Univerzitet u Novom Sadu

Fakultet tehničkih nauka

Dokumentacija za projektni zadatak

Student: Bogdanović Vukašin, SV09/2020

Predmet: Paralelno Programiranje

Tema projektnog zadatka: Paralelizacija PSO algoritama

**Sadržaj**

[**Uvod** 3](#_Toc105525000)

[**Analiza problema** 4](#_Toc105525001)

[**Implementacija** 6](#_Toc105525002)

[**Inicijalizacija Podataka** 6](#_Toc105525003)

[**Serijska Implementacija** 6](#_Toc105525004)

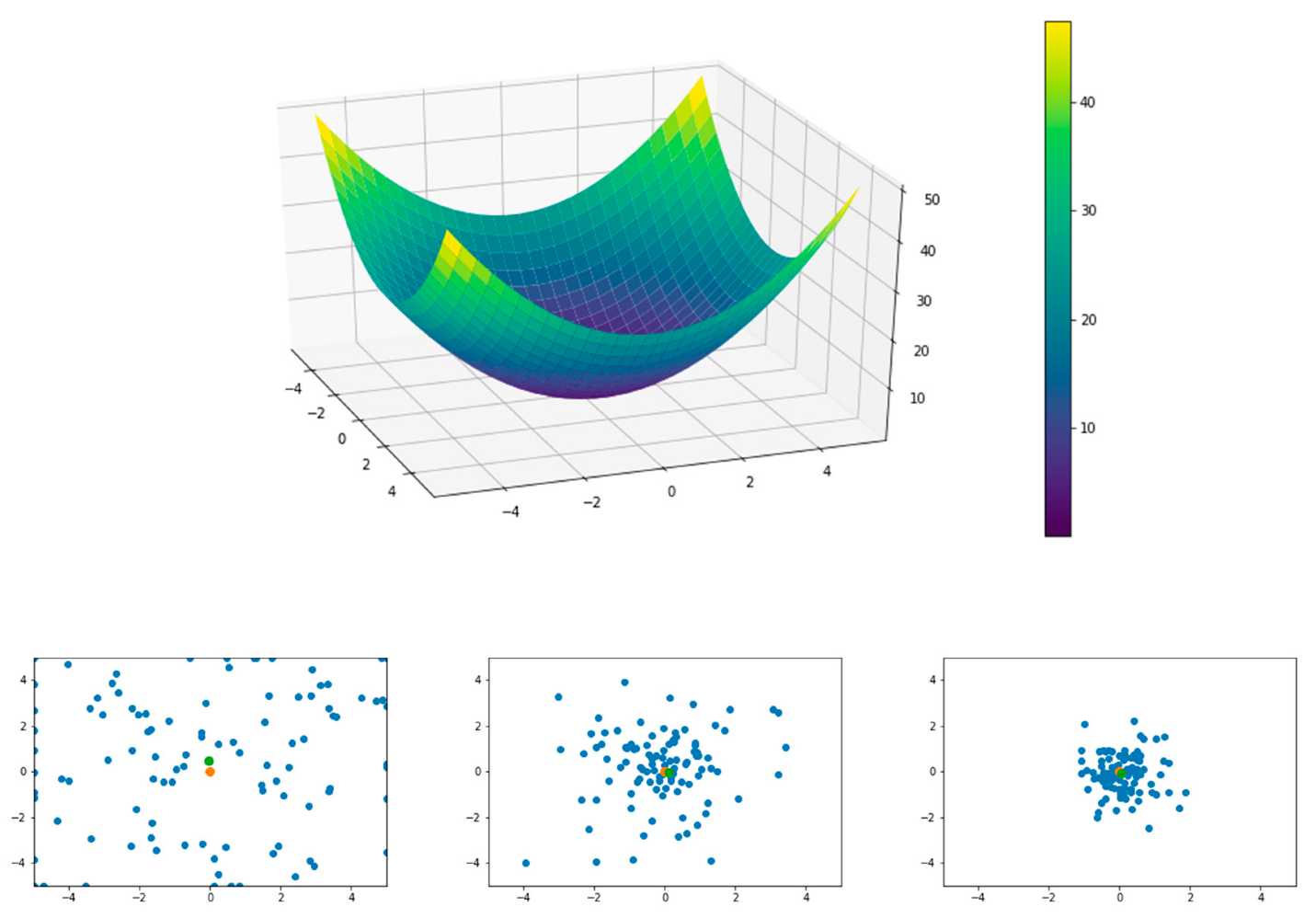
[**Paralelna Implementacija** 7](#_Toc105525005)

[**Analiza Rezultata** 10](#_Toc105525006)

[**Zaključak** 14](#_Toc105525007)

# **Uvod**

Particle swarm optimization algoritam ima veliku primenu u veštačkoj inteligenciji gde se koristi za izračunavanje kompleksnih funkcija koje su analitički nepoznate, odnosno za one funkcije čiju je vrednost moguće računarski evaluirati. Kako ovaj algoritam u tim slučajevima prima više desetina hiljada čestica koje su smeštene u sisteme sa više desetina dimenzija, on je idealan za paralelizovanje. Često je potrebno nekoliko sati za izvršavanje ovog algoritma u realnim primerima i tada bi paralelizacija značajno uštedela vreme za izračunavanje minimuma funkcije.



# **Analiza problema**

PSO algoritam funkcioniše slično kao i jato ptica u potrazi za hranom. Počinjemo sa određenim brojem tačaka koje su slučajnim putem raspoređene u prostoru i koje ćemo u daljem tekstu nazivati „čestice“, a zatim ćemo ih pustiti da traže minimume u krećući se u slučajno odabranim pravcima. Svakim sledećim korakom, svaka čestica će tražiti minimume oko najminimalnije tačke koju ona ikada pronašla, kao i oko minimalne tačke koju je pronađena od strane celog roja čestica. Posle određenog broja koraka, minimumom funkcije možemo smatrati najminimalniju tačku koju je ceo roj čestica pronašao.

Ako imamo N čestica, tada ćemo česticu k tokom iteracije i obeležavati sa Xk(t), pri čemu Xk(t) predstavlja tačku sa 60 koordinata.

Xk(t)=( X1k(t), X2k(t),..., X60k(t))

Analogno tome, brzinu ćemo označavati sa Vk(t), i važiće

Vk(t)=( V1k(t), V2k(t),..., V60k(t))

Koordinate čestica u svakoj narednoj iteraciji ćemo računati po formuli

**Xk(t+1)= Xk(t)+ Vk(t+1)**

Brzine čestica u svakoj sledećoj iteraciji ćemo računati po formuli

**Vk(t+1) = wk Vk(t) + cprp(pb- Xk(t)) + cgrg(gb- Xk(t))**

Iz jednačine za računanje brzine čestice u t+1 iteraciji vidimo da se ona sastoji od 3 komponente:

1. Inerciona komponenta - wk Vk(t)

Inerciona komponenta definiše stepen mobilnosti čestice, odnosno koliko je čestici lako da se kreće kroz prostor. Ova komponenta je definisana proizvodom:

* + - * 1. Konstante inercije wk koja uzima vrednosti od 0.9 do 0.4 po formuli

wk=0.9-k/N(0.9-0.4)

N- ukupan broj iteracija, k- trenutna iteracija

* + - * 1. Brzinom čestice k u prethodnoj iteraciji Vk(t)

1. Kognitivne komponente- cprp(pb- Xk(t))

Kognitivna komponenta definiše koliko će se svaka čestica oslanjati na sopstveno iskustvo. Ova komponenta je definisana proizvodom:

* + - * 1. Konstanti ubrzaznja cp irp, pri čemu cp uzima vrednosti od 2.5 do 0.5, a rp uzima slučajnu vrednost u opsegu od 0 do 1 pri čemu se njena vrednost menja prilikom svake iteracije
        2. Trenutne udaljenosti od lične najbolje pozicije pb

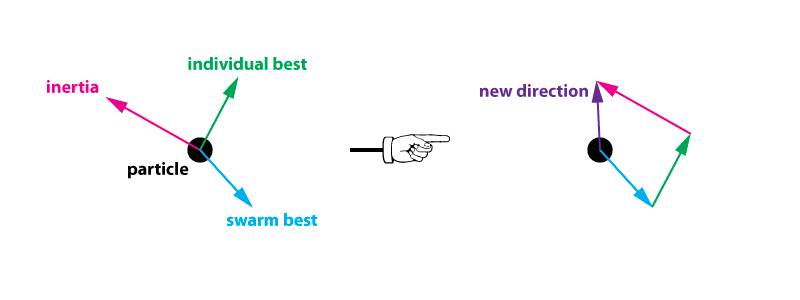
1. Socijalne komponente komponente- cgrg(gb- Xk(t))

Socijalna komponenta definiše koliko će se svaka čestica „slušati“ iskustvo drugih čestica iz roja. Ova komponenta je definisana proizvodom:

* + - * 1. Konstanti ubrzaznja cg irg, pri čemu cp uzima vrednosti od 0.5 do 2.5, a rg uzima slučajnu vrednost u opsegu od 0 do 1 pri čemu se njena vrednost menja prilikom svake iteracije
        2. Trenutne udaljenosti od globalne najbolje pozicije gb

Nakon podešavanja novih vrednosti koordinata i brzina poredimo vrednosti trenutne najbolje lične vrednosti funkcije sa novim vrednostima podešavamo nove najbolje pozicije tako što biramo manju od svake dve. Takođe, ažuriramo i globalnu najbolju poziciju poređenjem vrednosti funkcija svih najboljih ličnih pozicija.

Zavisnost kretanja čestice od ove tri komponente najbolje sledeća opisuje slika



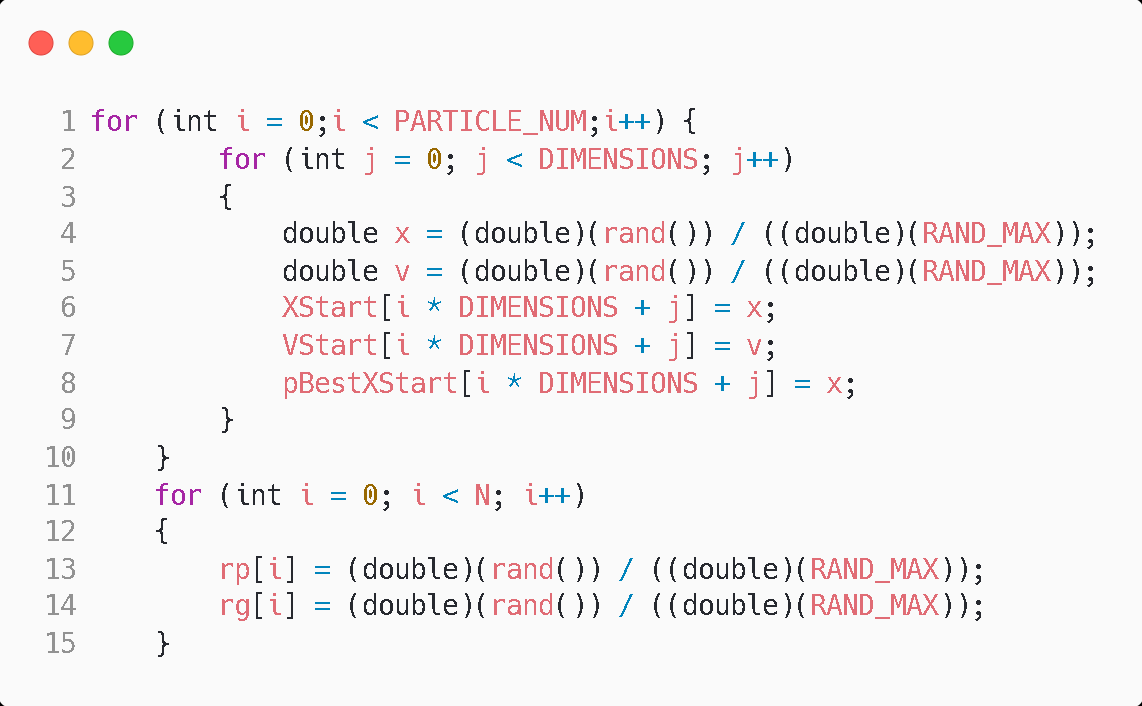
Naš algoritam ima 4 kritične tačke. Prva tačka predstavlja prvo izračunavanje vrednosti funkcije i postavljanje personalnih i globalnih minimuma. Kako je potrebno da program prodje kroz svaku od 12000 čestica, u našem slučaju, i da za svaku izračuna vrednost kompleksne funkcije, taj deo bi bilo poželjno paralelizovati tako što ćemo paralelno izračunavati vrednosti funkcija za više čestica. Drugi problem koji uočavamo je kalkulacija novih koordinata i brzina za svaku česticu, koji može da traje dosta dugo zbog velikih dimenzionalnosti samih funkcija. Potrebno je i ovaj deo podeliti na više delova i paralelizovati. Treći problem koji predstavlja i najveći problem u vremenu izvršavanja ovog algoritma jeste računanje vrednosti funkcije N\*M puta (N-broj iteracija, M- broj čestica).Rešavanjem ovog problema ćemo najviše doprineti brzini našeg programa, a to ćemo uraditi tako što će više niti istovremeno računati vrednost funkcije. Poslednji problem sa kojim se suočavamo u ovom tekstu je problem pronalaska najbolje vrednosti u jatu od nekoliko desetina hiljada čestica. Ovakav problem će se takođe rešiti na sličan način korišćenjem paralelnih zadataka.

# **Implementacija**

Algoritam se sastoji iz 4 koraka, inicijalizacije, ažuriranja pozicije i brzine, ažuriranje personalno i globalno najboljih vrednosti i prekida ažuriranja nakon predefinisanog broja iteracija. Pošto ovaj algoritam spada u grupu Blackbox algoritama za optimizaciju, određen broj parametara u izvršavanju zasniva se na slučajnom odabiru, a kako bismo videli pravi učinak paralelizacije iste „slučajne“ brojeve koristimo i prilikom serijskog i prilikom paralelnog izvršavanja algoritma. Koristimo konstante N za definisanje broja iteracija izvršavanja algoritma, DIMENSIONS za broj dimenzija funkcije sa kojom interagujemo i PARTICLE\_NUM za broj čestica sa kojim interagujemo prilikom izvršavanja algoritma.

## **Inicijalizacija Podataka**

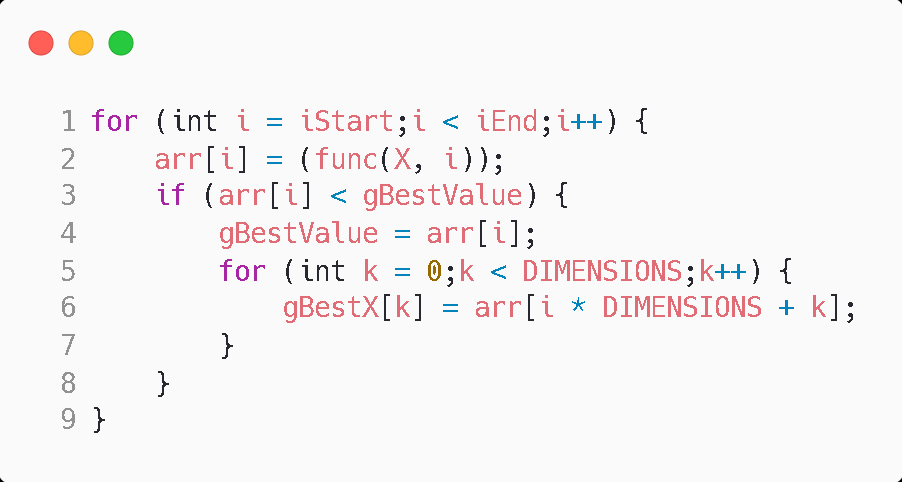
Inicijalizaciju podataka započinjemo tako što svakoj čestici dodeljujemo nasumične koordinate kao i nasumičnu brzinu kretanja. Kako čestice uopšte nisu istraživale, njihova trenutna pozicija će takođe biti i njihova najbolja lična pozicija. Nakon toga na linijama 13 i 14 izračunavamo personalni i globalni nasumični faktor za svaku iteraciju.



## **Serijska Implementacija**

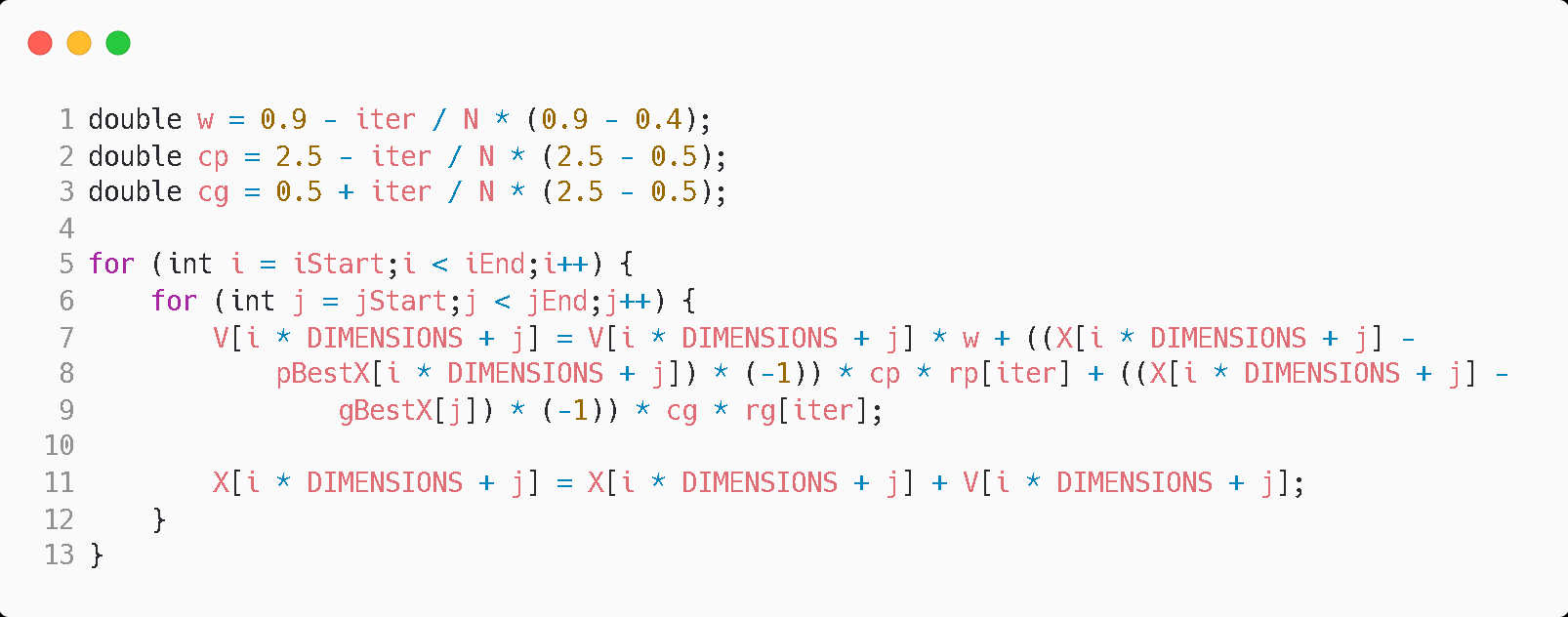
1. **Inicijalizacija**

U fazi inicijalizacije prolazimo kroz sve čestice i za svaku česticu izračunavamo vrednost funkcije u tački u kojoj se nalazi data čestica. Nakon toga , na 3. liniji, proveravamo da li je vrednost funkcije u toj čestici bolja, odnosno manja od globalne najbolje. Ukoliko je uslov ispunjen na linijama 4-6 postavljamo trenutnu česticu za globalno najbolju česticu.



1. **Ažuriranje pozicije i brzine pomoću formula**

U ovom delu implementacije algoritma pristupamo petlji koja se izvršava prethodno zadati broj puta (N). Na linijama 1-3 izračunavamo konstantu inertnosti, kognitivnu konstantu i socijalnu konstantu koje ažuriramo prilikom svake iteracije po pravilima koja smo definisali u uvodu. Nakon toga prolazimo kroz sve čestice i za svaku njenu dimenziju ažuriramo brzinu po formuli koja je takođe objašnjena i definisana u uvodnom poglavlju. Zatim se na liniji 11 postavlja nova vrednost čestice na osnovu njene prethodne kordinate i nove brzine.



1. **Ažuriranje ličnih i globalno najboljih pozicija**

Na kraju iteracije ažuriramo podatke ličnih i globalnih pozicija čestica. Za svaku poredimo njenu ličnu najbolju vrednost sa trenutnom vrednoscu, zatim biramo manju od dve i uzimamo je kao ličnu najbolju vrednost. Sličnu proveru radimo i za ažuriranje globalno najbolje vrednosti.



1. **Stajanje nakon zadatog broja iteracija**

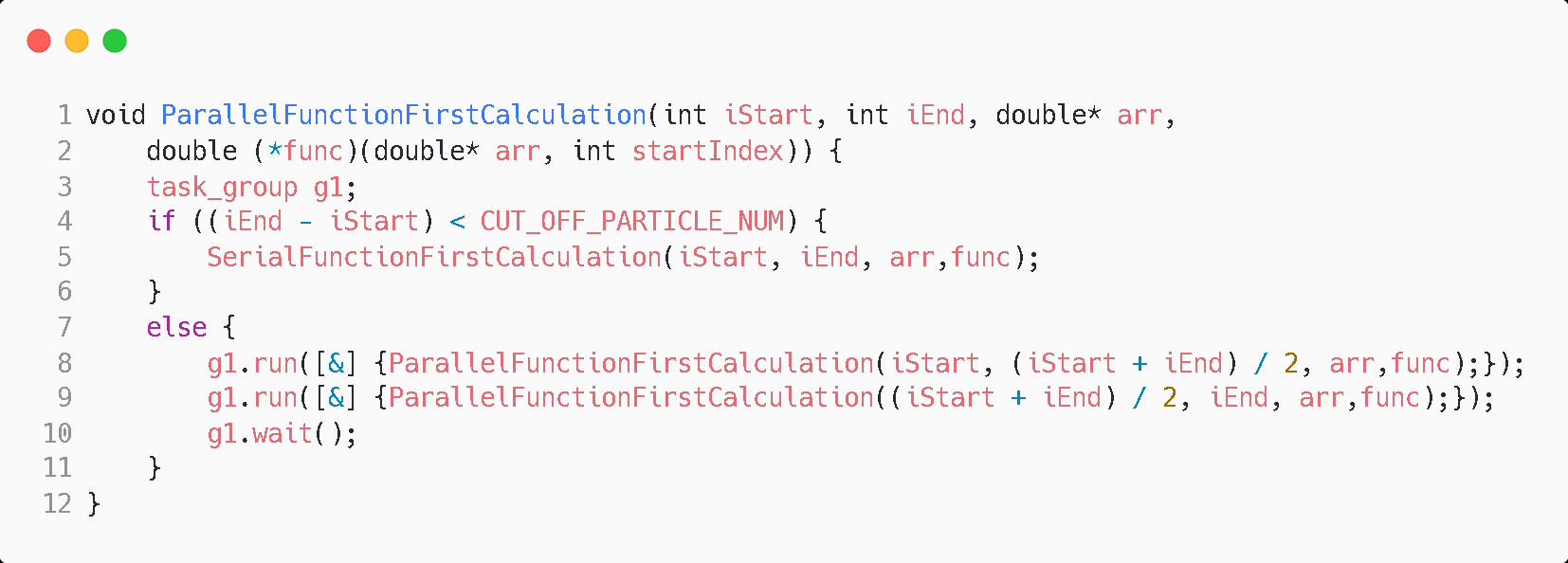
Kada iteriramo kroz petlju zadati broj puta trenutne globalne najbolje pozicije i vrednost funkcije u toj tački predstavlja, respektivno, tačku u kojoj se nalazi globalni minimum i vrednost globalnog minimuma. Važno je napomenuti da preciznost izračunavanja zavisi od zadatog broja iteracija od strane korisnika.

## **Paralelna Implementacija**

Za realizaciju paralelnih algoritama koristi´cemo Intel-ovu TBB biblioteku. Pri opisivanju generalnih karakteristika paralelnog PSO algoritma neophodno je definisati i cut-off čija je uloga da inicira prelazak u serijski režim ukoliko je interval iteratora dovoljno uzak. Za ovu implementaciju koristićemo cut-off koji je vezan za broj čestica (definisan kao CUT\_OFF\_PARTICLE\_NUM)

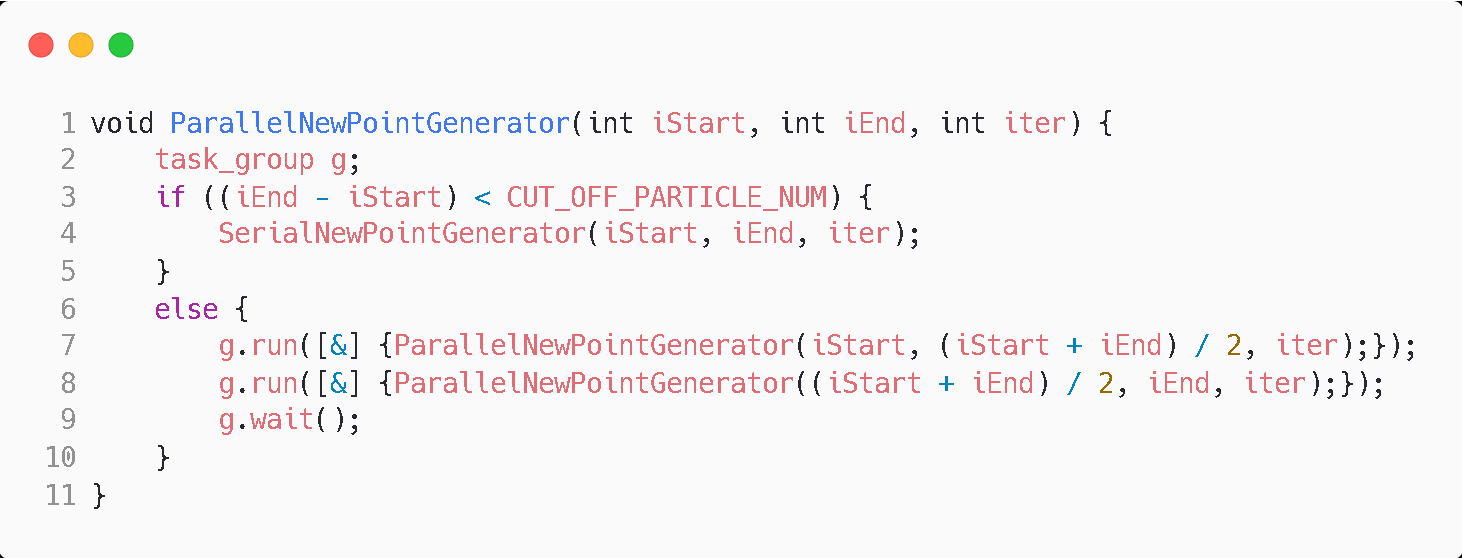
1. **Inicijalizacija**

U fazi inicijalizacije želimo da prvi put izračunamo vrednosti funkcija za svaku česticu i postavimo te vrednosti za personalno najbolje zato što su to i jedine vrednosti, kao i da postavimo globalno najbolju vrednost. Paralelnoj funkciji je potrebno da dostavimo širinu intervala niza koji treba da obrađujemo, niz u koji je to potrebno upisati i funkciju koju koristimo za računanje. Na 3. liniji smo definisali task\_group promenljivu, a nakon toga, na sledećoj liniji, proveravamo da li je interval dovoljno uzak da bi inicirao prelazak u serijski režim. Ukoliko taj uslov nije ispunjen, trenutni interval polovimo na dva dela i rekurzivno pozivamo istu funkciju sa novim intervalima.

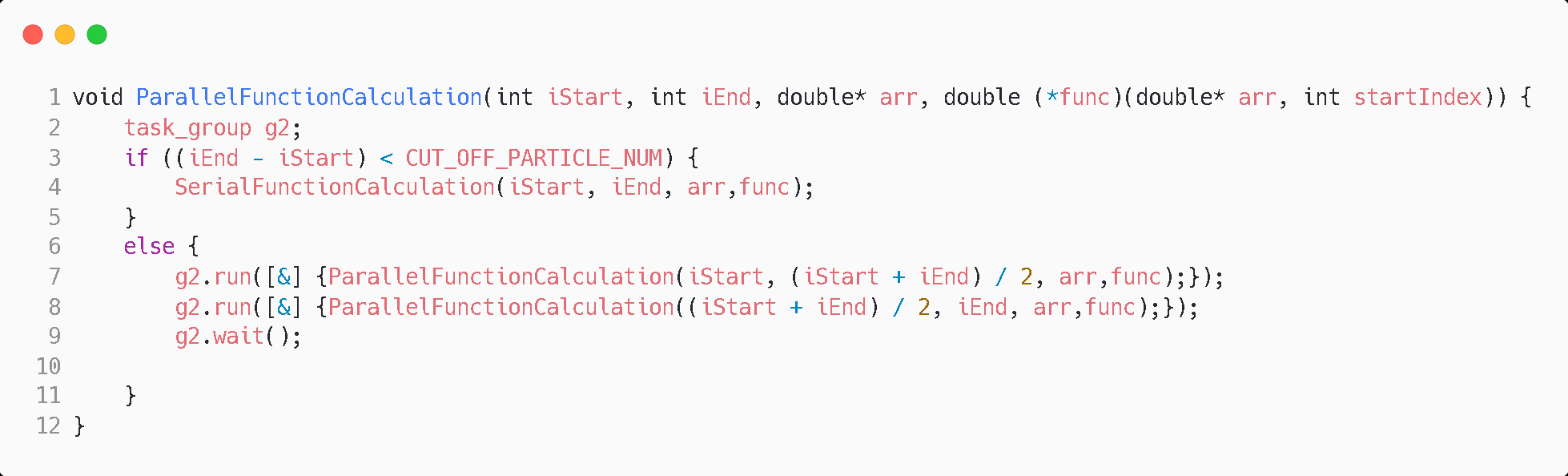


1. **Ažuriranje pozicije i brzine pomoću formula**

U drugoj fazi želimo da ažuriramo vrednosti brzine i koordinata. Paralelnoj funkciji je potrebno da dostavimo širinu (broj čestica) intervala koji treba da obrađujemo, kao i redni broj iteracije. Na 2. liniji smo definisali task\_group promenljivu, a nakon toga, na sledećoj liniji, proveravamo da li je interval dovoljno mali da bi inicirao prelazak u serijski režim. Ukoliko taj uslov nije ispunjen, trenutni interval delimo na dva podintervala nad kojima pozivamo istu funkciju.

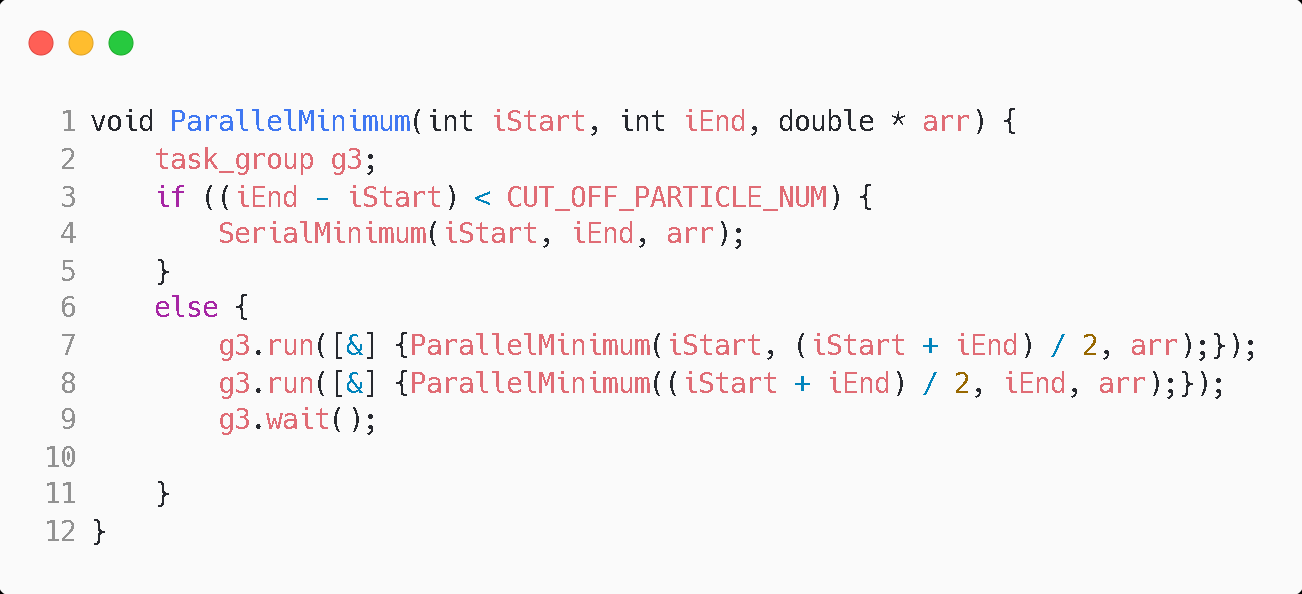


Nakon izračunavanja novih pozicija za svaku česticu se ponovo računa vrednost funkcije u toj tački. Iako je ova paralelizacija najjednostavnija, ona nam donosi najveće ubrzanje jer je vremenski najzahtevnija. Parametri i način paralelizacije su identični kao i tokom prvog koraka, inicijalizacije.



1. **Ažuriranje ličnih i globalno najboljih pozicija**

Treća faza ima za cilj da, nakon izračunatih novih vrednosti i ispunjenih uslova, promeni vrednost personalno i globalno najboljih vrednosti. Paralelnoj funkciji je potrebno da dostavimo širinu (broj čestica) intervala koji treba da obrađujemo, kao i redni broj iteracije. Na 2. liniji smo definisali task\_group promenljivu, a nakon toga, na sledećoj liniji, proveravamo da li je interval dovoljno mali da bi inicirao prelazak u serijski režim. Ukoliko taj uslov nije ispunjen, trenutni interval delimo na dva podintervala nad kojima pozivamo istu funkciju.



1. **Stajanje nakon zadatog broja iteracija**

Algoritam se završava kada se 2. i 3. korak ponovo prethodno definisani broj puta. Kako vrednosti svake naredne iteracije zavise od vrednosti prethodnih iteracija, ne možemo da paralelizujemo izvršavanje izvan jedne iteracije.

# **Analiza Rezultata**

Ovaj algoritam je testiran na dvojezgrenom AMD Ryzen 3 procesoru druge generacije sa četiri thread-a. Da bi testovi bili validni sve podatke koje odredjujemo nasumičnim odabirom smo pre samog izvršavanja algoritma izračunali i koristićemo iste nasumične brojeve i tokom serijskog i tokom paralelnog izvršavanja algoritma. Prvo smo fiksirali broj iteracija na hiljadu, broj čestica koje će učestvovati u traženju minimuma na 12000 i koristili smo četvorodimenzionalnu funkciju, dok smo vrednost cut-offa menjali i empirijskom metodom smo dobili najbolju vrednost vremena izvršavanja za dati algortiam. Priloženi programski kod je modularan i svaki od navedenih parametara je moguće vrlo lako izmeniti i prilagoditi potrebama. Svaki rezultat prikazan na grafikonima meren je 3 puta i u grafikonima se nalazi medijan tih merenja zato što su se, povremeno, javljale anomalije prilikom izvršenja programa. U nastavku teksta prikazani su grafikoni sa dobijenim rezultatima.

*Slika 1: Vreme izvršavanja paralelnog programa sa 12000 čestica nad četvorodimenzionalnom funkcijom u zavisnosti od cut-off parametra*

*Slika 2: Vreme izvršavanja paralelnog programa sa 24000 čestica nad četvorodimenzionalnom funkcijom u zavisnosti od cut-off parametra*

*Slika 3: Vreme izvršavanja paralelnog programa sa 12000 čestica nad dvodimenzionalnom funkcijom u zavisnosti od cut-off parametra*

*Slika 4: Vreme izvršavanja paralelnog programa sa 24000 čestica nad dvodimenzionalnom funkcijom u zavisnosti od cut-off parametra*

Kao što se vidi sa datih grafikona, najbolje rezultate daju vrednosti cut-offa između 400 i 1000 kada dobijamo ubrzanje i do dva i po puta u odnosu na serijski algoritam. Za merenje vremena izvršavanja u odnosu na broj čestica koristićemo cut-off iz gore navedenog intervala, tačnije 400. Slede merenja vremena izvršavanja u odnosu na broj čestica.

*Slika 5: Vreme izvršavanja paralelnog programa u odnosu na brojčestica za, do 12000 čestica*

*Slika 6: Vreme izvršavanja paralelnog programa u odnosu na brojčestica za, do 24000 čestica*

Rezulate koje smo dobili se poklapaju za našim pretpostavkama, uz korišćenje cut-offa dobili smo u proseku 1.8 puta manje vreme izvršavanja za broj čestica koji je veći od 4000. Dok kod slučajeva sa malim brojem čestica primećujemo neznatno ubrzanje, a nekad čak i sporiji rad paralelnog programa u odnosu na serijski.

# **Zaključak**

Najentuzijastičniji deo PSO algoritma je stabilna topologija koja obezbeđuje **komunikaciju između čestica u roju** i pomaže bržem učenju svake jedinke u postizanju globalnog minimuma. Ovaj algoritam obavlja prilično dobar posao u potrazi za globalnim minimumom, međutim njegova tačnost nije zagarantovana i zato smo se, prilikom istraživanja ovog algoritma, sreli sa mišljenjima da PSO algoritam rezultuje najverovatnijim globalnim minimumom. Takođe, tokom istraživanja otkrili smo da „black-box“ optimizacije kao i PSO algoritam imaju potencijal i široku primenu u nauci.

Što se paralelizacije ovog programa tiče, u pravoj primeni ovog algoritma ona je često nužna. Danas je vreme jedan od najbitnijih faktora prilikom istraživanja, a paralelizacija doprinosi velikoj uštedi vremena uz adekvatnu primenu.

Jedan od „nedostataka“ naše implementacije paralelnog PSO algoritma je paralelizacija 2. koraka, ažuriranja brzine i koordinata , koja bi mogla dodatno da se proširi. Inicijalna ideja je bila da se matrica, koja predstavlja listu višedimenzionih čestica deli na četiri podintervala umesto na dva kako je implementirano u projektu. Razlog tome su naši testni slučajevi sa funkcijama čije dimenzije nisu prelazile 5 dimenzija i kalkulacije nisu bile previše složene, pa je zbog toga deljenje matrice na 4 dela bilo značajno usporilo izvršavanje algoritma jer smo ga nepotrebno opteretili velikom količinom zadataka koji se sastoje iz relativno jednostavnih kalkulacija. Podela matrice na 4 podintervala bi imala smisao samo kada bismo imali veliki broj čestica čije su dimenzije reda veličine nekoliko stotina.