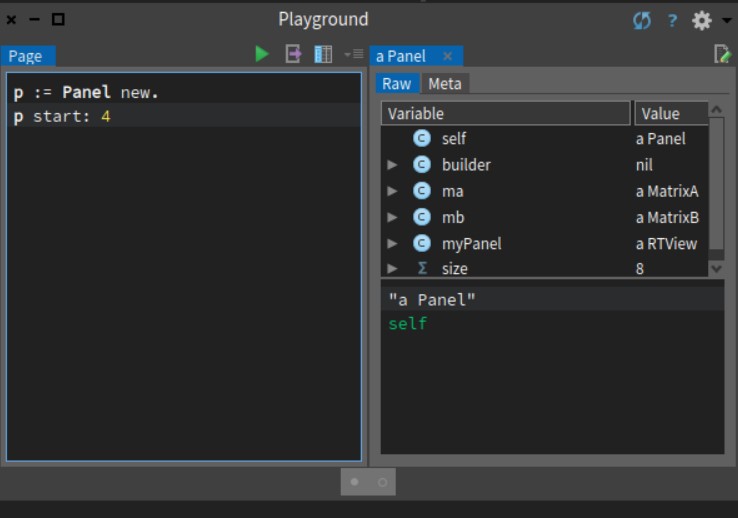


Pošto se Go verzija brže izvšava, eksperimenti za Python verziju su rađeni sa matricama 100x100, a za Go verziju sa matricama 1000x1000.  
Eksperimenti su vršeni na procesoru sa 6 fizičkih i 12 logičkih jezgara.

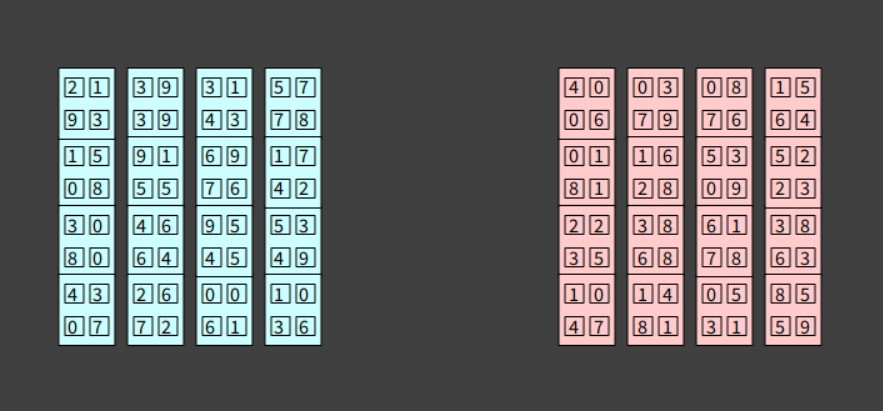
# Pharo vizualizacija algoritma

Vizualizacija je implementirana uz pomoć programskog jezika Pharo, uz biblioteku Roassal2.   
Nakon skidanja paketa Cannons sa repozitorijuma, potrebno je ubaciti ulazne matrice u datoteke matrixA.txt i matrixB.txt. Pokrenuti animaciju uz sledeće komande:

p := Panel new.  
p start: broj\_blokova\_u\_redu



Nakon pokretanja koda, pokreće se animacija koja prikazuje kako se ulazne matrice pomeraju u Kanonovom algoritmu, korak po korak.



**Organizacija koda:**U paketu se nalaze sledeće klase:  
 - BlockElement  
 - Matrix  
 - MatrixA  
 - MatrixB  
 - Panel

**BlockElement:** Klasa koja modeluje jedan blok podataka i popunjava blok na osnovu podmatrice koja se nalazi u bloku.  
**Matrix:** Klasa koja modeluje matricu. Ova klasa sadrži metode zajedničke za obe matrice, i ne instancira se, već će biti nasleđena.  
**MatrixA:** Klasa koja nasleđuje klasu Matrix i predstavlja prvu, levu matricu. Implementira metode za pomeranje matrica u levo- inicijalno pomeranje i pomeranje u narednim koracima.   
**MatrixB:** Klasa koja nasleđuje klasu Matrix i predstavlja drugu, desnu matricu. Implementira metode za pomeranje matrica na gore- inicijalno pomeranje i pomeranje u narednim koracima.  
**Panel:** Klasa koja instancira obe matrice, uvezuje ih i pravi animaciju.