SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 1361

# Višedretveni modeli evolucijskih algoritama

Roman Vazdar

Zagreb, lipanj 2010.

SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 1361

# Višedretveni modeli evolucijskih algoritama

Roman Vazdar

Zagreb, lipanj 2010.

## Sadržaj

# 1. Uvod

U ovom završnom radu obrađuje se tema višedretvene implementacije evolucijskih algoritama. Specifičnije, koriste se genetski algoritmi iz programskog okruženja ECF (eng. *Evolutionary Computation Framework*), a paraleliziraju se pomoću tehnologije OpenMP.

U sljedećem se poglavlju daje pojašnjenje genetskih i evolucijskih algoritama, arhitekture ECF-a. Nakon toga slijedi opis tehnologije OpenMP, kako se koristi, zbog kojih razloga te način primjene na evolucijske algoritme unutar ECF-a. Navedena implementacija OpenMP-a u algoritmima prikazuje se kasnije.

Nakon toga slijedi ispitivanje programskih rješenja. Ispituje se brzina samih paraleliziranih algoritama kako bi se vidjelo dolazi li do ubrzanja u izvođenju. Rezultati ispitivanja su dani u tablicama te pomoću njih možemo doći do zaključka oko korisnosti paralelizacije danih algoritama.

# 2. Evolucijski i genetski algoritmi

## 2.1. Evolucijski algoritmi

Evolucijski algoritmi su metode stohastičke pretrage koje slijede princip prirodne biološke evolucije. Funkcioniraju na populaciji potencijalnih rješenja primjenjujući načelo preživljavanja najboljih jedinki kako bi pružili bolju aproksimaciju rješenja nekog problema. Nakon svake generacije kreira se novi skup aproksimacija pomoću procesa odabira najboljih jedinki po kriteriju dobrote u domeni problema te njihovim međusobnim parenjem po principu iz prirodne genetike. Navedeni proces dovodi do evolucije populacije jedinki bolje prilagođenih okolini za razliku od prijašnjih jedinki iz kojih su nove jedinke stvorene, kao što je i slučaj u stvarnom svijetu [1] [3].

Evolucijski algoritmi modeliraju prirodne procese kao što su: selekcija, rekombinacija, mutacija, migracija, lokalnost. Prvu generaciju neke populacije čine slučajno generirana rješenja. Na slučajno generirana rješenja i na svo naknadno potomstvo se primjenjuje funkcija dobrote. U selekciji biramo roditelje sljedeće generacije rješenja s pristranošću prema boljoj dobroti. Roditelji reproduciraju potomstvo kopirajući njihove gene odabirom jedne od dviju metoda: križanje rekombinira gene roditelja, dok mutacija mijenja genotip jedinke slučajnim odabirom mogućih vrijednosti. Novonastale jedinke se natječu sa svojim roditeljima i ostalim jedinkama za mjesto u populaciji sljedeće generacije. Proces se ponavlja dok se ne nađe rješenje koje zadovoljava uvjete ili dok se ne dosegne limit pretrage.

Evolucijski algoritmi rade na populaciji jedinki te se stoga mogu izvršavati po paralelnom principu. Sastoje se od više različitih podtipova/klasa koje se razlikuju po implementacijskim detaljima i problemima koje rješavaju. Jedni od najpopularnijih tipova su genetski algoritmi.

## 2.1. Genetski algoritmi

Genetski algoritmi služe kao tehnika pretraživanja u računarstvu pri pronalaženju točnih ili aproksimiranih rješenja u optimizaciji ili problemima pretraživanja. Inspirirani su Darwinovom teorijom evolucije [1] [2].
Zajedničke karakteristike genetskih algoritama su biološki procesi koje simuliraju kao: selekcija, mutacija, križanje i nasljeđivanje.

Implementiraju se u računalnim simulacijama u kojima je populacija apstraktna reprezentacija (kromosomi ili genotipovi nekog genoma) potencijalnih rješenja (jedinke) nekom problemu optimizacije koji evoluira prema boljim rješenjima. Uobičajeno, rješenja se prikazuju pomoću nizova nula i jedinica, ali i druga kodiranja su moguća. Sama evolucija započinje od neke slučajno odabrane, početne populacije i događa se u generacijama.U svakoj generaciji uzima se dobrota (*engl. fitness*) svake jedinke (*engl. individual*) te se evaluira. Slijedi odabir tj. selekcija više jedinki po vrijednosti njihove dobrote iz trenutne populacije. Odabrane jedinke se križaju i po mogućnosti mutiraju kako bi se formirala nova populacija. Novoformirana populacija se koristi u sljedećoj iteraciji samog algoritma. Algoritam prestaje s radom nakon što dobrota neke populacije dosegne zadovoljavajuću vrijednost, ili nakon određenog broja generacija. Ako se dosegne broj generacija koje su određene za prestanak rada algoritma moguće je i dobivanje i nedobivanje zadovoljavajućeg rješenja. Navedeni opisani postupak rada genetskog algoritama je vrlo sličan postupku rada evolucijskog algoritma pošto je genetski algoritam podskup evolucijskog algoritma.

Genetski algoritmi nalaze primjenu u mnogobrojnim granama znanosti, kao što su bioinformatika, računarska znanost, matematika, fizika, ekonomija, kemija, itd.

## 2.2. Arhitektura i evolucijski algoritmi u ECF-u

### *2.2.1. Arhitektura ECF-a*

ECF- *Evolutionary Computation Framework* je programsko okruženje u C++ programskom jeziku za aplikacije bilo kojeg tipa u području evolucijskog računarstva [4]. Na slici 1. prikazan je klasni dijagram ECF-a.

Za mnoge postupke evolucijskog računanja potrebno je razvijati posebna programska rješenja, što se pokazuje izuzetno neučinkovitim te oduzima programerima i ostalim zainteresiranim stranama previše vremena.
Izrada programskog okruženja za postupke evolucijskog računanja rješava taj nedostatak na način da predstavlja neku vrstu "univerzalnog" alata koji pruža izrazitu fleksibilnost i mogućnost konstantne implementacije novih postupaka te korisnicima omogućava puno manje kontakta s izvornim kodom i jednostavniji način definiranja problema kojeg će programsko okruženje obraditi (npr. unos pretpostavljenih vrijednosti u konfiguracijsku datoteku).


**Slika 1. Klasni dijagram ECF programskog okruženja**

# 3. Tehnologija OpenMP

## 3.1. Opis OpenMP-a

OpenMP je aplikacijsko programsko sučelje (*engl. application programming interface*) sa svojstvima koja su bazirana na ciljevima izvršavanja paralelnih programa i aplikacija s dijeljenom memorijom.
Nije službeno sankcioniran standard nego je nastao kao sporazum članova ARB-a, koji dijele interes prema prenosivom, jednostavnom i efikasnom paralelnom programiranju. OpenMP je namijenjen implementaciji na SMP[1] arhitekturama. Osnovni opis se nalazi na Internetu [5]. Detaljan opis ideje, rada, i korištenja OpenMP-a se može naći u knjizi *Using OpenMP: Portable Shared Memory Parallel Programming* [7].

OpenMP nije novi programski jezik već vrsta notacije koja se dodaje sekvencijalnom programu u Fortranu, C-u ili C++-u, kako bi se opisao posao koji se dijeli između dretvi koje se izvode na različitim procesorima ili jezgrama i kako bi se moglo pristupiti dijeljenim podacima po potrebi. Pravilan način uključivanja OpenMP značajki u sekvencijalni program će omogućiti mnogim aplikacijama da se okoriste paralelno izvedenom arhitekturom dijeljene memorije, često uz minimalnu izmjenu teksta programa. U praksi mnoge se aplikacije mogu paralelizirati barem djelomično.

Sami uspjeh OpenMP-a se može pripisati snažnom naglasku na strukturno paralelno programiranje, jednostavnoj uporabi jer se detalji paralelizacije programa ostavljaju kompajleru umjesto samom programeru, te širokoj rasprostranjenosti i prihvaćenosti OpenMP-a, kako bi se OpenMP aplikacija mogla izvoditi na mnogim različitim platformama. Ali najvažnija karakteristika OpenMP-a je njegova budućnost. Naime, kako su SMP arhitektura i drugi

[1] Symmetric_multiprocessing – sklopovska arhitektura računala sa više procesora gdje su procesori spojeni na jednu dijeljenu glavnu memoriju i kontrolirani od strane jednog operacijskog sustava

višedretveni procesori u sve većoj proizvodnji i prodaji, potreba za programskim standardom dijeljene memorije koji je lagan za učenje i upotrebu je sve veća. Članovi koji nadgledaju razvoj OpenMP-a su proizvođači velikog dijela sklopovlja SMP arhitekture. Njihovo učešće u razvoju OpenMP-a jamči mogućnost njegove primjene na njihovim arhitekturama.

## 3.2. Ideja OpenMP-a

Kao što je već prije spomenuto, cilj OpenMP API-ija jest da omogući prenosivo paralelno programiranje po principu dijeljene ili zajedničke memorije. Također kreatori OpenMP-a su htjeli olakšati njegovu uporabu, primjenu i učenje. Dizajniran je da dopusti inkrementalno paraleliziranje postojećeg teksta programa, gdje se dijelovi programa paraleliziraju, po mogućnosti u sukcesivnim koracima, što je suprotno drugim paradigmama paralelnog programiranja koje zahtjevaju konverziju cijelog programa u jednom koraku.

Također se smatralo poželjnim da se omogući programeru rad sa jednim izvornim kodom, tj. ako jedan skup izvornih kodova sadrži kod za sekvencijalnu i paralelnu verziju programa tada je održavanje programa olakšano. Svi ti ciljevi i odluke u razvoju OpenMP-a su presudni na njegov današnji izgled.

OpenMP paralelizira program izvršavajući ga na različitim dretvama. Što je dretva? Dretva je entitet u izvođenju koji je u mogućnosti neovisno izvoditi niz instrukcija. OpenMP je namijenjen izvođenju programa sa skupom dretvi koje surađuju. Operacijski sustav kreira proces da izvrši program, daje dio sredstava tom procesu, stranice u memoriji i registre za vrijednosti objekata.

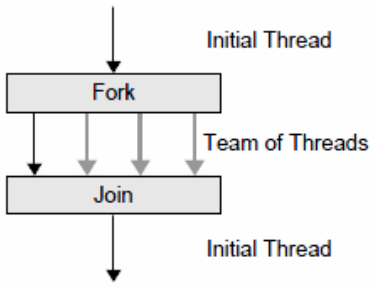
Ako više dretvi surađuju u izvedbi programa, dijeliti će sredstva, uključujući adresni prostor, procesa u kojem su sadržane. Dretva sama po sebi treba malo sredstava: programski brojač i prostor u memoriji za pohranu varijabli specifičnih za tu dretvu (uključujući registre i stog).

Više dretvi mogu biti izvođene na jednom procesoru ili jezgri pomoću zamjena konteksta[2].

Dretve koje se izvode na više procesora ili jezgri mogu raditi istovremeno kako bi se program izveo. Sami višedretveni programi mogu biti napisani na mnogo načina, te su u nekim načinima dopuštene kompleksne interakcije između dretvi.

OpenMP pruža *fork-join* programski model (vidi: Slika 2.). Specifičnije to je programski model gdje dretva na kojoj se početno izvršavao program stvara nove dretve kako bi paralelno izvršavao neki program. Dretva koja stvara nove dretve tj. prva dretva se zove *initial thread* ili *master thread*. Nove dretve se stvaraju kada se naiđe na paralelno područje tj. naredbu unutar OpenMP programa koja označava paralelno područje. Nakon završetka paralelnog područja, sve dretve osim inicijalne se terminiraju. Navedeni postupak je prikazan na slici 2. Svaki dio teksta programa koji je omeđen paralelnim naredbama se naziva paralelno područje.

Od programera se očekuje da zada specifikacije visokog nivoa paralelizma u programu i metode kojima će se to iskoristiti. Na programeru je da upotrijebi OpenMP notaciju tj. ključne riječi za označavanje paralelnih područja. Kao na primjer ključna riječ *parallel*. Ta ista OpenMP notacija ili sintaksa se sastoji od dodatnih ključnih riječi koje pobliže opisuju neko paralelno područje. Sam OpenMP obavlja stvaranje dretvi i pridruživanje posla tim dretvama kako bi se program izvodio paralelno.




**Slika 2. Programski model fork-join u OpenMP aplikaciji**

[2] Engl. *context switch* je računalni proces pohranjivanja i obnavljanja stanja (konteksta) procesora kako bi se izvršavanje nekog procesa u budućnosti moglo nastaviti tamo gdje je proces stao.

## 3.3. OpenMP opis rada

OpenMP API je sastavljen od skupa oznaka prevodioca, rutina biblioteka za izvođenje i varijabli okoline, kako bi se specificirao paralelizam na principu dijeljene memorije u Fortran i C/C++ programima. OpenMP oznaka je specijalno formatiran komentar ili *pragma* koja se primjenjuje na kod koji slijedi. Većina oznaka se primjenjuje na strukturirane blokove teksta programa koje imaju jedan ulaz i jedan izlaz u Fortran programima ili u slučaju C++ programa, gdje se oznake primjenju na izvršne izraze i naredbe (to također mogu biti naredbe sa jednim ulazom i izlazom). Program se ne smije razgranati unutar blokova teksta programa koji su ograničeni oznakama. To jest nije dopušteno koristiti *goto* naredbe koje „skaču“ van tog bloka teksta programa koji se paralelizirao, jer svaki dio teksta programa koji se paralelizira mora doći do svog kraja koji je najćešće implicitno naveden na kraju naredbe, najćešće parallel naredbe za podjelu poslova. U Fortran programima početak i kraj blokova teksta programa koje možemo paralelizirati se označavaju sa OpenMP oznakama, dok u C/C++ programima blokove teksta programa koje paraleliziramo označavamo samo sa početnom oznakom.

OpenMP korisniku pruža mogućnosti:

- kreiranja timova dretvi za paralelno izvršavanje,
- opisivanje dijeljenja posla između članova tima dretvi,
- deklariranje dijeljenih tj. zajedničkih i privatnih varijabli,
- sinkronizacija dretvi i davanje zasebnog zadatka pojedinoj dretvi.

Ove mogućnosti će pobliže biti opisane u daljnjem dijelu rada. Budući da unutar ovog rada koristimo OpenMP u C++ programskom jeziku pobliže će se opisati oznake samo za taj navedeni programski jezik.

### *3.3.1. Kreiranje timova dretvi*

Kako bi se program u nekom paralelnom području izvršavao paralelno stvara se tim ili skup dretvi. To se postiže tako da programer naznači početak paralelnog područja pomoću *parallel* oznake. Dodatne informacije mogu biti navedene pri pisanju *parallel* oznake. One obično daju informacije dretvama da li dijele neku varijablu ili da dio nekih podataka rezerviraju samo za sebe. Na kraju paralelnog područja je implicitna sinkronizacijska ograda. Ona označava lokaciju gdje dretva ili dretve koje su trenutno na toj lokaciji čekaju sve ostale dretve da ih dostignu. Nakon što sve dretve dođu na tu lokaciju program se nastavlja izvršavati, ali samo sa onom dretvom, ili onim dretvama koje su postojale prije samog paralelnog područja. Ako program unutar paralelnog područja naiđe na još jednu `parallel` oznaku, onda svaka aktivna dretva unutar tog paralelnog područja stvara svoj tim dretvi kojima postaje inicijalizacijska dretva. Time je podržano gniježđenje koje omogućuje višerazinsku paralelizaciju.

### *3.3.2. Dijeljenje poslova između dretvi*

Ako programer ne navede raspodjelu poslova između dretvi unutar nekog paralelnog područja, svaka od njih će izvršiti sav kod, što je redundantan posao te sprječava brže izvođenje programa. OpenMP oznake za dijeljenje poslova služe za opisivanje načina na koji se izvršavajući kod raspodjeljuje između dretvi. Dobar odabir metode za raspodjelu poslova može dati značajno ubrzanje izvođenja programa.

Najčešća upotreba raspodjele poslova jest unutar *for* petlji. Da bi se to postiglo programer umeće prikladne oznake odmah prije svake petlje unutar paralelnog područja koje se dijeli između dretvi. Mnogi programi veliki dio svojeg vremena provode unutar petlji vršeći računske operacije nad elementima nekog polja te je ova strategija razdjele poslova učinkovita i primjenjiva.

Sve OpenMP strategije za razdjelu poslova unutar petlji dodjeljuju neovisne skupove iteracija svakoj dretvi. Programer navodi metodu podjele tih skupova iteracija. Najizravnija strategija dodjeljuje „komad“ iteracija (engl. *chunk of iterations*) svakoj dretvi. Kompleksnije strategije dinamički računaju komade tj. skupove iteracija za svaku dretvu. Ako se ne navede metoda raspodjele koristi se implementacijski, unutar OpenMP-a definirana, strategija.

Za primjenu strategija raspodjele poslova unutar petlji, potrebno je unaprijed znati broj iteracija neke petlje. Zato recimo `while` petlja nije prikladna za upotrebu gore navedenih strategija. Također svaka iteracija mora biti nezavisna s obzirom na drugu, tj. redoslijed izvršavanja iteracija ne smije imati utjecaj na rezultat petlje. Postoji i mogućnost tzv. podatkovne ovisnosti (*engl. data dependence*) koja označava situaciju kada se vrijednost, koja se upisuje u jednoj iteraciji petlje, čita ili upisuje u sljedećoj iteraciji petlje.

### *3.3.3. OpenMP memorijski model*

OpenMP je zasnovan na dijeljenom memorijskom modelu, stoga se podrazumijeva da su podaci dijeljeni između dretvi i vidljivi svima. Ponekad su potrebne varijable koje imaju specifične vrijednosti za pojedinu dretvu. Ako svaka dretva ima vlastitu kopiju neke varijable, kako bi postojala zasebna vrijednost varijable za svaku dretvu, kažemo da je varijabla privatna (*engl. private*). Kada skup dretvi izvršava paralelnu petlju, svaka dretva treba vlastitu vrijednost iteracijske varijable. Ovaj slučaj je toliko važan da ga prevodioc sam primjenjuje, iako u drugim slučajevima programer mora sam odlučiti koje varijable su dijeljene, a koje privatne. Podaci općenito mogu biti deklarirani privatnima ili dijeljenima (zajedničkima) s obzirom na paralelno područje. Korištenje privatnih varijabli ima nekoliko prednosti: smanjuju frekvenciju ažuriranja dijeljene memorije, te smanjuju potrebu za sinkronizacijom i pristupom vanjskim podacima na cc-NUMA platformama (*engl. Cache Coherent Non-Uniform Memory Access*).

Još jedno svojstvo OpenMP-a jest *flush*, za sinkronizaciju memorije.

Operacija *flush* osigurava da dretva koja ju pozove ima istu vrijednost dijeljenih objekata podataka (npr. dijeljenih varijabli) kao i glavna dretva.
Stoga nove vrijednosti dijeljenih objekata ažurirane od strane neke dretve su zapisane u dijeljenoj memoriji, a i ta dretva dohvaća bilo koje nove vrijednosti proizvedene od strane drugih dretvi koje se čitaju iz dijeljene memorije.

### *3.3.4. Sinkronizacija dretvi*

Sinkroniziranje ili koordiniranje akcija (posla) dretvi je ponekad nužno kako bi se osigurao pravilan redoslijed njihovom pristupanju dijeljenim podacima i kako bi se spriječila nevaljanost podataka (*engl. data corruption*). OpenMP ima mali skup razumljivih sinkronizacijskih svojstava.
Kod programiranja sa dijeljenom memorijom najteže je postići dobru koordinaciju dretvi. Uobičajeno je kod OpenMP-a da se na kraju paralelnog područja ili dijela sa dijeljenom memorijom čeka da sve dretve završe sa svojim poslom. Sinkronizaciju podskupa dretvi u OpenMP programu je teže izvesti i zahtijeva pažnju pri programiranju jer nema eksplicitne potpore za tu vrstu problema.

Ponekad je potrebno unutar OpenMP programa osigurati da samo jedna dretva izvodi dio teksta programa u nekom vremenskom intervalu. Postoji više mehanizma koji to omogućuju. Na primjer, ako neka dretva A pokuša izvoditi dio teksta programa koji je zaštićen takvim mehanizmom, a već ga izvodi neka druga dretva B, onda će prije spomenuta dretva A morati čekati na završetak izvođenja teksta programa od strane dretve B. Alternativno, postoji mogućnost da dretva A obavlja neki drugi korisni posao dok čeka da dretva B završi.
Također, ako je potrebno zaštiti jednu varijablu tj. neku memorijsku lokaciju, učinkovitije je koristiti *atomic* oznaku unutar OpenMP-a, nego tu memorijsku lokaciju tj. varijablu zaštiti *critical* oznakom.

Sinkronizacijske točke su mjesta u kodu gdje je sinkronizacija specificirana, ili eksplicitno ili implicitno. Njihova dodatna funkcija u OpenMP programskom

kodu je omogućavanje konzistentnih vrijednosti objekata unutar dijeljene memorije. Naime, važno je da svi podaci u dijeljenoj memoriji imaju najnovije ažurirane vrijednosti kako bi druge dretve rukovale s aktualnim, a ne zastarjelim podacima. Sinkronizacijske točke uključuju eksplicitne i implicitne ograde, početak i kraj kritičnih područja ili odsječaka, specijalne točke koje se zaključavaju ili oslobađaju i mjesta navođenja `flush` oznaka.

### 3.4. Performanse prilikom uporabe OpenMP-a

Glavni razlog korištenja paralelizacije nekog programa ili aplikacije je ubrzanje njegova izvođenja. Ako se trajanje izvođenja programa na jednom procesoru označi sa T1, onda u idealnoj situaciji trajanje izvođenja programa na P procesora jest T1/P. Ako je Tp trajanje izvođenja na P procesora onda

$$S = \frac{T_1}{T_p}$$

označava ubrzanje i mjera je uspjeha paralelizacije. No uvijek postoje neke prepreke u postizanju savršenog ubrzanja. Gotovo svi programi se sastoje od dijelova ili područja teksta programa koje nije moguće paralelizirati i onih koje jest. Korištenjem većeg broja procesora trajanje provedeno u paraleliziranim dijelovima teksta programa je smanjeno, ali sekvencijalni dio programa ima jednako trajanje izvođenja programa. Trajanje izvođenja programa je ograničeno trajanjem izvođenja sekvencijalnog dijela programa, što stavlja granicu na mogućnost cjelokupnog ubrzanja. Taj je efekt poznat kao Amdahlov zakon i glasi:

$$S = \frac{1}{\left(f_{par}/P + \left(1 - f_{par}\right)\right)}$$

gdje fpar je paralelni udio teksta programa, a P označava broj procesora. U idealnom slučaju cijeli kod je paraleliziran te je fpar=1, a očekivano ubrzanje je jednako broju procesora. Stoga je poželjno paralelizirati što je više teksta programa moguće posebice pri korištenju većeg broja procesora. Druge prepreke prema savršenom linearnom ubrzanju obuhvaćaju utrošak vremena

pri stvaranju i uništavanju dretvi kod *fork-join* modela, sinkronizaciju dretvi i pristup memoriji. S druge strane, mogućnost stavljanja više podataka programa u priručnu memoriju može nadoknaditi utrošak vremena kod *fork-join* modela. Mjera sposobnosti programa da smanji trajanje izvršavanja programa pomoću povećanja broja procesora se naziva paralelna skalabilnost (*engl. parallel scalability*).

Treba primijetiti da programer sam mora sam odrediti koje dijelove teksta programa paralelizirati. Naime postoji mogućnost odabiranja samo nekih dijelova teksta programa za paralelizaciju, a ostatak ostaviti sekvencijalnim. Također se ubrzanje pomoću paralelizacije često definira kao poboljšanje naspram najbolje izvedenog sekvencijalnog algoritma za određeni problem. Ta mjera ukazuje donosi li paralelizacija prednosti koje nije moguće postići pri biranju drugog načina rješavanja problema.

## 3.5. OpenMP naredbe

Nadalje slijedi opis sintakse, oznaka i klauzula OpenMP API-ija. Budući da je naglasak na korištenje u C++-u neke će se oznake zaobići. Iako OpenMP biblioteka ima relativno malen skup oznaka, još manji dio njih je stvarno neophodno za paralelizaciju. Ostale služe za posebne slučajeve, bolju efikasnost u paralelizaciji, itd.

### *3.5.1.* Naredba *parallel*

Parallel naredba (*engl. parallel construct*) je ključna za izvođenje programa pisanog OpenMP-ijem, program bez te naredbe će se izvoditi sekvencijalno. Sintaksa je:

**`#pragma omp parallel`** *[clause[[,] clause]. . . ]*

*structured block*

Treba navesti da se *parallel* naredba može ograničiti i vitičastim zagradama {,}. Paralelno područje završava na kraju strukturiranog bloka što je obično označeno zatvarajućom vitičastom zagradom **}**.

Kada dretva naiđe na *#pragma omp parallel* naredbu, stvara skup ili tim dretvi koji paralelno izvode program u paralelnom području, ali ne osigurava raspodjelu poslova između dretvi niti način pristupa dijeljenoj memoriji. Dretva koja je stvorila ostale dretve u timu ili skupu dretvi se naziva inicijalna ili glavna dretva (*engl. master thread*). Na kraju paralelnog područja postoji implicitna ograda koja uništava sve dretve osim glavne.

### *3.5.2. Naredbe za podjelu poslova u OpenMP programu*

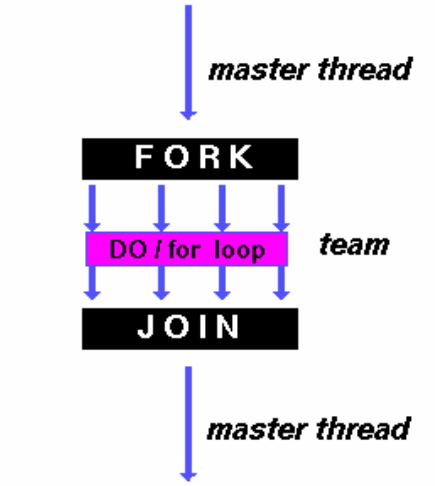
Naredbe za podjelu poslova su vrlo važno svojstvo OpenMP-a. C/C++ koristi tri naredbe za podjelu poslova. Naredba za podjelu poslova zajedno sa naredbom koja terminira naredbu za podjelu poslova, ako je potrebna, specificira područje teksta programa u kojem se posao distribuira između dretvi koje ga izvršavaju, također specificira na koji način će se taj posao podijeliti. Naredba podjele poslova mora biti vezana na aktivno paralelno područje (ključna riječ *parallel*). Ako nije vezana na aktivno paralelno područje, ili je navedena u sekvencijalnom dijelu programa, jednostavno se ignorira. Također, naredbe za podjelu poslova ne pokreću nove dretve i imaju implicitnu ogradu na kraju, ali ne i na početku.
Dva glavna pravila kod naredbi za podjelu poslova su:

- Sve dretve pojedinog tima ili skupa dretvi moraju naići na svako područje podjele poslova.

- Redoslijed pristupanja pojedine dretve području podjele poslova ili nekoj ogradi mora biti jednak.

#### 3.5.2.1. Oznaka For

Oznaka *do/ for* specificira da iteracije neke petlje moraju biti izvršavane paralelno od strane tima ili skupa dretvi (vidi: Slika 3.). Podrazumijeva se da je paralelno područje već navedeno, inače se izvodi slijedno na jednom procesoru.




**Slika 3. Primjer for petlje kao bloka podjele posla**

Sintaksa *DO/for* oznake jest:

**`#pragma omp for`** *[clause[[,] clause]. . . ]*

*for-loop*

Blok ne sadrži vitičaste zagrade jer su one implicitno navedene sa samom oznakom. Za bolje izvođenje programa navode se klauzule.

Klauzula *schedule* opisuje kako se iteracije neke petlje raspoređuju između dretvi u skupu dretvi. Podrazumijevano raspoređivanje ovisi o implementaciji. Sastoji se od više tipova:

- `static-` iteracije petlje se dijele u dijelove veličine *chunk* (to je integer varijabla) i onda se statički dodjeljuju dretvama. Ako varijabla *chunk* nije navedena, iteracije se jednako (ako je moguće) dijele između dretvi
- `dynamic-` iteracije petlje se dijele u komade veličine *chunk* i dinamički se raspoređuju između dretvi, kad neka dretva završi sa svojim dijelom, dinamički joj se dodjeljuje drugi. Podrazumijevana veličina *chunk* varijable je 1.

- `guided-` za veličinu *chunk*=1, veličina svakog dijela je proporcionalna broju nedodijeljenih iteracija podijeljenih sa brojem dretvi, padajući do 1. Za veličinu *chunk*=k (gdje je k veće od 1), veličina svakog dijela je određena na isti način kao i prije uz restrikciju da dijelovi ne sadržavaju manje od k iteracija (osim zadnjeg dijela koji se dodjeljuje , te koji može imati manje od k iteracija). Podrazumijevana veličina *chunk* varijable je 1.
- `runtime` – odluka o raspodjeli se odgađa do izvođenja teksta programa kada se čita varijabla okoline OMP_SCHEDULE. Nije dopušteno definirati varijablu *chunk.*
- `auto-` odluka o raspodjeli se prepušta kompajleru ili sistemu na kojem se program pokreće

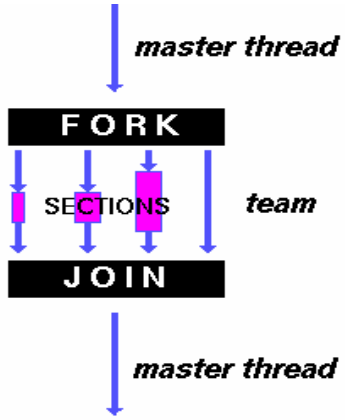
Klauzula *nowait* određuje da se dretve ne sinkroniziraju na kraju paralelne petlje.
Klauzula *ordered* određuje da se iteracije petlje izvode onim redom kojim bi se izvodile u slijednom programu.

#### 3.5.2.2. Oznaka *sections*

Oznaka *sections* je ne-iterativna oznaka bloka za podjelu poslova
(vidi: Slika 4.).
Specificira da se odvojeni odjeljak teksta programa raspodjeljuje između dretvi u skupu dretvi. Neovisne oznake *section* se ugnježđuju unutar *sections* oznake. Svaki odjeljak (*section*) se izvršava od strane jedne dretve. Različit odjeljci mogu biti izvršavane od različitih dretvi. Moguće je da jedna dretva izvrši više od jednog odjeljka.

Sintaksa *sections* oznake jest:

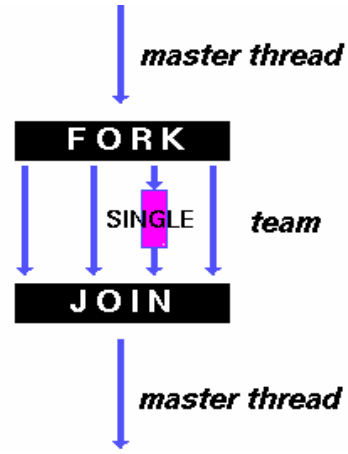
**#pragma omp sections** *[clause[[,] clause]. . . ]*{
[**#pragma omp section** ]
*structured block*
[**#pragma omp section**
*structured block* ]
. . .}




**Slika 4. Oznaka sections bloka podjele poslova**

### 3.5.2.2. Oznaka *single*

Oznaka *single* specificira da se označeni i odvojeni kod izvršava od strane jedne dretve u skupu (vidi: Slika 5.). Oznaka je korisna kada jedan dio teksta programa nije siguran za rukovanje sa više dretvi, kao stalno pisanje i čitanje.




**Slika 5. Oznaka *single***

Dretve koje se ne koriste unutar *single* oznake čekaju na njenom kraju, osim ako se ne navede *nowait* klauzula.
Sintaksa *single* oznake jest:

```
#pragma omp single [clause[[,] clause]. . . ]
                   structured block
```

### *3.5.3. Sinkronizacijske naredbe*

#### 3.5.3.1. Oznaka *master*

Oznaka *master* specificira da se područje teksta programa izvršava samo od strane glavne ili inicijalne dretve. Sve druge dretve u timu preskaču taj dio teksta programa. Ova oznaka nema pridruženu implicitnu ogradu.
Sintaksa jest:

```
#pragma omp master
   structured block
```

#### 3.5.3.2. Oznaka *critical*

Oznaka *critical* specificira područje teksta programa koja se mora izvršavati od strane samo jedne dretve istovremeno (kritični odsječak).
Potrebno je napomenuti da ako se jedna dretva izvršava unutar *critical* područja i druga dretva stigne do tog *critical* područja i pokuša ga izvršiti, biti će blokirana dok prva dretva ne izađe iz tog *critical* područja.
Opcionalno može se navesti ime *critical* područja koje omogućava postojanje više *critical* područja. Imena su globalni identifikatori. Različita *critical* područja istog imena se smatraju istim područjem. Sva neimenovana kritična područja se smatraju istim područjem.

Sintaksa *critical* oznake jest:

```
#pragma omp critical [(name)]
structured block
```

#### 3.5.3.3. Oznaka *barrier*

Oznaka *barrier* sinkronizira sve dretve u timu. Kada izvršavanje dođe do *barrier* oznake, dretva će čekati na toj točki dok je sve ostale dretve ne dostignu. Nakon toga sve dretve paralelno nastave sa izvršavanjem teksta programa poslije ograde.

Sintaksa jest:

```
#pragma omp barrier
```

Sve dretve u timu (ili nijedna) moraju izvršiti *barrier* područje.
Sekvenca područja podjele poslova i područja ograda na koje se naiđe u programu moraju biti iste za svaku dretvu u timu.

#### 3.5.3.4. Oznaka *ordered*

Oznaka *ordered* specificira da iteracije odvojene petlje će biti izvršene u istom redoslijedu u kojem bi bile izvršene slijedno. Dretve će trebati čekati prije izvršavanja vlastitih dijelova iteracija ako prijašnje iteracije nisu gotove sa izvođenjem. Koristi se najčešće unutar for petlje pomoću *ordered* klauzule. Oznaka *ordered* omogućuje usavršavanje unutar petlje u kojoj se primjenjuje. Inače nije potrebna.
Sintaksa *ordered* oznake jest:

```
#pragma omp ordered
structured block
```

#### 3.5.3.5. Oznaka *atomic*

Oznaka *atomic* specificira da se memorijska lokacija mora osvježavati (tj. mijenjati) periodički i individualno, umjesto da se dopušta više dretvi da pokušaju pisati u nju. To je ustvari, mini verzija kritičnog odjeljka.

Sintaksa *atomic* oznake jest:

```
#pragma omp atomic
statement
```

Oznaka se primjenjuje na jednu naredbu, koja slijedi odmah poslije nje.

### *3.5.4. Klauzule za kontrolu paralelizacije i naredbi podjele poslova*

#### 3.5.4.1. Klauzula *private*

Klauzula *private* deklarira varijable u listi koje su privatne za svaku dretvu.
Ponašanje privatnih varijabli:

- Novi objekt istog tipa se deklarira za svaku dretvu u timu dretvi.
- Sve reference na originalni objekt se zamjenjuju s referencama na novi objekt.
- Varijable deklarirane kao privatne nisu početno inicijalizirane za svaku dretvu.

Format klauzule:

```
#pragma omp parallel for private(var i, var a, ...)
```

#### 3.5.4.2. Klauzula *shared*

Klauzula *shared* deklarira varijable u listi kao dijeljene između svih dretvi u timu. Dijeljena varijabla postoji na samo jednoj lokaciji u memoriji i sve dretve mogu pisati na tu adresu ili čitati sa nje. Na programeru je da osigura pravilo pristupanje dijeljenoj varijabli od strane više dretvi.

Format klauzule:

```
#pragma omp parallel for shared(var a, ...)
```

#### 3.5.4.3. Klauzula *lastprivate*

Klauzula *lastprivate* kombinira ponašanje *private* klauzule sa kopijom iz zadnje iteracije petlje ili odjeljka i sprema u originalnu varijablu objekta.
Vrijednost koja se kopira u originalni objekt varijable se dohvaća iz zadnje (gledano po redoslijedu) iteracije ili odjeljka teksta programa.

Format klauzule:

```
#pragma omp parallel for lastprivate(var a, ...)
```

#### 3.5.4.4. Klauzula *firstprivate*

Klauzula *firstprivate* kombinira ponašanje klauzule *private* sa automatskom inicijalizacijom varijable u svojoj listi. Lista je sastavljena od varijabli navedenih u oblim zagradama klauzule.
Varijable u listi se inicijaliziraju shodno vrijednostima njihovih originalnih objekata prije ulaska u paralelni odsječak ili blok podjele poslova.

Format klauzule:

```
#pragma omp parallel for firstprivate(var a, ...)
```

#### 3.5.4.5. Klauzula *default*

Klauzula *default* dopušta korisniku da specificira podrazumijevani doseg svih varijabli u leksičkom rasponu nekog paralelnog područja.
Format klauzule jest:

```
default (shared | none)
```

C/C++ OpenMP specifikacija ne sadrži *private* ili *firstprivate* kao vrijednost *default* klauzule. No moguće ih je izvesti pomoću implementacije. Upotreba *none* vrijednosti zahtijeva da programer eksplicitno navede doseg svih varijabli.

#### 3.5.4.6. Klauzule *schedule* i *nowait*

Klauzule *schedule* i *nowait* su detaljnije objašnjene u poglavlju
3.5.2.1.. Detaljniji opis ostalih klauzula, oznaka i naredbi OpenMP-a se mogu naći na Internetu[6].

### 3.6. OpenMP primjer

Slijedi jedan jednostavan primjer jednog programa u OpenMP-u koji je naveden na slici 6.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE   10
#define N  100

int main (int argc, char *argv[]) {
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0; /* Some initializations */
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
        tid = omp_get_thread_num();
                if (tid == 0)
                {
                        nthreads = omp_get_num_threads();
                        printf("Number of threads = %d\n", nthreads);
                }
        printf("Thread %d starting...\n",tid);
        #pragma omp for schedule(dynamic,chunk)
                for (i=0; i<N; i++)
                {
                        c[i] = a[i] + b[i];
                        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
                }
        }  /* end of parallel section */
}
```

**Slika 6. Primjer paraleliziranog programa primjenom OpenMP API-ija**

Pojašnjenje primjera:

Za primjenu OpenMP-a u C/C++ programu potrebno je u početku navesti oznaku:

```
#include <omp.h>
```

Ta oznaka uključuje biblioteku funkcija, oznaka, klauzula i naredbi OpenMP-a.

Nakon ostalih biblioteka definiramo i deklariramo dvije konstante, N i CHUNKSIZE. Konstanta N nam govori koliko iteracija će imati for petlja, a CHUNKSIZE konstanta se koristi za postavljanje varijable *chunk* na njenu vrijednost. Nakon ulaska u glavni program dolazi do deklaracije varijabli i 3 polja tipa *float*.

Slijedi *for* petlja gdje inicijaliziramo polja a i b na neke vrijednosti, te poslije *for* petlje varijablu *chunk* postavljamo na konstantu CHUNKSIZE, koja je navedena da se izbjegne upotreba tzv. „magičnih brojeva“.

Linija teksta programa:

```
#pragma omp parallel shared (a,b,c,nthreads,chunk)
```

služi za definiciju paralelnog dijela teksta programa. Klauzula *shared* definira da sve dretve koje izvode paralelni dio teksta programa dijele zajedničke varijable navedene unutar klauzule. Klazula *private* nam govori koje varijable su privatne tj. posebne za svaku dretvu.

Funkcija `omp_get_thread_num()` vraća redni broj dretve te se postavlja varijabla tid na njenu vrijednost, pošto je ona privatna za svaku dretvu se radi zasebna kopija koja sadrži redni broj te dretve.

*If* izraz uvjeta provjerava je li varijabla *tid* jednaka nuli, što označava glavnu dretvu. Ako je trenutna dretva glavna, ispisuje se broj dretvi unutar paralelnog bloka pomoću varijable *nthreads*. Istovjetno se kasnije ispisuje koja je trenutna dretva.

Slijedi nova oznaka koja definira da se paralelizira *for* petlja.

```
#pragma omp for schedule(dynamic,chunk)
```

Gore navedena oznaka pomoću klauzule *schedule* dodjeljuje dinamički svakoj dretvi dijelove petlje veličine određene brojem u varijabli *chunk*. U našem slučaju to je 10. Unutar same *for* petlje vrši se neki posao te ispisuje trenutačna dretva, iteracija i rezultat posla. Nakon toga završavamo paralelni odjeljak.

# 4. Paralelizacija evolucijskih algoritama pomoću OpenMP-a

## 4.1. Algoritam Roulette wheel

Algoritam *Roulette Wheel* je generacijski algoritam sa *roulette wheel* selekcijskim operatorom [10]. Funkcija dobrote pridružuje dobrotu svakoj jedinki, te s obzirom na vrijednost dobrote svake jedinke dobiva se vjerojatnost izbora neke jednike. Što je dobrota jedinke veća, vjerojatnost njenog izbora je veća. Algoritam se zove *Roulette Wheel* jer ako se kolo za rulet podijeli na dijelove određene vrijednostima dobrota svih jedinki u populaciji, njegovom vrtnjom biramo jednike. Pseudokod algoritma je prikazan na slici 7.

```
Registriraj parametre
Inicijaliziraj parametre

Generacija
        kopiraj trenutnu najbolju jedinku
        selektiraj grupu jedinki po dobroti, spremi u wheel

        for petlja- za sve jedinke u grupi wheel
                kopiraj grupu jedinki u novu populaciju
        end for

        for petlja- za sve jedinke u grupi wheel
                kopiraj grupu jedinki u novu populaciju
        end for

        for petlja- za sve jedinke u populaciji
                zamijeni staru populaciju novom
        end for

        noCrx= veličina populacije * vjerojatnost rekombinacije/2
        for petlja- od 0 do noCrx
                izaberi dvije jedinke populacije kao roditelje
                kopiraj dva roditelja kao djecu
                spari dva roditelja i prvo dijete
                spari dva roditelja i drugo dijete
                zamijeni prvog roditelja i prvo dijete
                if- ako roditelji imaju različite indekse u populaciji
                        zamijeni drugog roditelja i drugo dijete
                end if
        end for
mutiraj cijelu populaciju

for-za cijelu populaciju
        evaluiraj novu populaciju
end for
slučajno izaberi jedinku
if- slučajno izabrana jedinka ima bolju dobrotu od najbolje
                zamijeni najbolju jedinku sa slučajnom
end if
end Generacija
```

**Slika 7. Pseudokod algoritma Roulette Wheel**

### *4.1.1. Paralelizirani dio algoritma*

U algoritmu *Roulette Wheel* paralelizirana su dva dijela algoritma. Prvi dio je *for* petlja gdje vršimo selekciju i križanje unutar cijele populacije (vidi: slika 8.). Prva linija koda deklarira brojač j petlje *for* izvan paralelnog dijela teksta programa kako bi taj brojač bio vidljiv svim dretvama koje će kasnije izvršavati algoritam. Linija teksta programa:

`#pragma omp parallel for shared(noCrx,deme) ordered`

označava početak paralelnog dijela algoritma, i odnosi se na *for* petlju, koja se izvodi paralelno te ne njenom kraju postoji implicitna ograda gdje se čeka s završetkom rada svih dretvi, i njihovim uništavanjem osim glavne dretve. Oznaka *shared* definira varijable *noCrx* i *deme* kao dijeljene varijable svih dretvi, te su one spremljene na dijeljenoj memorijskoj lokaciji.

Klauzula *ordered* specificira da se sve iteracije petlje izvode istim redoslijedom kao i u slijednom izvršavanju algoritma. To se odnosi na odjeljak teksta programa gdje upotrebljavamo oznaku *ordered* točnije linija teksta programa:

`#pragma omp ordered`

Jer poželjno je da se križanje roditelja i djece izvodi slijedno.

```
 int j;
#pragma omp parallel for shared(noCrx,deme) ordered
for( j = 0; j < noCrx; j++){
                IndividualP parent1 = selRandomOp->select(*deme);
                IndividualP parent2 = selRandomOp->select(*deme);
                state->getLogger()->log(5, "Parents: " +
                dbl2str(parent1->fitness->getValue()) + ", " +
                dbl2str(parent2->fitness>getValue()));
                IndividualP child1 = copy(parent1);
                IndividualP child2 = copy(parent2);
                mate(parent1, parent2, child1);
                mate(parent1, parent2, child2);
                #pragma omp ordered
                replaceWith(parent1, child1);
                if (parent1->index != parent2->index ){
                        replaceWith(parent2, child2);
                }
}
```

**Slika 6. Paralelizirani dio algoritma Roulette Wheel**

Drugi dio paraleliziranog dijela algoritma *Roulette Wheel* paralelizira for petlju gdje se vrši evaluacija jedinki, gdje je najveće trajanje izvršavanja algoritma (vidi: Slika 9.). Linija teksta programa:

```
#pragma omp parallel for shared(deme,j)
```

označava početak paralelnog dijela algoritma, i odnosi se na *for* petlju, koja se izvodi paralelno te ne njenom kraju postoji implicitna ograda gdje se čeka s završetkom rada svih dretvi, i njihovim uništavanjem osim glavne dretve. Oznaka *shared* definira varijable *deme* i j spremljene na dijeljenim memorijskim lokacijama i dostupne svim dretvama.

```
#pragma omp parallel for shared(deme,j)
for( j = 0; j < deme->size(); j++){
        if(!deme->at(j)->fitness->isValid()) {
                evaluate(deme->at(j));
        }
}
```

**Slika 9. Paralelizirani dio algoritma Roulette Wheel**

## 4.2. Eliminacijski algoritam

Algoritam *Elimination* je generacijski algoritam sa eliminacijskim selekcijskim operatorom [11]. Eliminacijska selekcija temelji se na izbacivanju loših jedinki radije nego odabiru boljih pri generiranju nove populacije. Kako bi odabrala lošu jedinku izračunava se kazna svake jedinke koja predstavlja vjerojatnost odabira jedinke za izbacivanje. Kazna jedinke je razlika maksimalne dobrote i dobrote jedinke. Ovim načinom odabira jedinke za izbacivanje iz postupka selekcije u potpunosti se uklanja jedinka s najvećom dobrotom, pošto ona ima kaznu nula. Zbog očuvanja broja jedinki u populaciji nakon izbacivanja stvara se nova jedinka nekim od genetskih operatora te se nastavlja proces selekcije. Pseudokod algoritma je prikazan na slici 10.

```
Registriraj parametre
Inicijaliziraj parametre

Generacija
        kopiraj trenutnu najbolju jedinku
        for – za sve jedinke u populaciji
                kopiraj sve jedinke u newGen
        end for
        generationGap= genGap * veličina populacije

        for- od 0 do generationGap
                eliminiraj postotak najlošijih jedinki
        end for

        for- od 0 do generationGap
                izaberi dvije jedinke populacije kao roditelje
                kopiraj prvog roditelja kao dijete
                spari dva roditelja i dijete
                pospremi dijete u newGen
        end for

        for – za sve jedinke u populaciji
                zamijeni staru populaciju sa newGen populacijom
        end for

        for-za cijelu populaciju
                evaluiraj novu populaciju
        end for
        slučajno izaberi jedinku
        if- slučajno izabrana jedinka ima bolju dobrotu od najbolje
                zamijeni najbolju jedinku sa slučajnom
        end if
end Generacija
```

**Slika 7. Pseudokod algoritma Elimination**

### *4.2.1. Paralelizirani dio algoritma*

U algoritmu *Elimination* paralelizirana su dva dijela algoritma. Prvi dio je *for* petlja gdje vršimo selekciju i križanje unutar cijele populacije (vidi: slika 11.). Prva linija koda deklarira brojač j petlje *for* izvan paralelnog dijela algoritma kako bi taj brojač bio vidljiv svim dretvama koje će kasnije izvršavati algoritam.

Linija teksta programa:

```
#pragma omp parallel for shared(deme,generationGap)
```

označava početak paralelnog dijela algoritma, i odnosi se na *for* petlju, koja se izvodi paralelno te ne njenom kraju postoji implicitna ograda gdje se čeka s završetkom rada svih dretvi, i njihovim uništavanjem osim glavne dretve. Oznaka *shared* definira varijable *generationGap* i *deme* kao dijeljene varijable svih dretvi, te su one spremljene na dijeljenoj memorijskoj lokaciji. Također unutar paraleliziranog dijela algoritma koristi se oznaka *critical* u liniji teksta programa:

```
#pragma omp critical
```

koja označava kritični odjeljak paraleliziranog dijela algoritma. Tj. spremanje jedinke djeteta u *newGen* vektor se vrši od strane jedne dretve u nekoj jedinici vremena.

```
int j;
#pragma omp parallel for shared(deme,generationGap)
        for( j = 0; j < generationGap; j++) {
                IndividualP parent1 = selRandomOp->select(*deme);
                IndividualP parent2 = selRandomOp->select(*deme);
                IndividualP child = copy(parent1);
                mate(parent1, parent2, child);
                #pragma omp critical
                newGen.push_back(child);
}
```

**Slika 8. Paralelizirani dio algoritma Elimination**

Drugi dio paraleliziranog dijela algoritma *Elimination* paralelizira *for* petlju gdje se vrši evaluacija jedinki, gdje je najveće trajanje izvršavanja algoritma (vidi: Slika 12.). Linija teksta programa:

```
#pragma omp parallel for shared(deme,j)
```

Označava početak paralelnog dijela algoritma, i odnosi se na *for* petlju, koja se izvodi paralelno te ne njenom kraju postoji implicitna ograda gdje se čeka s završetkom rada svih dretvi, i njihovim uništavanjem osim glavne dretve. Oznaka *shared* definira varijable *deme* i j spremljene na dijeljenim memorijskim lokacijama i dostupne svim dretvama.

```
#pragma omp parallel for shared(deme,j)
        for( j = 0; j < deme->size(); j++)
                if(!deme->at(j)->fitness->isValid()) {
                        evaluate(deme->at(j));
}
```

**Slika 9. Paralelizirani dio algoritma Elimination**

# 5. Analiza rezultata paralelizacije algoritama

## 5.1. Ispitivanje

Slijedi ispitivanje implementacije paraleliziranih algoritama u ECF-u.
Ispitivanje se provodi na osobnom računalu, unutar Visual Studio 2008 IDE-a. Konfiguracija osobnog računala je kako slijedi: Intel (R) Core (TM) i5 CPU 750 @ 2.67GHz četverojezgreni procesor.
Prati se brzina izvođenja algoritma s obzirom na genotip kojeg koriste i problem kojeg rješavaju. Pri pokretanju algoritma gleda se ukupno trajanje izvršavanja. Trajanje izvršavanja algoritma ne obuhvaća samo izvorni kod algoritma nego i poziva brojne druge metode i izvorne kodove, te se sveukupno trajanje izvršavanja proteže na izvršavanje algoritma unutar ECF-a a ne samog algoritma. To naravno ima utjecaja na samo trajanje izvršavanja.
Sa strane paralelizacije, budući da se paralelizirao dio samog algoritma, nije moguće postići linearno ubrzanje, ali je moguće postići ubrzanje približno tome.

Tokom ispitivanja kao što je prije napomenuto koriste se dva algoritma: *Roulette Wheel* i *Elimination*. Detalji njihove izvedbe se navode u konfiguracijskim datotekama problema kojeg rješavaju. Za potrebe ispitivanja neki parametri genotipova će se ostaviti istima tokom cijelog procesa zbog jednolikosti rezultata i njihove moguće usporedbe. Navedeni parametri su:

- veličina populacije od 500, postavljena je na tu vrijednost zbog povećanja broja iteracija te većeg efekta na paralelizaciju;
- zaustavljanje algoritma nakon 100 generacija (vrijednost 100 je podrazumijevana vrijednost).

Ostali parametri genotipova će imati podrazumijevane vrijednosti.

Također odabiru se 2 problema koja će rješavati algoritmi. Prvi je problem simboličke regresije, drugi je problem minimizacije funkcije.
Problem minimizacije funkcije se rješava traženjem te biranjem realnih i cijelih brojeva iz nekog skupa brojeva [8]. Simbolička regresija je problem gdje uz neke zadane vrijednosti parova realnih ili cijelih brojeva želimo naći jednadžbu funkcije koja predstavlja te vrijednosti [9].
Sukladno tome za prvi problem koristi se genotip *Tree* [9] dok se za drugi problem koristi genotip *Binary* [9].

Mjerenja će se vršiti nad podrazumijevanim vrijednostima parametara algoritama. To su za algoritam *Roulette Wheel*: vjerojatnost rekombinacije (crxprob) od 0.5 i selekcijski pritisak (selpressure) vrijednosti 10. algoritam *Elimination* ima parametre: generacijski jaz (gengap) vrijednosti 0.6 i selekcijski pritisak (selpressure) vrijednosti 10.

Nadalje, mjerenja će se vršiti za svaki algoritam kada se izvršavaju serijski ili slijedno bez OpenMP-a, te uz pomoć OpenMP-a sa dvije, tri i četiri dretve. Ispitivanje za veći broj dretvi ne mora nužno značiti poboljšanje performansi. Naime, potrebno je određeno trajanje za stvaranje svake dretve te odvajanje za njen stog itd. Poželjno je stvarati onoliki broj dretvi koliko računalo na kojem se program izvodi ima procesorskih jezgri ili samih procesora.

## 5.2. Ispitni primjeri

Izabrana su dva paralelizirana algoritma te svaki od njih rješava dva problema. Budući da je funkcija dobrote relativno jednostavna, petlju je moguće dodatno opteretiti po potrebi. Sama odstupanja u mjerenjima su zanemariva i prosjek je približne vrijednosti kao i sama mjerenja. Za izračun ubrzanja korištena je formula $S = \frac{T_1}{T_p}$ .

Popis ispitnih primjeraka je:

1. algoritam *Roulette Wheel* rješava problem simboličke regresije
2. algoritam *Roulette Wheel* rješava problem minimizacije funkcije
3. algoritam *Elimination* rješava problem simboličke regresije
4. algoritam *Elimination* rješava problem minimizacije funkcije

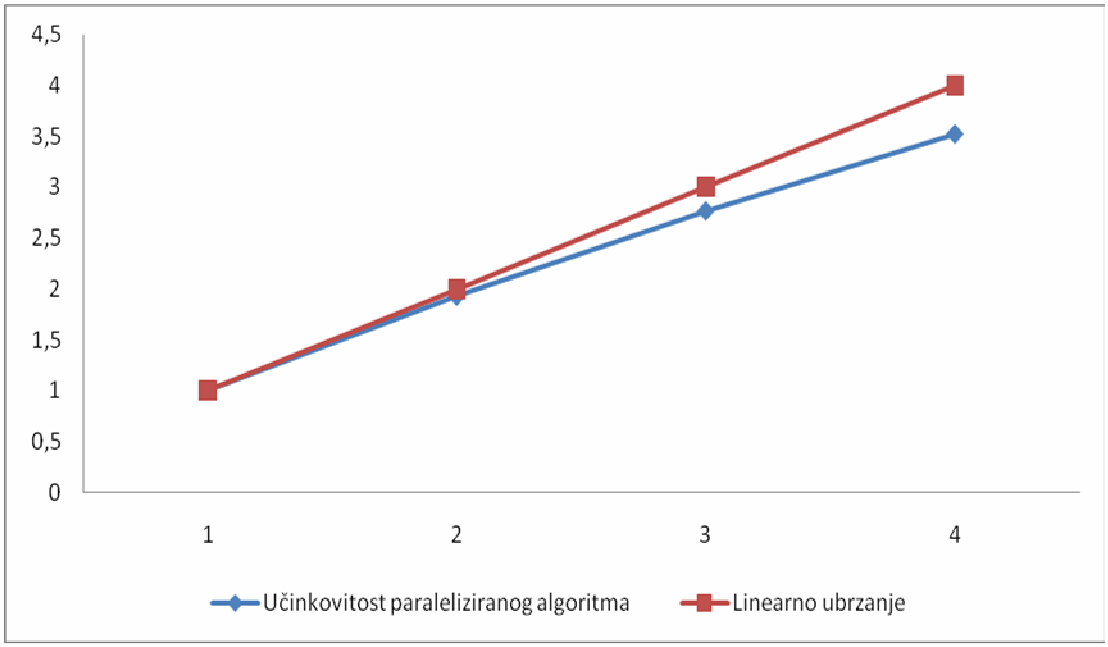
## 5.3. Rezultati ispitivanja

### *5.3.1. Ispitni primjer 1*

U prvom ispitnom primjerku se testira *Roulette Wheel* algoritam za problem simboličke regresije prvo slijedno, a onda se izvršava i testira na dvije, tri i četiri dretve. Dobiveni rezultati su prikazani u tablici 1. Prosjek izvršavanja algoritma na jednoj dretvi jest 26,4823 sekundi dok je na dvije dretve trajanje izvršavanja 13,7407 sekundi. Ubrzanje je približno 1,9273. Nadalje, ako se koriste tri dretve ubrzanje jest 2,7647, a za četiri ubrzanje jest 3,5193. Što je veći broj procesora učinkovitost pomalo pada zbog samog načina izvedbe algoritma gdje je svaka dretva morala stvarati privatne kopije varijable unutarnje petlje, a i samog algoritma i problema.

**Tablica 1. Trajanje izvršavanja algoritma Roulette Wheel na problemu simboličke regresije**

| Ispitni primjerak 1 | Vremena mjerenja izvršavanja algoritma | | | |
|---|---|---|---|---|
| **Broj dretvi** | **Jedna(slijedno izvođenje)** | **Dvije** | **Tri** | **Četiri** |
| Vremena u sekundama: | 26,5031 | 13,7144 | 9,5655 | 7,5609 |
| | 26,4480 | 13,7624 | 9,5866 | 7,5464 |
| | 26,4802 | 13,7346 | 9,5591 | 7,5022 |
| | 26,4833 | 13,7749 | 9,5665 | 7,5417 |
| | 26,5012 | 13,7477 | 9,6052 | 7,4640 |
| | 26,5070 | 13,7443 | 9,5472 | 7,5113 |
| | 26,4684 | 13,7323 | 9,5819 | 7,5529 |
| | 26,5161 | 13,7544 | 9,5745 | 7,5488 |
| | 26,5083 | 13,7453 | 9,5805 | 7,5111 |
| | 26,4337 | 13,7259 | 9,5897 | 7,5417 |
| | 26,4667 | 13,7145 | 9,6009 | 7,5482 |
| | 26,4716 | 13,7375 | 9,5852 | 7,4684 |
| Prosječno trajanje | 26,4823 | 13,7407 | 9,5786 | 7,5248 |



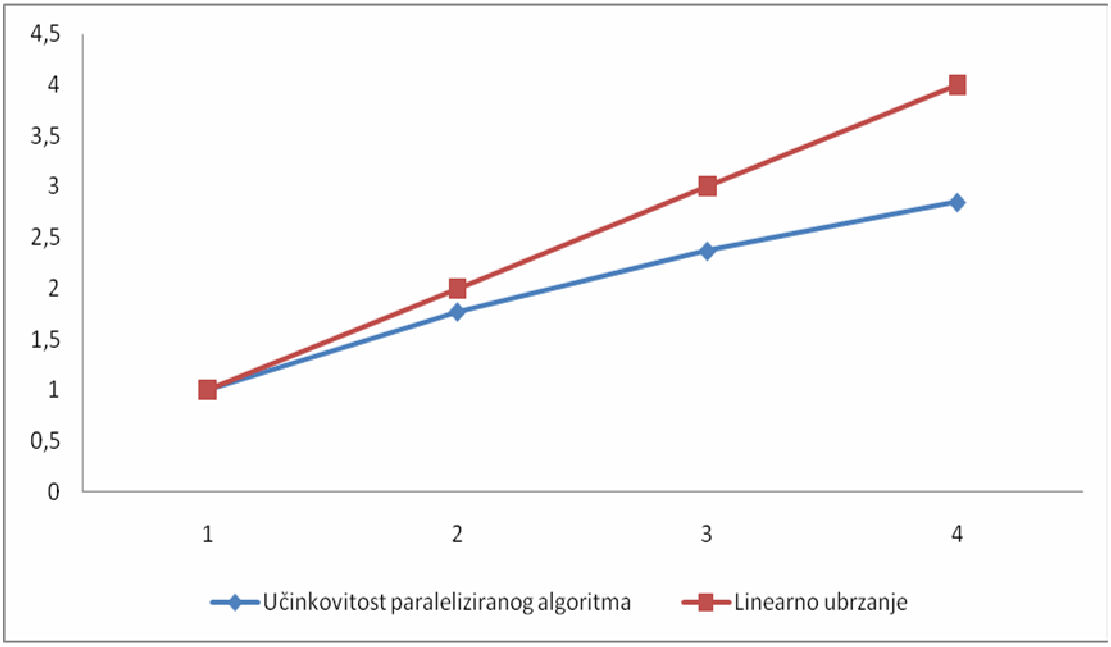

**Slika 10. Graf ubrzanja algoritma Roulette Wheel za problem simboličke regresije**

### *5.3.2. Ispitni primjer 2*

U drugom ispitnom primjerku je testiran *Roulette Wheel* algoritam za problem minimizacije funkcije prvo slijedno, a onda se izvršava na dvije, tri i četiri dretve. Dobiveni rezultati su prikazani u tablici 2. Prosjek izvršavanja algoritma na jednoj dretvi jest 29,1812 dok je na dvije dretve trajanje izvršavanja 16,4847. Ubrzanje je približno 1,7702. Što je relativno dobro ubrzanje. Nadalje kad se koriste tri dretve ubrzanje jest 2,3685, a za četiri ubrzanje jest 2,8487. Što je veći broj procesora učinkovitost pomalo pada zbog samog načina izvedbe algoritma gdje je svaka dretva morala stvarati privatne kopije varijable unutarnje petlje, a i samog algoritma i problema. Nadalje pri korištenju problema minimizacije funkcije utjecaj dobrote nije bio toliko složen kao kod simboličke regresije te samo ubrzanje nije doseglo vrijednost kao i kod problema simboličke regresije.

**Tablica 2. Trajanje izvršavanja algoritma Roulette Wheel na problemu minimizacije funkcije**

| **Ispitni primjerak 2** | **Vremena mjerenja izvršavanja algoritma** | | | |
|---|---|---|---|---|
| **Broj dretvi** | **Jedna(slijedno izvođenje)** | **Dvije** | **Tri** | **Četiri** |
| Vremena: | 29,1854 | 16,4823 | 12,3298 | 10,2938 |
| | 29,1765 | 16,4889 | 12,3179 | 10,2337 |
| | 29,1812 | 16,4748 | 12,3183 | 10,2307 |
| | 29,1852 | 16,4949 | 12,3160 | 10,2473 |
| | 29,1823 | 16,4771 | 12,3214 | 10,2504 |
| | 29,1805 | 16,4804 | 12,3133 | 10,2472 |
| | 29,1861 | 16,4748 | 12,3214 | 10,2550 |
| | 29,1806 | 16,4894 | 12,3276 | 10,2258 |
| | 29,1832 | 16,4878 | 12,3152 | 10,2420 |
| | 29,1739 | 16,4821 | 12,3177 | 10,2400 |
| | 29,1806 | 16,4880 | 12,3247 | 10,2324 |
| | 29,1792 | 16,4962 | 12,3211 | 10,2226 |
| Prosječno trajanje | 29,1812 | 16,4847 | 12,3204 | 10,2434 |



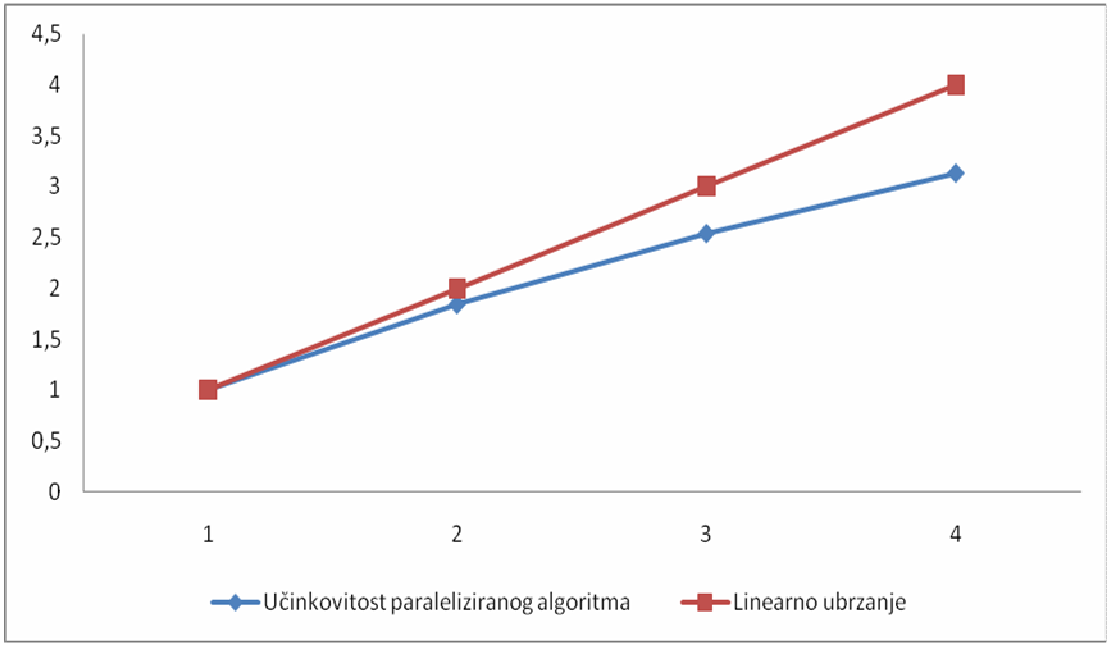

**Slika 11. Graf ubrzanja algoritma Roulette Wheel za problem minimizacije funkcije**

### *5.3.3. Ispitni primjer 3*

U trećem ispitnom primjerku se testira *Elimination* algoritam za problem simboličke regresije prvo slijedno, a onda se izvršava na dvije, tri i četiri dretve. Dobiveni rezultati su prikazani u tablici 3. Prosjek izvršavanja algoritma na jednoj dretvi jest 28,4391 sekundi dok je na dvije dretve trajanje izvršavanja 15,4459 sekundi. Ubrzanje je približno 1,8412. Što je relativno dobro ubrzanje. Nadalje kad su korištene tri dretve ubrzanje jest 2,5387, a za četiri ubrzanje jest 3,1318. Što je veći broj procesora učinkovitost pomalo pada zbog samog načina izvedbe algoritma gdje je svaka dretva morala stvarati privatne kopije varijable unutarnje petlje, a i samog algoritma i problema. Izvršavanje algoritma *Elimination* na tri dretve pokazuje da je došlo do ubrzanja, ali i do pada u učinkovitosti, dok za četiri dretve trajanje izvršavanja pokazuje da je učinkovitost vrlo niske vrijednosti. Razlog je zbog samog stvaranja dretvi, algoritma i razdjele memorijskih sredstava.

**Tablica 3. Trajanje izvršavanja algoritma Elimination na problemu simboličke regresije**

| **Ispitni primjerak 3** | **Vremena mjerenja izvršavanja algoritma** | | | |
|---|---|---|---|---|
| **Broj dretvi** | **Jedna(slijedno izvođenje)** | **Dvije** | **Tri** | **Četiri** |
| Vremena: | 28,4338 | 15,4364 | 11,1986 | 9,1051 |
| | 28,4596 | 15,4294 | 11,2268 | 9,0846 |
| | 28,4374 | 15,4287 | 11,1968 | 9,0870 |
| | 28,4727 | 15,4586 | 11,2083 | 9,0690 |
| | 28,4251 | 15,4482 | 11,2003 | 9,0895 |
| | 28,4197 | 15,4523 | 11,1954 | 9,0916 |
| | 28,4586 | 15,4750 | 11,1978 | 9,0744 |
| | 28,4414 | 15,4491 | 11,2187 | 9,0985 |
| | 28,4285 | 15,4399 | 11,1959 | 9,0583 |
| | 28,4144 | 15,4491 | 11,1908 | 9,0697 |
| | 28,4429 | 15,4374 | 11,2130 | 9,0632 |
| | 28,4345 | 15,4469 | 11,1839 | 9,0764 |
| Prosječno trajanje | 28,4391 | 15,4459 | 11,2022 | 9,0806 |



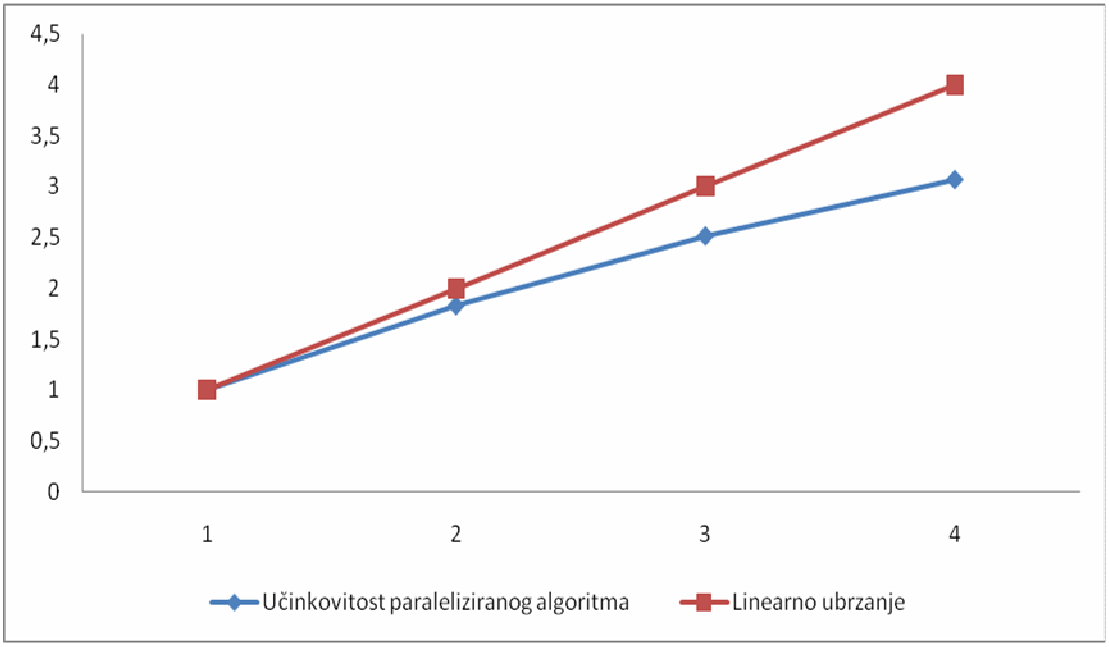

**Slika 12. Graf ubrzanja algoritma Elimination za problem simboličke regresije**

### *5.3.4. Ispitni primjer 4*

U četvrtom ispitnom primjerku je testiran *Elimination* algoritam za problem minimizacije funkcije prvo slijedno, a onda se izvršavava na dvije, tri i četiri dretve. Dobiveni rezultati su prikazani u tablici 4. Prosjek izvršavanja algoritma na jednoj dretvi jest 30,7027 sekundi dok je na dvije dretve trajanje izvršavanja 16,7811 sekundi. Ubrzanje je približno 1,8296. Što je relativno dobro ubrzanje. Nadalje kad su korištene tri dretve ubrzanje jest 2,5156, a za četiri ubrzanje jest 3,0704. Što je veći broj procesora to ubrzanje pomalo pada zbog samog načina izvedbe algoritma gdje je svaka dretva morala stvarati privatne kopije varijable unutarnje petlje, a i samog algoritma i problema. Izvršavanje algoritma *Elimination* na tri dretve pokazuje da je došlo do ubrzanja, ali i do pada u učinkovitosti, dok za četiri dretve trajanje izvršavanja pokazuje da je učinkovitost vrlo niske vrijednosti. Razlog je zbog samog stvaranja dretvi, algoritma i razdjele memorijskih sredstava.

**Tablica 4. Trajanje izvršavanja algoritma Elimination na problemu minimizacije funkcije**

| **Ispitni primjerak 4** | **Vremena mjerenja izvršavanja algoritma** | | | |
|---|---|---|---|---|
| **Broj dretvi** | **Jedna(slijedno izvođenje)** | **Dvije** | **Tri** | **Četiri** |
| Vremena: | 30,7047 | 16,7851 | 12,2028 | 10,0128 |
| | 30,7043 | 16,7935 | 12,2124 | 9,9416 |
| | 30,6967 | 16,7847 | 12,2027 | 9,9348 |
| | 30,7108 | 16,7768 | 12,2029 | 9,9304 |
| | 30,7027 | 16,7698 | 12,2024 | 9,9356 |
| | 30,7007 | 16,7849 | 12,2123 | 9,9439 |
| | 30,7075 | 16,7803 | 12,2013 | 9,9285 |
| | 30,7036 | 16,7841 | 12,2042 | 9,9531 |
| | 30,6957 | 16,7759 | 12,2066 | 9,9286 |
| | 30,6948 | 16,7912 | 12,2105 | 9,9236 |
| | 30,6995 | 16,7683 | 12,1980 | 9,9513 |
| | 30,7111 | 16,7781 | 12,2026 | 10,6087 |
| Prosječno trajanje | 30,7027 | 16,7811 | 12,2049 | 9,9994 |




**Slika 13. Graf ubrzanja algoritma Elimination za problem minimizacije funkcije**

# 6. Zaključak

OpenMP je API za paralelizaciju programa u Fortran i C/C++ programskim jezicima. Ima jednostavnu sintaksu i potrebno je malo uređivanja teksta programa za implementaciju paralelnog programa.

Očekivano i idealno ubrzanje koristeći dvije dretve je je dvostruko ubrzanje, dok je u ispitivanjima u ovom radu dobiveno ubrzanje od 1,77 do 1,92 za algoritme *Elimination* i *Roulette Wheel* koja su paralelizirana za oba rješavana problema. Nakon toga korištenjem tri i četiri dretvi dolazi do ubrzanja vremena izvršavanja algoritma, ali im je učinkovitost smanjena. Za poboljšanje ubrzanja na tri ili više dretvi trebalo bi dodatno specificirati korištenje dijeljene memorije, smanjiti uporabu privatnih varijabli ako je to moguće, te paralelizirati veći dio teksta programa. Međutim, ta poboljšanja ovise o mogućnosti implementacije same višedretvenosti na neke dijelove algoritama, te i o samom algoritmu koji može vršiti operacije koje se ne mogu učinkovito ili uopće paralelizirati. U svakom slučaju, rezultati izvršavanja algoritma u više dretvi su zadovoljavajući i dovode do smanjenog vremena izvršavanja algoritama.

## 7. Literatura

[1] Hartmut Pohlheim, Evolutionary Algorithms 2 Overview, 22.12.2006, *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation*, http://www.geatbx.com/docu/algindex-01.html#TopOfPage, 31.05.2010.

[2] Genetic algorithm, G*enetic algorithm,* http://en.wikipedia.org/wiki/Genetic_algorithm, 30.05.2010.

[3] Evolutionary algorithm, 02.03.2003, *Evolutionary algorithm*, http://en.wikipedia.org/wiki/Evolutionary_algorithm, 30.05.2010.

[4] Jakobović Domagoj, 23.05.2009., *ECF - Evolutionary Computation Framework*, http://gp.zemris.fer.hr/ecf/, Preuzeto 28.05.2010.

[5] OpenMP, http://en.wikipedia.org/wiki/OpenMP, 28.05.2010.

[6] Blaise Barney, OpenMP, https://computing.llnl.gov/tutorials/openMP/, *OpenMP*, 28.05.2010.

[7] Chapman B., Jost G., van der Pas R., Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, Cambridge, Massachusetts, London, England, 2008 Massachusetts Institute of Technology

[8] Optimization (mathematics), *Optimization (mathematics),* http://en.wikipedia.org/wiki/Optimization_(mathematics), 17.06.2010.

[9] Jakobović Domagoj, Minimal examples – by genotype, 23.05.2009., ECF - Evolutionary Computation Framework, http://gp.zemris.fer.hr/ecf/genotype.html, 17.06.2010.

[10] Fitness proportionate selection, http://en.wikipedia.org/wiki/Fitness_proportionate_selection, 19.11.2003.

[11] Mišljenčević, N., Spasojević, B. Genetski algoritmi. Seminar. Fakultet elektrotehnike i računarstva, svibanj, 2008.

## 8. Sažetak

Unutar završnog rada općenito su opisani genetski i evolucijski algoritmi, ECF programsko okruženje te i OpenMP-a. Nakon toga pobliže su definirane oznake, klauzule, naredbe podjele poslova u OpenMP-u, i dodatne upute za njihovo korištenje. Nakon svega naveden je primjer OpenMP programa. Da bi se pobliže opisala dva paralelizirana algoritma naveden je njihov pseudokod, dijelovi teksta programa nadopunjeni OpenMP sintaksom. Poslije njih dolaze rezultati testiranja izvršavanja tih algoritama za različite probleme. Rezultati su bili adekvatni i očekivani pojašnjeni su u ispitnim primjercima, detaljne tablice sadrže vremena mjerenja rezultata te su ispod njih i dani grafovi sa ubrzanjem izvršenog algoritma s obzirom na broj dretvi. Sama korisnost testiranja u završnom radu je navedena u zaključku.

Genetic and evolution algorithms are described within the bachelors thesis, as well as Evolution Computation Framework and OpenMP. Afterwards we define OpenMP directives, clauses, work-sharing constructs more closely and instructions for their usage. Example of a simple OpenMP program follows. In order to describe the two algorithms we parallelized, their program codes supplemented with OpenMP syntax and their pseudocodes are given. Then the testing results for the parallelized algorithms with different examples follow. Results were adequate and expected. They are explained further in test specimens, detailed tables consist of time measurements for solutions. Below them are graphs depicting the speedup for runtime of algorithms in relation to thread number. Usefulness of testing in the thesis is defined in conclusion of the thesis.

**Ključne riječi**

OpenMP, paralelizacija, naredbe za podjelu poslova, ograde, oznake, paralelno područje, genetski algoritam, problem minimizacije funkcije, problem simboličke regresije, učinkovitost paralelizacije