SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1988

Automatska paralelizacija uporabom genetskog programiranja

Ivo Majić

Zagreb, srpanj 2011.

Zahvaljujem se svojoj obitelji na potpori i razumijevanju u periodu izrade ovog rada te općenito na potpori tijekom studija.
Također bih zahvalio mentoru Doc. dr. sc. Domagoju Jakobovića na pruženoj pomoći u nabavci literature te savjetima.

SADRŽAJ

1. Uvod

Uobičajeno se računalni programi pišu za slijednu obradu. Tako napisani programi izvršavaju se na centralnom procesoru jednog računala. Samo jedna instrukcija se može izvršiti u jednom periodu vremena, a nakon njenog izvršenja sljedeća. Paralelno programiranje s druge strane paralelno izvodi više neovisnih instrukcija. Izvođenjem dijelova algoritma na takav način možemo postići značajna poboljšanja u brzini izvršavanja algoritma.

U ovom radu se korištenjem genetskog programiranja nastoji iz algoritma pisanog za slijednu obradu dobiti istovjetni paralelni algoritam. Kako kombinacija u pogledu koje grupe instrukcija možemo izvoditi paralelno, a pritom dobiti najbržu (ali potpuno ispravnu) izvedbu ima mnogo, genetsko programiranje je odabrano kao način rješavanja navedenog problema.

Nakon uvodnog poglavlja slijedi poglavlje koje ukratko opisuje temeljnu ideju paralelnog programiranja te kakva ubrzanja u izvođenju programa ono može donijeti. U trećem poglavlju opisuje se genetsko programiranje te najvažniji pojmovi vezani uz razumijevanje rada algoritma. U četvrtom poglavlju je objašnjena cjelokupna implementacija sustava izrađenog u okviru ovog rada. Objašnjeno je kako generiramo paralelne algoritme iz slijednih te kako ocjenjujemo njihovu ispravnost. U petom poglavlju je dana analiza rezultata primijene implementacije na slijedni algoritam za različite parametre pokretanja algoritma.

2. Paralelno programiranje

Paralelno programiranje je način oblikovanja programa i algoritama kod kojega se više instrukcija može obrađivati istovremeno (paralelno). Ideja se temelji na činjenici da je većinu problema moguće razložiti na više manjih neovisnih koje se zatim mogu neovisno paralelno izvršavati. U zadnjih nekoliko godina paralelna paradigma je postala jako popularan način ubrzanja računalnih programa, najviše zbog ubrzanog razvoja višejezgrenih procesora.
Jednu od najranijih klasifikacija računalnih sustava po načinu obrade podataka je osmislio Michael J. Flynn. Flynn je klasificirao programe i računala po tome koriste li jedan ili više skupova instrukcija te koriste li instrukcije jedan ili više skupova podataka 2.1.

Tablica 2.1: Flynnova kalsifikacija

	Jedna instrukcija	Više instrukcija
Jedan podatak	**SISD**	**MISD**
Više podataka	**SIMD**	**MIMD**

- **SISD** predstavlja Von Neumannov model računala s jednim procesorom i jednim memorijskim spremnikom. Jedna instrukcija obrađuje jedan podatak.
- **SIMD** arhitektura radi na principu da jedna instrukcija obrađuje više podataka istovremeno. Najčešće je izvedeno kao polje procesora koji izvode jednake instrukcije nad različitim podacima. Model je ograničen na one probleme koji zahtijevaju jednoliku obradu velikog skupa podataka (kao na primjer obrada slike).
- **MISD** arhitektura nije vrlo raširena jer su za svakodnevne primijene MIMD i SIMD najčešće mnogo korisnije. Koriste ju sustavi koji moraju biti otporni na greške (računala koja se koriste u svemirskoj industriji su MISD računala).
- **MIMD** arhitektura na više procesora izvodi različite instrukcije nad različitim podacima istodobno. Prednost je što je svaki procesor neovisan o drugima. Ovo je arhitektura za koju su namijenjeni paralelni algoritmi u ovom radu.

2.1. Amdahlov zakon

Teoretski gledano, ubrzanje dobiveno paralelizacijom polovice slijednih instrukcija trebalo bi prepoloviti vrijeme izvođenja, a paralelizacijom druge polovice također bi se vrijeme izvođenja trebalo opet prepoloviti. Amdahl je prvi 60-ih godina definirao zakon (2.1) kojim se može iskazati učinkovitost paralelnih sustava. [2]

$$ubrzanje = \frac{1}{1-P} \tag{2.1}$$

Amdahlov zakon uzima u obzir da se većina paralelnih izvedbi sastoje od paralelnih ali i malih slijednih dijelova koji ograničavaju ukupno ubrzanje algoritma. P pritom predstavlja paralelni udio programa, dok S predstavlja slijedni. U jednadžbu je moguće uključiti i broj procesnih jedinica na kojima je moguće paralelno izvođenje instrukcija (2.2)

$$ubrzanje = \frac{1}{S + \frac{P}{N_P}} \tag{2.2}$$

U tablici 2.2 je prikazan utjecaj broja procesnih jedinica N_P na ukupno ubrzanje uz različite vrijednosti udjela paralelnih instrukcija.

Tablica 2.2: Utjecaj N_P na ukupno ubrzanje

	P=50%	P=90%	P=99%
$N_P = 10$	1.82	5.26	9.17
$N_P = 100$	1.98	9.17	50.25
$N_P = 1000$	1.99	9.91	90.99
$N_P = 10000$	1.99	9.91	99.02

3. Genetsko programiranje

Genetsko programiranje pripada skupini evolucijskih algoritama te je jako srodno genetskom algoritmu. Cilj evolucijskih algoritama je rješavanje optimizacijskih problema koji bi klasičnim metodama pretraživanja bili teško rješivi. Svaki od algoritama iz te skupine svoju funkcionalnost temelji na nekom fenomenu iz prirode. Genetski algoritam svoju funkcionalnost temelji na Darwinovoj teorije evolucije vrsta koja se temelji na 5 pretpostavki:

- **Plodnost vrsta** - Potomaka uvijek ima više nego je potrebno.
- **Veličina populacije** - Populacija je uvijek približno stalne veličine.
- **Količina hrane** - Količina hrane je ograničena.
- **Varijacija jedinki** - Vrste koje se seksualno razmnožavaju nemaju identičnih jedinki nego postoje varijacije.
- **Nasljeđe** - Najveći dio varijacije se prenosi nasljeđem

Uzimajući u obzir ove pretpostavke jasno je da sve jedinke neće moći preživjeti jer je hrana ograničeni resurs. Opstati će samo najjače jedinke s najboljim karakteristikama koje će se zatim razmnožavati te prenijeti svoje karakteristike na potomke. Preslikavanjem ovog obrasca ponašanja na optimizacijske probleme dobivamo učinkovit način rješavanja problema čije bi rješavanje klasičnim metodama bilo dugotrajno. [4]
Dok jednostavni genetski algoritam koristi evoluciju kako bi pronašao optimalno rješenje, genetsko programiranje evoluira programe tj. načine kako dobiti najbolje rješenje. Postoje dva načina izvođenja algoritma:

- **Eliminacijski (engl. *steady-state*)**
- **Generacijski**

U nastavku (Algoritam 1) je prikazana te objašnjena eliminacijska izvedba algoritma koja je korištena u ovom radu. Eliminacijski algoritam provodi evoluciju nad stalnom populacijom jedinki u odnosu na generacijski koji u svakoj iteraciji stvara novu populaciju.

Algoritam 1 Eliminacijski genetski algoritam

```
P = stvori_inicijalnu_populaciju(POPSIZE)
evaluacija_dobrote(P)
while uvjet_zaustavljanja_nije_zadovoljen do
    jedinka1 = selektiraj_iz(P)
    jedinka2 = selektiraj_iz(P)
    dijete = krizaj(jedinka1, jedinka2, CPROB)
    mutiraj(dijete, MPROB)
    evaluacija_dobrote(dijete)
    umetni dijete u P koristeći operator zamjene
end while
```

U početku algoritam stvara inicijalnu populaciju veličine POPSIZE te svakoj jedinki u novostvorenoj populaciji ocjenjuje dobrotu. Zatim algoritam kreće kroz iteracije pri čemu u svakoj bira dvije jedinke iz populacije nekim od načina selekcije te odabrane jedinke križa s određenom vjerojatnošću (CPROB). Rezultat je jedno ili više djece nad kojima se onda provodi mutacija s određenom (najčešće vrlo niskom) vjerojatnošću (MPROB). Tako stvorena djeca se odabranim operatorom zamijene ponovo umeću u populaciju. Algoritam ponavlja ovaj postupak sve dok uvjet zaustavljanja nije zadovoljen. Na kraju algoritam iz tako evoluirane populacije bira najbolju jedinku te nju vraća kao konačno rješenje.

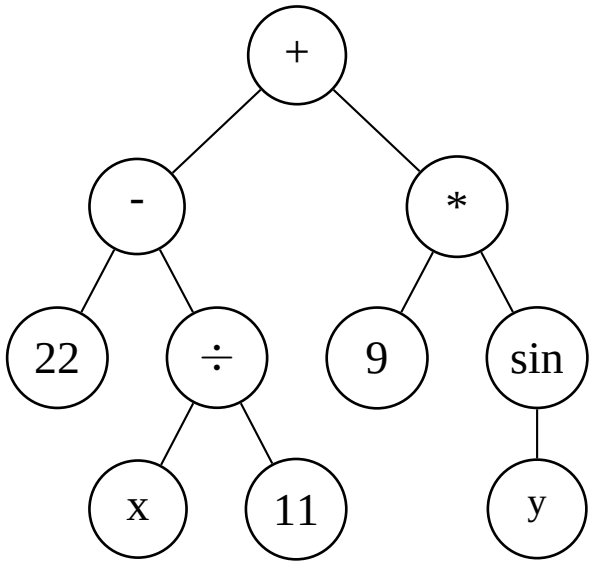
3.1. Prikaz jedinke

Genetsko programiranje koristi strukturu stabla za prikaz programa koji algoritam evoluira. [1] Stablo se u takvom prikazu sastoji od:

- **Funkcija** koje predstavljaju elemente koji će u stablu predstavljati unutarnje čvorove.
- **Terminala** koji će formirati listove ili krajnje točke (terminale) stabla.

Programi koje algoritam evoluira se najčešće prikazuju kao sintaksna stabla koja se generiraju tako da se rekurzivno za svaku podgranu stabla bira hoće li čvor biti funkcija ili terminal. Ako prikazujemo aritmetičke izraze kao u primjeru 3.1 funkcijske čvorove predstavljaju računske operacije (+, -, *, ÷) ili funkcije (cos, sin) dok terminali predstavljaju operande nad kojima te funkcije obavljaju svoje operacije.

$$(22 - \frac{x}{11})) + (9 * sin(y))$$



Slika 3.1: Primjer sintaksnog stabla

Pri stvaranju inicijalne populacije najčešće se koriste dvije metode generiranja koje stabla razvijaju do zadane maksimalne dubine:

- **Full** koja će svaku podgranu stabla izgenerirati do zadane maksimalne dubine tj. svi listovi stabla će se nalaziti na maksimalnoj dopuštenoj dubini stabla.
- **Grow** metoda generira stabla koja ne moraju biti izgenerirana do maksimalne dubine nego su nejednolika jer u svakom rekurzivnom koraku odabira čvora možemo odabrati i element iz skupine funkcije i element iz skupine terminala.

3.2. Genetski operatori

3.2.1. Selekcija

Selekcija jedinki se može raditi na dva načina:

- **Proporcionalna selekcija** Ova selekcija je poznata još kao ‘roulete-wheel selection‘. Ideja je da svakoj jedinki pridružimo vjerojatnost odabira pri čemu bolje jedinke imaju veću vjerojatnost dok lošije imaju manju vjerojatnost da će biti odabrane za križanje. Vjerojatnost odabira pojedine jedinke se računa prema formuli (3.1).

$$vjerojatnost(i) = \frac{dobrota(i)}{\sum_{j=1}^{n} dobrota(j)} \tag{3.1}$$

- **K-turnirska selekcija** Kod k-turnirske selekcije odabire se k nasumičnih jedinke iz populacije koje zatim sudjeluju u turniru. Zatim od tih k jedinke biramo najbolju (ili najgoru - ovisno o primjeni) za križanje. Ako trebamo n roditelja obaviti ćemo n turnira.

3.2.2. Križanje

Križanje je proces u koji kombinira genetske materijale dviju roditeljskih jedinki. Kako kod genetskog programiranja genetski materijal jedinki predstavljaju stabla, križanje se provodi na sljedeći način:

- Kod svakog roditelja odaberi jedan nasumični čvor.
- Izdvoji podstabla ispod odabranih čvorova
- Zamijeni izdvojena podstabla

Hoće li se križanje dviju jedinki stvarno i dogoditi određuje parametar CPROB (engl. *crossover probability*) koji određuje vjerojatnost križanja.

3.2.3. Mutacija

Mutacijom se u djecu uvodi novi genetski materijal koji može poboljšati dobrotu jedinke (ali i pogoršati). Kako je genetski materijal predstavljen stablom mutaciju se najčešće provodi na sljedeći način:

- Odaberi jedan slučajan čvor (ili list) u stablu.
- Odabrani čvor (ili list) i sva njegova podstabla se brišu.
- Na njegovom mjestu se slučajnim odabirom jednom od metoda generiranja stabla (*full* ili *grow*) stvara novo podstablo.

Vjerojatnost mutacije jedinke određena je parametrom MPROB (engl. *mutation probability*).

4. Programsko ostvarenje

Cilj implementacije je sustav koji može zadani slijedni algoritam preoblikovati u istovjetni paralelni. Pri tome se na originalni algoritam primjenjuju transformacije koje će rezultirati paralelnim algoritmom. Temelj tih transformacija je relacija (4.1)

$$SEQ(A, B) = PAR(A, B) \tag{4.1}$$

koja kaže da se dvije slijedne instrukcije A i B mogu izvesti paralelno ako između njih nema podatkovnih ovisnosti. Kako bi mogli odrediti postoji li podatkovna ovisnost između instrukcija A i B moramo na njih primijeniti Bernsteinove uvjete [3]. Neka U_A i U_B predstavljaju skupove ulaznih varijabli navedenih instrukcija te I_A i I_B skupove izlaznih varijabli navedenih instrukcija. Bernsteinovi uvjeti su zadovoljeni ako vrijedi relacija (4.2). Time je dokazano da između tih dviju instrukcija nema podatkovnih ovisnosti te se mogu izvesti paralelno.

$$(U_A \cap I_B) \cup (I_A \cap U_B) \cup (I_A \cap I_B) = \emptyset \tag{4.2}$$

Relacija (4.2) je naravno primjenjiva i na više instrukcija pri čemu se Bernsteinovi uvjeti onda moraju provjeriti međusobno za sve instrukcije. Kao primjer su navedene dvije instrukcije prikazane pod (4.3)

$$\begin{aligned} A &: a = (b + c) * 3 \\ B &: d = a * (d + 5) \end{aligned} \tag{4.3}$$

Njihovi odgovarajući skupovi ulaznih i izlaznih varijabli te rezultat provedbe relacije (4.2) su prikazani pod (4.4)

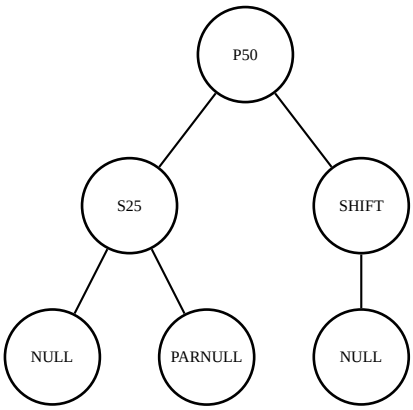
$$\begin{aligned} U_A &= \{b, c\} & I_A &= \{a\} \\ U_B &= \{a, d\} & I_B &= \{d\} \\ (U_A \cap I_B) &\cup (I_A \cap U_B) \cup (I_A \cap I_B) = \{a\} \end{aligned} \tag{4.4}$$

Kako rezultat provedbe relacije (4.2) nije prazan skup ove dvije instrukcije se ne mogu izvesti paralelno. Razlog tomu je što instrukcija A zapisuje u varijablu {a} dok instrukcija B iz nje čita. Bitno je naglasiti da se dvije instrukcije mogu izvoditi paralelno ako obje samo čitaju istu varijablu (no ne i ako zapisuju u nju).

4.1. Prikaz jedinke

Na početku poglavlja je definirano na koji način će sustav provjeravati mogu li se dvije ili više instrukcija izvoditi paralelno. Kako bi znali koje instrukcije će se izvoditi paralelno nad algoritmom ćemo primijeniti skup transformacija. Primjenom tih transformacija na slijedni algoritam dobivamo njegovu paralelnu verziju koju zatim treba ocijeniti po pitanju korektnosti i razine paralelizacije. Skup transformacija koje pritom koristimo je definiran u [3].

Redoslijed izvođenja tih transformacija ćemo prikazati pomoću strukture stabla koja se često koristi za prikaz jedinki u genetskom programiranju.



Slika 4.1: Primjer prikaza jedinke kao stablo tranformacija

Transformacije koje je moguće prikazati u stablu su podijeljene u četiri skupine pri čemu prve tri predstavljaju funkcije koje mogu biti čvorovi stabla dok zadnja skupina (NULL/PARNULL) predstavlja terminale (listove) stabla.

- PXX/SXX
- FXXX/LXXX
- SHIFT
- NULL/PARNULL

Transformacije se izvršavaju nad instrukcijskim segmentima tj. odsječkom algoritma koji transformacija trenutno obrađuje. Ovisno o vrsti transformacije odsječak (ili dio odsječka) će nakon obrade biti poslan na daljnju obradu sljedećoj transformaciji u stablu.

PXX/SXX transformacije dijele trenutni instrukcijski segment ovisno o postotku 'XX'. P transformacija će zatim ta dva nova dobivena odsječka izvesti u paraleli dok će ih S izvesti slijedno. U tablici 4.1 su prikazani primjeri izvođenja P40 i S60 transformacija nad slijednim segmentom [ABCDE]

Tablica 4.1: Primjer PXX/SXX tranformacije

Transformacija	Ulaz	Izlaz
P40	[ABCDE]	[AB]
		[CDE]
S60	[ABCDE]	[ABC] [DE]

LXXX/FXXX transformacije mogu doći u obliku FPAR/LPAR ili FSEQ/LSEQ pri čemu uzimaju početnu ili završnu instrukciju segmenta te ostatak šalju na daljnju obradu niz stablo sljedećoj transformaciji. Instrukcija koja je uzeta se izvodi paralelno ili slijedno s rezultatom ostalih transformacija nad navedenim segmentom koji je poslan da daljnju obradu.

Tablica 4.2: Primjer LXXX/FXXX tranformacije

Transformacija	Ulaz	Izlaz
FSEQ	[ABCDE]	[A] [BCDE]
LSEQ	[ABCDE]	[ABCD] [E]
FPAR	[ABCDE]	[A]
		[BCDE]
LPAR	[ABCDE]	[E]
		[ABCD]

SHIFT transformacija odgađa izvršavanje cjelokupnog segmenta za jedan vremenski korak. U primjeru u tablici 4.3 su prikazana dva segmenta [AB] i [CDE] koji se izvode paralelno nakon što nad [CDE] segmentom provede SHIFT transformacija.

Tablica 4.3: Primjer SHIFT tranformacije

	0	1	2	3
0	A	B		
1		C	D	E

Vrijeme

NULL/PARNULL su završne transformacije (listovi stabla). NULL transformacija će sve preostale instrukcije u segmentu izvesti slijedno dok će ih PARNULL sve izvesti paralelno.

4.2. Inicijalna populacija

Inicijalna populacija se stvara generiranjem određenog broja slučajnih jedinki nad kojima ćemo provesti genetski algoritam. Populacija se stvara tako da rekurzivno generiramo grane stabla (Algoritam 2) pri čemu izbor parametara maksimalne dubine stabla te način rasta (*grow* ili *full*) radimo prema *Ramped half-and-half* metodi.

Algoritam 2 GenerirajStablo(maksdubina, metoda)

if $maksdubina = 0$ **then**
  $root = random(terminal)$
  return
else if $metoda = full$ **then**
  $root = random(funkcija)$
  for all $grana$ **in** $root$ **do**
    $GenerirajStablo(maksdubina - 1, metoda)$
  end for
else if $metoda = grow$ **then**
  $root = random(funkcija, terminal)$
  if $root = funkcija$ **then**
    for all $grana$ **in** $root$ **do**
      $GenerirajStablo(maksdubina - 1, metoda)$
    end for
  end if
  return
end if

- **Ramped half-and-half** metoda radi na principu jednolikog generiranja stabala kako po metodi generiranja kako po dubini. Ako želimo dobiti populaciju od N jedinki onda ćemo ju podijeliti na N-1 dijelova te će svakom dijelu parametar maksimalne dubine biti zadan u cjelobrojnom intervalu $[2, N]$. Tako će prvi dio imati parametar $maksdubina = 2$, drugi dio $maksdubina = 3$, ...itd. Za svaku vrijednost $maksdubine$ pola stabala će biti generirano full metodom, a drugi dio grow metodom. Uporabom ove metode osigurana je raznolikost inicijalnog genetskog materijala jedinki koje algoritam evoluira.

Pri izboru funkcija koje će činiti čvorove stabla svaka funkcija ima jednaku vjerojatnost biti odabrana. Ako izbor padne na PXX/SXX transformaciju tablica 4.4 prikazuje vjerojatnosti izbora postotka 'XX' u kojem će transformacija dijeliti instrukcijski segment koji obrađuje. Najveću vjerojatnost odabira ima vrijednost 50 jer se i najmanji segment od dvije instrukcije može podijeliti na pola. Određene transformacije će svoj postotak imati slučajno odabran u intervalu $[1 - 99]$ kako bi se jedinke što bolje mogle prilagođavati segmentima koje obrađuju.

Tablica 4.4: Vjerojatnosti PXX/SXX transformacija

Vjerojatnost	Vrijednost
25%	50
15%	25
15%	75
15%	33
15%	66
15%	random(1,99)

4.3. Evaluacija dobrote jedinke

Na početku evaluacije dobrote jedinke ćemo svakoj instrukciji u slijednom algoritmu pridružiti vremenski korak koji određuje poredak u kojem će se instrukcije izvršavati. U tablici 4.5 je prikazan izgled algoritma koji se sastoji od pet instrukcija **[ABCDE]** prije nego se nad njime provedu transformacije zapisane u jedinki.

Tablica 4.5: Stanje algoritma na početku evaluacije dobrote

Instrukcija	A	B	C	D	E
Vremenska jedinica	1	2	3	4	5

Nakon provedbe transformacija neke instrukcije će se izvoditi u istom vremenskom koraku (paralelno) dok će druge biti izvedene slijedno prema redoslijedu vremenskih koraka. Takvom transformiranom algoritmu sada moramo ocijeniti dobrotu pri čemu ocjenjujemo dva faktora:

Korektnost Ocjena korektnost jedinke se određuje ispitivanjem podatkovnih ovisnosti instrukcija koje se izvode paralelno. Ispitivanje se provodi tako da se nad instrukcijama koje bi trebale biti izvedene u paraleli provedu Bernsteinovi uvjeti. Jedinka se kažnjava za svake dvije instrukcije koje ne zadovoljavaju navedene uvjete. Uz provjeru paralelnih instrukcija radi se i provjera korektnosti toka programa što znači da provjeravamo je li očuvan redoslijed instrukcija koje su podatkovno vezane. Cjelokupni postupak je opisan pod (Algoritam 3).

Paralelnost Ocjena paralelnosti se dobiva tako da odredimo broj vremenskih koraka potrebnih za izvođenje konačnog paralelnog algoritma. Pri tome se više instrukcija koje se izvode paralelno broje kao da se sve izvode u jednom vremenskom koraku. Na početku evaluacije dobrote prije nego primijenimo transformacije slijedni algoritam s N instrukcija ima ocjenu paralelnosti N (jer se svaka instrukcija slijedno izvodi u jednom vremenskom koraku). Idealni slučaj je jedinka koja ima ocjenu paralelnosti 1 što znači da se sve instrukcije u odsječku izvode paralelno u jednom koraku.

Pri usporedbi dobrote dviju jedinki najveću važnost ima korektnost jedinke jer ona osigurava da će paralelni algoritam imati istu funkcionalnost kao slijedni algoritam iz kojeg je generiran. Potom se u obzir uzima paralelnost algoritma pri čemu želimo da je ukupno vremensko trajanje (broj vremenskih jedinica u kojima se algoritam izvodi) što kraće.

Algoritam 3 Evaluacija dobrote jedinke - korektnost

Inicijalna korektnost
$korektnost = 0$

Provjera podatkovne neovisnosti paralelnih instrukcija
$grupiraj\_paralelno =$ **group all** $inst$ **in** $vrem\_tablica$ **by** $vrem\_korak$
for all $grupa$ **in** $grupiraj\_paralelno$ **do**
  $korektnost+ = bernstein\_check(grupa)$
end for

Provjera korektnosti redoslijeda instrukcija
for all $index, (instA, vrem\_korakA)$ **in** $vrem\_tablica$ **do**
  for all $instB, vrem\_korakB$ **in** $vrem\_tablica[index + 1 :]$ **do**
    if $vrem\_korakB < vrem\_korakA$ **then**
      $korektnost+ = bernstein\_check(instA, instB)$
    end if
  end for
end for

Pri evaluaciji korektnosti jedinke kako je prikazano u (Algoritam 3) prvo se utvrđuje ima li podatkovnih ovisnosti između instrukcija koje se izvode u istom vremenskom koraku ($vrem\_korak$). Sve instrukcije koje se izvode u istom vremenskom koraku šalju se na provjeru Bernsteinovih uvjeta ($bernstein\_check()$). Nakon toga se provjerava ispravnost programskog toka, tako da se za svaku instrukciju provjeri izvršavaju li se o njoj podatkovno ovisne instrukcije u ispravnom redoslijedu (kao u slijednoj verziji). Pritom se provjerava odnos vremenskog koraka tekuće instrukcije ($vrem\_korakA$) koja se provjerava i svih instrukcija koje u slijednoj verziji algoritma dolaze nakon nje ($vrem\_korakB$).

4.3.1. Primjer evaluacije jedinke

Za primjer će biti uzet odsječak koda s 5 slijednih instrukcija koji je prikazan pod 4.2 te su pojedine instrukcije označene slovima [ABCDE] radi lakšeg praćenja postupka transformacije.

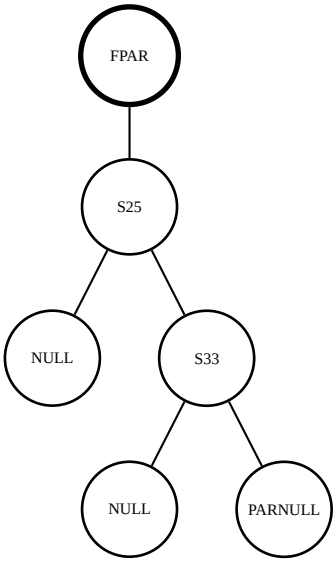
$$A : a = (3 + c) * a$$
$$B : d = c + 7$$
$$C : d = d * (a * (e - 4))$$
$$D : e = (d - 3) * 4$$
$$E : f = a * d$$

	0	1	2	3	4
0	A	B	C	D	E
1					
2					

Vrijeme

Slika 4.2: Odsječak s 5 slijednih instrukcija

Trenutni poredak izvršavanja instrukcija prikazujemo pomoću tablice na kojoj horizontalna os označava vremenske korake (poredak izvršavanja instrukcija) dok vertikalna os označava procesne jedinice. Kako će se postupkom transformacije neke instrukcije naći u istom vremenskom koraku svaka od njih će biti izvršena na zasebnoj procesnoj jedinici paralelno s ostalima u tom vremenskom koraku.
Za primjer je uzeto stablo transformacija koje bi trebalo generirati optimalno rješenje za ovaj odsječak.

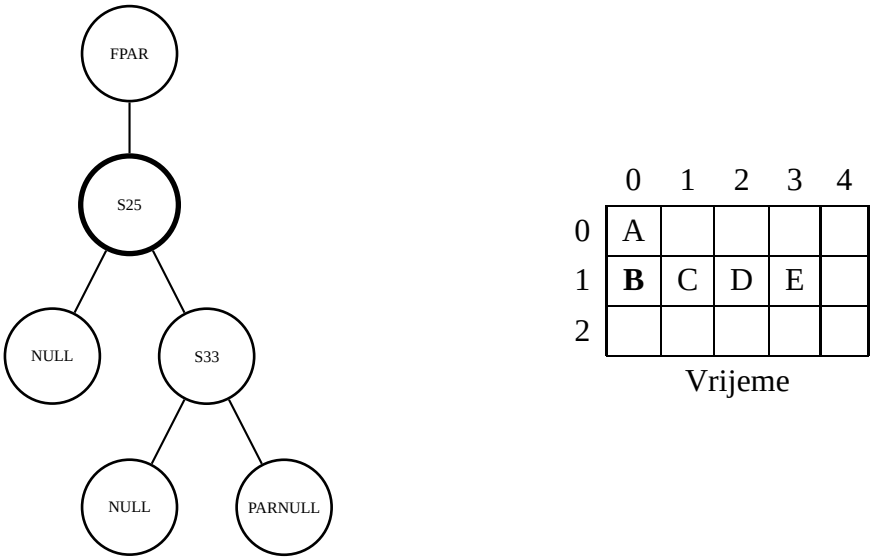


	0	1	2	3	4
0	A				
1	**B**	**C**	**D**	**E**	
2					

Vrijeme

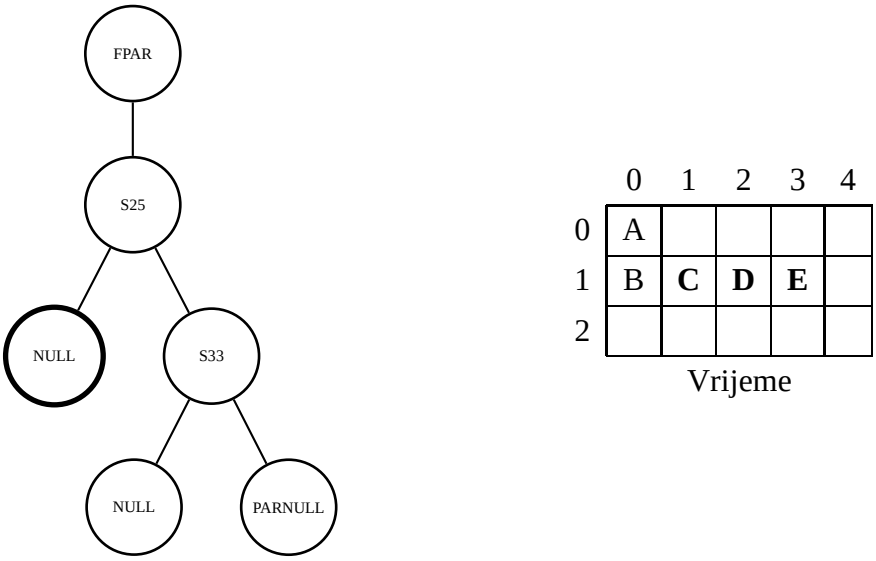
Slika 4.3: Korak 1: FPAR

Prva transformacija u stablu je FPAR transformacija koja uzima prvu instrukciju u segmentu [ABCDE] i izvodi je paralelno s ostatkom instrukcija u segmentu. Nakon transformacije ostatak instrukcija ([BCDE]) se šalje na daljnju obradu sljedećem čvoru.



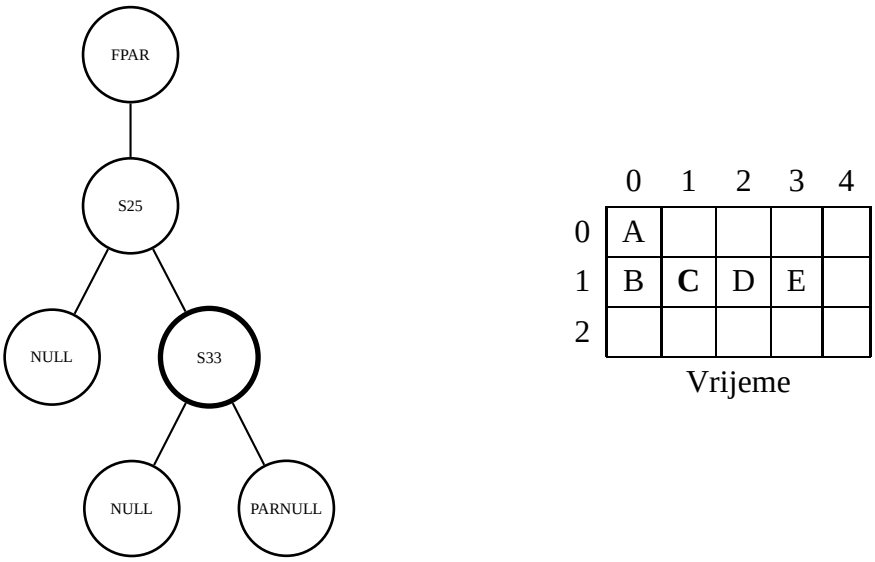
Slika 4.4: Korak 2: S25

Sljedeća transformacija je S25 transformacija koja uzima 25% segmenta te ga šalje lijevo kroz stablo na obradu sljedećem čvoru dok ostatak segmenta šalje desno kroz stablo. U ovom primjeru 25% segmenta je instrukcija [B] dok se ostatak [CDE] šalje desno niz stablo na daljnju obradu.



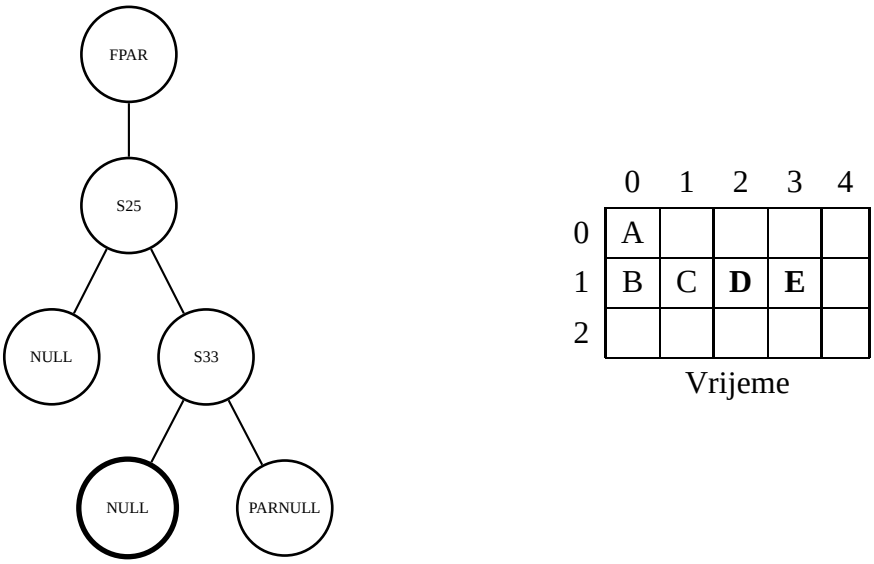
Slika 4.5: Korak 3: NULL

Slijedi NULL transformacija koja uzima sve instrukcije u segmentu koji obrađuje te ih izvršava slijedno. Kako NULL transformacija spada u terminale (listove) stabla ovdje obrada segmenta za taj dio stabla završava te se nastavlja s obradom segmenta [CDE] u desnoj podgrani stabla.



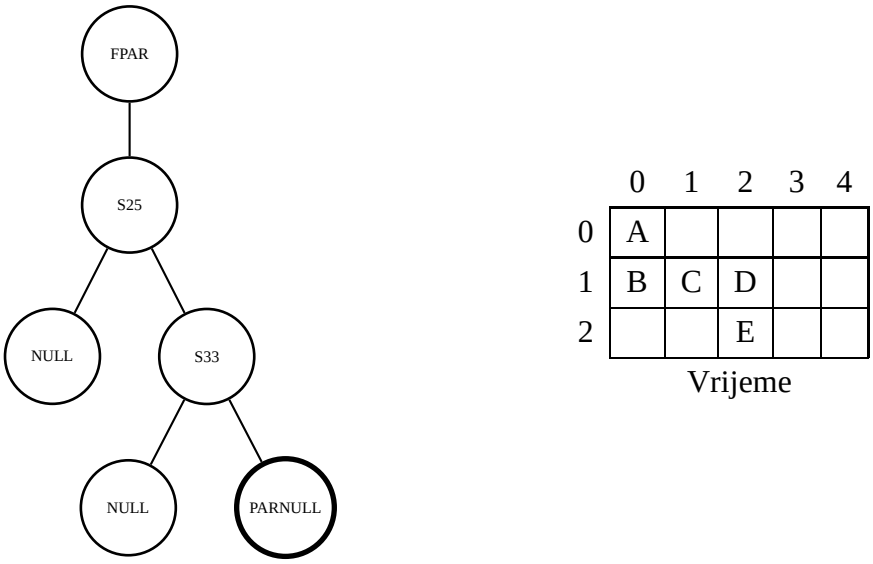
Slika 4.6: Korak 4: S33

Transformacija S33 uzima 33% segmenta koji obrađuje te ga šalje lijevo niz stablo na daljnju obradu dok ostatak šalje desno niz stablo. Kod segmenta [CDE] 33% je upravo instrukcija [C] te ju se šalje lijevo niz stablo.



Slika 4.7: Korak 5: NULL

Instrukcijski segment [C] dolazi do NULL transformacije te obrada za taj segment tu završava te se nastavlja s obradom segmenta [DE] u desnoj podgrani stabla.



Slika 4.8: Korak 6: PARNULL

Zadnja transformacija u stablu je PARNULL transformacija koja uzima sve instrukcije u segmentu i izvodi ih paralelno. Kako se trenutni segment sastoji od instrukcija [D] i [E] te dvije instrukcije će biti izvedene u istom vremenskom koraku. Ovime je završena transformacija početnog slijednog koda te tablica vremenskih koraka sad izgleda kao na 4.6

Tablica 4.6: Slijed izvršavanja instrukcija nakon provedbe transformacije

Instrukcija	A	B	C	D	E
Vremenska jedinica	0	0	1	2	2

Nakon što se provedu sve transformacije u stablu nad odsječkom pristupa se evaluaciji dobrote jedinke tj. ocjeni korektnosti i razine paralelizacije generiranog paralelnog odsječka.

Korektnost Ocjena korektnosti se određuje tako da se provedu Bernsteinovi uvjeti nad grupama instrukcija koje bi se trebale izvoditi paralelno. U ovo slučaju to su instrukcije [A,B] te [D,E]. Nakon toga ispituje se izvode li se instrukcije koje su međusobno podatkovno ovisne u ispravnog redoslijedu tj. redoslijedu u kojem su se izvodile i u originalnom slijednom programu. Kako u tim grupama instrukcija nema podatkovnih ovisnosti te je očuvan redoslijed podatkovno ovisnih instrukcija jedinka dobiva savršenu ocjenu korektnosti.

Paralelnost Ocjenu paralelnosti generiranog algoritma se dobiva prebrojavanjem broja vremenskih koraka algoritma. U ovom slučaju instrukcije [A,B] se izvode u jed-

nom koraku, [C] u drugom te [D,E] u trećem koraku. Dakle ocjena paralelizacije je 3.

Iako ukupnu dobrotu zatim možemo izraziti kao zbroj ovih dviju vrijednosti, prilikom odabira najbolje jedinke prvo ćemo populaciju sortirati po korektnosti te zatim po paralelnosti.

Nakon obavljanja ovih provjera rezultat je paralelni algoritam na slici 4.9. Sve instrukcije koje se nalaze unutar [PARBEGIN] i [PAREND] oznaka se izvode paralelno.

$$
\begin{aligned}
&[PARBEGIN] \\
&\quad a = (3 + c) * a \\
&\quad d = c + 7 \\
&[PAREND] \\
&d = d * (a * (e - 4)) \\
&[PARBEGIN] \\
&\quad e = (d - 3) * 4 \\
&\quad f = a * d \\
&[PAREND]
\end{aligned}
$$

Slika 4.9: Rezultirajući paralelni algoritam

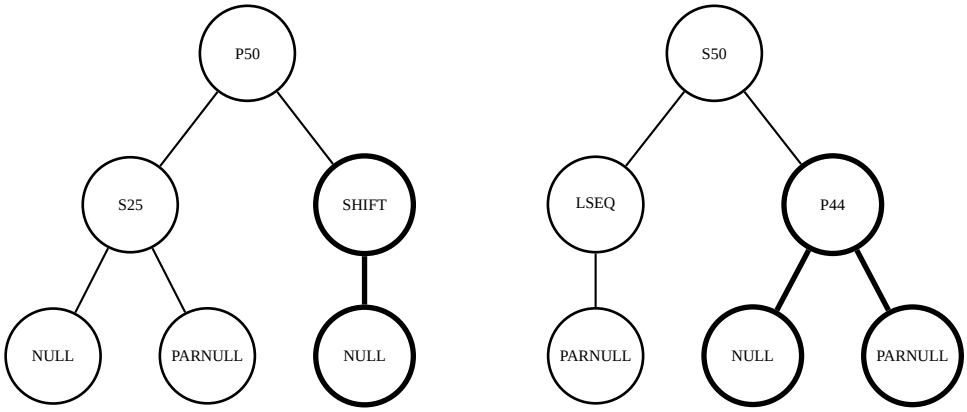
4.4. Genetski operatori

4.4.1. Selekcija

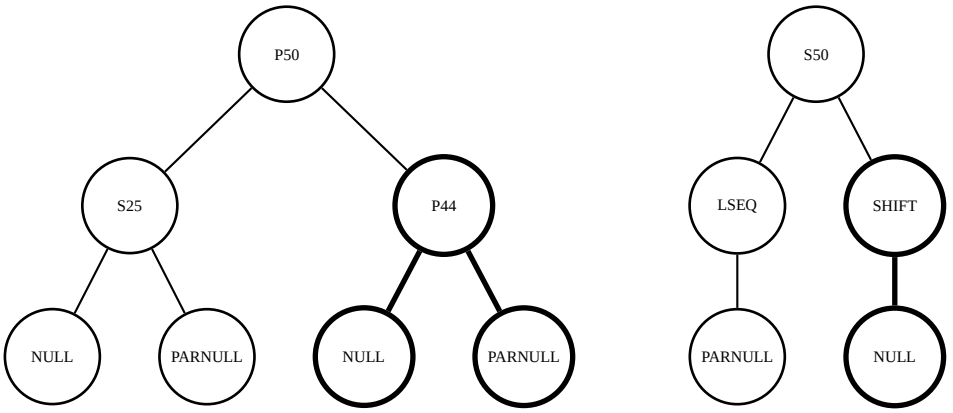
Za algoritam selekcije odabrana specijalan inačica k-turnirske selekcije pri kojoj u svakoj iteraciji iz populacije uzimamo 4 nasumične jedinke te provodimo turnir. Najbolje dvije jedinke će imati pravo na reprodukciju što će rezultirati s dvoje djece. To dvoje djece je zatim mutirano te dodano u novu populaciju umjesto jedinki koje su izgubile turnir. Postupak se ponavlja sve dok nije zadovoljen uvjet zaustavljanja.

4.4.2. Križanje

Križanje provodimo tako da odaberemo dva nasumična čvora u roditeljskim jedinkama te ih zamijenimo te time dobivamo dvoje djece koje nakon mutacije možemo dodati u populaciju. Na slici 4.10 vidimo roditeljske jedinke kojima je nasumično odabrana grana za provođenje križanja. Rezultat su dvije jedinke s obilježjima oba roditelja 4.11.



Slika 4.10: Roditeljske jedinke odabrane za križanje



Slika 4.11: Rezultat križanja

4.4.3. Mutacija

Mutacija se provodi na način da u stablu odaberemo jedan nasumični čvor te nad njim pokrenemo algoritam za generiranje stabla. Hoće li se mutacija nad nekim djetetom dogoditi ovisi o parametru $MPROB$ koji određuje vjerojatnost mutacije (u intervalu [0,1]).

4.5. Petlje i uvjetni izrazi

Kako se pravi algoritmi ne sastoje samo od jednostavnih instrukcija pridruživanja, potrebno je uvesti i neki način obrađivanja programskih petlji (npr. FOR petlje) te uvjetnih izraza (npr. IF). Programske petlje su i najzanimljivije jer njihovim paraleliziranjem se može dobiti i najveće ubrzanje algoritma. Kako se navedeni programski elementi mogu gnijezditi jedni u druge cijeli problem postaje još kompliciraniji.
Sustav opisan u ovom radu ovakvom problemu pristupa analizirajući razine instrukcija u algoritmu. U primjeru (Algoritam 4) se nalazi slijedni algoritam koji računa broj π.

Algoritam 4 Računanje broja π

$b\_tocaka = 100000$
$b\_tocaka\_krug = 0$
for b_tocaka **do**
  $R1 = slucajan\_broj(1, 2r)$
  $R2 = slucajan\_broj(1, 2r)$
  if $tocka(R1, R2)$ **in** $krug$ **then**
    b_tocaka_krug++
  end if
end for
$pi = 4 * (b\_tocaka\_krug/b\_tocaka)$

Navedeni algoritam se sastoji od tri razine instrukcija zbog postojanja FOR petlje te IF uvjetnog grananja unutar nje. Instrukcijske razine algoritma prikazane su pod 4.12.

$A : b\_tocka = 10000$
$B : b\_tocaka\_krug = 0$
$C : metaFOR$
$D : pi = 4 * (b\_tocaka\_krug/b\_tocaka)$

(Razina 1)

$A : R1 = slucajan\_broj(1, 2r)$
$B : R2 = slucajan\_broj(1, 2r)$
$C : metaIF$

(Razina 2 - metaFOR)

$A : b\_tocaka\_krug++$

(Razina 3 - metaIF)

Slika 4.12: Instrukcijske razine algoritma

Algoritam se izvodi nad svakom od razina te za svaku od razina vraća konačni paralelni raspored instrukcija. Bernsteinov zapis ulaznih i izlaznih varijabli kod meta-instrukcija sadrži podatke o svim instrukcijama unutar meta-instrukcije (čak i onih koje se nalaze u ugniježđenim elementima unutar meta-instrukcije).

$$I_{metaFOR} = \{R1, R2, b\_tocaka\_krug\}$$
$$U_{metaFOR} = \{b\_tocaka\_krug, b\_tocaka\}$$

Slika 4.13: metaFOR - ulazne i izlazne varijable

FOR petlje su tu od posebne važnosti jer se svaka njihova iteracija može izvoditi na posebnoj procesnoj jedinici. To će biti moguće ako su iteracije međusobno neovisne tj. trenutna iteracija ne čita ili zapisuje u podatak iz neke od prethodnih iteracija. Ova provjera FOR petlji se izvodi analitički pregledom ulaznih i izlaznih varijabli korištenih u petlji. Ako FOR petlja zadovoljava ovaj uvjet označava se kao $PARFOR$ petlja te se instrukcije unutar nje izvode slijedno (jer će se njezine iteracije ionako izvoditi paralelno).

Konačni paralelni algoritam je vidljiv pod (Algoritam 5).

Algoritam 5 Računanje broja π - paralelno

[PARBEGIN]
$b\_tocaka = 100000$
$b\_tocaka\_krug = 0$
[PAREND]
parfor b_tocaka **do**
  $R1 = slucajan\_broj(1, 2r)$
  $R2 = slucajan\_broj(1, 2r)$
  if $tocka(R1, R2)$ **in** $krug$ **then**
    b_tocaka_krug++
  endif
endparfor
$pi = 4 * (b\_tocaka\_krug/b\_tocaka)$

5. Rezultati

Kako bi se utvrdio utjecaj pojedinih parametara na razinu paralelizacije dobivenih paralelnih algoritama, uzet je programski odsječak koji se sastoji od 9 podatkovno neovisnih instrukcija. Idealna jedinka (tj. paralelna verzija algoritma) bi imala ocjenu paralelnosti 1, što znači da bi se sve instrukcije izvodile paralelno u jednom vremenskom koraku. Ocjena korektnosti će u ovom slučaju uvijek biti 0 jer su svih 9 instrukcija međusobno neovisne te ne može doći do podatkovnih ovisnosti. Kako bi implementaciji dodatno otežali pronalaženje idealne jedinke, izuzeta je $PARNULL$ transformacija. Kako su sve instrukcije međusobno neovisne algoritam bi s uključenom $PARNULL$ transformacijom težio tome da samo nju primijeni na zadani problem te time dobije idealnu jedinku.

	$[PARBEGIN]$
$a = a + 1$	$a = a + 1$
$b = b + 2$	$b = b + 2$
$c = c + 3$	$c = c + 3$
$d = d + 4$	$d = d + 4$
$e = e + 5$	$e = e + 5$
$f = f + 6$	$f = f + 6$
$g = g + 7$	$g = g + 7$
$h = h + 8$	$h = h + 8$
$i = i + 9$	$i = i + 9$
	$[PAREND]$
(Slijedno)	(Paralelno)

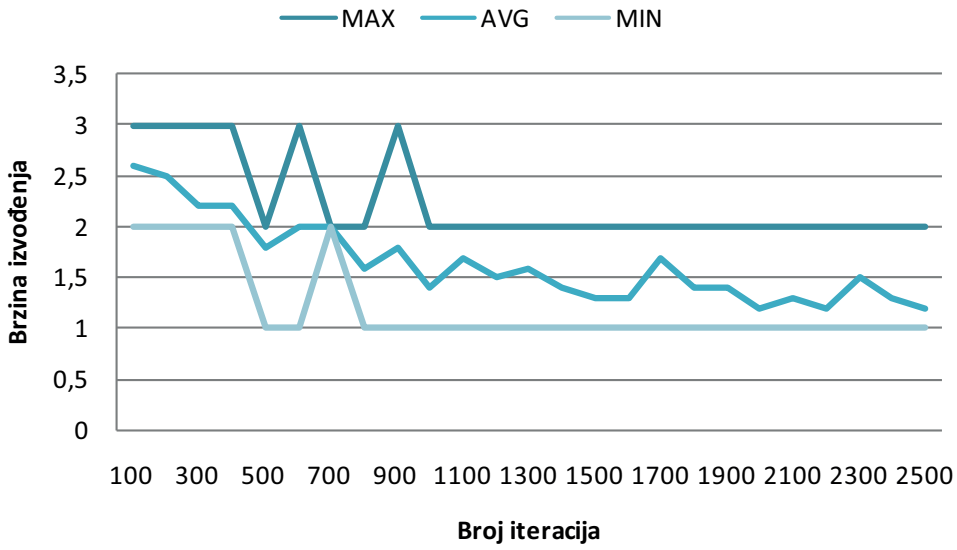
Slika 5.1: Odsječak sa 9 neovisnih instrukcija

Kako se u rezultatima želi prikazati utjecaj pojedinih parametara na ukupnu dobrotu jedinke (u ovom slučaju razinu paralelizacije) odabrani su inicijalni parametri pokretanja. Korišteni parametri su vidljivi u tablici 5.1

Tablica 5.1: Inicijalne vrijednosti parametara

Parametar	Broj iteracija	Populacija	Mutacija	Križanje	Dubina stabla
Vrijednost	600	200	0.2	0.9	9

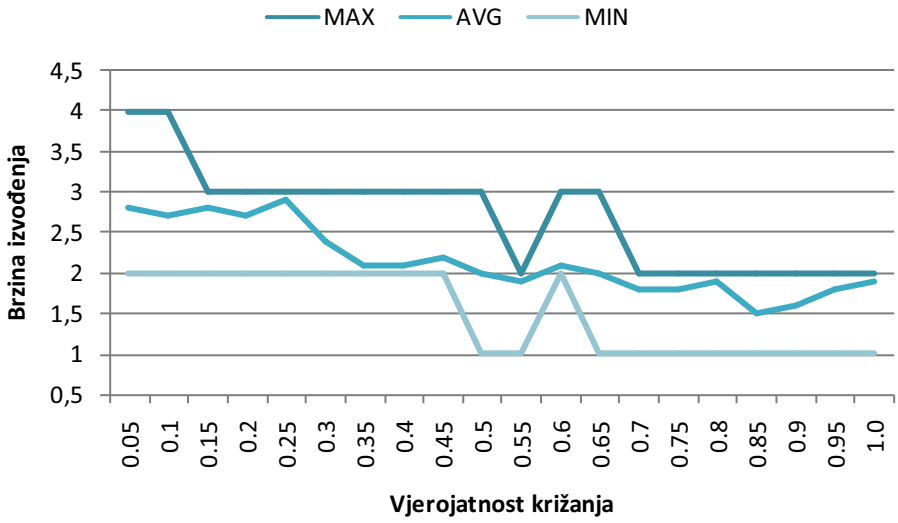
Utjecaj broja iteracija



Slika 5.2: Utjecaj broja iteracija

Graf na slici 5.2 prikazuje utjecaj broja iteracija na ukupnu dobrotu jedinke. Pri mjerenju je parametar mijenjan u intervalu $[100, 2500]$ s korakom 100. Vidljivo je da se povećanjem broja iteracija povećava dobrota jedinke tj. u ovom slučaju broj vremenskih jedinica potrebnih za izvođenje odsječka konvergira prema idealnoj vrijednosti od jednog vremenskog koraka. Rezultati su očekivani jer se uklanjanjem $PARNULL$ transformacije algoritmu značajno otežava pronalaženje idealne jedinke za mali broj iteracija. Za primjer, tijekom provođenja ispitivanja s uključenom $PARNULL$ transformacijom algoritam je već pri vrijednost 100 vraćao idealnu jedinku.

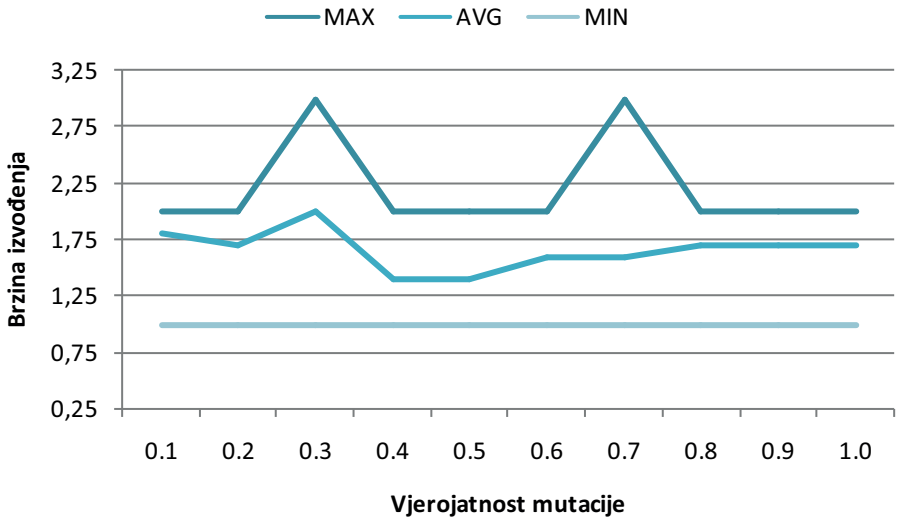
Utjecaj vjerojatnosti križanja



Slika 5.3: Utjecaj vjerojatnosti krizanja

Graf na slici 5.3 prikazuje utjecaj vjerojatnosti križanja dviju jedinki na razinu paralelizacije najbolje jedinke. Iz grafa je vidljivo da vjerojatnost križanja od $0.8 - 0.9$ daje u prosjeku najbolje rezultate. Ovakvi rezultati su očekivani zbog same prirode rada genetskog algoritma gdje je križanje jedinki jedan od temeljnih pokretača evolucije prema globalnom optimumu.

Utjecaj vjerojatnosti mutacije



Slika 5.4: Utjecaj vjerojatnosti mutacije

Graf na slici 5.4 prikazuje utjecaj vjerojatnosti mutacije jedinki na razinu paralelizacije najbolje jedinke. Kako se iz rezultata može zaključiti da algoritmu prosječno odgovara nešto veća vjerojatnost mutacije (oko $0.4 - 0.5$) to se može objasniti time da zbog uklanjanja $PARNULL$ transformacije algoritam češće ulazi u područja lokalnih optimuma. Većom vjerojatnosti mutacije jedinke algoritam pokušava zaobići takve lokalne optimume u potrazi za globalnim optimumom.

6. Zaključak

U ovom radu je prikazana primjena genetskog programiranja na automatsko generiranje paralelnih algoritama. Najveći problem pri stvaranju ovakvog sustava je bio odabir načina prikaza jedinke koji omogućava dinamičko određivanje redoslijeda izvršavanja instrukcija unutar algoritma. Uz navedeno jedinku je moralo biti moguće podvrgnuti standardnim genetskim operatorima koji se koriste u genetskom programiranju.

Pri evaluaciji dobrote jedinke trebalo je u obzir uzeti dva faktora - korektnost te razinu paralelizacije generiranog paralelnog algoritma. Potpuna korektnost je pri tome bila strogi uvjet za prihvaćanje jedinke kao najbolje, dok je ocjena razine paralelizacije (tj. teoretsko vrijeme potrebno za izvođenje algoritma) sekundarna.

Bitno je naglasiti da genetsko programiranje ne garantira pronalazak najboljeg rješenja, što je u ovoj implementaciji vidljivo u razlici razine paralelizacije generiranih algoritama. Iako ovaj sustav garantira vraćanje potpuno ispravnih paralelnih algoritama (zbog strogih zahtjeva pri odabiru najbolje jedinke), rezultati pokazuju da ispravan odabir parametara genetskog algoritma znatno utječe na razinu paralelizacije generiranih algoritama.

LITERATURA

[1] Wolfgang Banzhaf. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*. Morgan Kaufmann Publishers, 1998. ISBN 155860510X.

[2] Domagoj Jakobović. Paralelno programiranje - predavanja. Fakultet elektrotehnike i računarstva, 2011.

[3] Conor Ryan. *Automatic re-engineering of software using genetic programming*. Springer, 2000. ISBN 0792386531.

[4] Marko Čupić. Prirodom inspirirani optimizacijski algoritmi. Fakultet elektrotehnike i računarstva, 2009.

Automatska paralelizacija uporabom genetskog programiranja

Sažetak

U ovom radu je dan kratki uvod u paralelno programiranje i njegove prednosti. Zatim je opisan problem generiranja paralelnih algoritama iz slijednih korištenjem genetskog programiranja. Implementiran je i opisan sustav koji je osmišljen za rješavanje tog problema (prikaz jedinke, primjena genetskih operatora). Implementacija omogućava jednostavan unos slijednih algoritama, parametara genetskog algoritma te ispis rezultata. Prikazan je utjecaj različitih parametara na razinu paralelizacije dobivenih algoritama.

Ključne riječi: paralelni algoritmi, genetski algoritmi, automatska paralelizacija

Automatic parallelization with genetic programming

Abstract

In this thesis a brief introduction to parallel programming and its benefits is given. Afterwards the problem of generating parallel algorithms from existing sequential ones using genetic programming is explained. A genetic algorithm is implemented and described to solve this problem (chromosome representation, genetic operators). The implementation allows easy data input and result representation. Usage of different parameters and the parallelism level of the resulting algorithms is examined.

Keywords: parallel algorithms, genetic algorithms, automatic parallelization