SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 955

Paralelni oblici evolucijskih algoritama

Tomislav Novak

Zagreb, 19. lipnja 2009.

Sadržaj

1. Uvod

Evolucijski algoritmi, klasa algoritama inspiriranih prirodnim procesom evolucije, pokazali su se vrlo učinkovitom metodom rješavanja različitih *teških*, determinističkim algoritmima u razumnom vremenu nerješivih optimizacijskih problema.

Evolucijski algoritmi spadaju u skupinu heurističkih algoritama. Rješenje dobiveno takvim algoritmima u velikom broju slučajeva nije optimalno (rješenje za koje funkcija dobrote postiže globalni maksimum, odnosno minimum), međutim često je dovoljno dobro (prihvatljivo) za zadanu instancu problema. Budući da se određenim dijelom oslanja na slučajan odabir operacija koje određuju smjer pretrage prostora rješenja, evolucijski algoritam je *stohastička* metoda optimizacije.

Osnovna ideja iza te klase algoritama leži u mehanizmu prirodnog odabira koji je sredinom 19. stoljeća u djelu *Podrijetlo vrsta* opisao Charles Darwin: u prirodi preživljavaju one jedinke koje su najprilagođenije životnim uvjetima. Evolucijski algoritam je simulacija prirodne selekcije: svako rješenje optimizacijskog problema predstavlja jedinku koja ima određenu kvalitetu (engl. *fitness*), odredivu upotrebom funkcije dobrote nad rješenjem koje ta jedinka predstavlja. Algoritam vrši operacije nad skupom jedinki (*populacijom*). U svakoj se iteraciji izvršavanja algoritma vrši selekcija nad jedinkama, čime se (prema određenom kriteriju) eliminiraju loše jedinke. Nad preživjelim jedinkama se zatim obavljaju genetske operacije križanja i mutacije. Križanjem nastaju nove jedinke koje sadrže (u idealnom slučaju samo dobre) osobine svojih roditelja - time se usmjerava postupak pretrage prostora rješenja. Međutim, kako bi se spriječila konvergencija prema lokalno optimalnom rješenju, koristi se i operator mutacije koji može uzrokovati drastičnu promjenu u kvaliteti određene jedinke.

Očito je kako se povećanjem broja iteracija, kao i veličine populacije, ukupno vrijeme izvršavanja evolucijskog algoritma povećava. Općenito, jedan od načina ubrzavanja algoritama je i njihova paralelizacija. Dijeljenjem ukupnog opterećenja, sustav koji se sastoji od N procesora u idealnom slučaju moći će isti zadatak izvršiti u N-terostruko kraćem vremenu (ili analogno, u istom vremenu izvršiti N puta više zadataka).[1]

Evolucijski algoritmi veoma su pogodni za paralelizaciju. Štoviše, kako je i sama evolucija u prirodi paralelan proces, određene paralelne varijante evolucijskih algoritama mogu pokazati bolje karakteristike od analognih slijednih inačica (Tomassini, 2000).

U ovom je radu prikazano nekoliko načina paralelizacije evolucijskih algoritama. Implementirane su dvije varijante *globalno paralelnog evolucijskog algoritma*

[1] Takvo *linearno* ubrzanje teško je postići u praksi zbog različitih faktora poput komunikacije između procesora, međusobnog isključivanja kod korištenja zajedničkih resursa itd.

te je izvršena njihova usporedba na dva različita problema.

U sljedećem su poglavlju opisane osnovne karakteristike evolucijskih algoritama. Treće poglavlje prikazuje paralelne arhitekture te metode paralelnog programiranja, dok je u kasnijim poglavljima fokus stavljen na paralelne evolucijske algoritme te njihovu usporedbu.

2. Evolucijski algoritmi

Evolucijskim algoritmima nazivaju se pripadnici klase metaheurističkih optimizacijskih algoritama koji oponašaju biološke procese: razmnožavanje, rekombinaciju, mutaciju i selekciju.

Prema načinu implementacije i vrsti problema za koji su predviđeni, evolucijski algoritmi dijele se na *genetske algoritme*, *genetsko programiranje*[1], *evolucijsko programiranje* te *evolucijske strategije*.

Evolucijski algoritam u svakom koraku raspolaže sa skupom jedinki koji se naziva *populacija*. Svaka jedinka predstavlja određeno rješenje problema. *Genotipom* jedinke naziva se računalni zapis rješenja koji određena jedinka predstavlja. Temeljem tog zapisa mora biti moguće izračunavanje kvalitete rješenja (*fitness*) određene jedinke. Cilj EA je maksimizacija kvalitete najbolje jedinke u populaciji.

Algoritam obično započinje populacijom sastavljenom od slučajno generiranih jedinki. Te jedinke predstavljaju *početnu generaciju*. U svakom koraku izvršavanja algoritma odabire se nekoliko jedinki iz populacije te ih se modificira (upotrebom **genetskih operatora**), čime nastaje nova generacija jedinki. Ta se nova generacija koristi u sljedećem koraku algoritma.

Evolucijski algoritam izvršava se tako dugo dok nije zadovoljen kriterija zaustavljanja. Neki od mogućih kriterija su vrijeme izvršavanja, kvaliteta rješenja te broj iteracija unutar kojih nije ostvareno poboljšanje rješenja.

2.1. Genetski operatori

Evolucijski algoritmi koriste tri genetska operatora: selekcijski, mutacijski te operator križanja.

Selekcija

Selekcijom se određuje koje jedinke preživljavaju, a koje bivaju eliminirane. Svaka jedinka ima određenu vjerojatnost preživljavanja, a ta je vjerojatnost obično proporcionalna dobroti jedinke. Dvije su osnovne vrste selekcije: kod *generacijske selekcije* nova se populacija stvara kopijom određenih jedinki iz stare populacije (s time da se bolje jedinke mogu pojavljivati više puta u novoj generaciji; zastupljenost pojedine jedinke razmjerna je njezinoj kvaliteti), dok se kod *eliminacijske*

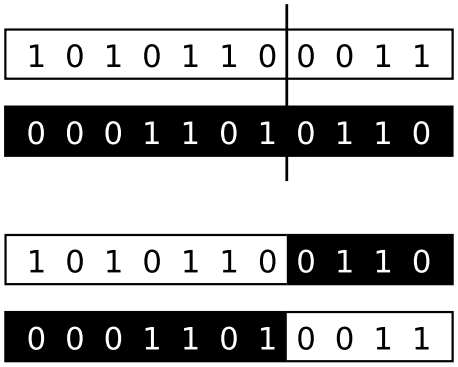
[1] Genetsko programiranje vrsta je evolucijskog algoritma gdje svaka jedinka ne predstavlja rješenje nekog optimizacijskog problema, već *računalni program*. GP je, dakle, jedna od metoda strojnog učenja.

selekcije neke jedinke eliminiraju, a zamijenjuju ih nove.

Selekcijski operator određuje način na koji se neke jedinke odabiru iz populacije (ili nekog njenog podskupa). Primjerice, kod *eliminacijske turnirske selekcije* slučajno se odabere k jedinki te se najlošija među njima eliminira. Nad preostalim jedinkama izvrše se operacije križanja i mutacije, te se eliminirana jedinka zamijeni jednom od tako dobivenih.

Križanje

Križanje je operacija kojom iz dvije jedinke roditelja nastaje jedna ili više novih jedinki, naslijedivši pritom dio genetskog materijala od svakog roditelja. Način na koji se izvodi operator križanja ovisi o zapisu jedinke te o samom problemu koji se rješava.

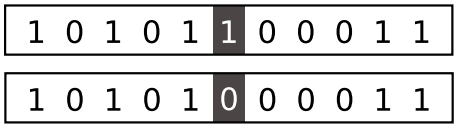


Slika 2.1: Križanje s jednom točkom prekida.

Ukoliko se za zapis jedinke koristi niz binarnih znamenaka, jedan od mogućih operatora križanja jest križanje s jednom točkom prekida (engl. *one-point crossover*), prikazano na slici 2.1: slučajno se odabere jedna točka u genotipu; u toj se točki jedinke roditelji raspolavljaju, a svaka od dvije novonastale jedinke dobiva jedan dio od prvog, a drugi od drugog roditelja.

Mutacija

Mutacija predstavlja promjenu u genetskom zapisu jedinke. Ukoliko je jedinka opisana nizom binarnih znamenaka, mutacija će predstavljati slučajnu promjenu jednog bita (slika 2.2).



Slika 2.2: Mutacija u jednoj točki.

Kao parametar evolucijskom algoritmu obično se zadaje vjerojatnost mutacije pojednog bita p_m. Ako je ukupan broj bitova n, tada je vjerojatnost da će jedinka biti izložena mutaciji (doći će do izmjene barem na jednom bitu) jednaka $p_M = 1 - (1 - p_m)^n$.

3. Paralelno programiranje

Paralelno programiranje posebna je metoda oblikovanja računalnih programa čiji je cilj povećanje performansi na način da se više naredbi izvršava istodobno. Osnova joj je činjenica da se velik broj problema može rastaviti na manje segmente koji se zatim mogu rješavati paralelno (često čak i neovisno jedni o drugima).

Promotrimo kao primjer *quicksort* algoritam. Quicksort je tipičan primjer takozvanog *podijeli pa vladaj* algoritma - u svakoj se iteraciji odabire određeni element niza (engl. *pivot*) koji se zatim smješta na konačno mjesto u nizu (proces particioniranja), nakon čega nastaju dva podniza (prvi podniz sadrži elemente manje od odabranog, a drugi veće). Na kraju se isti postupak (rekurzivno) ponavlja za svaki od ta dva podniza. Jedna od mogućih implementacija tog algoritma u programskom jeziku Haskell je sljedeća:

```
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs)
                  ++ [x] ++ qsort (filter (>= x) xs)
```

Složenost tog algoritma u prosječnom slučaju (nad svim permutacijama ulaznog niza) je $\Theta(n \log_2 n)$.

Uočimo kako su dva podniza nastala particijom neovisna jedan o drugome te se mogu dalje sortirati **paralelno**. Pretpostavimo da imamo na raspolaganju p procesora. Tada je moguće sa složenošću $\Theta(n)$ particionirati početni niz na p podnizova veličine n/p. Svaki se od tih podnizova može sortirati u prosječnom vremenu od $\Theta(\frac{n}{p} \log_2 \frac{n}{p})$, čime se dobiva (ukoliko se zanemari početno pretprocesiranje) **linearno ubrzanje**.[1] Postoje i paralelne inačice quicksort algoritma koje mogu postići vrijeme izvršavanja $O(\log_2 n)$ (Powers, 1991).

Tradicionalno, računalni su programi bili predviđeni za slijedno izvođenje. Procesor računala izvodio je instrukciju po instrukciju - sljedeća se instrukcija izvodila tek nakon što je prethodna završila (zanemarimo na trenutak instrukcijski *cjevovod*, tehniku kojom se obrada instrukcije dijeli na više koraka, od kojih se mnogi mogu istovremeno izvršavati - primjerice, izvršavanje jedne instrukcije i dekodiranje sljedeće). Povećanjem radnog takta procesora smanjuje se i vrijeme izvršavanja programa, za koje vrijedi izraz $R = I \times C \times T$, gdje je R vrijeme

[1] Ubrzanje se kod paralelnih algoritama definira kao kvocijent između vremena koje je potrebno sekvencijalnom algoritmu te paralelnoj varijanti tog istog algoritama na p procesora, tj. $S_p = \frac{T_1}{T_p}$. Ako je ubrzanje linearno ($S_p = p$), udvostručenjem broja procesora udvostručava se i brzina izvođenja algoritma.

izvršavanja, I broj instrukcija programa, C broj ciklusa po instrukciji, a T trajanje jednog ciklusa. Međutim, trajanje ciklusa sve se više približava fizikalnim ograničenjima, te je paralelizacija već neko vrijeme trend u razvoju procesora.

Paralelna računala mogu se ugrubo podijeliti prema razini na kojoj je paralelizacija hardverski podržana - od višejezgrenih i višeprocesorskih računala (veći broj jedinica izvođenja na jednom računalu) do grozdova (engl. *cluster*) i spletova (engl. *grid*) računala (više mrežno povezanih računala koja komuniciraju i zajednički rješavaju određeni problem).

3.1. Vrste paralelizma

Postoji nekoliko hijerarhijskih razina na kojima se paralelizacija izvodi: bitovna razina, instrukcijska razina, razina dretvi te razina podataka.

3.1.1. Paralelizam na razini bitova

Ovaj se tip paralelizma temelji na povećanju veličine procesorske riječi. Time se smanjuje broj instrukcija koje su potrebne kako bi procesor izvršio operaciju nad podacima većim od veličine riječi.

Primjerice, kako bi se zbrojila dva 32-bitna broja na 16-bitnoj arhitekturi, potrebno je izvršiti dvije operacije - zbrajanje nižih 16 bitova te zbrajanje viših 16 bitova, uz dodavanje prijenosnog bita (engl. *carry bit*). Povećanjem veličine riječi na 32 bita, operacija zbrajanja može se izvršiti u jednom koraku.

Paralelizam na razini bitova najniža je razina paralelizma.

3.1.2. Paralelizam na razini instrukcija

Postupak izvršavanja instrukcije je složen te je u pravilu razdvojen na nekoliko faza, kao što su dekodiranje instrukcije i spremanje rezultata u registar ili memoriju, gdje svaka faza traje jedan ili više procesorskih ciklusa. Takva podjela omogućuje paralelno izvođenje različitih faza nekoliko instrukcija (tzv. instrukcijski **cjevovod**), čime se može postići propusnost od jedne instrukcije po ciklusu.

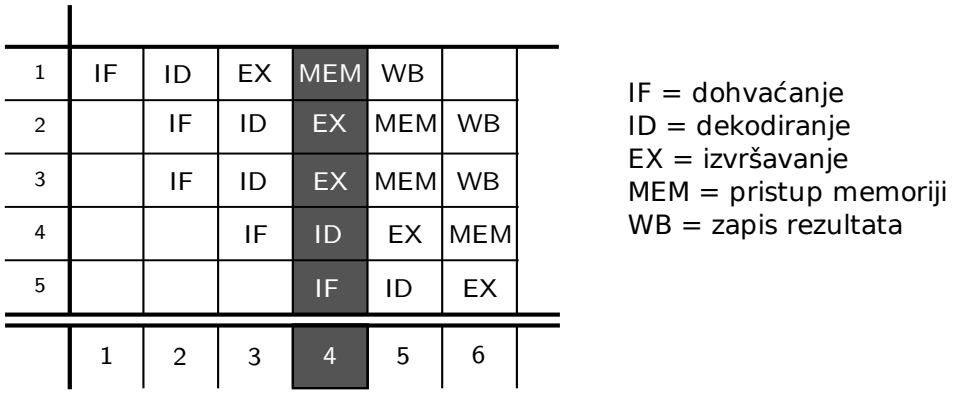
Cjevovod RISC (engl. *reduced instruction set computer*) procesora tipično se sastoji od sljedećih osnovnih faza:[2] (slika 3.1)

- **dohvaćanje instrukcije** iz memorije prema adresi koju sadrži programsko brojilo (engl. *program counter*)
- **dekodiranje**, postupak kojim se prema početnom dijelu instrukcije (tzv. *opcode*) određuje o kojoj se vrsti instrukciji radi
- **izvršavanje**, čije trajanje ovisi o vrsti instrukcije (jednostavne aritmetičko-logičke operacije u pravilu traju jedan ciklus)
- **pristup memoriji**

[2]Zanimljivo je napraviti usporedbu s Pentium 4 procesorom čiji se cjevodod sastoji od 31 faze (Prescott).

- **zapis** rezultata u odgovarajući registar (engl. *writeback*)

Jedan od problema (tzv. kontrolni hazard) koji se pojavljuju kod ovakvih arhitektura uzrokuju naredbe grananja. Primjerice, kod uvjetnog grananja često sljedeća instrukcija koja se treba izvršiti neće biti ona koja se nalazi u memoriji nakon trenutne (dok je ta već prošla kroz fazu dohvaćanja i dekodiranja). Za rješavanje takvih problema koriste se različite metode predviđanja grananja (engl. *branch prediction*).



Slika 3.1: Cjevovod RISC procesora. Na horizontalnoj osi prikazani su procesorski ciklusi, dok su na vertikalnoj osi prikazane instrukcije programa.

Važno je spomenuti i tzv. *superskalarnu* arhitekturu, nazvanu tako jer omogućuje izvršavanje više od jedne instrukcije po ciklusu. Superskalarni procesor ima više funkcijskih jedinica koje vrše dohvaćanje, dekodiranje i izvođenje instrukcija **istodobno** te svaku pristiglu instrukciju prosljeđuje jednoj od tih jedinica. Takav procesor dinamički provjerava podatkovne međuovisnosti između pojedinih instrukcija (primjerice, instrukcije `ADD EAX, 5` i `MOV EBX, EAX` ne mogu se izvršavati paralelno).

Superskalarni procesor nije jednak višejezgrenom procesoru - potonji može paralelno izvršavati više različitih dretvi, dok superskalarni procesor paralelno izvodi više (neovisnih) instrukcija iz iste dretve. Dakako, svaka jezgra višejezgrenog procesora može ujedno biti i superskalarna.

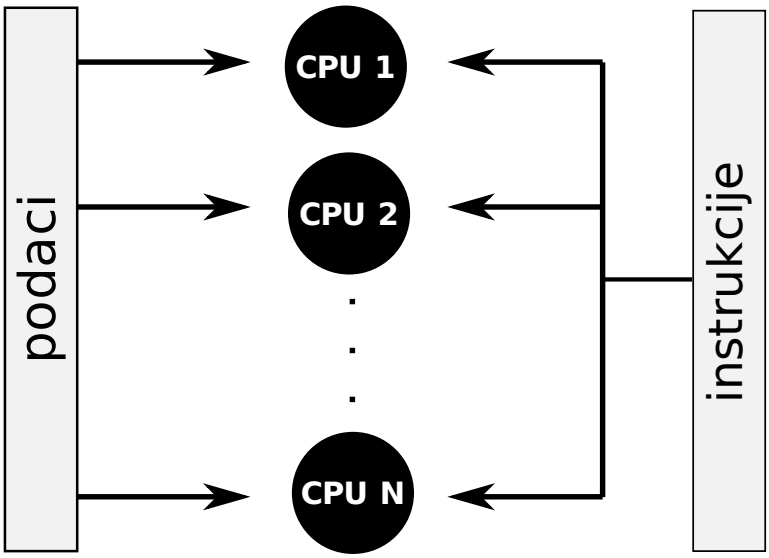
3.1.3. Paralelizam na razini dretvi

Za razliku od paralelizma na razini instrukcija gdje se određeni podskup instrukcija može (u cijelosti ili samo neke od faza u izvođenju) izvršavati istodobno na **jednom procesoru**, paralelizam na razini dretvi (tzv. MIMD model, prema engl. *multiple instructions, multiple data*) oblik je paralelizacije naredbi na **više procesora** (pritom se na pojedinom procesoru mogu izvršavati različite instrukcije na različitom skupu podataka).

3.1.4. Paralelizam na razini podataka

Paralelizam na razini podataka (SIMD model, prema engl. *single instruction, multiple data*) postiže se u višeprocesorskom sustavu gdje svaki procesor izvršava **isti skup naredbi** nad **različitim podacima**.

Primjerice, ako imamo dva procesora te želimo kvadrirati svaki element nekog niza, možemo podijeliti taj niz na dva jednaka dijela te svakom procesoru dodijeliti jednu polovicu.



Slika 3.2: SIMD model paralelnog računala. Više procesora dobiva iste instrukcije koje primjenjuje nad različitim podacima. Primjer SIMD arhitekture je grafički procesor.

3.2. Paralelna računala

Paralelnim računalom naziva se skup procesora koji zajednički mogu rješavati neki računalni problem. Ti se procesori mogu nalaziti na istoj silikonskoj pločici (**višejezgreni procesori**) ili mogu biti udaljeni kilometrima, povezani Internetom (**splet računala**).

Važan čimbenik u dizajnu paralelnih računala je i način na koji procesori pristupaju i međusobno dijele podatke. U **modelu zajedničke memorije** (engl. *shared memory*), primjerice kod višeprocesorskih računala, svi procesori dijele isti adresni prostor. Prednost takve arhitekture je jednostavan pristup zajedničkim podacima i komunikacija između procesora. Nedostatak je loša skalabilnost - povećanjem broja procesora pristup memoriji postaje usko grlo. Zbog postojanja jedinstvene sabirnice, samo jedan procesor može u određenom trenutku pristupati memoriji. Također, potrebno je poduzeti dodatne korake kako bi se osigurala koherentnost podataka koje procesori drže u priručnoj memoriji (engl. *cache coherence*).

U **modelu raspodijeljene memorije** (engl. *distributed memory*) svaki procesor ima svoj memorijski prostor te barata s lokalnim podacima. Podjela podataka i sinkronizacija odvija se porukama.

Primjer arhitekture raspodijeljene memorije je NUMA (engl. *non-uniform memory access*), kao alternativa SMP (engl. *symmetric multiprocessor*) arhitekturi. Svaki procesor ima lokalnu memoriju, čime se rješava problem smanjenih performansi zbog istovremenog pristupa memoriji od strane više procesora, no moguć je pristup i memorijama drugih procesora (taj je pristup sporiji; odatle i naziv arhitekture). Budući da više procesora često mora koristiti iste podatke, NUMA sustavi implementiraju (hardverski ili softverski) metode za kopiranje podataka između memorija pojedinih procesora.

Kombinacija spomenutih modela je **raspodijeljena zajednička memorija** (engl. *distributed shared memory*). U toj arhitekturi svaki procesor, osim lokalne memorije, ima pristup i zajedničkom adresnom prostoru.[3]

3.3. Komunikacija porukama

Važan element u razvoju paralelnih algoritama je način na koji pojedini čvorovi međusobno komuniciraju. Način komunikacije ovisi o arhitekturi - kod zajedničke memorije komunikacija je olakšana jer svi čvorovi imaju pristup istom adresnom prostoru (ključno je koristiti neku od metoda međusobnog isključivanja), dok se kod raspodijeljene memorije komunicira porukama.

OpenMP te različite implementacije dretvi (npr. *pthreads*) najčešće su korišteni API-ji kod arhitekture zajedničke memorije. Za komunikaciju porukama uglavnom se koristi *Message Passing Interface* (MPI) o kojemu će biti više riječi u nastavku, budući da se koristi u implementaciji praktičnog dijela ovog rada.

3.3.1. Message Passing Interface

MPI je standard kojim je definiran način na koji procesi mogu međusobno komunicirati porukama. Ne postoji jedinstvena, službena implementacija - neke od najpoznatijih su *MPICH*, *LAM/MPI* te *Open MPI*. Osnovne karakteristike MPI su praktičnost, portabilnost, učinkovitost i fleksibilnost.

MPI standardom definirano je sučelje za programske jezike C i Fortran, no danas postoje implementacije za ogroman broj jezika.

Najčešće svi procesi koji koriste MPI za komunikaciju izvode isti program, no moguće je i pokrenuti različite programe za različite procese.

Svaka tehnika paralelnog programiranja mora za svaki proces omogućiti sljedeće mehanizme (Jakobović, 2009):

- otkrivanje ukupnog broja procesa (`MPI_Comm_size`)
- identifikacija vlastitog procesa u grupi (`MPI_Comm_rank`)

[3]NUMA se razlikuje od DSM arhitekture utoliko što kod NUMA-e svaki procesor može pristupiti memoriji drugog procesora, dok kod raspodijeljene zajedničke memorije to nije moguće, već se može pristupiti samo dijeljenoj memoriji.

- slanje poruke određenom procesu (`MPI_Send`, `MPI_Isend`)
- primanje poruke od određenog procesa (`MPI_Recv`, `MPI_Irecv`)

MPI podržava sljedeće oblike komunikacije:

- komunikacija **od točke do točke** (engl. *point-to-point*) između bilo koja dva procesa
- **kolektivna** komunikacija (između grupe procesa)

Dva su osnovna načina komunikacije od točke do točke: **blokirajući** (npr. funckije `MPI_Send`, `MPI_Probe`) i **neblokirajući** (`MPI_Isend`, `MPI_Iprobe` itd.).

Sam MPI standard uključuje više od 100 funkcija, no većina se funkcionalnosti može postići sa znatno manjim podskupom.

4. Paralelni evolucijski algoritmi

Motivacija za paralelizaciju evolucijskih algoritama je dvojaka. Prvi je razlog poboljšanje performansi - paralelni EA može pronaći rješenje podjednake kvalitete u znatno kraćem vremenu (alternativno, u istom raspoloživom vremenu može pronaći bolje rješenje). Dodatno, kako je i sama evolucija u prirodi paralelan proces, moguće je da paralelna varijanta evolucijskog algoritma pokaže bolja svojstva (npr. izbjegavanje preuranjene konvergencije) u odnosu na slijednu verziju.

Postoji nekoliko razina na kojima se sekvencijalni evolucijski algoritam može paralelizirati: razina *populacije*, razina *jedinke* te razina *evaluacije dobrote*. Osim osnovnih modela paralelnih evolucijskih algoritama opisanih u nastavku, postoje i različiti hibridni modeli. Primjerice, *hibridni paralelni genetski algoritam* kombinira paralelizaciju na jednoj od spomenutih razina s nekim od algoritama lokalne pretrage (primjerice penjanjem uzbrdo).

Također, potrebno je spomenuti i tzv. *trivijalni paralelni genetski algoritam*. Takav algoritam se istodobno izvodi na više potpuno neovisnih računala (koja ne moraju biti čak ni međusobno povezana), te na zasebnim problemima.[1] Ova je metoda vrlo korisna za statističku analizu algoritama. Npr. isti se algoritam može pokrenuti na različitim računalima s drugačijim parametrima, te se zatim analizirati kvaliteta dobivenih rješenja.

4.1. Raspodijeljeni evolucijski algoritam

Raspodijeljeni evolucijski algoritam (engl. *distributed evolutionary algorithm*) oblik je paralelnog evolucijskog algoritma kod kojeg se paralelizacija vrši na razini populacije.

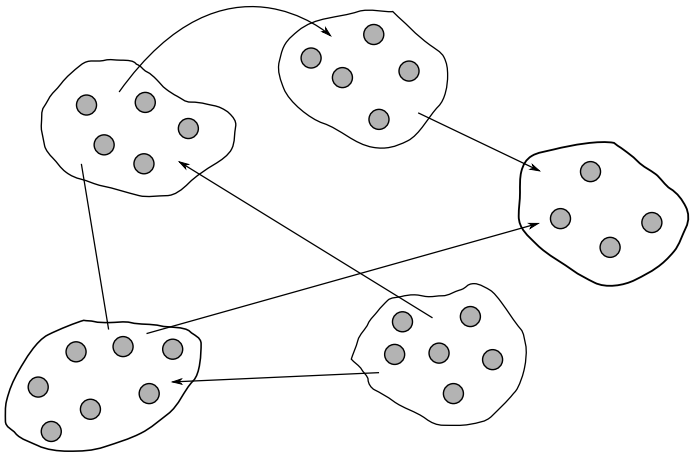
Umjesto djelovanja nad jednom velikom populacijom, DEA djeluje nad više manjih subpopulacija (otoka). Takav se model još naziva i otočnim paralelnim genetskim algoritmom (engl. *island distributed evolutionary algorithm*) ili krupnozrnatim (engl. *coarse-grained*) paralelnim evolucijskim algoritmom, upravo zbog svojstva da barata s većim skupinama jedinki (subpopulacijama).[2]

Subpopulacije su djelomično izolirane - sva računala izvode algoritam nad svojom subpopulacijom (moguće je da računala izvode različite evolucijske algoritme s različitim vrijednostima parametara - nužno je jedino da koriste isti zapis

[1] Takav se algoritam često u stranoj literaturi naziva i *embarassingly parallel*, upravo zbog toga što nije potreban gotovo nikakav trud da ga se prilagodi paralelnom okruženju.

[2] Za razliku od sitnozrnatog algoritma (masovno paralelni) koji radi s pojedinačnom jedinkom.

te rješavaju isti optimizacijski problem), no povremeno međusobno razmjenjuju genetski materijal (**migracija**). Model je prikazan na slici 4.1.



Slika 4.1: Otočni model djelomično izoliranih subpopulacija.

Ukratko, svako računalo izvršava sekvencijalni evolucijski algoritam nad svojom populacijom te svakih M_i iteracija izmjenjuje M_s jedinki s drugim računalima (prema definiranoj topologiji).

U idealnom slučaju svaki čvor pretražuje različit dio prostora problema, a migracijama se uvodi određena raznolikost u subpopulacije, čime se smanjuje mogućnost preuranjene konvergencije lokalnim optimumima.

Važno je napomenuti kako pojam migracije ne podrazumijeva doslovno prelazak jedinke iz jedne subpopulacije u drugu, već odabir boljih jedinki iz jedne subpopulacije te nadomještanje lošijih iz druge njihovim kopijama (u skladu sa strategijom odabira).

Parametri koji određuju migraciju (Golub, 2004) su *migracijski interval* M_i (broj iteracija nakon kojih se vrši migracija), *migracijska stopa* M_s (broj jedinki koje se izmjenjuju), *strategija odabira* (način na koji se odabiru bolje jedinke te jedinke za eliminaciju) te *topologija razmjene jedinki* (usmjereni graf čiji su vrhovi subpopulacije, a bridovi određuju smjerove kojima se vrši migracija).

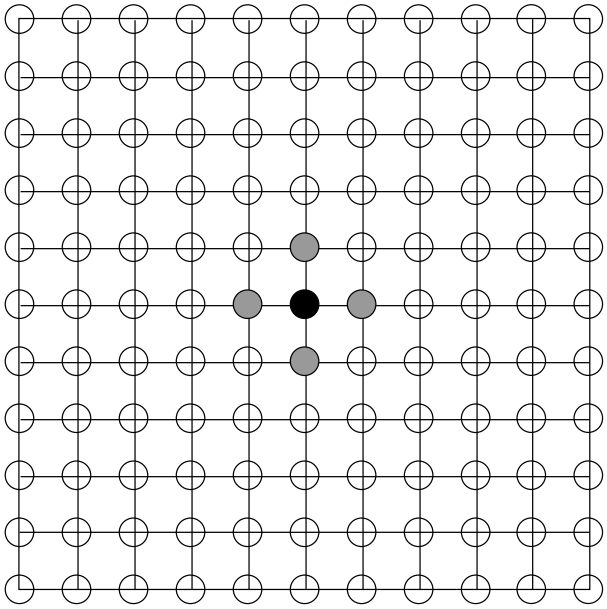
Detaljniji opis distribuiranog evolucijskog algoritma može se pronaći u (Cantú-Paz, 1998) i (Golub, 2004).

4.2. Masovno paralelni evolucijski algoritam

Masovno paralelni evolucijski algoritam, nazvan još i sitnozrnatim[3] (engl. *fine-grained*) evolucijskim algoritmom, vrši podjelu na razini jedinki. Svaki čvor paralelnog računala nadležan je za određenu jedinku, a migracija je moguća samo između susjednih čvorova. Na slici 4.2 prikazano je paralelno računalo gdje su čvorovi (logički) smješteni u pravokutnu mrežu; određenom su čvoru susjedni oni

[3]Podjela je sitnozrnata (fina) jer je svaka subpopulacija veličine jedne jedinke.

čvorovi koji se nalaze u jednom od četiri smjera (iznad, ispod, lijevo ili desno) u odnosu na nj.



Slika 4.2: Dvodimenzionalni prikaz raspodjele jedinki. Sivo su označeni mogući susjedi određene jedinke (označene crnom bojom).

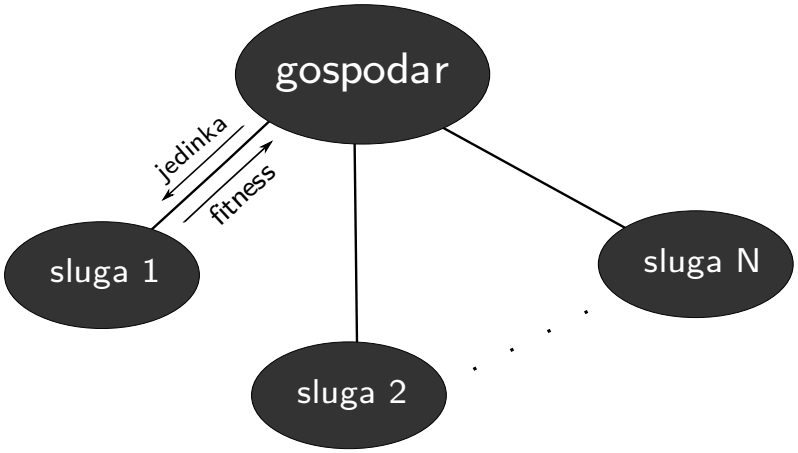
Svaki čvor paralelno obavlja sve genetske operatore. Unarne operatore (evaluacija te mutacija) čvor vrši nad njemu dodjeljenom jedinkom, dok se križanje izvodi nad vlastitom jedinkom te jednom od susjednih.

Parametri MPEA su *topologija* i *broj susjeda* S. Subpopulacije su veličine $S+1$ jedinki (određenom čvoru pridružena jedinka te S susjednih) te se međusobno preklapaju. Podjela na subpopulacije omogućava nezavisno istraživanje prostora rješenja, dok preklapanje subpopulacija omogućava širenje dobrih rješenja po populaciji (Golub, 2004).

4.3. Globalni paralelni evolucijski algoritam

Globalni paralelni evolucijski algoritam najsličniji je slijednom evolucijskom algoritmu. Kod GPEA najčešće se paralelizira samo evaluacija jedinki (izračun funkcije dobrote). Algoritam radi na principu *sluge i gospodara*. Jedan proces (gospodar) izvršava sve genetske operatore kao i kod slijednog algoritma, no posao evaluacije proslijeđuje slugama. Algoritam je globalan jer, za razliku od dva prethodno opisana algoritma, uvijek održava samo jednu populaciju - svaka se jedinka može križati s bilo kojom drugom iz populacije.

GPEA se koristi i kod arhitekture raspodijeljene memorije, kao i kod računala sa zajedničkom memorijom. Korištenje tog algoritma isplativo je ako je vrijeme potrebno za evaluaciju jedinke veće od vremena koje se troši na komunikaciju



Slika 4.3: GPEA - model gospodar-sluga. Nakon selekcije, križanja i mutacije gospodar šalje jednom od sluga novu jedinku, na što mu sluga uzvraća s vrijednošću funckije dobrote te jedinke.

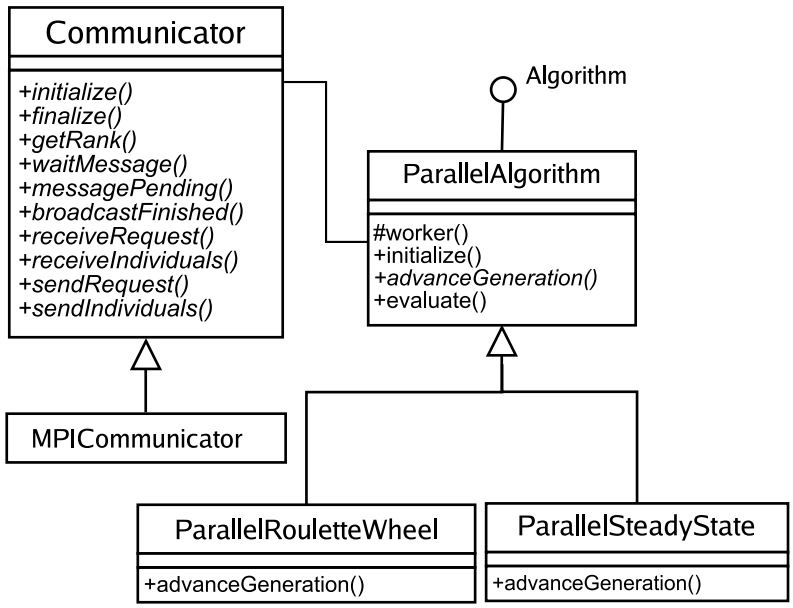
između gospodara i sluge - u protivnom, takav je algoritam po performansama čak lošiji od istovjetne slijedne varijante.

Dvije su vrste globalnog paralelnog evolucijskog algoritma: *sinkroni* i *asinkroni*. U sinkronom GPEA gospodar čeka dok i najsporiji sluga ne izračuna vrijednost funkcije dobrote. Tek kad dobije odgovore od svih slugu, gospodar prelazi u sljedeću iteraciju. Takav algoritam ima, izuzev veće brzine, ista svojstva kao i slijedni algoritam. S druge strane, asinkroni GPEA ne čeka da sluge završe s evaluacijom, već odmah prelazi u sljedeću iteraciju. Zbog toga se može dogoditi da se selekcija vrši nad jedinkama za koje spremljena vrijednost funkcije dobrote ne odgovara samom genotipu (evaluacija još nije završena).

5. Ostvarenje paralelnih evolucijskih algoritama

U sklopu praktičnog dijela završnog rada implementirane su dvije varijante globalnog paralelnog evolucijskog algoritma: *sinkroni generacijski GPEA* te *asinkroni GPEA s eliminacijskom selekcijom.*

Algoritmi su izvedeni kao proširenje radnog okvira ECF (*Evolutionary Computation Framework*)[1], sustava za različite vrste evolucijskog računanja. Slika 5.1 sažeto prikazuje podskup klasa ECF-a vezanih uz paralelizaciju.



Slika 5.1: Paralelni ECF - sažeti dijagram klasa.

Razred `MPICommunicator` implementira metode za komunikaciju između pojedinih procesa (slanje i primanje jedinki, zahtjev za poslom itd.) koristeći *Message Passing Interface* API. Budući da paralelni algoritmi pristupaju komunikatoru preko sučelja, jednostavno je komunikaciju preko MPI zamijeniti s, primjerice, OpenMP-om. Korišteni komunikator definira se u konfiguracijskoj datoteci.

`ParallelAlgorithm` apstraktna je klasa što implementira određenu funkcionalnost koju koriste izvedene klase, poput inicijalizacije komunikatora, sinkrone

[1] `http://gp.zemris.fer.hr/ecf/`

evaluacije čitave populacije itd. Budući da kod standardnog modela GPEA procesi sluge samo evaluiraju jedinke (neovisno o tome o kojem se algoritmu radi), ta je funkcionalost izvedena putem virtualne metode `worker` klase `ParallelAlgorithm`. Posao procesa gospodara pojedini algoritam implementira u metodi `advanceGeneration`.

5.1. Sinkroni generacijski GPEA

Kod generacijskog evolucijskog algoritma nova se populacija stvara od kopija određenih jedinki iz stare populacije, nad kojima se zatim izvrše operatori križanja i mutacije. Tako dobivena populacija predstavlja novu generaciju.

Selekcija jedinki iz prethodne generacije može se izvesti na mnogo načina. Jedan od njih je i tzv. *roulette-wheel* selekcija, gdje je vjerojatnost odabira pojedine jedinke proporcionalna njezinoj vrijednosti funkcije dobrote. Iz toga proizlazi da će se bolje jedinke pojavljivati više puta u novoj populaciji.

Generacijski evolucijski algoritam uglavom provodi evaluaciju nove populacije na kraju svake iteracije, nakon izvršavanja svih ostalih genetskih operatora. Upravo zbog te činjenice, takav je algoritam jednostavno paralelizirati.

U nastavku je dan pseudokod jedne varijante sinkronog generacijskog GPEA[2]. Parametri algoritma su *vjerojatnost križanja* p_c, vjerojatnost mutacije p_m, selekcijski pritisak (parametar koji određuje koliko puta najbolja jedinka ima veću vjerojatnost odabira od najlošije) te broj jedinki S koje se u jednom navratu šalju slugama.

SINKRONI-GENERACIJSKI-GPEA

```
1  odaberi N jedinki iz populacije, proporcionalno njihovom fitnessu
2  kopiraj odabrane jedinke u novu populaciju
3  slučajno odaberi p_c × N jedinki iz nove populacije te ih križaj
4  izvrši mutaciju s vjerojatnošću p_m
5  Q ← jedinke za evaluaciju
6  sve dok sve jedinke nisu evaluirane
7      radi pričekaj poruku od sluge
8          ako je sluga poslao evaluirane jedinke
9              onda ažuriraj populaciju
10         ako je primljena poruka o spremnosti i skup Q nije prazan
11             onda pošalji S jedinki iz Q sluzi na evaluaciju
```

Paralelna evaluacija jedinki obavlja se u linijama 6-11. Sve dok nije primljena i zadnja jedinka, proces gospodar prima poruke od slugu. Poruka može sadržavati:

- signal o spremnosti (koji šalje sluga gospodaru na početku te nakon evaluacije svake grupe od S jedinki)
- evaluirane jedinke

[2]Pseudokod prikazuje samo jednu generaciju, tj. opisuje implementaciju `advanceGeneration` metode procesa gospodara.

Ako je $|Q| > 0$ (a skup Q inicijalno sadrži sve jedinke iz nove populacije), sluzi se šalje S jedinki iz Q. Ukoliko poruka nosi rezultate evaluacije, vrijednosti funkcije dobrote odgovarajućih jedinki u novoj populaciji bivaju ažurirane.

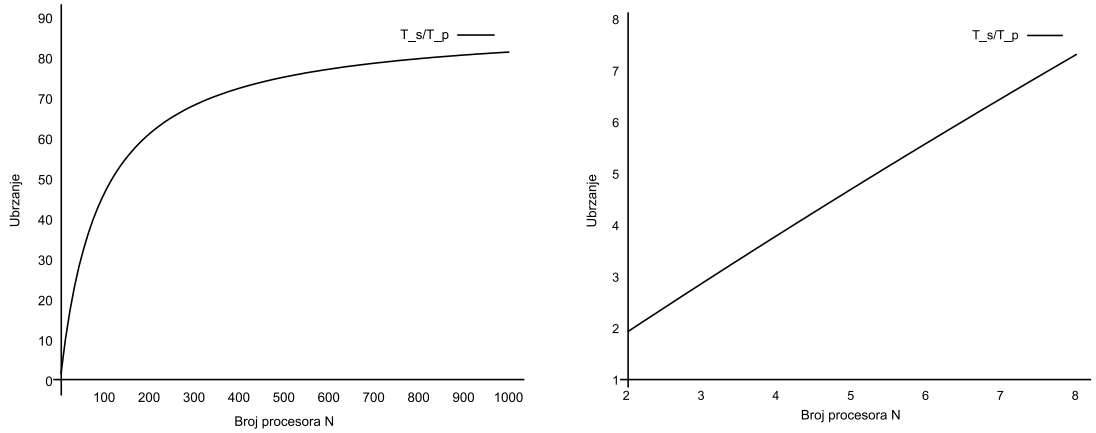
Promotrimo ubrzanje u odnosu na slijednu inačicu generacijskog evolucijskog algoritma. Pretpostavimo radi jednostavnosti analize da je vrijeme evaluacije pojedine jedinke konstantno te iznosi T_e. Također, pretpostavimo da je vrijeme potrebno za slanje grupe jedinki proporcionalno broju jedinki, uz dodani trošak slanja poruke T_m.[3] Dakle, vrijedi $T_c(S) = S \times T_{c_0} + T_m$ (ovdje koeficijent proporcionalnosti T_{c_0} predstavlja vrijeme slanja jedne jedinke)[4].

Nadalje, neka P predstavlja broj procesora-slugu, a N veličinu populacije. Vrijeme koje je potrebno slijednom generacijskom algoritmu za evaluaciju svih jedinki iz populacije iznosi $T_{slijedni} = N \times T_e$. Budući da kod paralelne inačice algoritma svih P slugu istodobno vrši evaluaciju, takvom je algoritmu potrebno $\frac{N}{P} \times T_e$ vremena. Međutim, potrebno je uračunati i vrijeme koje se utroši na komunikaciju. Kako algoritam uvijek čeka najsporijeg slugu (Golub, 2004), to je vrijeme jednako $(\frac{N}{S} + 1) \times T_c(S)$.

Kombinacijom navedenih izraza dobiva se:

$$T_{paralelni} = \frac{N}{P} \times T_e + (N + S) \times T_{c_0} + (\frac{N}{S} + 1) \times T_m$$

Ubrzanje paralelnog algoritma jednako je $\frac{T_{slijedni}}{T_{paralelni}}$. Slika 5.2 prikazuje graf ovisnosnti ubrzanja o broju procesora, uz neke konkretne vrijednosti parametara N, S, T_e, T_{c_0} i T_m.



Slika 5.2: Ubrzanje paralelnog sinkronog generacijskog GPEA. Drugi graf prikazue ograničenje funkcije na interval $[2, 8]$. Parametri su $N = 500$, $S = 10$, $T_e = 1$, $T_{c_0} = T_m = 0.01$.

Na slici se može uočiti kako se nagib krivulje smanjuje povećanjem broja procesora. Zaključujemo kako povećanje broja procesora nakon neke granice postaje

[3]Parametar T_m objedinjuje vrijeme potrebno za slanje zaglavlja poruke, kontrolnih podataka itd.

[4]Kao i za vrijeme evaluacije, ovdje je pretpostavljeno kako je i vrijeme slanje pojedine jedinke konstantno.

neisplativo zbog troškova komunikacije. Također, jasno je da je takav paralelni algoritam neučinkovit kad je $T_e < T_{c_0}$, bez obzira na broj procesora.

5.2. Asinkroni turnirski eliminacijski GPEA

Sinkroni paralelni algoritam opisan u prethodnom odjeljku nikad neće nastaviti sa sljedećom iteracijom dok nisu primljeni rezultati evaluacije svih jedinki. Zbog toga je takav algoritam, osim što je znatno brži, svojstvima potpuno ekvivalentan istovrsnoj slijednoj varijanti. Međutim, kod mnogih problema (osobito kod genetskog programiranja) evaluacija fitnessa ne traje uvijek podjednako. Također, česte su situacije (uglavnom kod paralelnih računala koja koriste model raspodijeljene memorije, npr. grozdovi) kad svi procesori nisu jednako brzi. Kao posljedica toga, proces gospodar uvijek će biti ograničen brzinom **najsporijeg** sluge.

Tom problemu može se doskočiti tako da se evaluacija izvodi **asinkrono**. Gospodar neprekinuto izvodi zadane operacije, dok sluge, čim završe s evaluacijom, ažuriraju fitness odgovarajućih jedinki.

Pseudokod jedne od mogućih implementacija asinkronog GPEA naveden je u nastavku. U algoritmu se koristi turnirska eliminacija kao način selekcije (u svakoj iteraciji odabere se k jedinki; najgora od njih se eliminira, dok se preostale križaju te se njihov potomak pridružuje populaciji).

```
ASINKRONI-TURNIRSKI-ELIMINACIJSKI-GPEA
1  n_g ← 0
2  sve dok n_g < N
3      radi eliminiraj S postojećih te stvori nove jedinke
4          ako je pristigla poruka od sluge
5              onda ažuriraj jedinke ukoliko poruka sadrži rezultate evaluacije
6                   pošalji sluzi S jedinki
7              inače samostalno evaluiraj istih S novih jedinki
8          n_g ← n_g + S
```

Prednost ovakvog algoritma je što ne troši vrijeme na sinkronizaciju. Ako u nekom trenutku nijedan od sluga nije slobodan, gospodar ili nastavlja sa sljedećom iteracijom (generira nove jedinke) ili sam evaluira jedinke.

Najveći problem takvog asinkronog algoritma je što se jedinke mogu relativno lako naći u **nepotpunom stanju**. Naime, budući da se fitness nove jedinke (nakon što ista zamijeni onu jedinku koja je prethodno eliminirana u turniru) ne izračunava i sprema odmah već tek kad neki od sluga postane slobodan te izvrši evaluacijski operator, moguće je da se u sljedećoj iteraciji za križanje odaberu jedinke čiji stvarni genotip ne odgovara spremljenoj vrijednosti funkcije dobrote (koja je preostala iz prijašnje iteracije). To povlači sljedeće (neka $\mathbf{x}$ predstavlja novu jedinku, a $\mathbf{x'}$ jedinku iz prijašnje iteracije; $f(\mathbf{x})$ je funkcija dobrote).[5]

- ako je $f(\mathbf{x}) > f(\mathbf{x'})$, može se dogoditi da se kvalitetna jedinka izgubi

[5]Ovdje je pretpostavljeno (bez gubitka općenitosti) da se funkcija $f(\mathbf{x})$ maksimizira.

- ako je $f(\mathbf{x}) < f(\mathbf{x'})$, loša jedinka može preživjeti duže nego što je očekivano

Vjerojatnost odabira nepotpune jedinke eksponencijalno opada porastom veličine populacije, pa taj problem za znatno veće N (u odnosu na broj procesora) ne dolazi do izražaja.

5.3. Usporedba

Postoji više kriterija za usporedbu paralelnih evolucijskih algoritama. Neki od njih su sljedeći:

- brzina izvršavanja i kvaliteta rješenja za predefinirani broj generacija
- kvaliteta rješenja koje algoritam postiže u određenom vremenu
- vrijeme koje je potrebno algoritmu za postizanje rješenja predefinirane kvalitete

Prvi kriterij upotrebljiv je kod usporedbe algoritama koji su funkcionalno identični, dok se drugi može koristiti kod međusobne usporedbe različitih algoritama.

U sklopu ovog rada izvršena je usporedba prethodno opisanog modela sinkronog generacijskog i asinkronog generacijskog GPEA sa njihovim slijednim inačicama. Korišten je kriterij **kvalitete rješenja** - bolji je onaj algoritam koji u manje vremena postiže rješenje određene kvalitete.

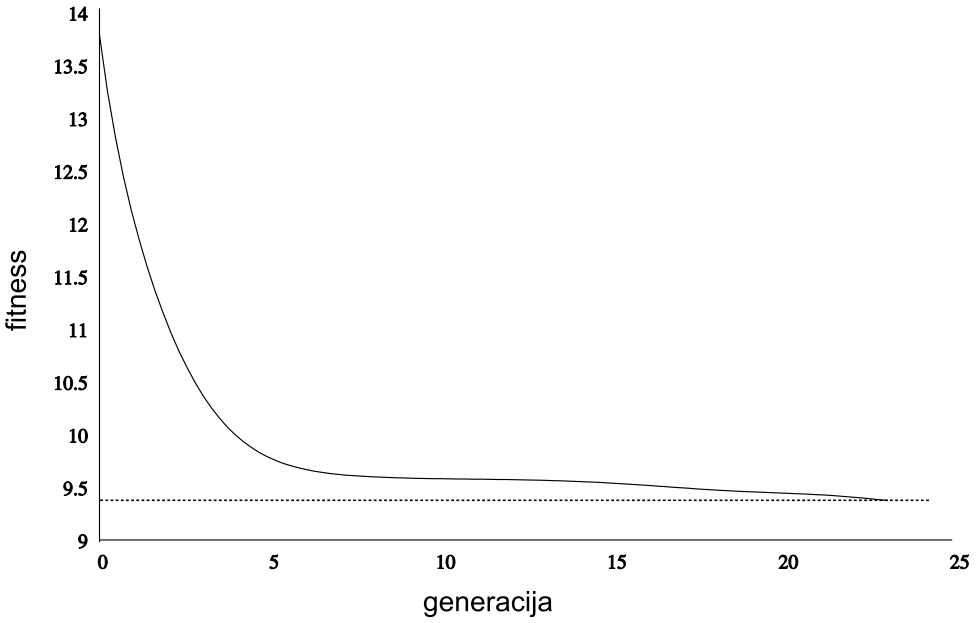
Optimizacijski problem koji je rješavan navedenim algoritmima jest *problem raspoređivanja u statičkoj okolini*, detaljno opisan u (Jakobović, 2005). Isti je primjer problema kod kojeg trajanje evaluacije jedinki nije konstantno, te je puno veće od trajanja ostalih genetskih operacija.

Korištene vrijednosti parametara evolucijskih algoritama su sljedeće:

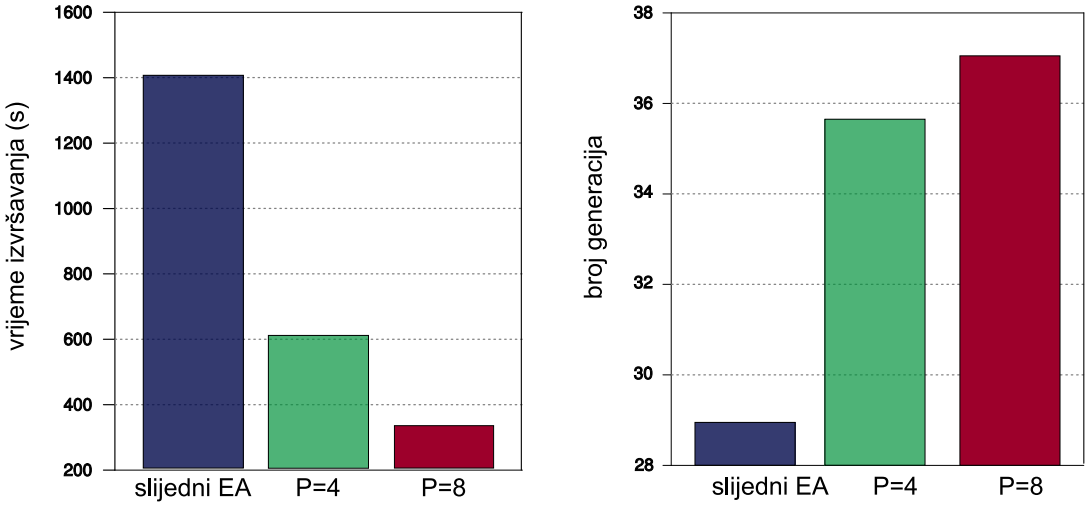
Veličina populacije (N):	500
Vjerojatnost mutacije:	0.3
Vjerojatnost križanja (generacijski algoritam):	0.5
Broj jedinki po poruci (S):	10

Na slici 5.3 prikazan je primjer izvršavanje slijednog EA s eliminacijskom selekcijom. Kao ciljna kvaliteta jedinke koristila se eksperimentalno dobivena vrijednost 9.35. Algoritam je na testnom računalu uz navedene parametre tu vrijednost postigao nakon nešto više od 26 minuta.

Slika 5.4 prikazuje prosječno vrijeme izvršavanja te broj generacija asinkronog eliminacijskog GPEA te njegove slijedne varijante. Algoritam se izvršavao do postizanja kvalitete rješenja vrijednosti 9.35. Paralelni algoritam pokrenut je prvo na četiri, a zatim na osam procesora. Ubrzanje je sublinearno zbog komunikacijskih troškova te sporije evolucije algoritma (primijetimo kako je paralelnom algoritmu prosječno potrebno više generacija za postizanje tražene kvalitete rješenja, što je posljedica problema nepotpunih jedinki).



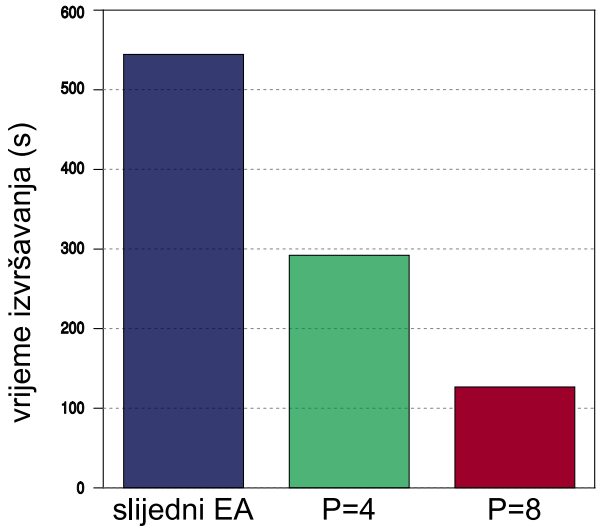
Slika 5.3: Izvršavanje slijednog evolucijskog algoritma s eliminacijskom selekcijom do postizanja tražene kvaltete jedinke. Na horizontalnoj osi prikazan je redni broj generacije, a na vertikalnoj iznos funkcije dobrote (fitness) najbolje jedinke u populaciji.



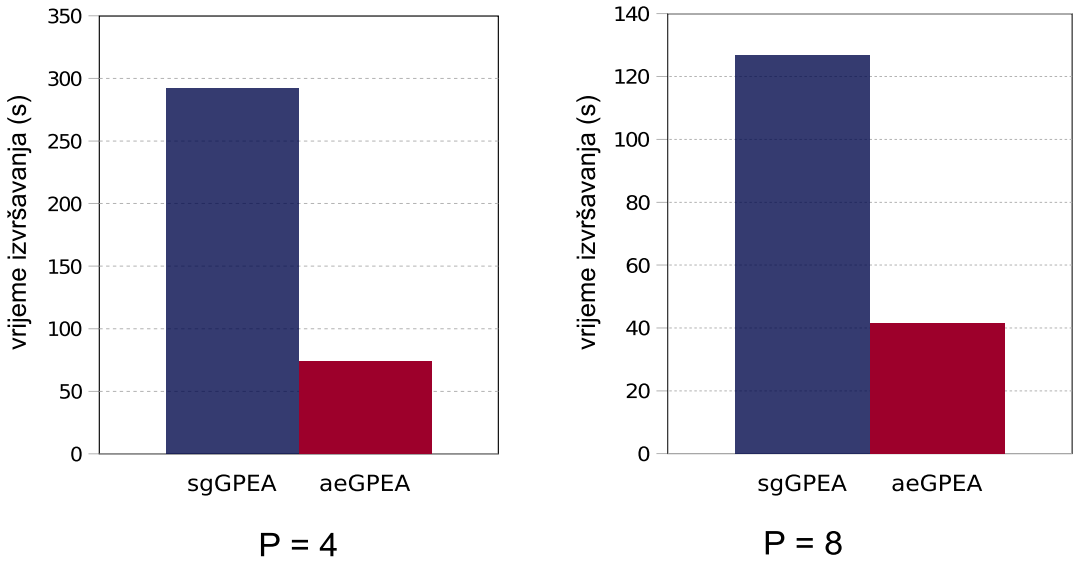
Slika 5.4: Usporedba asinkronog eliminacijskog GPEA sa slijednom inačicom. Na vertikalnoj osi prikazano je prosječno vrijeme, odnosno prosječan broj generacija potrebnih algoritmu da postigne željenu kvalitetu rješenja.

Na slici 5.5 nalazi se prikaz vremena izvršavanja sinkronog generacijskog GPEA, te istovrsnog slijednog algoritma.

Mjerenja su pokazala (slika 5.6) da paralelni asinkroni algoritam s eliminacijskom selekcijom usprkos nešto sporijoj evoluciji daje daleko najbolje rezultate.



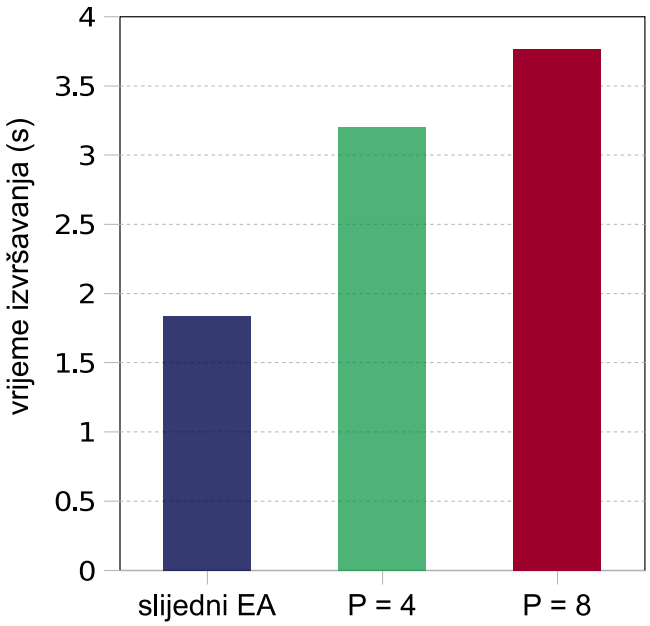
Slika 5.5: Sinkroni generacijski GPEA te njegova slijedna inačica. Ciljno rješenje je kvalitete vrijednosti 10.



Slika 5.6: Usporedba sinkronog generacijskog GPEA i asinkronog eliminacijskog GPEA. Oba algoritma izvršavala su se do postizanja fitnessa 10.

Važno je ponovo naglasiti kako prednosti paralelizacije na razini evaluacije jedinki dolaze do izražaja prvenstveno u situacijama kad upravo evaluacija predstavlja usko grlo, kao što je slučaj s problemom raspoređivanja. Ukoliko je evaluacija jedinki samo vremenski nezamjetni dio cjelokupnog evolucijskog ciklusa, vrijeme izvršavanja GPEA biti će čak i duže od slijednog EA zbog dodatnog troška uzrokovanog prijenosom jedinki i kontrolnih poruka između procesora. Slika 5.7 prikazuje vrijeme izvršavanja slijednog eliminacijskog EA te asinkronog elimi-

nacijskog GPEA na tzv. *onemax* problemu (problem maksimizacije broja jedinica u nizu binarnih znamenaka), kod kojeg je evaluacija jedinki procesorski nezahtjevan posao - izvodi se sa $\Theta(N)$ (gdje je N broj bitova u zapisu pojedine jedinke) jednostavnih bitovnih operacija.



Slika 5.7: Usporedba paralelnog i slijednog algoritma na *onemax* problemu. Veličina populacije je 100, a svaka jedinka sadrži niz od 1000 binarnih znamenaka. Uvjet završavanja algoritma je postizanje optimalne jedinke (one koja sadrži samo jedinice).

Literatura

Erick Cantú-Paz. A survey of parallel genetic algorithms, 1998.

Marin Golub. Genetski algoritam, drugi dio, 2004.

Domagoj Jakobović. Predavanja iz kolegija paralelno programiranje, 2009.

Domagoj Jakobović. Raspoređivanje zasnovano na prilagodljivim pravilima, 2005.

David M. W. Powers. Parallelized quicksort with optimal speedup, 1991.

Marco Tomassini. Parallel and distributed evolutionary algorithms: A review, 2000.