SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2581

Raspodijeljeni evolucijski algoritmi u okruženju za evolucijsko računanje u Javi

Marijan Šuflaj
Voditelj: dr. sc. Domagoj Jakobović

Zagreb, lipanj 2012.

Sadržaj

1. Uvod

Većina problema s kojima se susrećemo u stvarnome životu i čijim rješavanjem imamo neke koristi, pripadaju skupini NP-teških problema. To su problemi za koje još nije poznat algoritam koji bi ih riješio polinomnom složenošću, već su im složenosti pretežno eksponencijalne i faktorijelne. Samim time, oni predstavljaju veliki izazov, kako za ljude, tako i za računala. Kako je za veći obim ulaznih podataka nemoguće doći do egzaktnog rješenja u prihvatljivom vremenu, potrebno je pronaći algoritme koji će u dovoljno kratkom vremenu dati zadovoljavajuće rješenje. Iz tog razloga, započet je razvoj heurističkih algoritama koji to omogućuju. Voljni smo napraviti kompromis između optimalnosti rješenja i vremena izvršavanja algoritma.

Jedna od takvih skupina algoritama su i evolucijski algoritmi (EA) (1). Oni predstavljaju skupinu nedeterminističkih metoda pretraživanja prostora koje oponašaju tijek biološke evolucije. Glavna prednost im je što svoje pretraživanje započinju iz više točaka prostora. Te točke predstavljaju populaciju, a svaka točka predstavlja jednu jedinku populacije. Populacija evoluira kroz generacije te time nastaju sve bolja i bolja rješenja. Kada je zadovoljen neki kriterij zaustavljanja, algoritam prestaje s radom i u populaciji se nalaze najbolja rješenja pronađena do trenutka zaustavljanja. Primijetimo da to rješenje ne mora nužno biti optimalno. Danas evolucijski algoritmi postaju sve popularniji zbog svoje apstraktne naravi što im omogućava primjenu na široki spektar problema. Također se provode sve iscrpnija istraživanja o kombinacijama parametara koji su se pokazali kao zadovoljavajuće dobri za neki uži skup problema.

Danas se ne razvijaju samo algoritmi, već i računala za koje se i pišu ti algoritmi. Glavne značajke današnjih računala su procesori s više jezgara, sustavi s više procesora i sustavi u kojima je više računala umreženo kako bi stvorili jedinstvenu cjelinu. Kako bi se iskoristila sva moć takvih sustava, potrebno je paralelizirati

evolucijske algoritme. Jedini problem predstavlja to što su sami algoritmi zamišljeni kao slijedni algoritmi. Tako je razvijeno više varijanti algoritama kojima su pojedini dijelovi paralelizirani.

Isto tako, radi sve većeg interesa za rješavanjem problema pomoću evolucijskih algoritama, pojavila se i potreba za stvaranjem razvojnih okruženja koji će omogućiti brzo razvijanje programa koji ciljano rješavaju neki problem. Jedan od takvih razvojnih okruženja je i okruženje za evolucijsko računanje u Javi (ECFJ).

U ovome radu bit će prikazane neke od paralelnih varijanti evolucijskih algoritama, pobliže objašnjen sam način rada algoritma i bit će ponuđeno proširenje razvojnog okruženja ECFJ kako bi se pružila mogućnost implementiranja paralelnih algoritama.

2. Genetski algoritmi

Genetski algoritmi (2) predstavljaju općeniti način heurističkog pretraživanja prostora. Kako se odluke o daljnjem napretku temelje na vjerojatnostima i operatori većinom funkcioniraju uz pomoć pseudoslučajnih brojeva, spadaju u skupinu nedeterminističkih, odnosno stohastičkih algoritama.

Priroda nas uči kako se životne karakteristike jedinki iz generacije u generaciju poboljšavaju prirodnom selekcijom i križanjem gena. To je bila inspiracija za stvaranje prvih inačica genetskih algoritama. Inženjeri su se vodili idejom predstavljanja potencijalnih rješenja kao gena, selekcijom pojedinih rješenja koji će predstavljati roditelje te će njihovim križanjem nastati nova, potencijalno bolja, jedinka.

```
Stvori početnu populaciju P
Inicijaliziraj početnu populaciju P
Dok god nije zadovoljen kriterij zaustavljanja
    Korištenjem selekcijskog algoritma selektiraj dvije jedinke
    Križaj odabrane jedinke
    Mutiraj djecu dobivenu križanjem
    Ubaci jedinke u populaciju
    Evaluiraj populaciju i odaberi jedinke za populaciju sljedeće generacije
Vrati najbolju jedinku iz populacije P
```

Slika 2.1 - Pseudokod genetskog algoritma

Slika 2.1**Error! Reference source not found.** prikazuje grubi pseudokod genetskog algoritma. Primijetimo vezu samog genetskog algoritma s prirodnom evolucijom. Kao što u prirodi imamo parenje partnera, kombinacije gena i slučajne mutacije pojedinih gena, tako sličnu stvar imamo i u samom genetskom algoritmu.

Za svako rješenje potrebno je odrediti koliko je dotično rješenje dobro u odnosu

na sva ostala rješenja. Brigu o tome vodi funkcija dobrote. Ona se modelira ovisno o tome koji se problem rješava. U slučaju da optimiziramo funkciju realne varijable, tada bi funkcija dobrote mogla biti predstavljena stvarnom vrijednošću te funkcije za danu točku. Gledano dijametralno u drugu stranu, za neki kombinatorni problem poput N kraljica, funkcija dobrote bi mogla biti broj kraljica koje se međusobno ne napadaju. U dosta slučajeva je funkcija dobrote komponenta genetskog algoritma koja je najzahtjevnija po pitanju samog izračuna, te se stoga isplati uložiti malo dodatnog vremena kako bi se pronašla algoritamski manje zahtjevna inačica.

2.1 Predstavljanje jedinke

Kako bismo mogli primijeniti genetski algoritam, potrebno je odabrati odgovarajuću reprezentaciju rješenja. U većini slučajeva se optimiziraju funkcije realnih varijabli, te su stoga pogodna dva predstavljanja rješenja.

- Predstavljanje realnim brojem gdje je taj broj stvarno potencijalno rješenje
- Binarni prikaz gdje je svako rješenje predstavljeno nizom binarnih brojeva

Prednost binarnog zapisa nad zapisom realnim brojem je to što se više različitih problema može predstavljati pomoću binarnih nizova te je moguće koristiti iste operatore mutacije i križanja. Na taj način smanjujemo količinu teksta programa koji je potrebno implementirati kako bismo mogli rješavati raznovrsnije probleme. Nedostatak binarnog zapisa jest to što smo fiksirali točke unutar intervala koje naše rješenje može poprimiti dok nam zapis realnim brojem ne zadaje to ograničenje. Iz tog razloga možemo naše rješenje fino ugađati.

Naravno, mi možemo optimizirati raznovrsne probleme, pa će tako i naša reprezentacija problema poprimati odgovarajući oblik. Ono što je odlična stvar kod genetskog algoritma jest to što je on neovisan o problemu koji se rješava dok god

postoji odgovarajuća reprezentacija i operatori koji znaju kako raditi s istom.

2.2 Operatori križanja

Operatori križanja su operatori koji od jedinki roditelja stvaraju djecu. Postoji puno načina na koji se isti mogu implementirati ovisno o tome kako se ponašaju kod pojedinog skupa problema i pojedinih reprezentacija. Radi lakšeg razumijevanja, razmotrimo primjer križanja zapisa binarnim nizom s jednom točkom sjecišta.

Slika 2.2 – Primjer križanja zapisa binarnim nizom s jednom točkom sjecišta

Slika 2.2 nam jasno prikazuje na koji način se izvodi križanje između dva roditelja. Svaki od roditelja ima neki binarni niz koji predstavlja njegovo rješenje pojedinog problema. Operator nasumično odabere točku sjecišta S koja je unutar intervala $< 0, duljinaNiza >$. Nakon toga stvaraju se dvije nove jedinke. Prva jedinka dobit će genetski materijal koji je kombinacija genetskog materijala prvog roditelja iz intervala $[0, S]$ i genetskog materijala drugog roditelja iz intervala $< S, duljinaNiza >$

. Druga jedinka dobit će genetski materijal koji je kombinacija genetskog materijala drugog roditelja iz intervala $[0, S]$ i genetskog

$< S.duljinaNiza >$

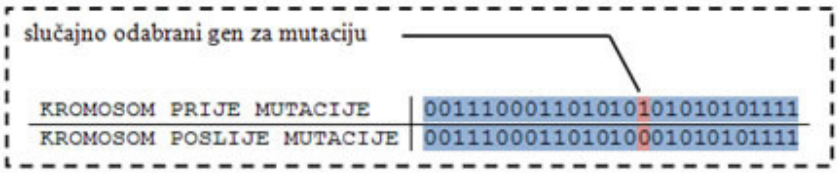
materijala prvog roditelja iz intervala .

Izvršavanjem operatora križanja dobivene su dvije nove jedinke koje su potencijano bolje od svojih roditelja. Nije nužno da su bolje, sve ovisi o tome kako je rješenje predstavljeno i na koji način sami operatori križanja rade.

2.3 Operatori mutacije

Kao kod operatora križanja, tako i kod operatora mutacije postoji velik broj različitih implementacija koje su bolje u nekim svojstvenim problemima od drugih. Isto tako, nisu svi operatori primjenjivi na sve moguće reprezentacije rješenja.

Ograničimo se opet na našu reprezentaciju binarnim nizom. Samo unutar te reprezentacije postoji velik broj operatora mutacije, te ćemo se stoga ograničiti na jedan koji se najlakše implementira. Operator jednostavno invertira jedan nasumično odabrani bit unutar našeg binarnog niza.



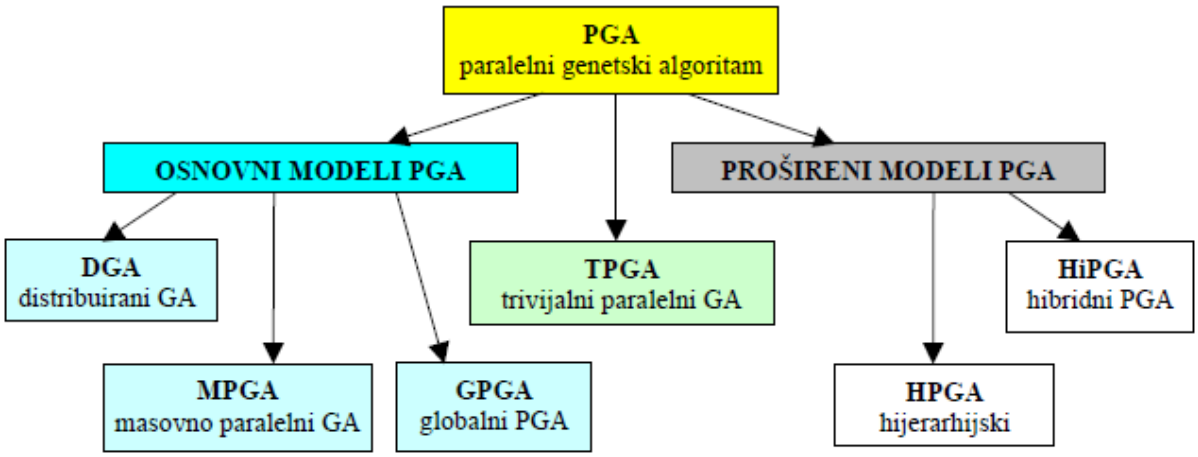
Slika 2.3 – Primjer operatora mutacije koji invertira jedan bit unutar binarnog niza

Slika 2.3 prikazuje na koji način radi operator invertiranja jednog bita. Nasumično se odabere jedna točka S unutar intervala $< 0.duljinaNiza >$. Nakon toga bit se na poziciji S invertira. Na taj način dobivena je jedinka s novim rješenjem.

Kao i kod operatora križanja, nije nužno da je tako dobivena jedinka bolja od one prije mutacije. Također, ovdje nije dobivena nova instanca jedinke, već je samo izmijenjen njen genetski materijal.

3. Paralelni genetski algoritmi

Kada govorimo o genetskim algoritmima, obično govorimo o slijednim inačicama. U današnje vrijeme, oni ne mogu u potpunosti iskoristiti svu procesnu moć računala i mreže računala. Da bi se tome doskočilo, započet je razvoj paralelnih inačica algoritama. Tako je razvijeno više inačica koje više ili manje odgovaraju slijednoj verziji algoritma od koje je razvoj inicijalno započeo.



Slika 3.1 – Podjela genetskih paralelnih algoritama

Slika 3.1 prikazuje hijerarhiju paralelnih genetskih algoritama (PGA) (3). Od svih navedenih algoritama, nama će biti najzanimljivija podjela osnovnog modela PGA o kojoj ćemo malo više reći. Unutar ovog rada, također će biti prikazana i implementacija globalnog PGA (GPGA).

Kada govorimo o paralelnim genetskim algoritmima, obično govorimo o sinkroniziranim inačicama, točnije sve migracije i komunikacije između instanci pojedinog algoritma su sinkronizirane. Nije na odmet napomenuti da postoje i asinkrone

verzije istih algoritama, no one u ovome radu neće bit razmatrane.

3.1 Raspodijeljeni GA

Ideja raspodijeljenog GA (DGA) je pokretanje više instanci GA koji rade s manjim populacijama te se periodički izmjenjuju jedinke između instanci. Algoritam je nastao s ciljem iskorištavanja umreženih računala gdje bi se na svakom računalu pokrenula jedna instanca te bi oni razmjenjivali jedinke.

```
Inicijaliziraj populacije P svih instanci
Evaluiaj početne populacije
Dok god nije zadovoljen kriterij zaustavljanja
   Za svaku instancu
      Ako je ispunjen uvjet migracije
         Migriraj jedinke
      Obavi jednu generaciju slijednog genetskog algoritma
Vrati najbolju jedinku iz populacije P dobivenom kao unija svih populacija instanci
```

Slika 3.2 – Raspodijeljeni genetski algoritam

Slika 3.2**Error! Reference source not found.** jasno prikazuje na koji način se ostvaruje paralelizacija u ovoj inačici algoritma. Instance algoritma su poprilično izolirane jedna od druge te svaka instanca pretražuje svoj prostor. Povremene migracije unose varijacije u populaciju te time pospješuju pronalaženje rješenja što bližeg optimanlnome.

3.2 Masovno paralelni genetski algoritam

Po strukturi je dosta sličan DGA uz dvije glavne iznimke.

1. Zahtjeva višeprocesorsko računalo s po nekoliko stotina ili čak tisuća

procesora

2. Migracija se obavlja samo između susjednih procesora

Svaki procesor (PE) ima pohranjenu jednu jedinku nad kojom izvršava slijedni genetski algoritam. To znači da će svaki PE nad svojom jedinkom izvršiti križanje i mutiranje dvije jedinke. Iznimka jest to što će se operator selekcije dva roditelja izvršavati paralelno budući da sam PE u sebi ima pohranjenu svoju jedinku, dok za križanje treba više jedinke. Iz tog razloga se jedinke za križanje paralelno selektiraju iz ostalih PE.

```
Inicijaliziraj paralelno populaciju P
Evaluiraj paralelno svaku jedinku
Dok god nije zadovoljen kriterij zaustavljanja
    Korištenjem selekcijskog algoritma paralelno selektiraj dvije jedinke
    Križaj odabrane jedinke
    Mutiraj djecu dobivenu križanjem
    Nadomjesti vlastitu jedinku sa novom
    Evaluiraj paralelno svaku jedinku
Vrati najbolju jedinku iz populacije P dobivenom kao unija svih populacija instanci
```

Slika 3.3 – Masovno paralelni genetski algoritam

Slika 3.3**Error! Reference source not found.** prikazuje pseudokod rada masovno paralelnog genetskog algoritma (MPGA). Glavna prednost ovakve vrste paralelizacije jest to što se porastom broja PE dobija gotovo linearno ubrzanje. Kako je veličina same populacije ograničena brojem PE, to je ujedno i veliki nedostatak samog MPGA budući da se zahtjeva računalo s velikim brojem PE. Također, uslijed velikog broja susjeda pojedinog PE, sam komunikacijski kanal za razmjenu informacija postaje usko grlo.

U slučaju da nije moguće pribaviti računalo s dovoljno velikim brojem PE, a

trenutna veličina populacije nije pogodna za dobivanje nama prihvatljivog rješenja, moguće je napraviti modifikacije samog algoritma i dozvoliti više jedinki po jednom PE.

3.3 Globalni paralelni genetski algoritam

Globalni paralelni genetski algoritam (GPGA) je vjerojatno jedna od najzastupljenijih implementacija i jedna od inačica koja najvjernije predstavlja slijednu verziju GA. Ideja ovog algoritma je postojanje jedne instance koja predstavlja voditelja (engl. *master*), dok ostale predstavljaju radnike (engl. *slave*). Voditelj obavlja sve što i klasični GA uz iznimku da koordinira radom radnika te im šalje jedinke na obradu i prima od njih obrađene rezultate. Radnici voditelja primaju jedinke te evaluiraju njihove funkcije dobrote.

Slika 3.4 jasno prikazuje na koji način se ostvaruje paralelizam kod GPGA. Dok kod MPGA jedna instanca mora biti jedan zasebni procesor, kod GPGA to nije slučaj. Na istom procesoru može biti pokrenuto više instanci radnika iako se pravo ubrzanje dobije kada koristimo jedan procesor za jednu instancu.

```
Ako sam voditelj
    Stvori početnu populaciju P
    Inicijaliziraj početnu populaciju P
Dok god nije zadovoljen kriterij zaustavljanja
    Ako sam Voditelj
        Korištenjem selekcijskog algoritma selektiraj dvije jedinke
        Križaj odabrane jedinke
        Mutiraj djecu dobivenu križanjem
        Ubaci jedinke u privremenu populaciju
        Dok nisu sve jedinke evaluirane
            Ako ima slobodnih radnika
                Odaberi određeni skup jedinki iz privremene populacije i pošalji ih
radnicima
                Čekaj na odgovor od svih radnika
                Vrati evaluirane jedinke u privremenu populaciju
            Inače
                Evaluiraj određeni skup jedinki iz privremene populacije
        Odaberi jedinke iz privremene populacije za populaciju sljedeće generacije
    Inače
        Čekaj na jedinke od voditelja
        Evaluiraj primljene jedinke
        Vrati evaluirane jedinke voditelju
Vrati najbolju jedinku iz populacije P
```

Slika 3.4 – Globalno paralelni genetski algoritam

Sam algoritam je najpogodniji u slučajevima kada imamo jako velike populacije, točnije, puno jedinki koje treba evaluirati. Tada voditelj šalje svakom od svojih radnika određenu količinu jedinki na evaluaciju. Posebno ubrzanje dobiva se u slučajevima kada je sama funkcija dobrote kompleksna i računski zahtjevna.

Valja uočiti da se za manje populacije i jednostavnije funkcije dobrote, ne isplati koristiti sam GPGA budući da bismo više vremena utrošili na sinkronizacije, kodiranje i dekodiranje jedinki te njihovo slanje.

Općenito se za manje populacije i jednostavnije funkcije dobrote ne isplati

koristiti paralelne verzije algoritama baš zbog samih migracija jedinki i sinkronizacija tih migracija.

4. Opis problema

Kao što je napomenuto na početku, danas su GA postali sve popularniji za rješavanje raznovrsnih problema iz svakodnevnog života. Uzmimo za primjer da se selimo iz jednog grada u drugi i imamo kutije različitih veličina. Kako kutija ima mnogo, unajmili smo kamion koji će ih prevesti, no najam kamiona je skup i želja nam je naći takav raspored kutija da ih možemo sve u jednom putovanju kamiona transportirati na novo odredište. Jedno od rješenja je napisati algoritam koji će grubom silom (engl. *brute force algorithm*) isprobati sve moguće kombinacije i ispisati onu koja je najpogodnija. Kako je složenost takvog problema faktorijelna, već za 13 kutija imamo preko 6 milijardi kombinacija. Druga mogućnost je implementirati jednostavan GA koji će uz dovoljno veliku populaciju i broj generacija, te pravi odabir mutacija i rekombinacija, unutar par minuta dati dovoljno dobar raspored kutija tako da uspijemo u jednom transportu prebaciti sve kutije.

Prisjetimo se jedne zanimljivosti o GA koju smo spomenuli na početku. Rekli smo da su oni općenito rješenje primjenjivo na široki skup problema, točnije sam algoritam ne zna koji problem rješava niti što je rješenje. On samo evoluira populaciju jedinki kroz generacije. Oni koji znaju kakav je problem i o čemu se radi su same interpretacije reprezentacija koji mi sami pišemo. Čak niti operatori križanja i mutacija ne znaju o čemu se radi, oni sami znaju o kakvoj se reprezentaciji radi i na koji način se s njom obavljaju određene operacije.

Vidimo da ne moramo, za svaki problem koji rješavamo, iznova pisati cijeli algoritam, već samo promijeniti kod koji interpretira reprezentaciju u naše rješenje te eventualno dodati nove reprezentacije kao i operatore mutacija i križanja za tu reprezentaciju. Vođeni takvom idejom, želimo sustav koji će nam uz malo programiranja omogućiti rješavanje širokog skupa problema. Upravo takve mogućnosti nam nudi

okruženje ECFJ. U slučaju da su odgovarajuće reprezentacije i njihovi operatori križanja i mutacije već implementirani, dovoljno je samo napisati funkciju interpretacije i odgovarajući glavni program koji će pozvati odgovarajući GA. Sve ostalo se automatski odvija budući da je već implementirano unutar samog okruženja. Također, okruženje nudi konfiguraciju preko XML (*EXtensible Markup Language*) datoteka tako da se parametri izvođenja poput broja generacija, sam GA ili veličina populacije mogu odrediti bez ponovnog prevođenja programa.

Problem samog okruženja je nedostatak podrške za paralelno računanje. Kako bismo to omogućili, potrebno je ponuditi rješenje koje će omogućiti upravo takav način računanja te jednostavnu implementaciju budućih inačica paralelnih algoritama. U svrhu toga potrebno je razviti sučelje koje će omogućiti transparentnu i neovisnu komunikaciju između instanci. Treba imati u vidu da se neće prenositi samo jedinke već i podatci poput zapisnika te bilo kakvi proizvoljni podatci.

Također je potrebno postojeće komponente, koje su implementirane unutar okruženja poput upravitelja zapisnicima i migratora, izmijeniti tako da mogu raditi i u paralelnom načinu rada.

5. Programsko rješenje

Glavnu komponentu u pružanju traženih mogućnosti predstavlja razred *Communicator*. Komponenta nudi lako ostvarivanje komunikacije i slanje raznovrsnih podataka. U pozadini se kao sredstvo komunikacije koristi sustav MPJ (4). S gledišta osobe koja će koristiti komponentu, nije joj bitno što se koristi u pozadini, ona samo vidi sučelje za komunikaciju s drugim sudionicima.

Trenutno ta komponenta nije sučelje, već je i konkretna implementacija koja koristi stvarni sustav MPJ. U slučaju da se iz nekog razloga ustanovi kako trenutna implementacija ne nudi sve potrebne mogućnosti, lako se mogu primijeniti sljedeći koraci kako bi se postigla nova funkcionalnost.

1. Izvući komponentu kao sučelje *Communicator*
2. Napraviti nove konkretne implementacije koje implementiraju sučelje *Coomunicator*
3. Omogućiti odabir implementacije komunikatora koja će se koristiti

Također, jedna od implementiranih komponenti je i GPGA. Algoritam za svoj rad koristi usluge komponente komunikatora.

5.1 Biblioteka MPJ

MPJ je sustav koji je pisan u Javi i predstavlja Java implementaciju standarda MPI (*Message Passing Interface*) (5). Sam sustav je standardizirani način na koji se šalju podatci i uspostavlja veza između dva člana komunikacijskog kanala. Sustav je neovisan o operacijskom sustavu na kojem se nalazi dok god postoji mogućnost pokretanja programa pisanih u Javi. Također, nismo ograničeni na komunikaciju s instancama koje su pokrenute na istim operacijskim sustavima, već bez problema mogu

komunicirati instanca pokrenuta na Windows operacijskom sustavu i druga koja je pokrenuta na Linux operacijskom sustavu.

Jedna od zanimljivih značajki sustava je mogućnost slanja grupnih i primanja grupnih poruka. Ako želimo poslati jednu poruku nekom podskupu sudionika koji su trenutno u sustavu, možemo definirati poruku, a MPJ će sam za nas osigurati da svaki sudionik primi poruku i još k tome obaviti potrebne sinkronizacije u slučaju sinkronog prijenosa. Alternativa bi bila da mi samo napravimo petlju ili odgovarajuću komunikacijsku strukturu koja će razaslati poruku svima koji se nalaze u tom podskupu.

Što se tiče samog slanja, na raspolaganju nam je slanje bilo kojeg primitivnog tipa kao i bilo kojeg objekta. U slučaju slanja objekta, bitno je da se taj objekt može seriajlizirati, odnosno da se može pretvoriti u niz bajtova koji će potom biti poslani preko komunikacijskog kanala i rekonstruirani na drugoj strani. To se ostvaruje tako da razred, čija instanca je taj objekt, implementira sučelje *java.io.Serializable* (6) (7).

5.2 Komponenta Communicator

Kao što je navedeno ranije, ta komponenta predstavlja glavni dio modula koji pruža podršku za implementaciju paralelnih algoritama. Nudi se nekoliko javnih metoda koje predstavljaju sučelje same komponente radi lakšeg slanja i komuniciranja sa ostalim sudionicima. U ovom kontekstu sudionik ne mora nužno biti instanca paralelnog algoritma, već može biti i komponenta koja se brine oko zapisnika. Kako se nudi slanje bilo koje vrste podataka, moguće je dodati nove komponente koje će koristiti usluge komunikatora poput komponente za praćenje statistike.

Neke od važnijih metoda dane su u nastavku.

- boolean initialize(State state, String[] args)
    - Kako se u pozadini koristi MPJ, potrebno je inicijalizirati sam MPJ

sustav. Zbog toga je potrebno proslijediti argumente naredbenog retka u polje *args*. Objekt tipa *State* predstavlja trenutni kontekst unutar kojeg se obavlja algoritam i taj kontekst je implementiran unutar samog ECFJ okruženja. Unutar ove metode se provode i inicijalizacije samog komunikatora poput određivanja naziva računala i našeg ranga unutar sustava.

- String[] getRealArgs()
    - Budući da MPJ koristi argumente naredbenog retka, naši stvarni argumenti se ne nalaze točno na onim pozicijama koje očekujemo. Radi toga je potrebno dohvatiti stvarne argumente koji bi bili proslijeđeni bez korištenja MPJ.
- boolean finish()
    - Završava komunikaciju tako da pokupi sve preostale zapisnike koji još nisu poslani sudioniku koji je započeo komunikaciju i pokrenuo sve ostale sudionike.
- boolean sendTerminateMessage(int processID, boolean termination)
    - Šalje poruku završetka predstavljenu varijablom *termination* sudioniku definiranom varijablom *processID*.
- boolean receiveTerminateMessage (int processID)
    - Čeka na poruku o zaustavljanju koju šalje sudionik definiran varijablom *processID* i vraća njen sadržaj.
- boolean sendIndividualsGlobal(Vector<Individual> pool, int processID, int numberOfIndividuals)
    - Šalje broj jedinki definiranih varijablom *numberOfIndiviruals* sudioniku definiranom varijablom *processID*. Koristi se komunikator na razini cijelog sustava. Jedinke koje se šalju uzimaju se iz kolekcije *pool*. U slučaju da je vrijednost pohranjena unutar

numberOfIndividuals manja od 1 ili veća od ukupnog broja jedinki sadržanih unutar kolekcije, šalju se sve jedinke.

- Vector<Individual> receiveIndividualsGlobal(int processID)
  - Prima kolekciju jedinki koje šalje sudionik definiran varijablom *processID*.
- boolean sendFitness(Vector<Individual> pool, int processID, int numberOfIndividuals)
  - Radi slično kao i metoda koja šalje jedinke. Jedina razlika je što se ne šalje cijela jedinka već se šalje samo objekt koji predstavlja vrijednost dobivenu evaluiranjem funkcije dobrote.
- int receiveDemeFitness(Vector<Individual> pool, int processID)
  - U jedinke koje se nalaze unutar kolekcija *pool* učitava objekt koji predstavlja vrijednost dobivenu evaluiranjem funkcije doborote. Metoda vraća broj učitanih objekata.

5.3 Problemi i ograničenja

Jedan od glavnih problema prilikom implementacije je bilo ispitivanje implementacije komunikatora. Kako se radi o paralelnim inačicama koje imaju internu sinkronizaciju, bilo je potrebno testirati na malo drugačiji način nego što se testiraju slijedni algoritmi. Kako MPJ sam stvara sve procese bilo je potrebno koristiti drugačiji način otkrivanja grešaka (engl. *debug*). U pomoć je priskočila službena dokumentacija koja objašnjava na koji način je moguće otkrivati greške u razvojnom okruženju Eclipse.

Trenutno je jedno od poznatih ograničenja nemogućnost pokretanja algoritama unutar mreže računala budući da sama implementacija MPJ ima problema s radom u tom načinu. No način rada unutar istog računala je uspješno ostvaren tako da je pokrenuto više različitih procesa. Razlog radi kojeg nije moguće pokretanje MPJ na

mreži je nepoznat, iako se u službenoj dokumentaciji navodi mogućnost takvog rada kao i demonstracijski primjeri koji se izvode na mreži.

No valja napomenuti da sama implementacija komunikatora ne ovisi o načinu na koji je u pozadini implementirana komunikacija između procesa. To ipak olakšava trenutnu situaciju budući da se može pomoću neke druge tehnologije implementirati komuniciranje preko mrežne opreme kako je to objašnjeno u poglavlju Programsko rješenje.

6. Prikaz rezultata

Kako je spomenuto ranije, paralelni algoritmi nisu najbrži u svim situacijama. Za većinu manjih problema, slijedni algoritmi bit će dosta brži od paralelnih inačica algoritama. To je iz razloga što je potrebno provesti više inicijalizacija, uspostaviti komunikacijski kanal, a i sama komunikacija između procesa zahtjeva određeno vrijeme. Također, jedinke koje se šalju potrebno je na neki način pretvoriti u zapis pogodan za prijenos komunikacijskim kanalom.

Iz tog razloga, u nastavku su dane usporedbe provedene između paralelne verzije algoritma i slijedne verzije algoritma. Rješavani problem je optimizacija funkcije sa 100 dimenzija, odnosni 100 varijabli. Provedena je usporedba ovisnosti vremena izvršavanja o veličini populacije. Broj generacija koji se koristi za oba slučaja je 10000 što je ujedno i kriterij zaustavljanja evoluiranja populacije.

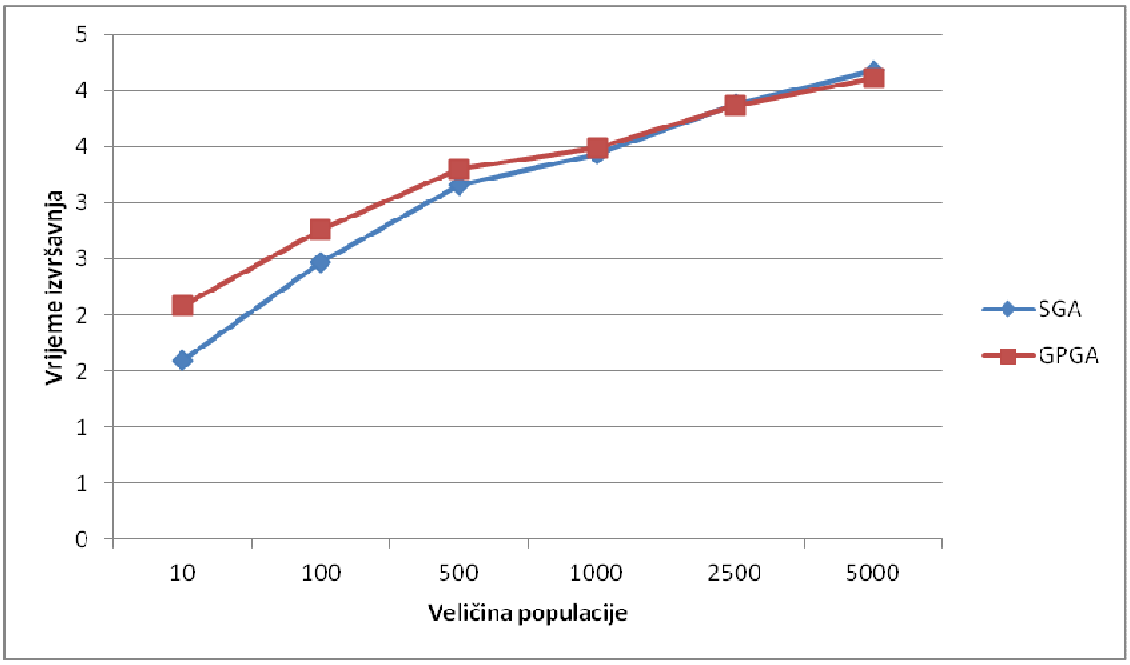
U slijednoj inačici programa se kao algoritam koristi *Steady state tournament* (jedan od algoritama okruženja ECFJ) s veličinom turnira postavljenom na 3. Paralelna inačica algoritma koristi GPGA s N-1 radnikom i jednim voditeljem gdje je N broj procesa.

U prvoj usporedbi, računalo na kojem se provodilo ispitivanje sadrži sljedeće bitne komponente.

- Procesor – Intel Core 2 Duo E4500 @ 2200 MHz
- Memorija – 4GB DDR2 @ 800 MHz (iskoristivo 3.5GB)
- OS – Windows 7 32bit

Paralelni algoritam se izvršavao na jednom računalu koristeći dva procesa koji simuliraju računala unutar mreže. Kako se ne koriste stvarna fizička računala, dobiveni

rezultati su lošiji nego što bi bili s fizički odvojenim procesorima zbog korištenja istog memorijskog prostora.

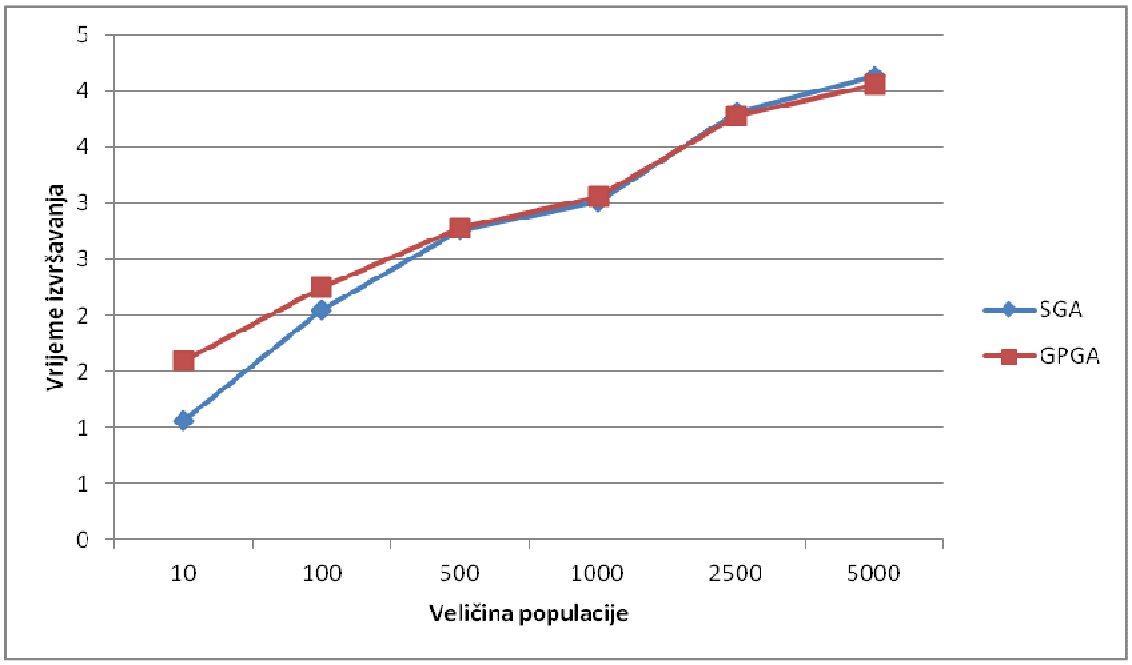


Slika 6.1 – Vremena izvršavanja u sekundama (logaritamska skala) u ovisnosti o veličini populacije za prvo računalo

U dugoj usporedbi je korišteno računalo sa sljedećim komponentama.

- Procesor – Intel i5-3570K @ 3400 MHz
- Memorija – 8GB DDR2 @ 800 MHz (iskoristivo 3.5GB)
- OS – Windows 7 32bit

Kako drugo računalo ima četiri jezgre, korištena su četiri procesa koji simuliraju računala unutar mreže.



Slika 6.2 – Vremena izvršavanja u sekundama (logaritamska skala) u ovisnosti o veličini populacije za drugo računalo

Slika 6.1 i Slika 6.2**Error! Reference source not found.** nam prikazuju ovisnost vremena izvršavanja u sekundama u ovisnosti o veličini populacije. Može se primijetiti kako za manje veličine populacija slijedni genetski algoritam (SGA) dominira nad GPGA. No kako se veličine populacija krenu povećavati, GPGA pokazuje svoju nadmoć nad SGA. Razlog tome je očit, dok kod SGA isti proces mora evaluirati sve jedinke, GPGA raspodijeli svojim radnicima jedinke koje treba evaluirati i na taj način se dobije ubrzanje. Pritom i sam voditelj vrši evaluaciju. Dio vremena koji se gubi na slanje jedinki i sinkronizaciju nadomjesti se upravo tom paralelnom evaluacijom.

Također vidimo razliku u broju procesa. Dok nam za slijednu inačicu veći broj jezgri procesora ne igra preveliku ulogu, za paralelnu inačicu se vidi znatno poboljšanje. Za slijednu inačicu bismo mogli ostvariti poboljšanje koristeći bolju arhitekturu samog

procesora, veći radni takt i bržu radnu memoriju.

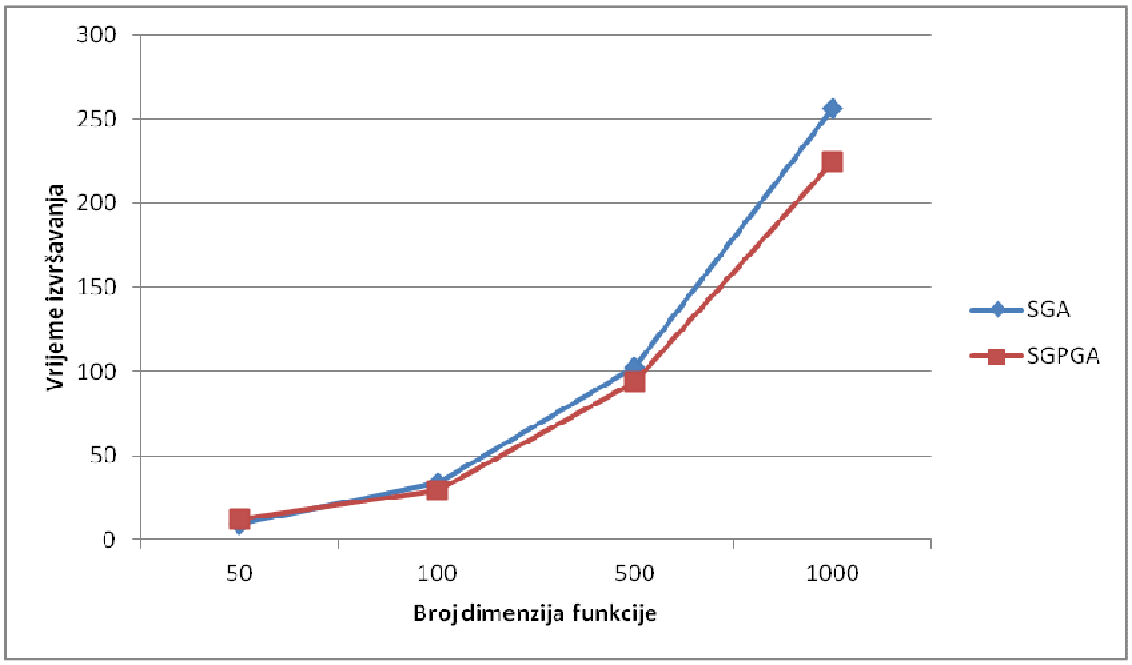
Da je funkcija dobrote bila računalno zahtjevnija, GPGA bi ranije počeo dominirati budući da bi se vrijeme izgubljeno na prijenos jedinki počelo ranije nadomještati paralelnom evaluacijom.

To nam pokazuje sljedeći eksperiment. Veličina populacije je postavljena na 100, broj generacije je postavljen na 1000 te je mijenjana kompleksnost funkcije dobrote. To je učinjeno tako da se povećavao broj dimenzija funkcije koja se optimizira.

Eksperiment je proveden na računalu sljedećih karakteristika.

- Procesor – Intel Core 2 Duo E4500 @ 2200 MHz
- Memorija – 4GB DDR2 @ 800 MHz (iskoristivo 3.5GB)
- OS – Windows 7 32bit

Kako se radi o procesoru sa dvije jezgre, stvorena su dva zasebna procesa. Jedan je predstavljao voditelja, dok je drugi predstavljao radnika.



Slika 6.3 – Vremena izvršavanja u sekundama u ovisnosti o kompleksnosti funkcije dobrote

Slika 6.3 – Vremena izvršavanja u sekundama u ovisnosti o kompleksnosti funkcije dobrote nam jasno prikazuje kako je puno veća učinkovitost paralelnog algoritma kod kompleksnije funkcije dobrote nego kod relativno jednostavne funkcije dobrote i veće populacije. Razlog tome je očit. U prva dva testiranja imali smo veći broj jedinki koje je trebalo prenijeti, evaluirati i vratiti rezultate. Sama evaluacija nije bila zahtjevna i nije došlo do prevelikog nadomještanja vremena. U zadnjem testiranju nismo imali toliko puno jedinki za prenošenje. Funkcija dobrote je bila računalno zahtjevnija u odnosu na prva dva testiranja i tu je došlo do zamjetne promjene u brzini izvršavanja.

7. Zaključak

Razvojna okruženja za genetske algoritme drastično ubrzavaju pisanje istih budući da se dobar dio koda ne mora svaki puta nanovo pisati. U većini dovoljno je samo napisati glavni program zajedno s funkcijom dobrote i povezati s okruženjem. Ako se pojavi potreba, moguće je vrlo lako dodati nove operatore i algoritme.

Kako se sve češće koriste paralelne verzije algoritama, bilo je potrebno proširiti postojeće razvojno okruženje ECFJ s podrškom za paralelno izvršavanje. To je učinjeno uz pomoć biblioteke MPJ koja omogućava relativno lako ostvarivanje komunikacije između računala unutar računalne mreže ili više procesa unutar istog računala.

Sami paralelni algoritmi nisu uvijek najbrže rješenje. U većini jednostavnih problema, slijedni algoritmi se pokazuju kao puno bolji izbor. Paralelne algoritme je puno bolje koristiti kod većih populacija ili kod računalno jako zahtjevnih funkcija dobrote.

8. Literatura

1. **Golub, Marin.** *Genetski algoritam, drugi dio.* Zagreb : an., 2004.

2. **Bradvica, Vinko.** Algoritmi koji oponašaju procese u prirodi. [Mrežno] 2007. [Citirano: 10. Ožujak 2012.] http://www.zemris.fer.hr/~golub/ga/studenti/seminari/2007_bradvica/index.html.

3. **Grupa autora.** Genetic algorithm. *Wikipedia.* [Mrežno] [Citirano: 5. Ožujak 2012.] http://en.wikipedia.org/wiki/Genetic_algorithm.

4. —. Message Passing Interface. *Wikipedia.* [Mrežno] [Citirano: 4. Ožujak 2012.] http://en.wikipedia.org/wiki/Message_Passing_Interface.

5. **Oracle.com.** Serializable interface. *Oracle.com.* [Mrežno] [Citirano: 5. Ožujak 2012.] http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html.

6. **Greanier, Todd.** Discover the secrets of the Java Serialization API. *java.sun.com.* [Mrežno] [Citirano: 5. Ožujak 2012.]

7. **Grupa autora.** MPJ Express. *MPJ Express.* [Mrežno] [Citirano: 3. Ožujak 2012.] http://mpj-express.org/.

9. Sažetak

Genetski algoritmi su se pokazali kao dobar izbor za rješavanje mnogih NP-problema kada želimo brzo pronaći dovoljno dobro rješenje. To rješenje ne mora biti čak niti blizu optimuma, dok god smo mi njime zadovoljni. Kako predstavljaju generalizirano rješenje problema, mogu se primijeniti na širok skup problema.

Razvojem računalne opreme, sve češće se pribjegava paralelnom računanju radi postizanja boljih performansi. Budući da su genetski algoritmi inicijalno osmišljeni kao slijedni algoritmi, bilo je potrebno redizajnirati algoritam kako bi on bio pogodan za paralelno izvršavanje. Osmišljeno je više inačica algoritama, no najrašireniji algoritam i algoritam koji najvjernije prikazuje genetski algoritam je globalni paralelni genetski algoritam (GPGA).

Kod GPGA postoji jedan proces koja predstavlja voditelja i više procesa koji predstavljaju radnike. Radnici primaju od voditelja jedinke na evaluaciju i njihova jedina zadaća je izračunavanje funkcije dobrote. Voditelj obavlja sve ostale zadaće.

U većini slučajeva je potrebno samo implementirati izračun funkcije dobrote i redundantno je svaki puta nanovo pisati cijeli genetski algoritam zajedno sa svim potrebnim operatorima. Upravo radi toga su se počela razvijati razvojna okruženja za genetske algoritme. U većini tih okruženja potrebno je napisati samo glavni program i funkciju dobrote. Sve ostalo je već implementirano unutar okruženja i potrebno je samo povezati komponente. Također, većina njih nudi konfiguraciju preko konfiguracijskih datoteka. Na taj način se mogu mijenjati parametri poput veličine populacije ili samog algoritma bez potrebe za ponovnim prevođenjem samog programa. Jedno od takvih okruženja je i ECFJ.

Paralelni algoritmi nisu u svim situacijama najbrži. U slučajevima kada su

populacije male ili funkcije dobrote nisu računalno zahtjevne, puno je efikasnije koristiti slijedni genetski algoritam. Razlog tome je što paralelni algoritam treba potrošiti određeno vrijeme na sinkronizacije s drugim procesima. Prenošenje jedinki komunikacijskim kanalom također troši vrijeme.

Ključne riječi: genetski algoritam, paralelni genetski algoritam, globalno paralelni genetski algoritam, razvojno okruženje, ECFJ, MPJ