Paralelizacija SAT-rješavača za alokaciju registara

Ivan Cvrk ivan.cvrk@fer.hr
Vinko Đurić vinko.duric@fer.hr
Fran Fodor fran.fodor@fer.hr
Andrija Krklec andrija.krklec@fer.hr
Anđelko Kućar andelko.kucar@fer.hr

***Sažetak*—Tema ovog projekta je ubrzanje alokacije registara koristeći paraleliziran SAT solver. Problem alokacije registara NP-potpun problem s kojim se suočava svaki kompilator pri prevođenju višeg programskog jezika u asemblerski jezik. Kao što je poznato, registri su način spremanja podataka u računalu čije je vrijeme pristupa najmanje, no broj registara u računalu je ograničen pa je upravljanje njima te način dodjele registra nekoj od varijabli međukoda od iznimne važnosti za brzo izvođenje nekog programa. Kako bi se alokacija uspješno napravila, za međukod određuje se raspon u kojem se neka od varijabli koristi te se na temelju toga gradi graf ovisnosti. Izgrađeni graf se pokušava obojiti s pomoću ograničenog skupa boja koje za naš problem predstavljaju registre kojih također imamo samo ograničen broj. Kako smo problem alokacije registara sveli na problem bojanja grafa (točnije bojanje vrhova grafa), za njegovo rješavanje koristit će se SAT solver. SAT solver će na temelju dane formule u konjunkcijskom normalnom obliku (CNF) uz primjenu DPLL (Davis-Putnam-Logemann-Loveland) algoritma pokušati dodijeliti registre (boje) pojedinim varijablama međukoda. U ovom projektu pokušat ćemo ubrzati rad SAT solvera tako što ćemo dodjelu logičke vrijednosti literalu izvoditi za više njih istovremeno na nekoliko dretvi. Rezultate za različit broj dretvi usporedit ćemo te na temelju njih donijeti zaključke o uspješnosti našeg projekta. Za izradu implementacije odabran je programski jezik Rust koji preko svojim knjižnica nudi vrlo dobru podršku za konkurentnost, paralelizam, višedretvenost te sinkronizaciju dretvi.**

***Index Terms*—SAT, prevodilac, paralelizacija, bojanje grafa, CNF, Rust, alokacija registara**

I. Uvod

Moderna računala pokušavaju poboljšati svoje performanse na razne načine, kroz sklopovska rješenja, ali i programska rješenja. Kako bi se povećala performansa izvođenja nekog programa na računalu, razmatra se kvaliteta strojnog koda koji se izravno izvodi na računalnom sklopovlju te načini kako povećati istu. S obzirom na to kako su računala fizički izvedena, resursi kojima računalo raspolaže su ograničeni. Rad procesora računala najopćenitije možemo svesti na dvije faze, a to su dohvaćanje i izvršavanje instrukcija. Dohvaćanje instrukcija uključuje pristup memoriji, što može biti skupo s obzirom na to kako današnji procesori rade na znatno većoj frekvenciji signala takta u odnosu na radnu memoriju. Tom problemu dugog pristupa radnoj memoriji doskočilo se priručnim memorijama. Osim instrukcija, procesor također dohvaća i podatke iz memorije te ih sprema u registre. Svi ili gotovo svi današnji procesori su registarski orijentirani, odnosno podatke spremaju u registre te sve operacije izvode nad njima. Glavna prednost registara nad memorijom bilo koje vrste je vrijeme pristupa koje kod registara poprima najmanji iznos. S druge strane, nedostatak registara je njihov ograničen broj. Lako je zaključiti kako je cilj da program napisan u nekom višem programskom jeziku preveden u asemblerski jezik što bolje upotrebljava ograničen skup registara s kojima procesor raspolaže. To je jedan od zadataka prevodioca, točnije kompilatora. Prevodilac na temelju ranije izgrađenog međukoda određuje raspon unutar kojeg se upotrebljava varijabla te se na temelju tih raspona gradi graf zavisnosti. Općenito graf se sastoji od svojih vrhova i bridova, a u ovom konkretnom slučaju vrhovi su varijable međukoda, a bridovi između dva vrha postoje ako postoji međuovisnost između raspona dvije varijable. Nad tako izgrađenim grafom potrebno je provesti bojanje. Jedan od načina za bojanje grafa je preko SAT-rješavača. SAT-rješavači na temelju dane Booleove formule pokušavaju odrediti postoji li rješenje za danu formulu. Dodjeljivanjem boja varijablama međukoda, zapravo se radi alokacija registara. Dijelovi rada SAT-rješavača su neovisni pa se stoga mogu pokušati izvoditi konkurentno, odnosno paralelno na više jezgri računala te tako postići ubrzanje rada, odnosno povećanje performansi.

II. Pozadina

A. Alokacija registara

Zadatak alokacije registara je NP-potpun problem koji je neizbježan dio svakog prevođenja. Alokacija registara nije sasvim jednostavan zadatak pri sintezi programa u ciljni (asemblerski) jezik te se sastoji od nekoliko koraka [1].

Početni korak pri alokaciji registara je određivanje raspona u kojem se varijabla međukoda koristi te taj raspon nazivamo mrežicom (engl. live range). Mrežice se određuju tako što se za varijablu međukoda određuje mjesto njene deklaracije te mjesto kada je ona zadnji put upotrebljavana. Sasvim je uobičajeno koristiti nekoliko varijabli kroz veći dio koda i u višem programskom jeziku pa je slično i u dobivenom međukodu. Iz tog razloga, mrežice za određene varijable međukoda će se preklapati te je taj odnos potrebno nekako prikazati.

Stoga se u sljedećem koraku gradi graf zavisnosti mrežica G=(V,E), gdje V predstavlja skup mrežica, odnosno vrhova, a E skup bridova kojima su vrhovi povezani. Graf se gradi tako što se svakoj mrežici pridruži jedan vrh. Zatim se za svaki vrh određuju susjedni vrhovi. Dva vrha su susjedna ako imaju zajedničkih naredbi ili su obje mrežice aktivne tijekom nekih naredbi. Primjerice, ako se unutar jedne mrežice definira druga mrežica, tada su njihovi pripadni vrhovi susjedni. Iznimka za prethodno opisani postupak su naredbe koje bi se u asemblerski jezik prevele u instrukciju koja samo kopira sadržaj jednog registra u drugi pa između ne postoji međuovisnost.

U sljedećem, trećem, koraku cilj je pronaći mrežice koje se mogu spojiti u jednu zajedničku mrežicu te tako pojednostaviti graf zavisnosti. Uvjet koji pri tome mora biti zadovoljen je

da postoji naredba koja samo kopira jednu varijablu u drugu te nakon toga te dvije mrežice ne ovise jedna o drugoj. Kopiranjem vrijednosti se ona ne mijenja, a ako se kasnije u početnu varijablu koja je kopirana ništa ne upisuje, tada dvije varijable možemo gledati kao jednu zajedničku bez operacije kopiranje jer ona tada postaje nepotrebna. Takve dvije mrežice su zapravo jedna mrežica te se one mogu spojiti. S obzirom na to da se takvim postupkom stapanja mrežica smanjuje broj vrhova grafa, potrebno je ponovno izgraditi graf zavisnosti, sada s novim skupom vrhova.

Broj registara je konačan, no broj varijabli i mrežica nije konačan te taj broj može biti veći od broja registara. Kod većih programa za koji prevodilac generira međukod s velikim brojem mrežica koje u grafu zavisnosti imaju puno susjeda, alokacija registara nije sasvim trivijalna jer za neke dvije ili više mrežica nije moguće za svaku instrukciju dodijeliti registre zbog njihovih međuovisnosti.

Četvrti korak alokacije registra je određivanje koje će se varijable privremeno pohraniti u memoriju te kasnije ponovno dohvatiti u registar. U ovom koraku pokušava se odrediti varijabla čija će privremena pohrana u memoriju biti optimalna, odnosno najmanje će utjecati na brzinu izvođenja programa.

Na temelju grafa zavisnosti i varijabli čije je pohranjivanje u memoriju optimalno izvršava se bojanje grafa. Za ovaj slučaj, riječ je o bojanju vrhova grafa. Općenito govoreći, bojanje vrhova grafa postupak je u kojem se svakom vrhu pokušava dodijeliti jedna od k boja tako da dva susjedna vrha nisu obojena istom bojom. Najmanji k za koji je moguće obojiti sve vrhove grafa naziva se kromatski broj. Za problem alokacije registara, vrhove grafa gledamo kao mrežice, dok boje gledamo kao fizičke registre procesora. Kromatski broj pri alokaciji registara može biti najviše jednak broju registara procesora. Za bojanje vrhova grafa postoje razni algoritmi, a postupak se može svesti i na Booleovu formulu te taj problem riješiti pomoću SAT-rješavača.

U zadnjem koraku ubacuju se memorijske naredbe ako su one potrebne. Memorijske naredbe su potrebne ako graf nije moguće obojiti s onoliko boja koliko procesor ima registara na raspolaganju. Ubacivanjem memorijskih naredbi za neku od mrežica, ona se uklanja pa je potrebno ponovno provesti kompletan postupak alokacije. Ako se s druge strane graf uspješno oboji u prethodnom koraku, tada postupak alokacije registara završava te se može prijeći na sljedeći korak sinteze programa u asemblerskom jeziku.

B. SAT-rješavači

Kako je prethodno spomenuto, bojanje grafa može se provesti pomoću raznih algoritama koji izravno rade s grafom, no također je moguće provesti i uz pomoć SAT-rješavača (engl. SAT-solver). Zadatak SAT-rješavača je odrediti zadovoljivost neke dane propozicijske formule. SAT-rješavači imaju široku primjenu. SAT-rješavači koriste se za formalnu verifikaciju računalnih sustava, od programskih do sklopovskih, imaju razne primjene u umjetnoj inteligenciji i sl. Svaku propozicijsku formulu moguće je prikazati u dva normalna oblika, disjunkcijskom normalnom obliku (DNF) i konjunkcijskom normalnom obliku (CNF). Normalne oblike propozicijske formule u Booleovoj algebri mogli bismo interpretirati kao sumu produkata za DNF, odnosno produkt suma za CNF. Oba normalna oblika sastavljena su od varijabli koje se nazivaju literali. Varijabla može biti u negiranom ili nenegiranom obliku. Literali su povezani logičkim veznicima konjunkcije, disjunkcije, implikacije ili ekvivalencije. Konjunkcijski normalni oblik čine klauzule povezane konjunkcijom, dok su literali unutar jedne klauzule povezani disjunkcijama. Uzimajući u obzir složenost rješavanja, formule u konjunkcijskom normalnom obliku imaju veću složenost, osobito one formule koje unutar jedne klauzule imaju 3 ili više literala te je takav problem NP-kompletan. Formule u CNF obliku s klauzulama koje su sastavljene od barem 3 literala još se nazivaju i 3SAT. SAT-rješavač na ulazu prima formulu te pokušava pridijeliti logičke konstante (istina, odnosno 1 i neistina, odnosno 0) literalima tako da cijela formula bude istinita. S obzirom na to kako je vremenska složenost rješavanje 3SAT problema eksponencijalna, iscrpno pridjeljivanje logičkih konstanti često ne daje zadovoljavajuće rezultate pa su iz tog razloga osmišljeni razni algoritmi koji u prosjeku daju bolje rezultate. Neki od takvih algoritama koji se najviše koriste u praksi su DPLL (Davis–Putnam–Logemann–Loveland) i konfliktom vođeno učenje klauzula (engl. conflict-driven clause learning) ili skraćeno CDCL.

III. Eksperiment

U ovom radu cilj je pokazati paralelizaciju SAT-rješavača koji se koristi pri prevođenju programskog jezika u asemblerski jezik, točnije pri alokaciji registara. SAT-rješavač bio bi zadužen za jedan od koraka pri alokaciji registara, a to je bojanje vrhova grafa zavisnosti mrežica. SAT-rješavač izgrađen je tako da zadovoljivost određuje s pomoću DPLL (Davis–Putnam–Logemann–Loveland) algoritma. Kao jezik izgradnje odabran je programski jezik Rust koji svojim knjižnicama kao što su std, rayon, crossbeam nudi podršku za pisanje programskih rješenja čija je tema paralelizacija, konkurentnost, višedretvenost i sinkronizacija dretvi. Za potrebe ovog rada, korištena je knjižnica rayon. Nešto popularniji algoritam je CDCL koji je svojevrsna nadogradnja DPLL algoritma.

A. DPLL algoritam

Davis–Putnam–Logemann–Loveland (DPLL) je algoritam koji se koristi u rješavanju 3SAT problema. Algoritam se razlikuje od iscrpnog pridjeljivanja logičkih konstanti po tome što uvodi heuristiku. Glavne prednosti ovog algoritma su jedinična propagacija (engl. unit propagation) i jednoznačna eliminacija literala (engl. pure literal elimination) [2].

Jedinična propagacija je postupak u kojem se traže klauzule u kojime je moguće dodijeliti logičku konstantu samo jednom literalu. Ako samo jedan literal nema dodijeljenu vrijednost, tada kako bi cijela klauzula bila istinita (što je nužan uvjet kako bi cijela formula mogla biti zadovoljiva) preostalom literalu se jednostavno može dodijeliti logička konstanta.

Jednoznačna eliminacija literala je traženje literala koji mogu poprimiti samo jednu vrijednost. Takve literale moguće je pronaći tako što se oni u svakoj klauzuli u kojoj se nalaze, pojave samo u jednom obliku, negiranom ili nenegiranom.

Za takve literale jednostavno je odrediti vrijednost čime se smanjuje složenost rada SAT-rješavača.

Rad algoritma može se opisati sljedećim pseudokodom:

Function `dpll(`l`)`**:**
  foreach $c \in jedinicneKlauzule$ **do**
    `jedinicnaPropagacija(c);`
  end
  foreach $l \in jednoznacniLiterali$ **do**
    `jednoznacnaEliminacijaLiterala(l);`
  end
  if φ *je prazna* **then**
    return zadovoljiva;
  end
  if $\exists prazna\_klauzula \in \varphi$ **then**
    return nije_zadovoljiva;
  end
  $y \leftarrow$ `odaberiLiteral();`
  return `dpll(`$l = 1$`)` $\vee$ `dpll(`$l = 0$`);`
End Function

Algorithm 1: Pseudokod DPLL algoritma

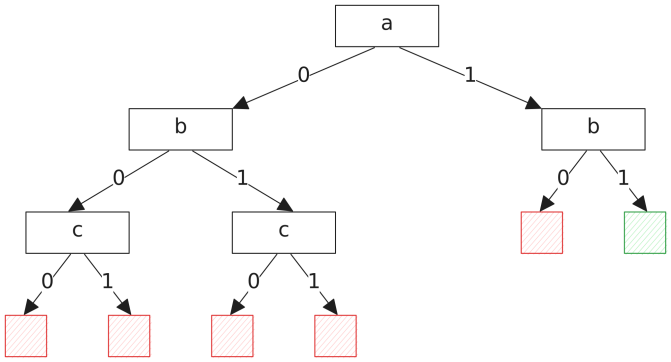
Algoritam radi tako da "odabere" neki literal kojem pridruži jednu od dvije logičke konstante. Na taj način, u ovaj algoritam uvodi se heuristika. Dodjelom logičke konstante potrebno je ispitati postoje li sada klauzule sa samo jednim literalom bez dodijeljene logičke konstante kao i postoji li jednoznačni literal. Ako je nešto od toga zadovoljeno, tada se može pojednostaviti formula. Tijekom provođenja postupka jedinične propagacije ili jednoznačne eliminacije literala, klauzule nad kojima se može primijeniti akcija trivijalne dodijele logičke konstante se brišu iz formule. Općenito, kada neka klauzula postaje zadovoljiva, ona se briše iz formule. Ako je nakon svi mogućih dodjela logičkih konstanti unutar funkcije (samo u tom pozivu funkcije) formula prazna, odnosno ne sadrži niti jednu klauzulu jer su prethodno obrisane, može se zaključiti kako je formula zadovoljiva. S druge pak strane, ako je neka od klauzula prazna, odnosno dodijeljene su joj sve logičke konstante, no ona nije izbrisana, može se zaključiti kako formula nije zadovoljiva za dodijeljene vrijednosti literalima. Algoritam gradi stablastu strukturu, točnije binarno stablo. Stablo se gradi kroz rekurzivne pozive funkcije `dpll`. Prije svakog poziva odabire se novi literal te se rekurzivno pozivaju dvije funkcije u kojima se formule razlikuju. Formule se razlikuju tako što je u jednom pozivu za prethodno odabrani literal pridružena njegov nenegiran oblik, dok je u drugom pozivu literalu pridružena njegov negiran oblik.

B. Paralelizacija

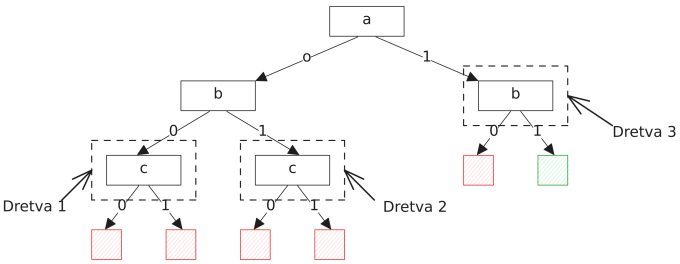
U ovom radu, naglasak je na iskorištavanju paralelizma. Paralelizam možemo podijeliti na 4 vrste, paralelizam na razini instrukcija (engl. instruction-level parallelism), na razini podataka (engl. data-level parallelism), na razini dretvi (engl. thread-level parallelism) i na razini zahtjeva (engl. request-level parallelism) [3]. SAT-rješavač izgrađen u sklopu ovog rada koristi paralelizam na razini dretvi. Paralelizam na razini dretvi karakterističan je za višeprocesorske/višejezgrene sustave, odnosno računala tipa MIMD (engl. multiple instructions multiple data) prema Flynnovoj taksonomiji.

Paralelizam se kod SAT-rješavača iskorištava tako što se prilikom rekurzivnog poziva funkcije `dpll` jednoj od dretvi iz bazena dretvi pridružuje pozvana funkcija. Na taj način, stablasta struktura se gradi paralelno, odnosno moguće je postići paralelno ispitivanje više kombinacija vrijednosti literala. Funkcije pozvane na jednoj razini stabla mogu raditi potpuno neovisno te je stoga potencijal za ubrzanje programa paralelizacijom velik.

Rad serijskog i paralelnog algoritma slikovito je prikazan na slikama 1 i 2.



Slika 1. Rad serijskog algoritma



Slika 2. Rad paralelnog algoritma

C. Testni skup

SAT-rješavač je dio kompilatora kojem je zadatak obojiti vrhove grafa. Prevodilac je u prethodnim koracima izgradio graf zavisnosti mrežica te je posao SAT-rješavača samo bojanje grafa. SAT-rješavač radi s formulama zapisanima u konjunkcijskom normalnom obliku, dok prevodilac u prethodnim koracima gradi graf. Prije bojanja je stoga potrebno pretvoriti graf u CNF formule.

Graf zavisnosti mrežica sastoji se od skupa vrhova koji predstavljaju mrežice, a mrežicu možemo promatrati kao životni vijek neke varijable. Boje kojima je potrebno obojiti vrhove promatramo kao fizičke registre procesora. Literali se stoga grade tako što se svakom vrhu dodijeli boja, tada je literal oblika xy, gdje je x vrh grafa, a y jedna od boja koje su na raspolaganju.

Pretvorba grafa može se napraviti prema danom pseudokodu [4].

```
foreach i ∈ V do
    izradi novu klauzulu c;
    izradi polje klauzula a[(m 2)];
    foreach j ∈ Boje do
        dodaj u c 'ij' odvojen disjunkcijom;
        foreach k ∈ SusjedniV(i) do
            dodaj klauzulu oblika (¬ ij v ¬ kj);
            /* moze se dodati i          */
               u drugom obliku
               ¬(ij ∧ kj) prema De
               Morganovim zakonima
        end
    end
end
```

Algorithm 2: Pretvorba grafa u CNF

Prema prikazanom pseudokodu, moguće je odrediti koliko bi klauzula CNF formula trebala imati. Označimo s m broj vrhova grafa, n broj boja, l broj bridova grafa. Tada je ukupan broj klauzula

$$l * n + n + m * \binom{n}{2} \tag{1}$$

Broj literala jednak je

$$2 * m * n \tag{2}$$

Ako je formula zadovoljiva, tada prevodilac završava s alokacijom registara, ali ako nije zadovoljiva, tada mora dodati memorijske naredbe te ponoviti postupak alokacije.

Za potrebe testiranja SAT-rješavača izgrađenog u sklopu ovog rada, koristi se gotov skup testnih primjera SATLIB [5]. U testnim primjerima, dobivena je formula u CNF obliku. Testni primjeri nisu realni primjeri formula koje bi prevodilac mogao dobiti, no reprezentativni su u svrhu testiranja brzine izvođenja sekvencijalno i paralelno te za testiranje ispravnosti rada.

IV. Rezultati

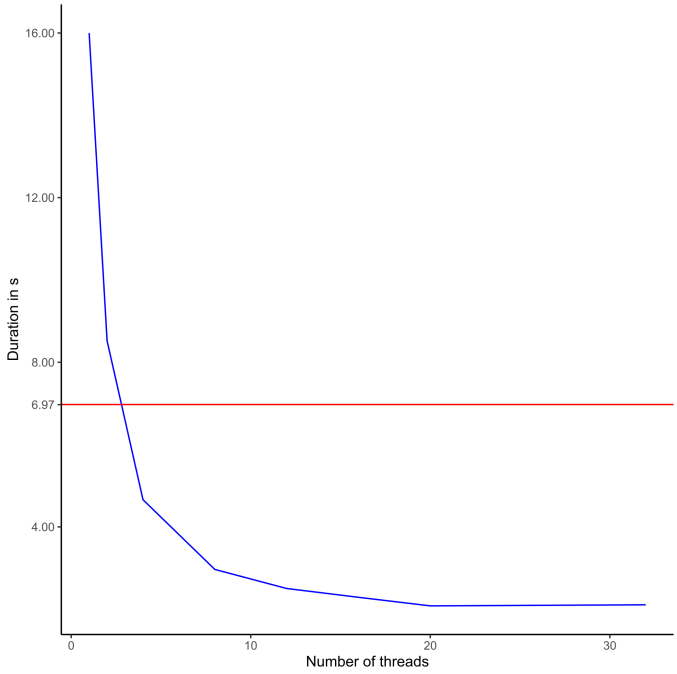
Slika 3 i tablica I prikazuju rezultatu izvođenja na procesoru Intel Core i9 12900H 4800MHz. Slika 4 i tablica II prikazuju rezultate izvođenja na procesoru AMD Ryzen 5 4600H 3.00 GHz. Prosječno vrijeme potrebno za paralelno izvođenje za različit broj dretvi prikazano je plavom linijom, dok je crvenom linijom prikazano prosječno vrijeme za serijsko izvođenje. U pripadnim tablicama je numerički prikazano prosječno vrijeme paralelnog izvođenja za različit broj dretvi. Procesor Intel Core i9 12900H sadrži 20 dretvi, dok procesor AMD Ryzen 5 4600H sadrži 12 dretvi. Prema prikazanim rezultatima jasno se vidi kako je postignuto ubrzanje paralelizacijom DPLL algoritma. Testiranje je provedeno za različit broj dretvi koji se mogu vidjeti u tablicama I i II. S obzirom na to kako Intel Core i9 12900H s pomoću kojeg je provedeno testiranje ima 20 dretvi, nakon što se broj dretvi u programu poveća na više od 20, program ne daje bolje rezultate u odnosu na program pokrenut s 20 dretvi. Isti zaključak može se izvući iz rezultata za AMD Ryzen 5 4600H procesor u kojem je vrijeme potrebno za određivanje zadovoljivosti dane CNF formule približno jednako za sva izvođenja u kojem program ima mogućnost rada s barem 12 dretvi. Za sve testove je osim paralelnog izvođenja, provedeno i serijsko izvođenje. Prosječno vrijeme za serijsko izvođenje prikazano je crvenom linijom te se s osi apscisa može iščitati koliko ono iznosi za pojedini procesor. Serijsko izvođenje pokazuje bolje rezultate od paralelnog izvođenja za 1 i 2 dretve. Razlog za bolje vrijeme serijskog izvođenja možemo pronaći u načinu provođenja algoritma. Naime prilikom serijskog izvođenja rekurzivni pozivi funkcije `dpll` događa se na samom kraju, prilikom povratka iz funkcije kako je prikazano u pseudokodu 1 i dodatku ovog rada V. S obzirom na to kako se prilikom povratka iz funkcije događaju dva rekurzivna poziva gdje je nekom literalu pri jednom pozivu dodijeljena logička vrijednost 1, a pri drugom pozivu istom literalu dodijeljena logička vrijednost 0 te kako su ova dva poziva odvojena logičkim operatorom disjunkcije, ako prvi rekurzivni poziv vrati istinu, drugi rekurzivni poziv se neće dogoditi. Ovakav način izračunavanja logičkih izraza je uobičajen kod svih programskih jezika i prevodioca. S druge strane, kod paralelnog izvođenja, dretvi se pridružuje jedan rekurzivni poziv te će stoga paralelni oblik algoritma više puta pozvati funkciju `dpll` u onim slučajevima kada je dana CNF formula zadovoljiva. U slučaju u kojem formula nije zadovoljiva, i jedan i drugi algoritam će pretražiti jednak broj kombinacija vrijednosti literala te neće biti značajne razlike u njihovim vremenima izvođenja. Za provjeru ove teze, uzet je jedan od testova čija CNF formula nije zadovoljiva te je provedeno mjerenje vremena izračuna za serijsko i paralelno izvođenje te su rezultati prikazani na slici 5. Iz rezultata se jasno može vidjeti kako je vrijeme izračuna jednako za serijsko i paralelno izvođenje s jednom dretvom u bazenu dretvi.

Tablica I
Prosječna vremena paralelnog izvođenja
Intel Core i9 12900H

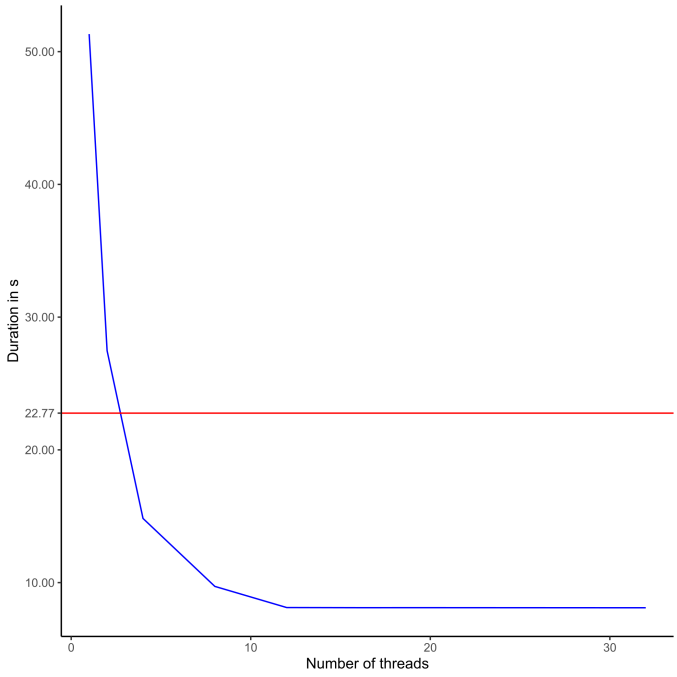
	Threads	Duration Mean
1	1	15.9984848484848
2	2	8.52034343434343
3	4	4.65970707070707
4	8	2.96639393939394
5	12	2.50292929292929
6	16	2.28892929292929
7	20	2.08137373737374
8	32	2.1070202020202

V. Zaključak i budući rad

U ovom radu predstavljen je pokušaj ubrzanja rada prevodioca, točnije jedne faze prevođenja, bojanje grafa zavisnosti mrežica pri alokaciji registara. Ubrzanje se pokušalo postići paralelizacijom. Za bojanje grafa zavisnosti koristio se DPLL algoritam. Paralelizacija je ostvarena tako što se novi rekurzivni poziv funkcije `dpll` pridružio jednoj dretvi iz prije određenog bazena dretvi. Eksperiment je proveden za uobičajenu, serijsku



Slika 3. Usporedba vremena paralelnog i serijskog izvođenja - Intel Core i9 12900H

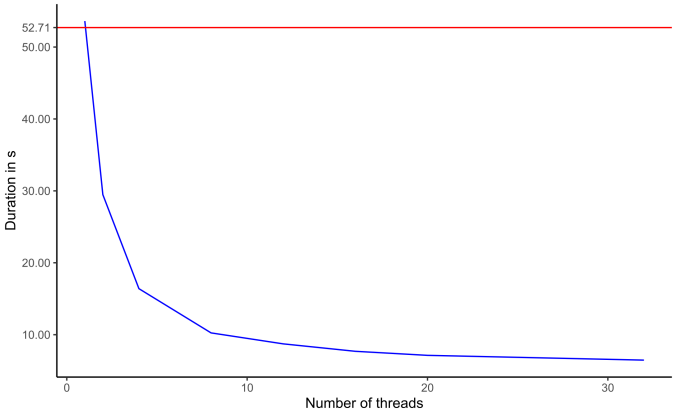


Slika 4. Usporedba vremena paralelnog i serijskog izvođenja - AMD Ryzen 5 4600H

izvedbu i za paralelnu izvedbu funkcije. Bazen dretvi, odnosno broj dretvi u bazenu dretvi mijenjao se između 1 i 32. Eksperiment je radi potpunosti proveden na više različit procesora. Rezultati testiranja su prikazani te se iz njih može zaključiti kako je paralelizacija ubrzala rad, odnosno kako je vrijeme potrebno za bojanja grafa zavisnosti mrežica uvelike smanjeno

Tablica II
PROSJEČNA VREMENA PARALELNOG IZVOĐENJA
AMD RYZEN 5 4600H

	Threads	Duration Mean
1	1	51.3228888888889
2	2	27.4716363636364
3	4	14.8346060606061
4	8	9.71472727272727
5	12	8.12158585858586
6	16	8.11188888888889
7	20	8.11309090909091
8	32	8.10627272727273



Slika 5. Usporedba vremena izvođenja za nezadovoljivu formulu

ako se u bazenu dretvi nalaze barem 4 dretve. Ako se u bazenu dretvi nalazi 1 ili 2 dretve, tada nije postignuto ubrzanje zbog načina provedbe DPLL algoritma.

Budući rad mogao bi se posvetiti poboljšanju sadašnjeg paralelnog algoritma na način da je serijska izvedba najlošiji način izvršavanja. Budući rad također bi se mogao posvetiti na traženju boljih algoritama za rad SAT-rješavača i njihovoj paralelizaciji. Osim toga, tema SAT-rješavača nije samo vezana uz bojanje grafova, već se SAT-rješavači mogu primijeniti na razna druga područja te je sigurno kako bi se i u nekim od tih područja mogli naći problemi na koje bi se mogla primijeniti paralelizacija u nekom svom obliku.

LITERATURA

[1] K. D. Cooper and L. Torczon, “Engineering a Compiler”, 3rd ed., Elsevier, 2022, pp. 663–712
[2] DPLL algorithm (2023). [Online] Dostupno: https://en.wikipedia.org/wiki/DPLL_algorithm
[3] J. L. Hennessy and D. A. Patterson, “Computer architecture: a quantitative approach”, 6th ed., Elsevier, 2017, p. 5
[4] M. Dey and A. Bagchi. “Satisfiability Methods for Colouring Graphs”, 2013. [Online] Dostupno: https://www.researchgate.net/publication/268460124_Satisfiability_Methods_for_Colouring_Graphs/fulltext/5721424d08ae0926eb45bd3f/Satisfiability-Methods-for-Colouring-Graphs.pdf
[5] “SATLIB - Benchmark Problems” (2011). [Online] Dostupno: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

DODATAK A
IMPLEMENTACIJA U RUSTU

U ovom dodatku prikazani su važni dijelovi koda u programskom jeziku Rust koji pokazuju funkcije za odabir sljedećeg literala za obradu, jediničnu propagaciju, jednoznačnu eliminaciju literala i središnju funkciju dpll u dvije verzije, za serijsko i paralelno izvođenje.

```rust
fn unit_propagate(l: i32, cnf_formula: &Vec<Vec<i32>>) -> Vec<Vec<i32>> {
    // initialize a new set of clauses
    let mut new_cnf_formula = Vec::new();
    // for each clause in cnf_formula
    for c in cnf_formula {
        // if l does not satisfy c
        if !c.contains(&l) {
            // remove the negation of l from c
            let new_c: Vec<i32> = c.iter().filter(|&x| x != &-l).cloned().collect();
            // add the new clause to the new set
            new_cnf_formula.push(new_c);
        }
    }
    return new_cnf_formula;
}

fn choose_literal(cnf_formula: &Vec<Vec<i32>>) -> Option<i32> {
    // for each clause in cnf_formula
    for c in cnf_formula {
        // if c is not empty
        if !c.is_empty() {
            // return the first literal
            return Some(c[0]);
        }
    }
    return None;
}

fn dpll_s(cnf_formula: Vec<Vec<i32>>) -> bool {
    let mut cnf_formula = cnf_formula;

    // unit propagation:
    while cnf_formula.iter().any(|c| c.len() == 1) {
        // get a unit clause
        let l = cnf_formula.iter().find(|c| c.len() == 1).unwrap()[0];
        cnf_formula = unit_propagate(l, &cnf_formula);
    }

    cnf_formula = pure_literal(cnf_formula.clone());

    // cnf_formula is empty
    if cnf_formula.is_empty() {
        return true;
    }
    // cnf_formula contains an empty clause
    if cnf_formula.contains(&Vec::new()) {
        return false;
    }

    let l = choose_literal(&cnf_formula).unwrap();

    let mut cnf_formula1 = cnf_formula.clone();
    cnf_formula1.push(vec![l]);

    let mut cnf_formula2 = cnf_formula.clone();
    cnf_formula2.push(vec![-l]);

    return dpll_s(cnf_formula1) || dpll_s(cnf_formula2);
}

```

```rust
fn dpll_p(cnf_formula: Vec<Vec<i32>>) -> bool {

    let mut cnf_formula = cnf_formula;

    // unit propagation:
    while cnf_formula.iter().any(|c| c.len() == 1) {
        // get a unit clause
        let l = cnf_formula.iter().find(|c| c.len() == 1).unwrap()[0];
        cnf_formula = unit_propagate(l, &cnf_formula);
    }

    cnf_formula = pure_literal(cnf_formula.clone());

    // cnf_formula is empty
    if cnf_formula.is_empty() {
        return true;
    }

    // cnf_formula contains an empty clause
    if cnf_formula.contains(&Vec::new()) {
        return false;
    }

    let l = choose_literal(&cnf_formula).unwrap();

    let mut cnf_formula1 = cnf_formula.clone();
    cnf_formula1.push(vec![l]);

    let mut cnf_formula2 = cnf_formula.clone();
    cnf_formula2.push(vec![-l]);

    let result = rayon::join(|| dpll_p(cnf_formula1), || dpll_p(cnf_formula2));
    return result.0 || result.1;
}
```