Univerzitet u Novom Sadu

Fakultet tehničkih nauka

Dokumentacija za projektni zadatak

Student: Bogdanović Vukašin, SV09/2020

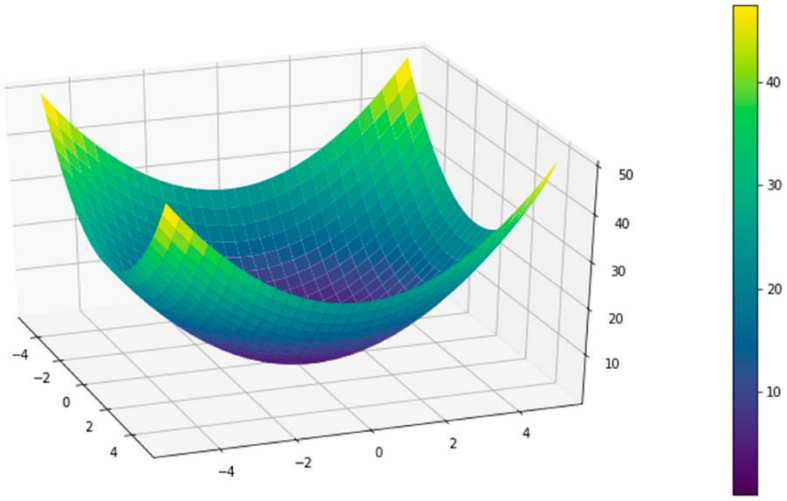
Predmet: Paralelno Programiranje

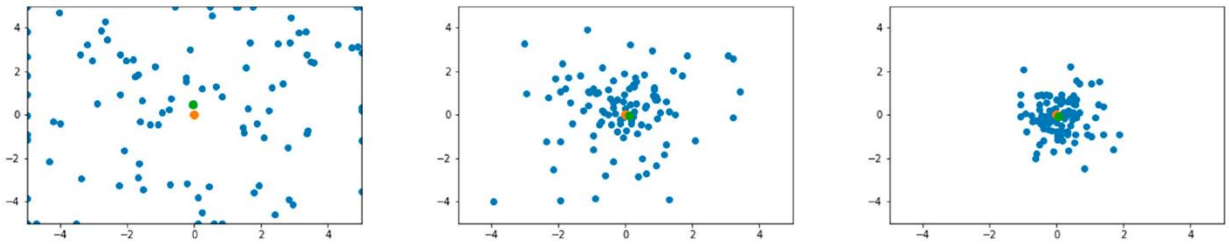
Tema projektnog zadatka: Paralelizacija PSO algoritama

Sadržaj

Uvod

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 $X^k(t)$, pri čemu $X^k(t)$ predstavlja tačku sa 60 koordinata.

$$X^k(t)=(X_1^k(t), X_2^k(t),..., X_{60}^k(t))$$

Analogno tome, brzinu ćemo označavati sa $V^k(t)$, i važiće

$$V^k(t)=(V_1^k(t), V_2^k(t),..., V_{60}^k(t))$$

Koordinate čestica u svakoj narednoj iteraciji ćemo računati po formuli

$$\mathbf{X^k(t+1)= X^k(t)+ V^k(t+1)}$$

Brzine čestica u svakoj sledećoj iteraciji ćemo računati po formuli

$$\mathbf{V^k(t+1) = w_k\, V^k(t) + c_p r_p(p_b - X^k(t)) + c_g r_g(g_b - X^k(t))}$$

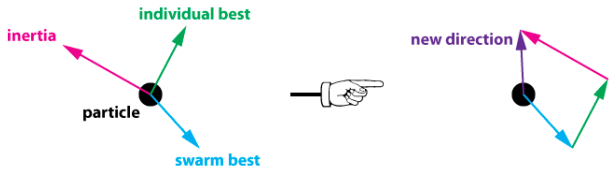
Iz jednačine za računanje brzine čestice u t+1 iteraciji vidimo da se ona sastoji od 3 komponente:

1. Inerciona komponenta - $w_k\, V^k(t)$
   Inerciona komponenta definiše stepen mobilnosti čestice, odnosno koliko je čestici lako da se kreće kroz prostor. Ova komponenta je definisana proizvodom:
   (a) Konstante inercije w_k koja uzima vrednosti od 0.9 do 0.4 po formuli
   $$w_k=0.9-k/N(0.9-0.4)$$
   N- ukupan broj iteracija, k- trenutna iteracija
   (b) Brzinom čestice k u prethodnoj iteraciji $V^k(t)$
2. Kognitivne komponente- $c_p r_p(p_b - X^k(t))$
   Kognitivna komponenta definiše koliko će se svaka čestica oslanjati na sopstveno iskustvo. Ova komponenta je definisana proizvodom:
   (a) Konstanti ubrzaznja c_p i r_p, pri čemu c_p uzima vrednosti od 2.5 do 0.5, a r_p uzima slučajnu vrednost u opsegu od 0 do 1 pri čemu se njena vrednost menja prilikom svake iteracije
   (b) Trenutne udaljenosti od lične najbolje pozicije p_b

3. Socijalne komponente komponente- $c_g r_g(g_b - X^k(t))$
   Socijalna komponenta definiše koliko će se svaka čestica „slušati“ iskustvo drugih čestica iz roja. Ova komponenta je definisana proizvodom:
   (c) Konstanti ubrzaznja c_g i r_g, pri čemu c_p uzima vrednosti od 0.5 do 2.5, a r_g uzima slučajnu vrednost u opsegu od 0 do 1 pri čemu se njena vrednost menja prilikom svake iteracije
   (d) Trenutne udaljenosti od globalne najbolje pozicije g_b

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.

```
1  for (int i = 0;i < PARTICLE_NUM;i++) {
2          for (int j = 0; j < DIMENSIONS; j++)
3          {
4              double x = (double)(rand()) / ((double)(RAND_MAX));
5              double v = (double)(rand()) / ((double)(RAND_MAX));
6              XStart[i * DIMENSIONS + j] = x;
7              VStart[i * DIMENSIONS + j] = v;
8              pBestXStart[i * DIMENSIONS + j] = x;
9          }
10     }
11     for (int i = 0; i < N; i++)
12     {
13         rp[i] = (double)(rand()) / ((double)(RAND_MAX));
14         rg[i] = (double)(rand()) / ((double)(RAND_MAX));
15     }
```

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 for (int i = iStart;i < iEnd;i++) {
2     arr[i] = (func(X, i));
3     if (arr[i] < gBestValue) {
4         gBestValue = arr[i];
5         for (int k = 0;k < DIMENSIONS;k++) {
6             gBestX[k] = arr[i * DIMENSIONS + k];
7         }
8     }
9 }
```

2. 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  double w = 0.9 - iter / N * (0.9 - 0.4);
2  double cp = 2.5 - iter / N * (2.5 - 0.5);
3  double cg = 0.5 + iter / N * (2.5 - 0.5);
4
5  for (int i = iStart;i < iEnd;i++) {
6      for (int j = jStart;j < jEnd;j++) {
7          V[i * DIMENSIONS + j] = V[i * DIMENSIONS + j] * w + ((X[i * DIMENSIONS + j] -
8              pBestX[i * DIMENSIONS + j]) * (-1)) * cp * rp[iter] + ((X[i * DIMENSIONS + j] -
9                  gBestX[j]) * (-1)) * cg * rg[iter];
10
11         X[i * DIMENSIONS + j] = X[i * DIMENSIONS + j] + V[i * DIMENSIONS + j];
12     }
13 }
```

3. 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  for (int i = iStart;i < iEnd;i++) {
2          if (arr[i] > pBestValue[i])
3          {
4              pBestValue[i] = arr[i];
5              for (int j = 0; j < DIMENSIONS; j++)
6              {
7                  pBestX[i * DIMENSIONS + j] = X[i * DIMENSIONS + j];
8              }
9              if (gBestValue > pBestValue[i]) {
10                 gBestValue = pBestValue[i];
11                 for (int k = 0;k < DIMENSIONS;k++) {
12                     gBestX[k] = pBestX[i * DIMENSIONS + k];
13                 }
14             }
15         }
16
17     }
```

4. 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  void ParallelFunctionFirstCalculation(int iStart, int iEnd, double* arr,
2      double (*func)(double* arr, int startIndex)) {
3      task_group g1;
4      if ((iEnd - iStart) < CUT_OFF_PARTICLE_NUM) {
5          SerialFunctionFirstCalculation(iStart, iEnd, arr,func);
6      }
7      else {
8          g1.run([&] {ParallelFunctionFirstCalculation(iStart, (iStart + iEnd) / 2, arr,func);});
9          g1.run([&] {ParallelFunctionFirstCalculation((iStart + iEnd) / 2, iEnd, arr,func);});
10         g1.wait();
11     }
12 }
```

2. **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.

```
1  void ParallelNewPointGenerator(int iStart, int iEnd, int iter) {
2      task_group g;
3      if ((iEnd - iStart) < CUT_OFF_PARTICLE_NUM) {
4          SerialNewPointGenerator(iStart, iEnd, iter);
5      }
6      else {
7          g.run([&] {ParallelNewPointGenerator(iStart, (iStart + iEnd) / 2, iter);});
8          g.run([&] {ParallelNewPointGenerator((iStart + iEnd) / 2, iEnd, iter);});
9          g.wait();
10     }
11 }
```

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 void ParallelFunctionCalculation(int iStart, int iEnd, double* arr, double (*func)(double* arr, int startIndex)) {
2     task_group g2;
3     if ((iEnd - iStart) < CUT_OFF_PARTICLE_NUM) {
4         SerialFunctionCalculation(iStart, iEnd, arr,func);
5     }
6     else {
7         g2.run([&] {ParallelFunctionCalculation(iStart, (iStart + iEnd) / 2, arr,func);});
8         g2.run([&] {ParallelFunctionCalculation((iStart + iEnd) / 2, iEnd, arr,func);});
9         g2.wait();
10
11    }
12 }
```

3. 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 void ParallelMinimum(int iStart, int iEnd, double * arr) {
2     task_group g3;
3     if ((iEnd - iStart) < CUT_OFF_PARTICLE_NUM) {
4         SerialMinimum(iStart, iEnd, arr);
5     }
6     else {
7         g3.run([&] {ParallelMinimum(iStart, (iStart + iEnd) / 2, arr);});
8         g3.run([&] {ParallelMinimum((iStart + iEnd) / 2, iEnd, arr);});
9         g3.wait();
10
11    }
12 }
```

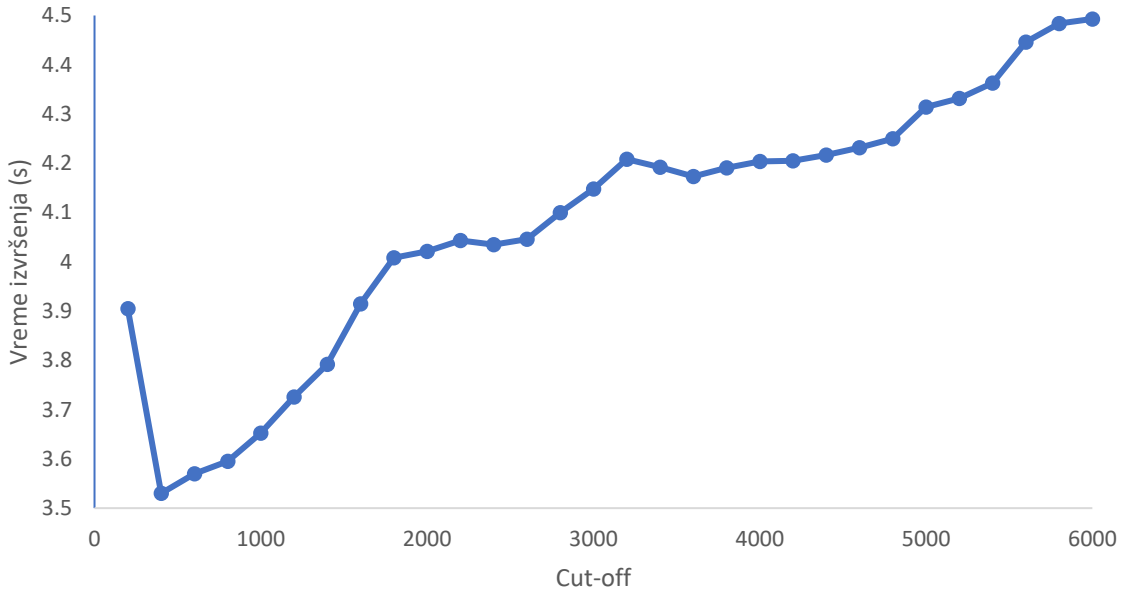
4. Stajanje nakon zadatog broja iteracija

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.

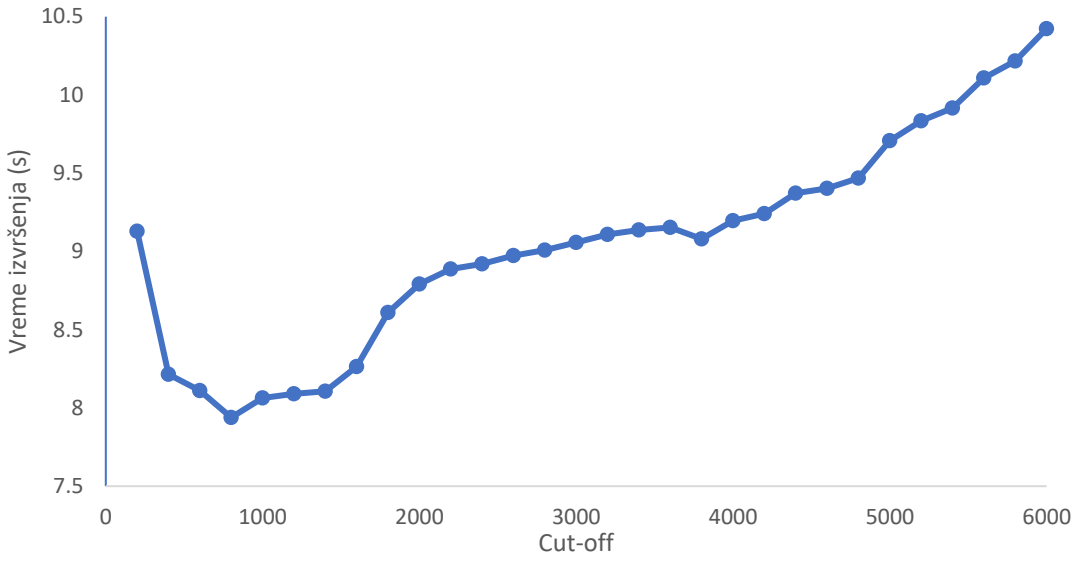
Analiza Rezultata

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.

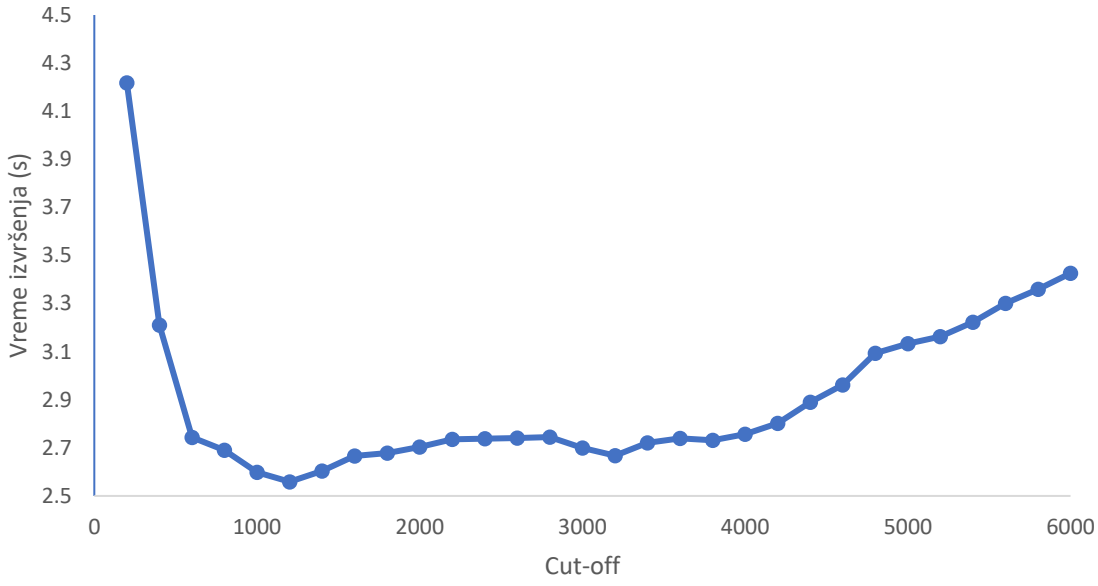
Slika 1: Vreme izvršavanja paralelnog programa sa 12000 čestica nad četvorodimenzionalnom funkcijom u zavisnosti od cut-off parametra



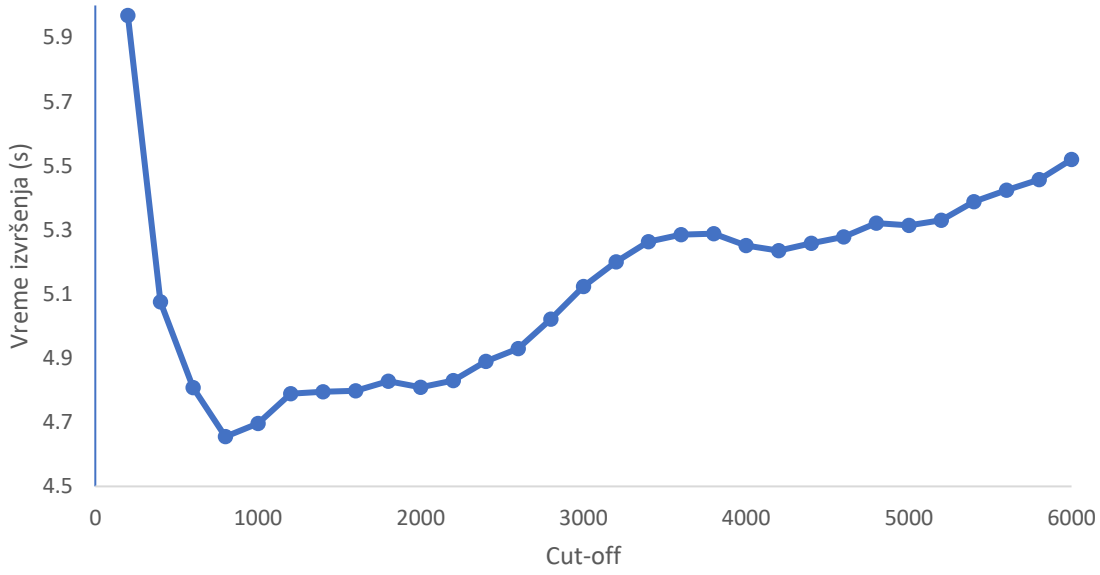
Slika 2: Vreme izvršavanja paralelnog programa sa 24000 čestica nad četvorodimenzionalnom funkcijom u zavisnosti od cut-off parametra



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

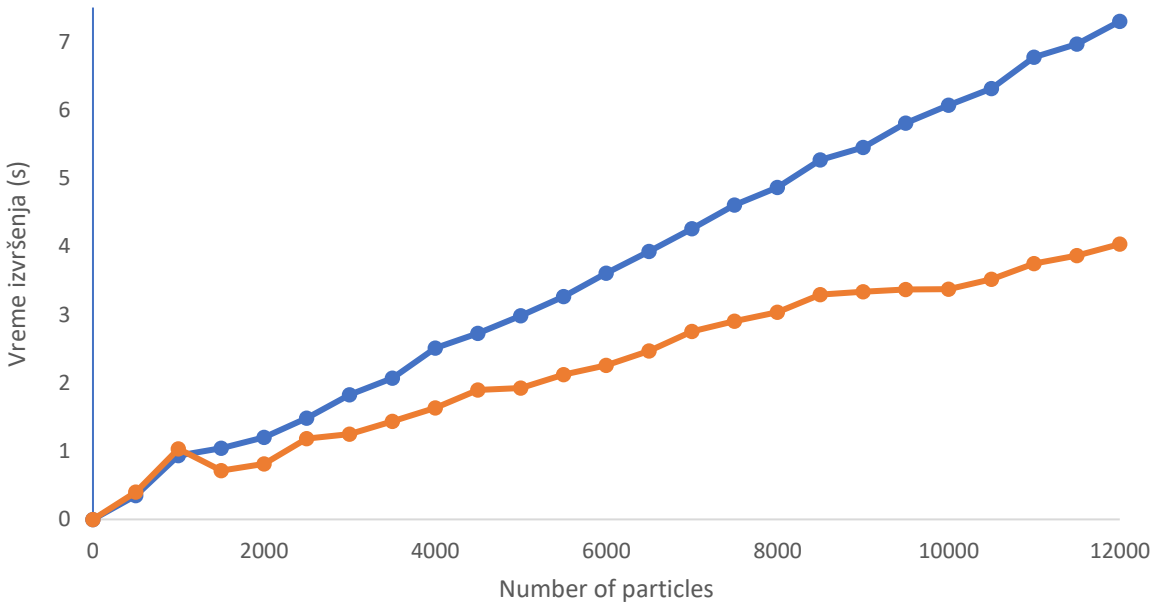


Slika 4: Vreme izvršavanja paralelnog programa sa 24000 čestica nad dvodimenzionalnom funkcijom u zavisnosti od cut-off parametra

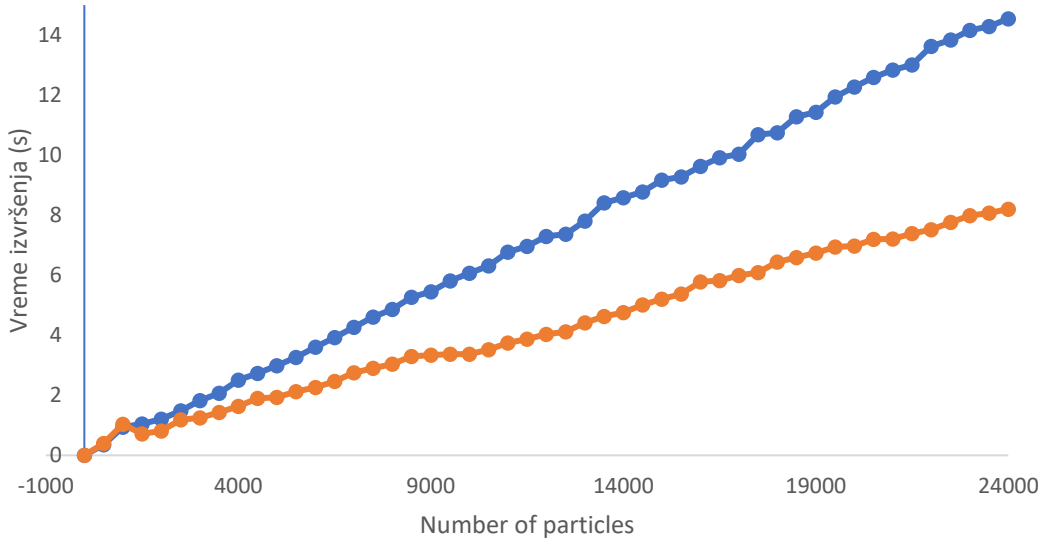


Kao što se vidi sa datih grafikona, najbolje rezultate daju vrednosti cut-offa između 400 i 1000 kada dobijamo ubrzanje i do dva i po puta u odnosu na serijski algoritam. Za merenje vremena izvršavanja u odnosu na broj čestica koristićemo cut-off iz gore navedenog intervala, tačnije 400. Slede merenja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