SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Financijski model za određivanje cijena opcija u stvarnom vremenu

Rijeka, rujan 2021.

Marko Matotan
0069085821

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Financijski model za određivanje cijena opcija u stvarnom vremenu

Mentor: doc.dr.sc. Goran Mauša

Rijeka, rujan 2021.

Marko Matotan
0069085821

Umjesto ove stranice umetnuti zadatak za završni ili diplomski rad

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad.

Rijeka, rujan 2021.

_____

Marko Matotan

Zahvala

Zahvaljujem mentoru Goranu Mauši na strpljenju i podršci, a posebice na pruženoj slobodi izbora vlastite teme za završni rad. Također zahvale A.M. na originalnoj inspiraciji i podršci za vrijeme izrade rada.

Sadržaj

Popis slika

Popis tablica

Poglavlje 1

Uvod

Još je 1965. godine uočeno da se otprilike svaka 24 mjeseca, odnosno svake dvije godine broj tranzistora u integriranim sklopovima duplo povećava. Mooreov zakon suosnivača Intela, Gordona Moorea, jest predviđanje budućnosti temeljeno na uočenom trendu koje se dan danas čini valjanim. Vode se mnoge debate o tome hoće li se taj trend nastaviti kretati u istome smjeru te je li već došao do kraja, no uz dodatne teme poput prisutne nestašice poluvodičkih čipova te povećanog naglaska prema smanjnju korištenja energije, a isto tako i boljoj iskoristivosti postojećih resursa mjesto proizvodnje novih, pozornost se sve više okreće prema optimizaciji programa kojeg računala izvršavaju [1].

Ova tema je uvelike izražena u financijskom svijetu koji svjedoči porast korištenja računalnog trgovanja na svjetskom tržištu, a posebice visoko-frekventnom trgovanju koje je zaslužno za gotovo 50% svog prometa američkog tržišta [2]. Pod visoko-frekventno trgovanje spadaju operacije koje računala moraju biti sposobna obaviti ekstremno velikim brzinama, a ono što spaja ovakvo izvršavanje programa i optimizaciju kôda je računarstvo visokih performansi, poznatije na engleskome kao High performace computing (HPC).

Svrha ovog rada jest ujediniti ta dva svijeta u jedinstveni program, odnosno primjeniti osnovne principe HPCa i paralelizacije kôda na računanje cijena financijskog instrumenta opcija u realnom vremenu. U poglavlju 2 je odrađena tema osnova superračunala i paralelizacije kôda. Slijedeće je poglavlje 3 koje se bavi primjenom HPCa u financijskom svijetu s detaljnijim fokusom na intrument opcija te njegov

izvod. Poglavlje 4 spaja obje spomenute teme te prolazi kroz program za računanje opcija, opisuje strukturu, funkcionalnost te primjenjenu paralelizaciju. Zadnje poglavlje uspoređuje dobivene rezultate za različite ulazne informacije sa hipotezom koja se pretpostavljala prije izrade programa te nakon iznesenih rezultata slijedi zaključak rada.

Poglavlje 2

Superračunala i paralelizacija kôda

2.1 Superračunala i arhitektura MPP

Zbog brzih i čestih promjena mnoge su tehnologije podložne tome da postanu nebitne, a posebice tome pripada računarski hardver, pa su tako računala napravljena u prošlom stoljeću u usporedbi s performansama današnjih gotovo zanemariva. No, iako su ona možda sada zanemariva, za svoje doba itekako nisu bila, stoga se superračunalima smatraju sva ona računala koja su bila najbrža za svoga vremena [3].

Njihove se performanse mjere u flop/s (eng. floating point operations per second), no za današnje standarde i ona postaje zanemariva pošto već postoje superračunala kao što je i japanski Fugaku koje može računati brzinom od 442 Pflop/s (petaflop/s), a sposobno je doseći i 2 Eflop/s (exaflop/s). No iako skala postaje zastarijela i dalje se upravo tom mjerom rangiraju sva svjetska računala na TOP500 listi, koja rangira 500 najbržih superračunala na svijetu od kojih, za vrijeme pisanja ovoga rada, spomenuti Fugaku zauzima prvo mjesto te je gotovo tri puta brži od idućeg superračunala na listi [4].

Sva računala koja se nalaze na TOP500 listi su arhitekture MPP (eng. Massively Parallel Processing), promjena koju je prvu potaknulo superračunalo ILLIAC IV iz 1966. godine. Za takvu je arhitekturu specifično da se računalo sastoji od jednog ili više klastera od kojih se svaki sastoji od velikog broja međupovezanih procesora,

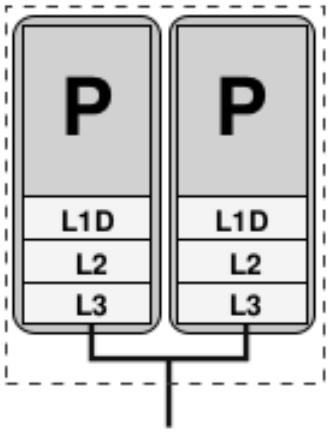
odnosno čvorova koji zajedno predstavljaju jedinstveni sustav. Zadatak čvorova jest paralelno raditi na određenom zadatku koji je unaprijed raspoređen po njima. Ovaj se princip primjenjuje i na zasebnim procesorima gdje se javlja u obliku višejezgrenih procesora.

2.2 Višejezgreni procesori i dretve

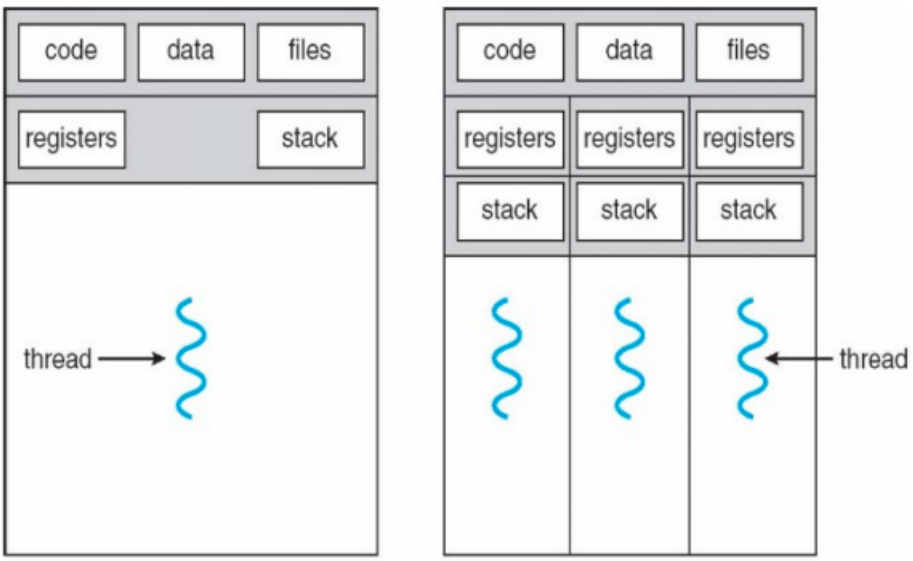
Pošto brzim napretkom standardni mikroprocesori dolaze do "toplinske barijere", odnosno zagrijavanje mikroprocesora postaje toliki problem da se njihovo hlađenje postavlja u prvi plan prilikom njihovog dizajna, a frekvencija procesora se nekako mora nastaviti povećavati, proizvođači se koriste višejezgrenim procesorima [1]. To su procesori koji na jedinstvenom integriranom sklopu sadrže dvije ili više procesne jedinice, odnosno jezgre kao što je prikazano i na slici 2.1. Sama implementacija procesora i njegovih jezgara može uvelike varirati, no sam razlog za ovakav dizajn je isti kao i kod arhitekutre MPP, a to je da omogućuje brže procesiranje naredbi, no umjesto da se poput jednojezgrenog procesora ograničenje postavlja na serijsko izvršavanje naredbi, ovim principom moguće je posao raspodijeliti na više jezgara ili procesora te ubrzati cjelokupno izvršavanje.

Iako veći broj jezgara ili procesora omogućuje brže izvršavanje naredbi, to itekako nije istina ukoliko naredbe nisu pravilno podijeljene. Takvi najmanji, neovisni tokovi instrukcija koji se mogu izvršavati istovremeno te dijele zajednički adresni prostor se nazivaju dretve (eng. threads). Proces sadrži najmanje jednu dretvu koja se u tom slučaju izvršava na jednoj jezgri, no ukoliko se proces sastoji od nekoliko dretvi, njih je moguće raspodijeliti tako da se istovremeno izvršavaju na više različitih jezgara. Takva vrsta računanja se naziva paralelno računanje. Usporedba procesa podijeljenog na jednu dretvu ili više njih je uočljiva na slici 2.2 [5].

Ovdje je također bitno spomenuti da ukoliko računalo raspolaže samo sa jednom jezgrom, iako neće moći izvršavati više dretvi istovremeno, ali ukoliko dođe do određene stanke(kašnjnja u pristupu memoriji, preduvjeti, neučinkovit slijed instrukcija itd.) u izvršavanju jedne, moguće je započeti s izvršavanjem druge dretve te obrnuto. Ovaj princip ne ubrzava izvršavanje jednako kao i paralelno računanje na više jezgara, no uvelike smanjuje vrijeme koje procesor provodi u stanju čekanja koje je češće



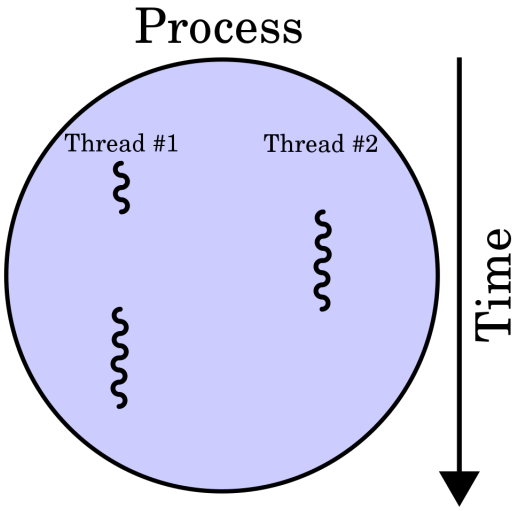
Slika 2.1 Prikaz dvo-jezgrenog procesora čije jezgre imaju vlastitu priručnu memoriju(L1, L2, L3), preuzeto iz [1]



Slika 2.2 Usporedba procesa koji sadrži jednu i istoga koji sadrži tri dretve, preuzeto iz [5]

pravilo, a ne iznimka. Takvo izmjenjivanje izvršavanja dretvi je vidljivo na slici 2.3 gdje je uočljivo da se dretve ne izvršavaju paralelno stoga se ovakvo računanje naziva

istodbno računanje (eng. concurrent computing). Iako ovaj pristup može relativno malo ubrzati vrijeme izvršavanja, ukoliko se proizvede preveliki broj dretava, vrijeme koje će se potrošiti na izmjenjivanje dretvi, odnosno promjenu konteksta(eng. context switching), ali i uz sam proces kreiranja i finaliziranja izvršavanja dretvi može nadmašiti vrijeme koje bi u protivnome bilo provedeno u stanju čekanja procesora [1, 6].



Slika 2.3 Prikaz izvršavanja dvije dretve na jednoj jezgri procesora, preuzeto s [6]

2.3 Paralelizacija kôda

Postupak kojim se ostvaruje podijela procesa na dretve započinje u kôdu programa te se naziva paralelizacija kôda. Danas se teži ostvarenju automatske paralelizacije, a ukoliko ju je potrebno manualno specificirati mnogi API-i (eng. application programming interface) su pojednostavljeni do te mjere da zahtijevaju minimalno poznavanje za implementaciju. No i dalje je korisno poznavati osnovne principe pisanja kôda sposobnog paralelizacije kako bi dobili što jednostavniji problem koji zahtijeva minimalno dodatno uloženog napora kako bi se paralelizirao. Takvi pro-

blemi se nazivaju sramotno paralelnim (eng. embarrassingly parallel).

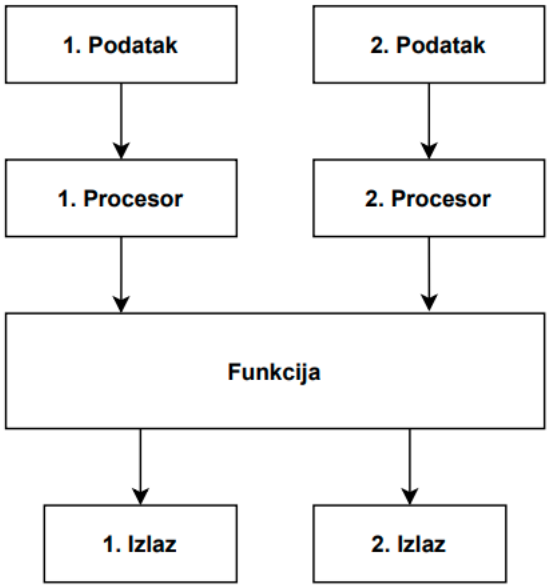
Dva dominantna koncepta za opis količine istodobne kontrole i toka podataka unutar paralelizacije su:

1. SIMD (eng. single instruction, multiple data) koji predstavlja samo jednu vrstu instrukcije na samo jednom procesoru ili više njih koja se paralelno izvršava na većem broju tokova podataka,
2. MIMD (eng. multiple instruction, multiple data) koji predstavlja veći broj instrukcija na nekoliko procesora koje se paralelno izvršavaju na većem broju tokova podataka.

Ovi koncepti dolaze iz perspektive rada računala, odnosno procesora, no analogno njima se prilikom pisanja kôda može naići na dvije vrste paralelizacije iz perspektive samoga programa. Također prije pisanja samoga paraleliziranog programa, identifikacija korištene paralelizacije bi trebao biti prvi korak.

Prva takva vrsta paralelizacije jest podatkovna paralelizacija(eng. data parallelism). Ona predstavlja onu paralelizaciju koja je zadužena procesirati veliku količinu podataka na isti način, a to uspijeva korištenjem većeg broja procesora na različitim dijelovima podataka. Ovo je glavni način paralelizacije te se naziva SPMD (eng. single program, multiple data). Iako se ovaj koncept na prvu čini sličnime SIMDu, oni se razlikuju po tome što SPMD podrazumijeva nezavisne instrukcije od procesora, dok SIMD za svaki procesor sadrži zasebnu i zavisnu instrukciju. Iz toga razloga je SPMD zapravo tip MIMD paralelizacije. Ova vrsta paralelizacije će biti uočljiva u poglavlju 4 gdje će paralelizacija proizaći iz izvršavanja istih funkcija nad velikim brojem različitih podataka. Prikaz primjera SPMD paralelizacije na dva procesora ili jezgre prikazan je na slici 2.4.

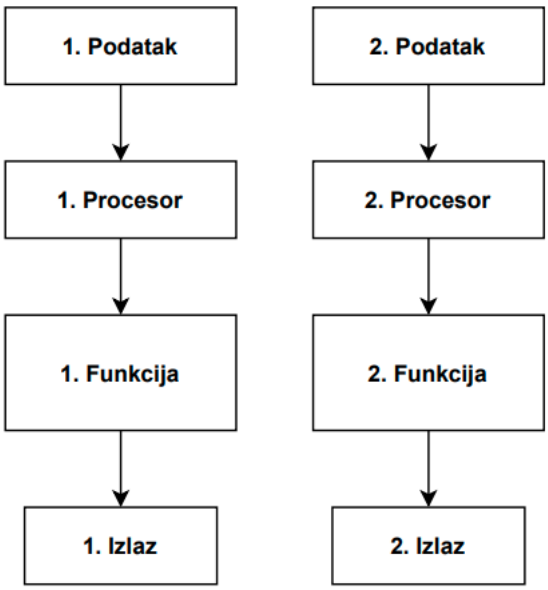
Druga vrsta paralelizacije jest funkcionalna paralelizacija. Ona za razliku od podatkovne funkcionira na način da se veliki računski zadatak rasporedi na više funkcija ili metoda koje se mogu podijeliti po različitim dretvama, odnosno različite instrukcije se izvode nad različitim dijelovima podataka od kuda i naziv MPMD (eng. multiple program multiple data). Ovakva vrsta paralelizacije iz perspektive implementacije je teža jer zahtijeva da različite funkcije imaju slična svojstva, a posebice slično vrijeme izvršavanja pošto rani završetak jedne, može uzrokovati potrebu za



Slika 2.4 Primjer SPMD paralelizacije

čekanjem kraja druge funkcije, do čega ne bi došlo prilikom serijske obrade podataka, te na taj način znatno produljiti vrijeme izvršavanja. Važno je napomenuti da korištenje podatkovne paralelizacije ne znači da se funkcionalna ne može implementirati, već se mogu implementirati zajedno ili zasebno. Primjer MPMD paralelizacije je prikazan na slici 2.5, no moguće ju je izvesti i na brojne druge načine, primjerice da druga funkcija kao ulaz prima izlaz prve funkcije, ovisno o potrebama programa [1].

Idući bitan koncept paralelnog programiranja se direktno nadovezuje na svojstvo dretvi koje dijele memoriju procesa. Ovo rezultira velikom mogučnošću da dvije različite dretve istovremeno pristupaju istome podatku u memoriji. Posao dodatno otežava činjenica da se sve naredbe granaju na procesorske instrukcije za koje postoji velika mogućnost da se ne zadrže u istome redoslijedu kakav je specificiran u kôdu programa. Ovaj se problem rješava ovisno o knjižnici kojom se implementira paralelizacija, no najčešće rješenje jest korištenje mutexa, varijable koja se zaključava prilikom pristupanja dretve dijeljenoj memoriji te time onemogućuje pristup drugih dretvi memoriji, a pri kraju modificiranja memorije mutex se otključava te ponovo



Slika 2.5 Primjer MPMD paralelizacije

ostalim dretvama daje normalan pristup. Također je važno spomenuti da nepravilno korištenje mutexa može uzrokovati trajni zastoj (eng. deadlock) u kojem dretve izgube mogućnost pristupa određenim sekcijama programa te se izvršavanje procesa zaustavlja.

Završno za temu paralelizacije kôda, prilikom pisanja programa, gotovo nikada neće biti moguće primijeniti paralelizaciju na čitavi kôd, već samo na specifičnim dijelovima programa. Takvi dijelovi se moraju što ranije prepoznati, pa kasnije i optimizirati. O tome govori jedan od važnijih koncepata paralelnog računanja, Amdahlov zakon koji predviđa ubrzanje izvršavanja programa uz korištenje većeg broja procesora, no također uzima u obzir koliki dio programa je zapravo moguće izvršavati paralelno, odnosno na većem broju procesora. Iako je važno spomenuti ovaj zakon, u poglavlju 4 će biti objašnjen točan podatak nad kojim se provodi mjerenje te iz kojeg razloga nije moguće objektivno provesti mjerenje vremena izvršavanja čitavog programa.

Poglavlje 3

Financijski instrument opcija

Radi razumijevanja programa opisanog u poglavlju 4 najprije je potrebno objasniti konept opcija u financijama te sam način njihovog računanja kako bi se isti način preslikao u kôd.

3.1 Koncept opcija

Za početak, opcije su financijski instrumenti, odnosno financijski derivati kojima njihov vlasnik ima pravo, ali ne i obvezu kupiti (eng. call) ili prodati (eng. put) imovinu specificiranu u tom ugovoru. Spomenuta imovina se najčešće odnosi na sto dionica određene firme. Dvije najvažnije varijable koje određuju svaki ugovor su vrijeme isteka opcije (eng. days until expiry, abrevijacija DTE) te strike price. Strike price predstavlja dogovorenu cijenu po kojoj će svaka dionica biti prodana, odnosno kupljena unutar perioda vremena isteka opcije što je krajnji datum do kojeg svaka opcija može biti izvršena te pri samom isteku ovisno je li ugovor profitabilan (eng. in the money, abrevijacija ITM) ili nije(eng. out of the money, abrevijacija OTM) opcija izvršava transakciju ili se poništava.

Iako se na prvu čini da trgovanje opcijama nosi brojne prednosti, najvažniji faktor koji pridodaje riziku trgovanja opcijama te je upravo i njega potrebno računati jest cijena premije prilikom kupnje svakog ugovora. Ta premija predstavlja cijenu ugovora za svaku pojedinu dionicu tako da je cijena čitave premije prilikom kupnje svake

opcije zapravo sto puta veća pošto se cijena premije odnosni na samo jednu dionicu, dok sam ugovor obuhvaća svih sto dionica.

Također je važno spomenuti da postoje dvije vrste opcija s obzirom na vrijeme unutar kojeg se mogu izvršiti. Američki tip opcija jest onaj opisan do sada, odnosno takvu opciju je moguće izvršiti u bilo kojem trenutku prije samog isteka ugovora dok europski tip opcija ograničava mogućnost izvršavanja opcija samo na datum isteka ugovora [7].

Kao primjer, važni elementi opcija su prikazani u tablici 3.1. Ovi podaci su preuzeti 28. srpnja 2021. godine u trenutku kada je jedna dionica AMDa iznosila $92, a datum isteka opcija jest 17. prosinca 2021. godine. Na tablici su uočljiva tri stupca, premije za call i put opcije, odnosno zadnja cijena po kojoj je svaka opcija prodana, te strike price koji je određen u inkrementima od $2.50 koji su prikazani u zasebnim retcima zajedno s odgovarajućim cijenama premija. Ukoliko bi se investitor odlučio kupiti call opciju čiji je strike price $100, on će platiti sveukupno $542 za jedan ugovor. Ta opcija je u trenu kupovanja OTM(cijena dionice AMDa je manja od strike pricea), no u slučaju da dana 17. prosinca ona bude veća od $100, tada opcija postaje ITM, ali samo ako je veća od $105.42 je ona zapravo profitabilna investitoru. Naravno to je primjer za kupnju call opcije, moguće je i kupiti put opciju koja djeluje na suprotan način. Investitor može kupiti put opciju stike pricea od $80, što znači da će on prodati do 17. prosinca sto dionica AMDa po cijeni od $80, iako je sada ta cijena puno veća, to ovu opciju trenutno čini OTM, no ukoliko ona padne ispod $80, preciznije na $76 kada se uračuna premija, investitor je u profitu.

Ovi primjeri ne podrazumijevaju broj dana do isteka opcije niti volatilnost specifične dionice koji drastično utječu na premiju opcija koji omogućuju da se na opcijama zaradi prodajom, a ne samo njihovim izvršavanjem. Ti detalji su vidljivi u idućoj sekciji koja se bavi računanjem cijena premija opcija.

Tablica 3.1 Prikaz lanca opcija koje ističu 17. prosinca 2021. godine [8]

Call opcije	Strike price	Put opcije
$23.80	$67.50	$1.24
$21.65	$70.00	$1.82
$20.86	$72.50	$2.09
$18.60	$75.00	$2.64
$16.78	$77.50	$3.25
$14.84	$80.00	$4.00
$13.50	$82.50	$4.87
$12.00	$85.00	$5.80
$10.50	$87.50	$6.90
$9.55	$90.00	$8.13
$8.10	$92.50	$9.75
$7.12	$95.00	$10.50
$6.21	$97.50	$11.89
$5.42	$100.00	$15.20
$4.20	$105.00	$18.49
$3.00	$110.00	$22.49
$2.23	$115.00	$25.05
$1.69	$120.00	$33.55
$1.24	$125.00	$41.25
$0.95	$130.00	$42.80

3.2 Računanje opcija

Iako ne postoje jedinstvene formule koje su sposobne u potpunosti točno izračunati cijene opcija, postoje precizni modeli koji ih mogu kvalitetno aproksimirati. Najbolji takav model, koji je još 1973. godine prouzrokovao nagli porast u trgovanju opcijama jest Black-Scholes model. Upravo se on i dan danas najviše koristi zbog jednostavnih formula i kao baza za ostale naprednije modele, no veliki mu je nedostatak što za vrijeme velikih fluktuacija u tržištu ne daje precizne rezultate. Iz parcijalne diferencijalne jednadžbe modela se mogu izraziti formule koje služe za

aproksimiranje cijena opcija.

Prva i osnovna formula jest ona za određivanje cijene call opcije, a glasi:

$$V_c = P_0 N_{d_1} - \frac{X}{e^{k_{RF} t}} N_{d_2} \tag{3.1}$$

gdje je V_c vrijednost call opcije, P_0 trenutna cijena dionice, N_{d_1} kumulativna površina pod normalnom distribucijskom krivuljom d_1. X je cijena imovine po kojoj će se prodati ili kupiti, odnosno strike price, k_{RF} kamatna stopa bez rizika te se za nju najčešće koristi stopa za desetogodišlje američke obveznice. t vrijeme do isteka opcije prikazani u dijelu godine, odnosno broj dana podijeljen s 365 te N_{d_2} kumulativna površina pod normalnom distribucijskom krivuljom d_2.

Kako bi se kumulativna distribucijska funkcija primjenila nad prvom normalnom distribucijskom krivuljom d_1, prvo ju je potrebno izračunati te njena formula glasi:

$$d_1 = \frac{\ln(\frac{P_0}{X}) + (k_{RF} + 0.5\sigma^2)t}{\sigma\sqrt{t}} \tag{3.2}$$

gdje se koriste sve već spomenute varijable uz dodatak implicirane volatilnosti σ.

Druga normalna distribucijska krivulja d_2 se dobiva korištenjem prve, a glasi:

$$d_2 = d_1 - \sigma\sqrt{t} \tag{3.3}$$

U konačnici još preostaje odrediti cijenu put opcije koja se određuje preko call opcije, a glasi:

$$V_p = V_c + \frac{X}{e^{k_{RF} t}} - P_0 \tag{3.4}$$

Sada kada su poznata pravila i glavne funkcije paralelizacije kôda, ali isto tako i formule koje su potrebne za računanje opcija, moguće ih je zajedno ukomponirati u jedinstveni program.

Poglavlje 4

Program za računanje opcija u realnom vremenu

4.1 Koncept i hipoteza programa

Iako već postoje mnogi programi koji su implementirali Black-Scholes formule te funkcioniraju kao kalkulatori za izračun opcija, oni gotovo svi dijele zajednički nedostatak što je potreba za manualnim unošenjem podataka za svaku varijablu u formuli. Ovaj pristup uvelike otežava brzo računanje, a u profesionalnom okruženju je gotovo neupotrebljiv.

Glavna ideja programa opisanog u ovome poglavlju jest omogućiti automatizirano računanje cijena opcija za proizvoljan broj dionica, ali isto tako i datuma isteka opcija, no kako bi praćenje velikog broja dionica uvelike produžilo vrijeme izvršavanja također je cilj omogućiti paralelno računanje opcija kako bi rezultati bili gotovo u realnom vremenu, što korištenjem serijskog računanja ne bi nužno bio slučaj.

Program ovakve vrste ima brojne primjene u financijskom svijetu, a glavni bi bio arbitraža. Arbitraža je zapravo uočavanje malih devijacija između cijena imovine na različitim tržištima te istovremeno kupovanje i prodaja te imovine kako bi se cijena na različitim tržištima izjednačila. Naravno ovaj primjer bi drugačije funkcionirao prilikom implementacije ovog programa, točnije ukoliko bi program s vrlo točnim modelom za aproksimaciju cijena opcija izračunao cijenu drugačiju od one izlistane

na tržištu to bi mogao biti znak da postoji devijacija u cijeni te da se ovisno o njoj opciju isplati kupiti ili prodati.

Hipoteza za rezultate takvog programa jest da je paralelno računanje opcija brže od serijskog posebice ako se opcije računaju na većem broju jezgara, a posebice ako se računaju cijene opcija za veliki broj dionica i datuma isteka opcija.

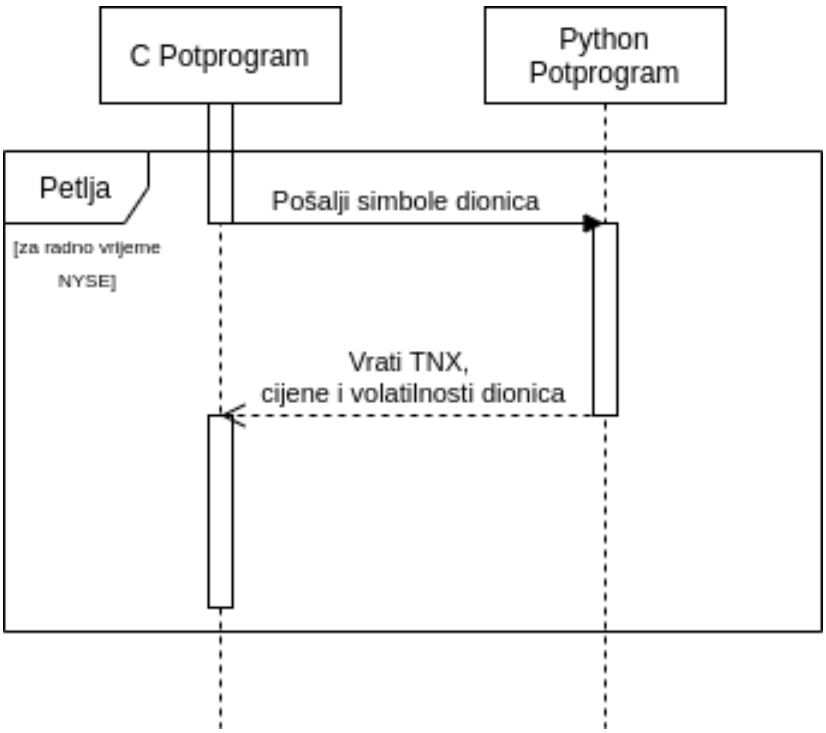
4.2 Opis rada programa

U nastavku slijedi makro opis čitavog programa te načina komunikacije u njemu nakon čega slijede opisi svaka od dva potprograma zasebno.

4.2.1 Izvođenje čitavog programa i unutarnja komunikacija

Čitavi program se sastoji od dva dijela, odnosno dva potprograma koji međusobno surađuju. Glavni je napisan u programskom jeziku C, on je zadužen za raspodijelu računanja cijena opcija po dretvama i zapisivanje rezultata u izlazne datoteke. Drugi potprogram je napisan u programskom jeziku Python te je pozvan od strane prvog potprograma svaki puta kada je potrebno prikupiti podatke o zasebnim dionicama koje zapisuje u datoteku kako bi prvi potprogram mogao računati cijene opcija na temelju tih podataka. Sekvencijalni dijagram prikazan na slici 4.1 prikazuje njihov odnos i komunikaciju. Za vrijeme rada američke burze, preciznije Njujorške burze, C potprogram opskrbljuje Python potprogram potrebnim simbolima koji tada svakom petljom učitava nove podatke potrebne za računanje cijena opcija. Prvi takav koji je zajednički svim dionicama jest već spomenuta kamatna stopa desetogodišnjih američkih obveznica(za čiji je simbol "TNX") nakon koje slijede cijene svih dionica te njihove implicirane volatilnosti.

Kako bi programi mogli komunicirati na ovaj način, potrebno je ugraditi Python potprogram u C potprogram. Python već dolazi sa knjižnicom "Python.h" koja omogućuje takvo direktno ugrađivanje. Postupak se sastoji od male nadogradnje na obično manipuliranje datotekama u C programskom jeziku, a u samom kôdu je izveden na sljedeći način:



Slika 4.1 Sekvencijalni dijagram čitavog programa

```
void run_python(char file_name[]){
    FILE* fp = _Py_fopen(file_name, "r"); // Otvaranje datoteke
    PyRun_SimpleFile(fp, file_name); // Pokretanje
    fclose(fp); // Zatvaranje

    return;
}
...
Py_Initialize(); // Inicijalizacija Python ugradnje
...
// "IV_and_stock_pc.py" je ime Python potprograma
run_python("IV_and_stock_pc.py");
...
Py_Finalize(); // Finalizacija Python ugradnje
```

U programu je ovaj proces automatiziran funkcijom run_python() koja pauzira pokretanje C potprograma te u potpunost prelazi na onaj dio za koji je zaslužen Python program.

Zbog jednostavnosti, ali i praktičnosti komunikacija se vrši zapisivanjem varijabli potrebnih za komunikaciju u CSV(Comma-separated values) datoteke. One služe za jednostavan zapis podataka koji je moguće i otvoriti u programima za manipuliranje proračunskih tablica što će uvelike biti korisno, posebice kod analize rezultata cijena opcija u poglavlju 5.

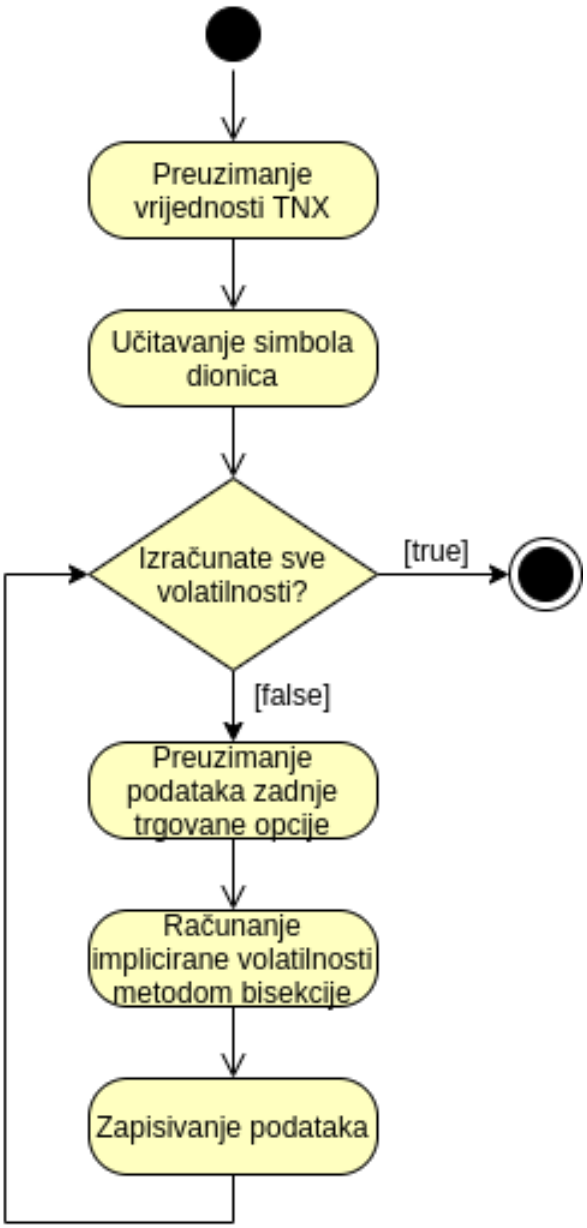
Prva takva datoteka zaslužna za komunikaciju jest "tickers.csv" u koju najprije C potprogram zapisuje simbole svih dionica koje korisnik želi pratiti. Nakon što se pozove Python potprogram, on te podatke učitava te prikuplja potrebne podatke koje zapisuje u "variables.csv". Koriste se još i dodatne CSV datoteke, no one ne služe za komunikaciju te su objašnjene u idućim podsekcijama.

4.2.2 Program za prikupljanje podataka

Spomenuti u prošloj posekciji kao potprogram napisan u Pythonu, ovaj program služi kako bi za potrebne dionice preuzeo potrebne podatke preko interneta, izračunava implicirane volatilnosti za svaku pojedinu dionicu te u konačnici sve potrebne podatke zapisuje u CSV datoteku.

Iako većina programa pretežno koristi osnovnu Python sintaksu, podatke potrebne za izračun volatilnosti i trenutne cijene dionica potrebno je dohvatiti pomoću APIa te upravo ovaj korak predstavlja srž ideje koja podrazumijeva računanje cijena opcija u realnom vremenu. Isto tako se u ovome dijelu javljaju najveći problemi. Izvori za dohvaćanje potrebnih varijabli su uvelike limitirani, posebice za širu javnost te su najčešće rezervirani za velike institucije koje se specijaliziraju u visokofrekventnom trgovanju. Najpristupačniji, javnosti dostupan takav API jest "yahoofin" koji preuzima zadnje informacije o dionicama preko servera Yahoo Finance-a. Upravo je i taj API korišten u ovome programu te služi za preuzimanje kamatne stope desetogodišnjih američkih obveznica, cijena dionica te cijenu zadnje prodane opcije.

Čitavo izvršavanje potprograma je vidljivo na slici 4.2.



Slika 4.2 Dijagram aktivnosti programa za prikupljanje podataka

Izvršavanje započinje preuzimanjem vrijednosti kamatne stope desetogodišnjih američkih obveznica. To je omogućeno korištenjem Python knjižnice APIa yahoofin te je stoga jedino potrebno navesti:

```
from yahoo_fin import stock_info as si
...
r = si.get_live_price("^TNX")
```

gdje je "TNX" simbol za već spomenute obveznice, a znak ^ služi kako bi se simbol protumačio kao indeks, a ne kao dionica.

Idući korak je bilježenje svih simbola dionica koje je potrebno pratiti. Naime njih C potprogram zapisuje u datotetku "tickers.csv" te ih je samo potrebno učitati u polje simbola. Tada je moguće otvoriti datoteku za zapisivanje podataka "variables.csv" te u nju zapisati kamatnu stopu i započeti s računanjem impliciranih volatilnosti za svaku dionicu.

Za svaki simbol, odnosno za svaku dionicu koju se korisnik odluči pratiti potrebno je najprije preuzeti podatke o zadnje trgovanoj opciji. Ovo je potrebno kako bi se izračunala volatilnost koja je potrebna za računanje dionica. Volatilnost je statistička mjera koja opisuje koliko je velika promjena cijene dionice u odnosu na njenu srednju vrijednost. Postoje dvije takve vrste volatilnosti od kojih je prva povijesna volatilnost koja podrazumijeva računanje standarne devijacije nad određenim setom podataka, u slučaju dionica, njene cijene u jednako udaljenim vremenskim periodima. Glavni je problem što ovakva vrsta volatilnosti sadrži podatke isključivo samo o povijesnim cijenama dionice te uopće ne promatra kakva bi volatilnost mogla biti u budućnosti što je upravo ono što računanje opcija mora sagledati.

Stoga se za računanje cijena premija opcija koristi druga vrsta volatilnosti, implicirana volatilnost. Ona se izračunava iz podataka o zadnje trgovanoj dionici te na taj način daje najbolji ugled u što šire tržište misli o kretanjima cijene dionice u budućnosti [9]. Fundamentalni problem koji se javlja u računanju implicirane volatilnosti je nemogućnost izražavanja volatilnosti iz formula za računanje opcija koja proizlazi iz nemogućnosti raspisivanja kumulativne distribucijske funkcije. Stoga se ona mora računati pogađanjem, odnosno jednom od metoda za pronalazak nultočke funkcije te je u ovome programu prijmenjena metoda bisekcije, a implementacija glasi ovako:

```
low_IV = 0
high_IV = 2000
IV = (low_IV + high_IV) / 2 #IV = Implied volatility

guess_price = price.call_price(IV / 100)
while guess_price != last_price:
    if guess_price < last_price:
        low_IV = IV
    else:
        high_IV = IV
    IV = (low_IV + high_IV) / 2
    IV = np.round(IV, 2)
    guess_price = price.call_price(IV / 100)
```

Ukratko, korištenjem podataka o dionici se pokušava dobiti cijena zadnje trgovane premije opcije pomoću različitih volatilnosti. Ukoliko je dobivena cijena manja od prave cijene, tada donja granica postaje zadnja probana volatilnost, no ukoliko je veća od prave cijene tada gornja granica preuzima tu vrijednost, a svakom iteracijom se za novu volatilnost uzima aritmetička sredina gornje i donje granice. Kada se cijena opcije koju dobivamo pogađanjem volatilnosti podudari s pravom cijenom opcije, tada je program došao do prave implicirane volatilnosti za tu dionicu.

U konačnici se uz impliciranu volatilnost u datotetku "variables.csv" zapisuje i trenutna cijena dionice te izvršavanje potrprograma prestaje, a izvršavanje dalje preuzima C potprogram u kojemu se i izračunavaju cijene premija opcija korištenjem paralelizacije kôda.

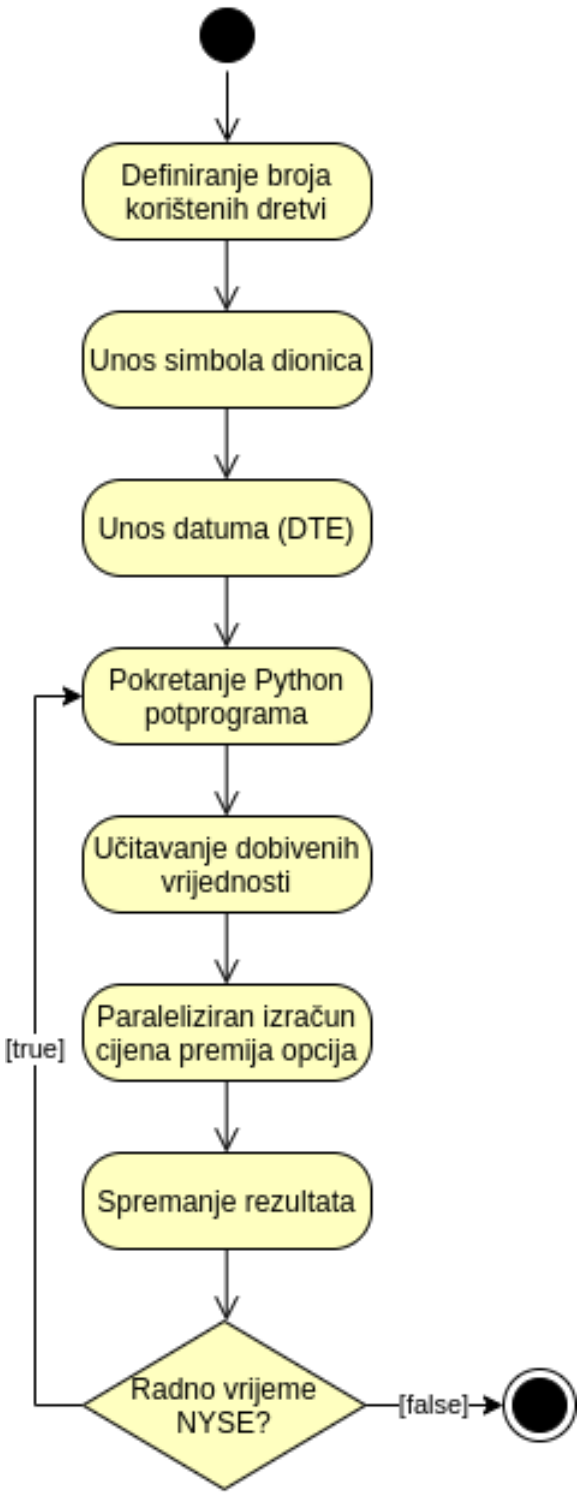
4.2.3 Program za računanje cijena opcija

Dvije osnovne funkcije koje ovaj potprogram mora moći izvršiti su pozvati Python potprogram na izvršavanje te nakon što prikupi te podatke izračunati iznose premija opcija korištenjem paralelizacije kôda. Prvi korak koji podrazumijeva ugrađivanje Python potprograma je već opisan u podsekciji 4.2.1, a drugi korak će biti omogućen

korištenjem APIa za pokretanje programa na većem broju dretava, OpenMP.

OpenMP pruža jednostavano, ali isto tako i vrlo fleksibilno sučelje čijim se pravilnim korištenjem mogu dobiti i bolji rezultati, nego korištenjem ostalih mogućih solucija za paralelizaciju kôda. Glavna razlika ovog APIa u usporedbi s knjižicama poput pthread.h jest što se paralelizacija, umjesto definiranja zasebnih dretava i njihovih pripadajućih funkcija, ostvaruje korištenjem specifičnih naredbi unutar kôda koje se odnose na samo ponašanje računalnog prevodilaca, odnosno kompajlera. Umetanjem ovih naredbi moguće je odrediti koji dio programa je potrebno razdijeliti u dretve kako bi se upravo taj dio izvršio paralelno. Takve naredbe se nazivaju pragmama te u C programskom jeziku svaka takva naredba započinje s `#pragma`.

Takva korištena naredba biti će opisana u procesu izvođenje potprograma koji je prikazan dijagramom aktivnosti na slici 4.3.



Slika 4.3 Dijagram aktivnosti C potprograma

Izvođenje programa započinje definiranjem maksimalnog broja dretvi na kojima će se program izvršavati. To je već omogućeno prije izvođenja programa navođenjem broja dretvi u argumentima prilikom pokretanja programa. Program će se izvršiti za svaki broj dretvi do unesenog u inkrementima od jedan kako bi se moglo doći do rezultata u promjeni performansa s promjenom broja dretvi. Uneseni broj dretvi se kasnije koristi kako bi prilikom izvođenja programa bilo moguće promijeniti broj korištenih dretvi funkcijom iz knjižnice omp.h `omp_set_num_threads(threads)`. Ova funkcija će omogućiti da svaka pragma koja slijedi razdvoji određeni blok kôda koji obuhvaća na točno specificirani broj dretava. Upravo je ovo fleksibilno kontroliranje korištenih dretava glavna prednost korištenja OpenMP nad ostalim knjižnicama.

Nakon specificiranja maksimalnog broja dretvi slijedi unos simbola dionica koje će biti promatrane, ali isto tako i datuma, odnosno broj dana za koje računamo cijene opcija. Dodatno je omogućeno zadati koliko puta je potrebno provesti računanja kako bi bilo moguće izračunati aritmetičku sredinu vremena izvršavanja programa za potpunije rezultate što će posebice biti analizirano u poglavlju 5. Sada je moguće povući brojne poveznice sa sekvencijalnim dijagramom prikazanim na slici 4.1 pošto unutar C potprograma započinje izvršavanje petlje koje traje za čitavo vrijeme rada Njujorške burze.

Prvi korak svake iteracije petlje je pokrenuti izvršavanje Python potprograma koje je opisano u podsekciji 4.2.1, a zatim učitati dobivene podatke. Nakon prikupljanja svih potrebnih podataka slijedi glavni dio čitavog programa, odnosno paralelizirano računanje cijena premija opcija. Taj dio kôda glasi:

```
omp_set_num_threads(threads);
dtime = omp_get_wtime();

int i, j, k;
#pragma omp parallel for private(j, k) schedule(dynamic) collapse(2)
for (i = 0; i < number_of_tickers; i++){
    for (j = 0; j < number_of_dates; j++){
        option *chain = malloc(number_of_strikes[i] * sizeof(option));
        for (k = 0; k < number_of_strikes[i]; k++){
            chain[k] = calculate_single_option(current_prices[i],
```

```
                volatilities[i] / 100, risk_free_rate,
                k + lowest_strikes[i], days[j] / 365);
            }
            option_prices[i][j] = chain;
        }
    }

    dtime = omp_get_wtime() - dtime;
```

Najprije se definira broj dretvi na kojima će se opcije računati. Varijabla "dtime" služi za mjerenje proteklog vremena za koje je program računao potrebne premije opcija, a njoj se pristupa korištenjem funkcije `omp_get_wtime()` iz knjižnice omp.h. Upravo ova razlika između vremena izmjerenog prije i nakon paraleliziranog računanja premija opcija će služiti kao osnovna mjera za prikaz rezultata. Razlog zašto se samo taj dio programa prati jest što ostali dijelovi programa, a posebice pristup podacima o opcijama preko interneta nisu dovoljno stabilni te ne pružaju konstantne brzine koje bi dale uvid u napredak izvršavanja programa s više dretava. Ukoliko dođe do problema u komunikaciji s poslužiteljem preko kojega program prikuplja informacije postoji opasnost od velikog vremenskog kašnjenja koje će u konačnici prikazati da je program sporije izvršen iako je paralelizacija računanja zapravo bila brža.

Iduće što slijedi jest definiranje pragme, odnosno naredbe koja će prevodilacu kazati na koji način da rasporedi izvršavanje tog bloka kôda na procesoru, odnosno procesorima. Svaka pragma OpenMP APIa započine s `#pragma omp`. Iduće ključne riječi `parallel for` omogućuju paralelno izvršavanje for petlje, preciznije u ovome slučaju izvršavanje 3 ugnježdene for petlje. U prvoj varijabla `i` služi kao iterator kroz sve dionice čije se opcije prate, `j` jest iterator kroz svaki od praćenih datuma, dok `k` predstavlja svaki strike price koji je smisleno pratiti, a njegova maksimalna i minimalna vrijednost ovise o trenutnoj cijeni dionice i njenoj volatilnosti. `collapse(2)` označuje da se dvije sljedeće petlje mogu povezati u jedinstevni iteracijski prostor, odnosno da svaka dretva započinje svoje izvršavanje od deklaracije strukture `option`, a zatim da računa cijene premija opcija u trećoj ugnježđenoj petlji. Ukoliko bi se umjesto dvije, ovdje povezale sve tri petlje u jedinstveni iteracijski prostor, iako bi

se činilo da bi se tim načinom ostvarila veća razina paralelizacije (što i jest istina pošto svaka operacija, odnosno svaki poziv funkcije je u tome slučaju paraleliziran) vrijeme konačnog izvršavanja će čak biti i sporije nego li prilikom serijskog izvršavanja. Razlog tome jest zbog velike potrebe za promjenama konteksta opisane u sekciji 2.2. `private(j, k)` osigurava da svaka dretva ima svoju privatnu verziju varijabla `j` i `k` kako ne bi dolazilo do istovremenog pristupanja istim varijablama, koje bi usporilo izvršavanje program te konačno `schedule(dynamic)` označuje da se iteracije dretvama dodijeljuju grupirano te nakon njihovog izvršavanja dretve zatražuju nove grupe zadataka [10].

Jedino što preostaje je konačno narediti programu da računa premije opcija što se događa pozivom funkcije `calculate_single_option()` koja obuhvaća sve postupke potrebne za izračun cijena premija opcija. Svi postupci, odnosno formule koje se pojavljuju u ovoj funkciji su ispisane u poglavlju 3.2, a unutar samog programa su zapisane u knjžnici calculations.h.

Nakon izračunatih vrijednosti konačno ih je potrebno spremiti. Vrijednosti svake dionice zasebno se spremaju u svoje datoteke formata "simbol_dionice.csv", a sam zapis nalikuje primjeru iz tablice 3.1, odnosno:

```
30,19/8/2021,13:59:46
10.95,27,0.01
9.95,28,0.01
8.95,29,0.01
7.95,30,0.01
6.96,31,0.01
5.97,32,0.01
5.00,33,0.04
4.06,34,0.10
3.18,35,0.22
2.39,36,0.43
1.71,37,0.75
1.17,38,1.21
0.75,39,1.79
0.46,40,2.50
```

```
0.27,41,3.30
0.15,42,4.18
0.08,43,5.11
0.04,44,6.07
0.02,45,7.05
0.01,46,8.04
0.01,47,9.04

60,19/8/2021,13:59:46
10.98,27,0.01
9.98,28,0.01
8.99,29,0.01
8.01,30,0.02
...
```

gdje prvi red sadrži redom informacije o broju dana do isteka opcije, datuma i točnog vremena izračuna cijena premija opcija, a svaki idući red sadrži cijenu call opcije za određeni strike price te isto tako i pripadajuću cijenu put opcije.

Poglavlje 5

Rezultati mjerenja programa

Kako bi bilo moguće potvrditi početnu hipotezu da se povećanjem broja dretvi smanjuje vrijeme izvršavanja programa, ali isto tako i prikazati da postoji optimalan broj dretvi koje je efikasno koristiti ovisno o kompleksnosti izvršavanja programa, potrebno je prikupiti podatke o izvršavanju programa te ih međusobno usporediti. Prikupljeni su podaci s prijenosnog računala s procesorom AMD Ryzen™ 3 2200U čija će razlika biti uvelike uočljiva usporedbom rezultata istih problema s superračunala "Bura".

5.1 Način mjerenja i program za vizualizaciju rezultata

Točan podatak koji se mjeri je prikazan u potpoglavlju 4.2.3, odnosno mjeri se vrijeme provedeno računajući cijene premija opcija. Važno je napomenuti da kako bi informacija o brzini bila točnija i potpunija uzeta je aritmetička sredina od tisuću različitih pokretanja programa te je zapravo ona prikazana na grafovima dalje u poglavlju. Broj korištenih dretava, aritmetička sredina vremena izvršavanja te broj operacija su zapisani u datoteci "results.csv" koja omogućuje daljnju vizualizaciju podataka programom napisanim u programskom jeziku Python. Program je jednostavniji od prošlih opisanih pretežito zbog knjižnica za manipulaciju i prikaz podataka

numpy, scipy i matplotlib.

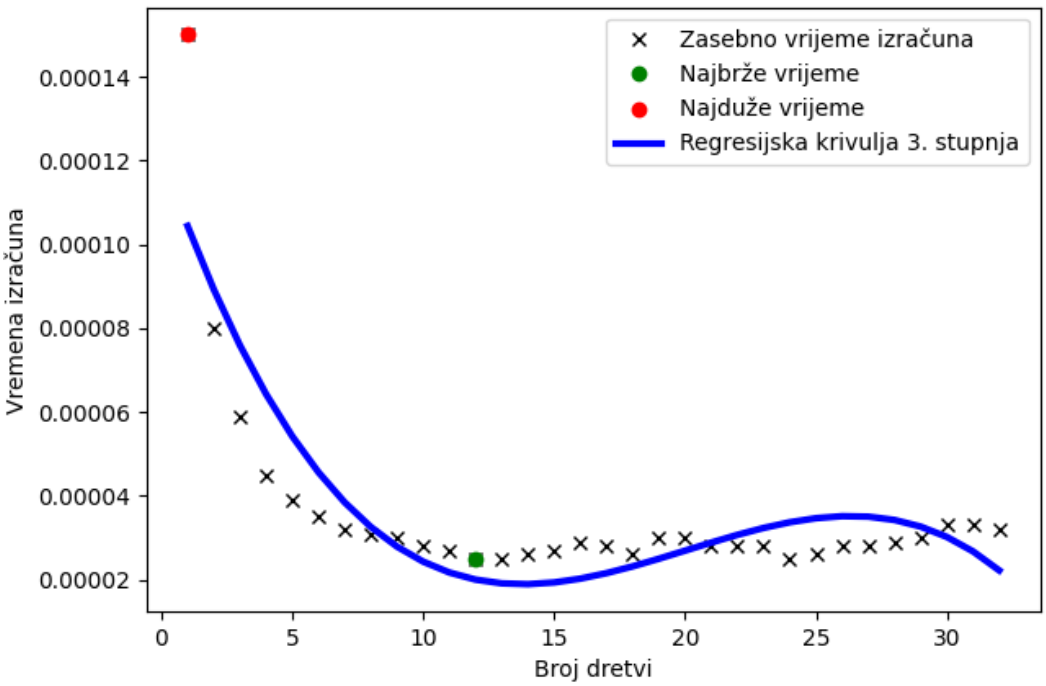
Njegovo izvođenje započinje učitavanjem rezultata u pripadajuća polja te crtanjem grafa kojime se vizualiziraju rezultati. To se postiže sučeljem matplotlib.pyplot gdje je najbitnije naglasiti postupak za izradu regresijske krivulje. Postupak započinje korištenjem funkcije `numpy.polyfit()` koja vraća koeficijente potrebne za konstrukciju polinoma što obavlja funkcija `numpy.poly1d()`. Ostatak crtanja obavlja funkcija `matplotlib.pyplot.plot()`.

Program rezultira primjerom grafa prikazanim na slici 5.1 koji ovisi o izmjerenim vremenima izračuna, no uvijek sadrži iste elemente. Na osi apcisa je prikazan broj dretvi korišten za izračun, a na osi ordinata vremena izračuna, a pojedini podaci, odnosno aritmetička sredina tisuću izračuna je prikazana simbolom "x". Od svih izmjerenih podataka, najduže je vrijeme prikazano crvenom točkom dok je najkraće prikazano zelenom. Također kako bi se lakše uočilo ubrzanje ili usporavanje dodavanjem većeg broja dretvi, povućena je regresijska krivulja, na grafu prikazana kao plava krivulja koja opisuje promjenu u vremenima izračuna promjenom broja dretvi za pokretanje programa.

Konačno je moguće definirati slučajeve, odnosno dionice te datume za koje će računalo i superračunalo izvršavati program. U tablici 5.1 koja slijedi je naveden broj dionica, datuma i cijeloukupan broj izračunatih premija opcija, a u idućim sekcijama su prikazani rezultati izvršavanja te njihova interpretacija.

Tablica 5.1 Popis slučajeva za testiranje izvođenja programa

Broj dionica	Broj datuma	Broj kalkulacija
1	1	40
2	2	132
4	4	720
8	8	2752
16	16	26896



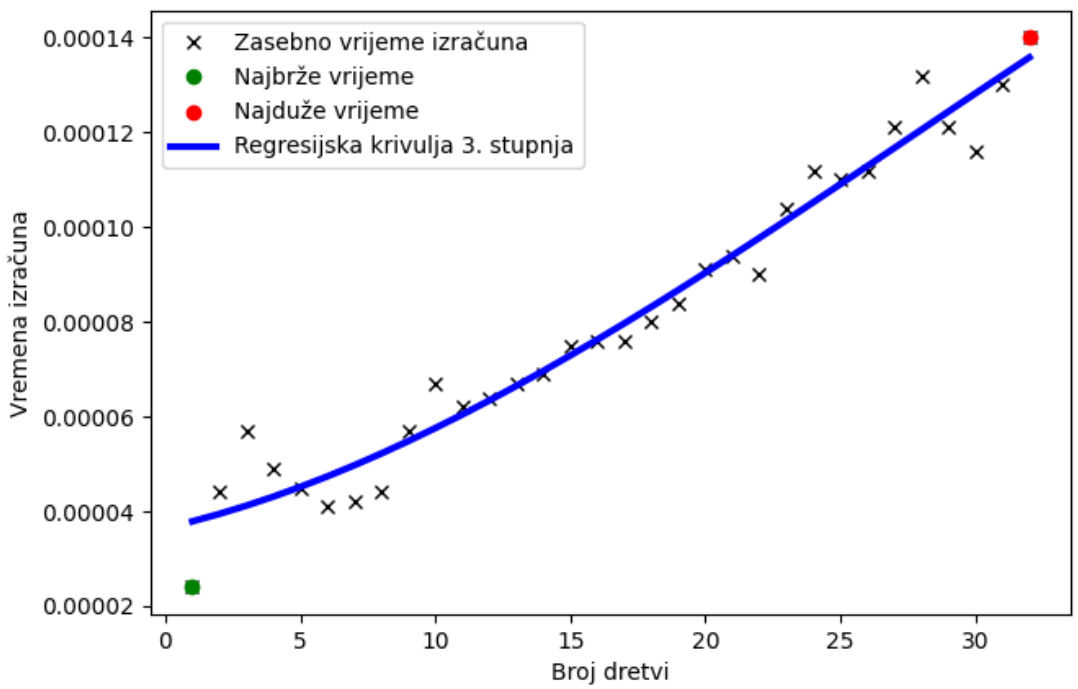
Slika 5.1 Primjer izgleda konačnog grafa

5.2 Rezultati s prijenosnog računala

Kako bi bilo moguće interpretirati rezultate potrebno je najprije specificirati pojedinosti procesora na kojemu će se program izvršavati. U prvim rezultatima testiranja, program je izvršavan na mobilnom procesoru AMD Ryzen™ 3 2200U. Ovaj procesor sadrži samo dvije jezgre te četiri dretve pri čemu je brzina rada procesora 2.5 GHz [11]. Hipoteza za rezultate koje je smisleno očekivati jest da će rad na malom broju dionica i datuma, odnosno u slučaju kada ne dolazi do velike paralelizacije u kôdu najbrže vrijeme izvršavanja biti na vrlo malenom broju jezgara, no ukoliko se radi o velikom broju korištenih dretvi, vrijeme izvršavanja na jednoj jezgri će sigurno biti najsporije dok će ostala vremena biti sve brža i brža. Iako se kao najbolja opcija čini koristiti što je više dretava moguće, s vremenom će brzina uvelike usporiti ukoliko dođe do prevelikog broja dretava, a najbrže vrijeme će u tome slučaju biti prije nego

što dođe do "gušenja" procesora.

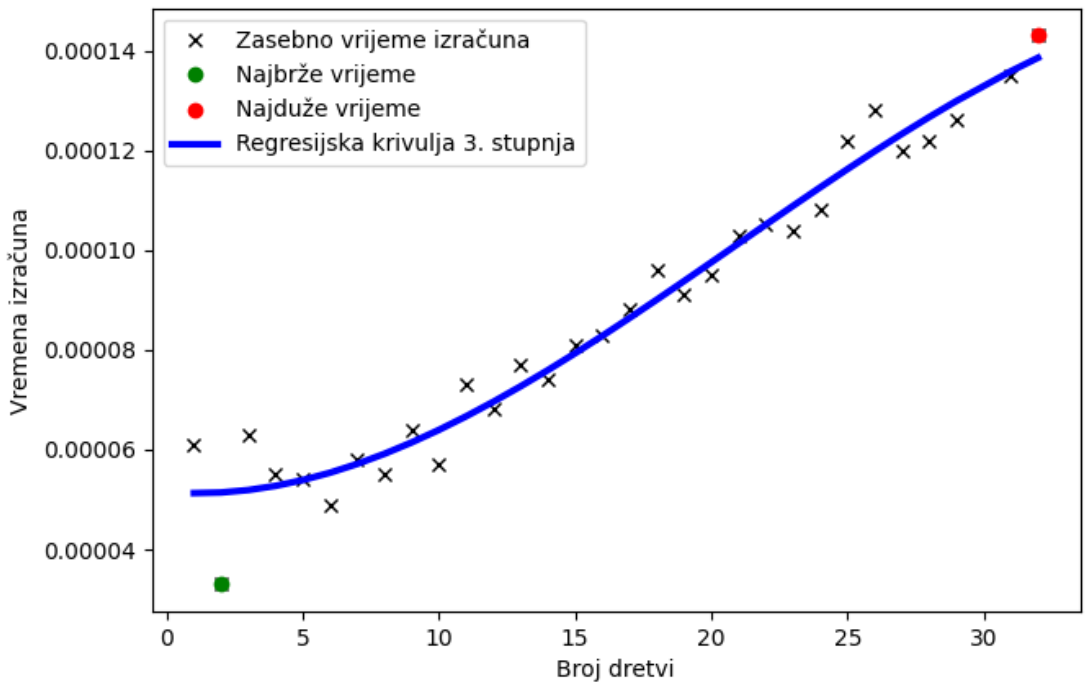
Prvi promatrani slučaj iz tablice 5.1 prati samo jednu dionicu te jedan datum isteka opcije, a vremena izvršavanja su prikazana na grafu u slici 5.2. Vidljivo je da je kao najkraće vrijeme izvršavanja označeno ono izvršavanje na samo jednoj dretvi, dok sva ostala izvršavanja, iako provedena na više dretava uvelike produljuju vrijeme izvršavanja, a samo povećanje vremena izvršavanja gotovo linearno raste.



Slika 5.2 Rezultati procesora AMD Ryzen™ 3 2200U na jednoj promatranoj dionici i jednom datumu (40 kalkulacija)

Idući promatrani slučaj prati dvije različite dionice i dva različita datuma isteka. Prikazan je na grafu u slici 5.3. Primjetljivo je da je regresijska krivulja vrlo slična u ovome te prethodnom slučaju, odnosno da vrijeme izvršavanja gotovo linearno raste, no sada je najbrže vrijeme izvršavanja ono na dvije dretve. Ovo je vrlo korisna informacija koja nam govori da čim se radi o većem stupnju paralelizacije da odmah dolazi do napretka u brzini izvršavanja programa. Također je bitno je primjetiti da

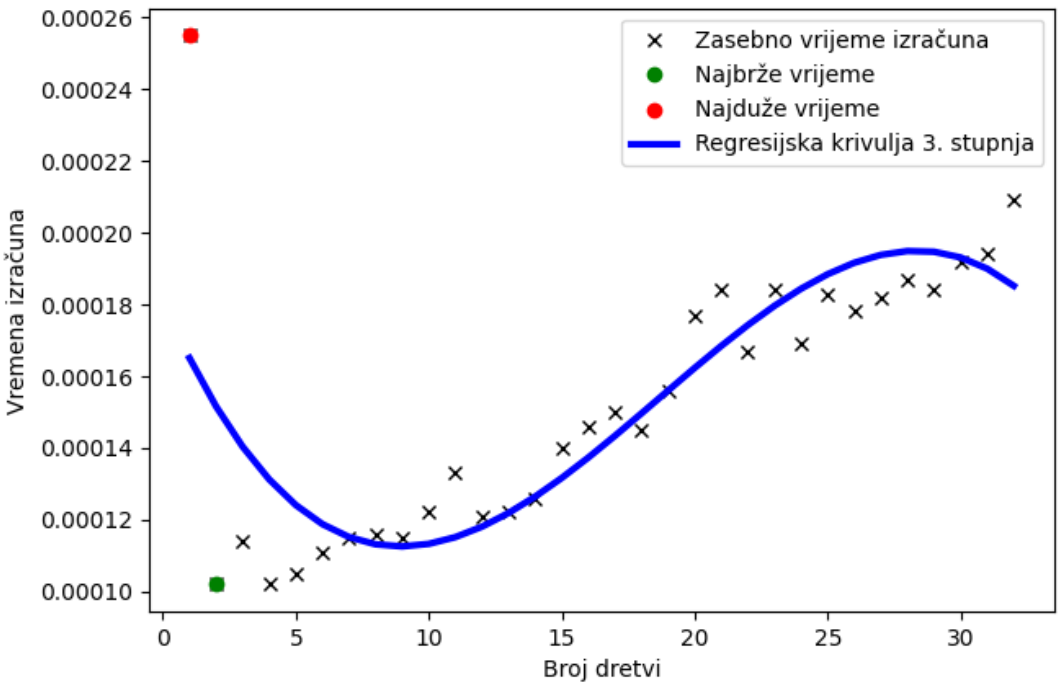
je vrijeme izvršavanja na jednoj dretvi puno bliže ostalim vremenima te iako je ono usporedivo s ostalima uz promatranje malo većeg broja dionica te datuma postoje šanse da će biti isplativije koristiti veći broj dretava.



Slika 5.3 Rezultati procesora AMD Ryzen™ 3 2200U na dvije promatrane dionice i dva datuma (132 kalkulacije)

Stoga slijedi slučaj koji prati četiri različite dionice te četiri različita datuma isteka, a prikazan je na grafu u slici 5.4. Ovaj slučaj drastično mijenja izgled regresijske krivulje, a uočljivo je da je izvršavanje na dvije dretve još uvijek najbrže, no dijeli gotovo isto vrijeme s onime na četiri dretve što u prošlome promatranju nije bio slučaj. Promatrajući hipotezu s početka poglavlja to je bilo za očekivati jer što više dretvi postoji, to su veće mogućnosti procesora iskoristiti taj dodatan broj. Pošto ovaj specifičan procesor sadrži samo dvije jezgre logično je zaključiti da najbolje vrijeme izvršavanja neće imati preveliki broj dretvi, već onaj s manjim brojem, najčešće oko dvije, no ukoliko procesor efikasno uspije u izmijenjivanju dretvi, taj

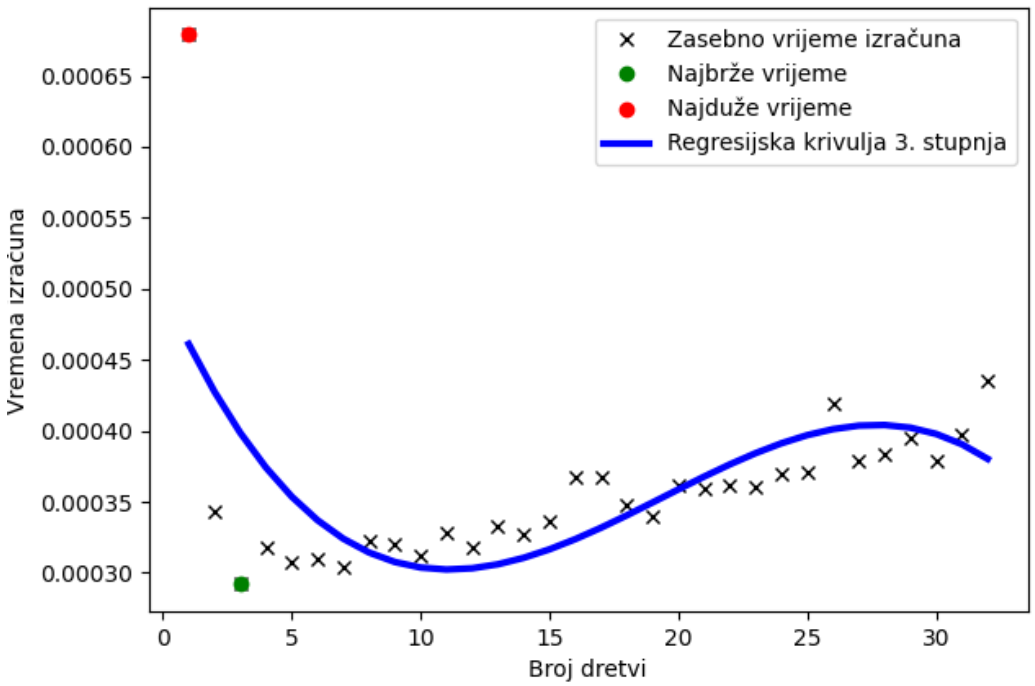
se broj može i povećati. Glavna opservacija koju je moguće učiniti na ovome grafu je vrijeme izračuna na jednoj dretvi, koje je daleko najveće u usporedbi s ostalima. Pogotovo sada kada postoji veći broj praćenih dionica i datuma serijsko izvršavanje na samo jednoj dretvi više uopće nije isplativo.



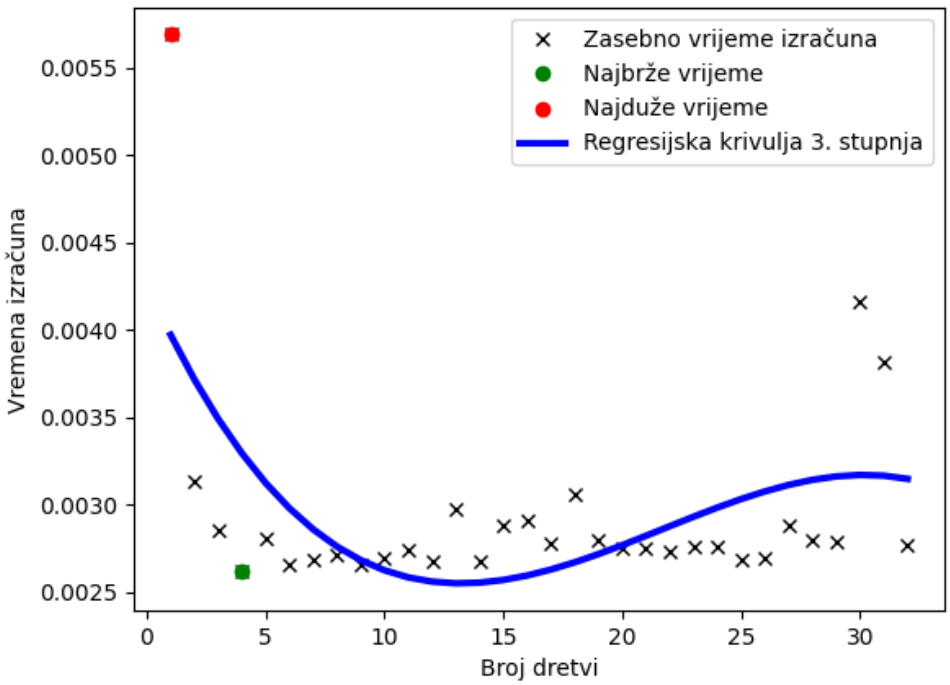
Slika 5.4 Rezultati procesora AMD Ryzen™ 3 2200U na četiri promatrane dionice i četiri datuma (720 kalkulacija)

Posljednja dva slučaja, onaj koji prati osam različitih dionica i osam datuma te je prikazan na slici 5.5 te onaj koji prati šestnaest dionica te šestnaset datuma, a prikazan je na 5.6 dijele vrlo sličnu regresijsku krivulju, a rezultati su vrlo slični uz male razlike. Daleko najduže je vrijeme još uvijek ono na samo jednoj dretvi, dok su najbrža ono na tri i četiri dretve. Naime zato što procesor fizički ne sadrži više jezgara, a njegova sposobnost izmijenjivanja četiri dretve ne pomaže previše u ubrzanju računanja, korištenje više dretvi unutar procesa ne može previše poboljšati njegovo izvršavanje. Do velikih promjena dolazi u idućoj sekciji koja promatra izvršavanje

na superračunalu Buri koje je zbog svojeg većeg broja jezgara sposobnije iskoristiti veliki broj dretvi koje proces sadrži.



Slika 5.5 Rezultati procesora AMD Ryzen™ 3 2200U na osam promatranih dionica i osam datuma (2752 kalkulacija)



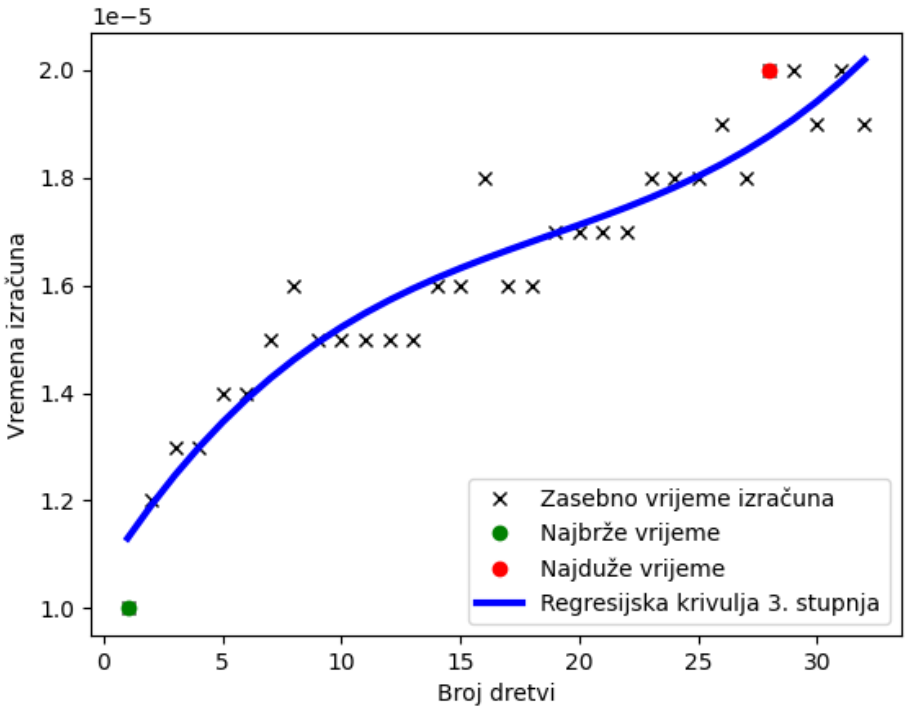
Slika 5.6 Rezultati procesora AMD Ryzen™ 3 2200U na šestnaest promatranih dionica i šestnaest datuma (26896 kalkulacija)

5.3 Rezultati sa superračunala Bura

Sada slijede rezultati izvršavanja istih slučajeva iz tablice 5.1 samo ovoga puta provedenih na superračunalu "Bura", odnosno na sustavu "Cluster" od kojih svaki čvor sadrži po dva Intel Xeon E5 procesora koja zajedno čine 24 jezgre, a svaki procesor sadrži 24 dretve. Rezultati koje je smisleno očekivati su slični i rezultatima sa mobilnog procesora iz prošle sekcije, no ovoga puta bi broj dretvi za optimalno izvršavanje programa trebao biti oko 24, ali naravno samo za slučajeve s velikim brojem kalkulacija kako bi došlo do najboljih rezultata primjenom paralelizacije kôda.

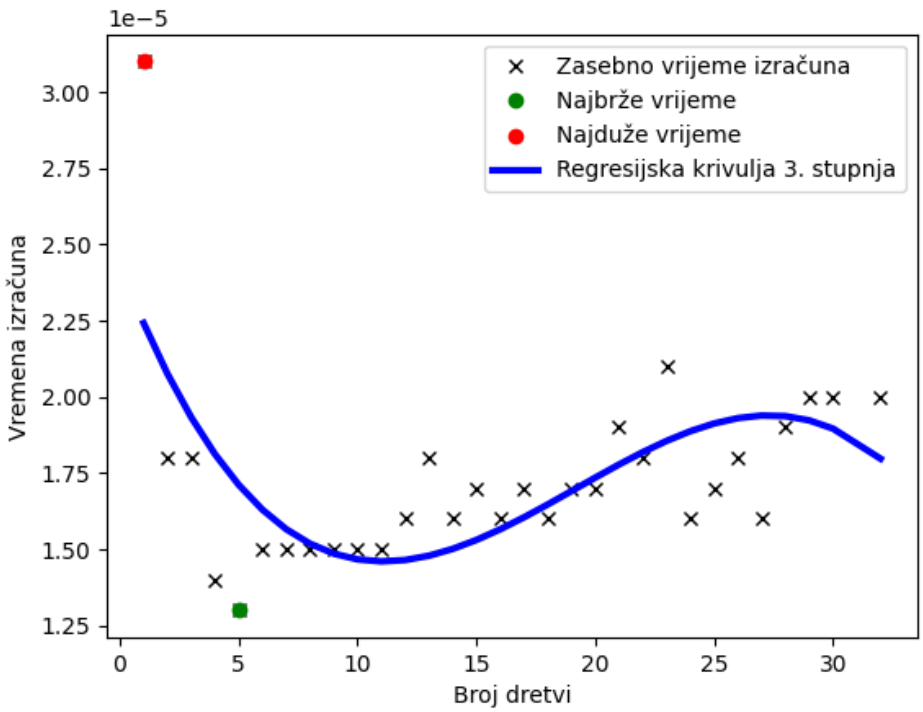
Prvi slučaj koji promatra samo jednu dionicu i jedan datum je prikazan na grafu u slici 5.7. Slično kao i u slici 5.2, izvršavanje na jednoj dretvi je daleko najbrže u usporedbi s ostalim vremenima, ta razlika bi se trebala smanjivati s povećanjem

broja kalkulacija.



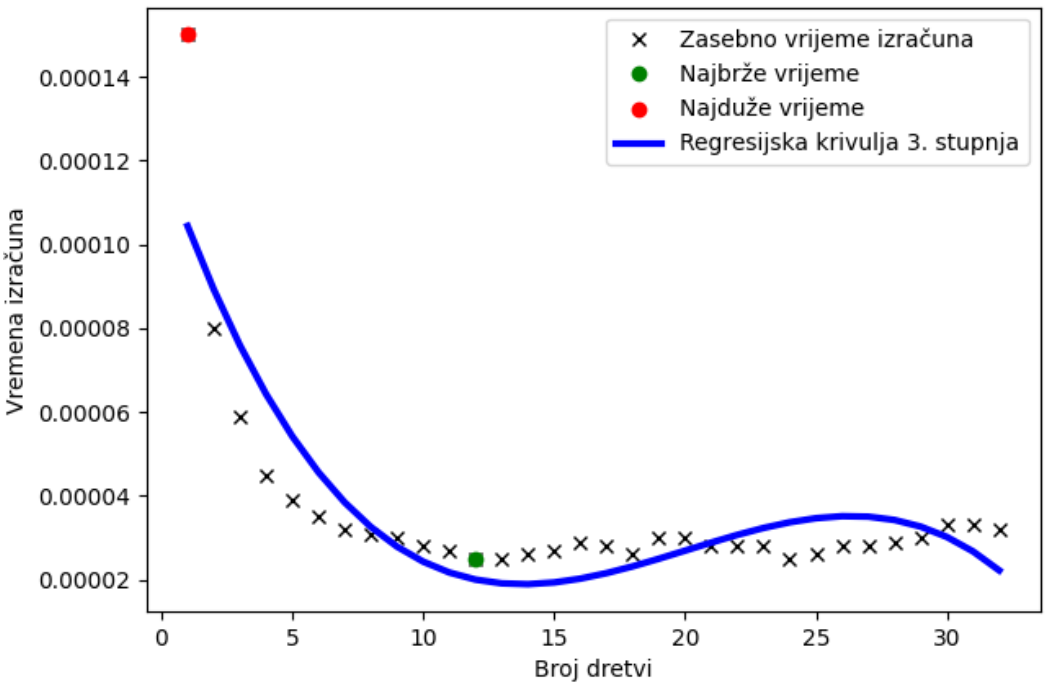
Slika 5.7 Rezultati čvora superračunala "Bura" na jednoj promatranoj dionici i jednom datumu (40 kalkulacija)

Drugi slučaj koji promatra dvije različite dionice i dva različita datuma je prikazan na slici 5.8. Čim se počne koristiti veći broj dretvi u procesu, procesor je sposoban to iskoristiti. Ovoga puta je za razliku od procesora na prijenosnom računalu, superračunalo počinje napredovati u vremenu izvršavanja s maksimalnim brojem ponuđenih dretvi. Pošto se radi o promatranih dvije dionice i dva datuma očekivano je da će najbrže vrijeme izvršavanja biti oko četiri dretve, no u ovome slučaju to mjesto zauzima malo brže, ono na pet dretvi.



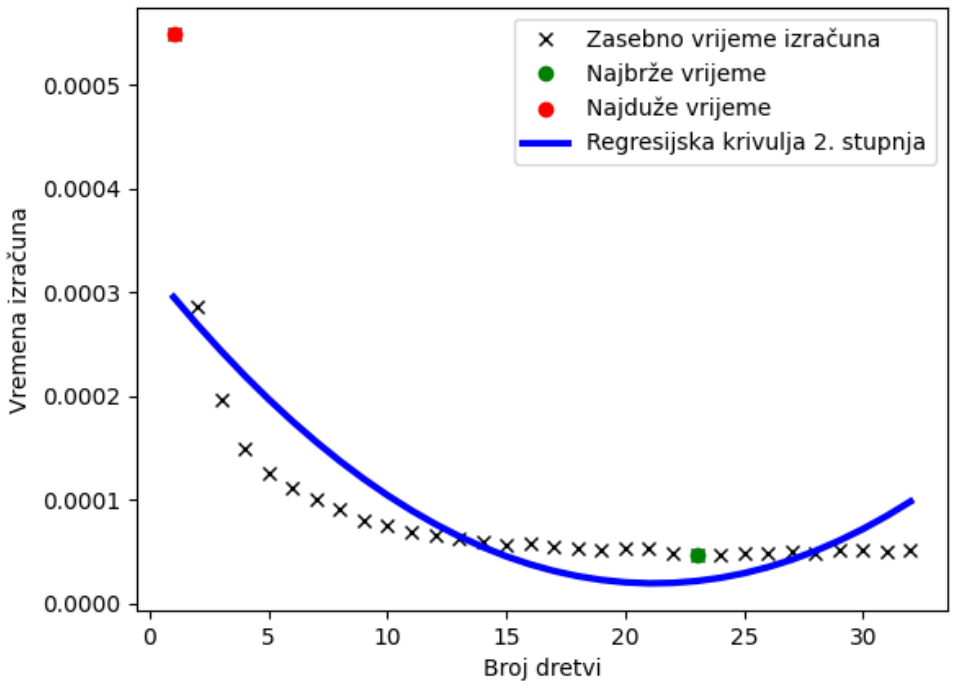
Slika 5.8 Rezultati čvora superračunala "Bura" na dvije promatrane dionice i dva datuma (132 kalkulacija)

Idući, treći slučaj promatra četiri različitih dionica i četiri različita datuma te rezultira grafom prikazanim u slici 5.9. Ovoga puta je ponovo serijsko izvršavanje na jednoj jezgri u potpunosti neisplativo naspram ostalih, a izvršavanja na manjem broju dretvi naglo padaju, a zatim se stabiliziraju. Ovakvo ponašanje će biti vidljivo i u kasnijim slučajevima. Glavna razlika je u konstantnom povećanju broja dretvi na kojima je izmjereno najbrže vrijeme izvršavanja. U ovome slučaju to je na čak dvanaest dretvi, a dodatnim povećanjem kompleksnosti praćenja dionica i datuma taj će broj nastaviti rasti.



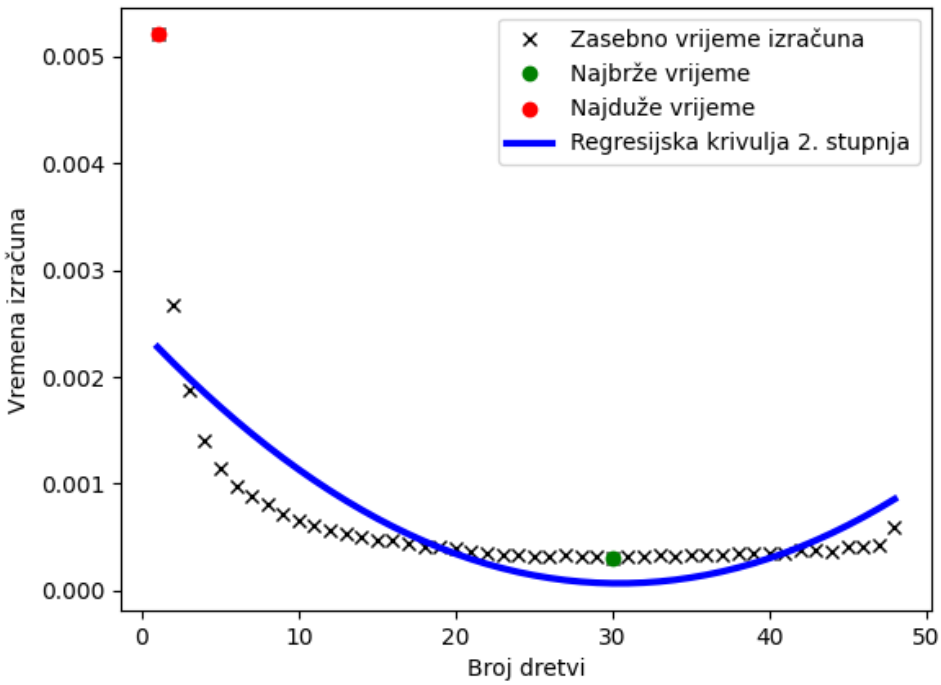
Slika 5.9 Rezultati čvora superračunala "Bura" na četiri promatranih dionica i četiri datuma (720 kalkulacija)

Predzadnji slučaj prikazan na slici 5.10 koji prati osam različitih dionica i osam datuma je grafom te regresijskom krivuljom sličan onome u slici 5.9, no ponovno, glavna promjena je u porastu broja dretvi koje obilježuju najbrže vrijeme, sada je taj broj na dvadeset i tri dretve što se bliži kapacitetu čvora koji sadrži dvadeset i četiri dretve. Naime to ne mora značiti da najbrže vrijeme ne može zauzeti izvršavanje na većem broju dretvi od dostupnih jezgri. Isto kao što je na prijenosnom računalu najbrže vrijeme zauzelo ono na četiri dretve, procesorima često dobro dođe veći broj dretvi kako bi imali više slobode s izvršavanjem, što će biti vidljivo na posljednjem, idućem primjeru.



Slika 5.10 Rezultati čvora superračunala "Bura" na osam promatranih dionica i osam datuma (2752 kalkulacija)

Konačno se dolazi do zadnjeg slučaja koji promatra šestnaest dionica te šestneast datuma prikazanog na slici 5.11. Ovoga puta je provedeno izvršavanje na šestdeset i četiri dretve upravo zato kako bi se dobio uvid u zaista najbrže vrijeme izvršavanja. Ispada da to ipak nije bilo potrebno jer je najbrže vrijeme izvršavanja na trideset dretvi što je malo veće od dvadeset i četiri jezgre koje čvor sadrži, a samim izgledom je graf vrlo sličan onome iz slike 5.10.



Slika 5.11 Rezultati čvora superračunala "Bura" na šestnaest promatranih dionica i šestnaest datuma (26896 kalkulacija)

5.4 Analiza rezultata

Uz navedene rezultate može se primjetiti da do najveće koristi paralelizacije dolazi u slučajevima koji sadrže velik broj kalkulacija, odnosno praćenih dionica i datuma, no samo ako se radi o prikladnom broju dretvi koje procesor može podnijeti. Stoga slijedi točan uvid u ubrzanje, povećanje u učinkovitosti i poboljšanje u performansama u usporedbi sa serijskim izvršavanjem na prijenosnom, ali i na superračunalu(sve metrike napravljene po uzoru na [12]). Metrike koje slijede su dobar indikator za rezultate paralelizacije, no zbog različitih sposobnosti procesora, ali i načina ostvarene paralelizacije unutar samog programa ne postoje objektivne metrike

koje je moguće izraziti, već su primarno korisne za međusobnu usporedbu.

Prva metrika je ubrzanje *S(n)*, odnosno omjer vremena potrebnog za izvršavanje na jednom procesoru i vremena potrebnog za izvršavanje na istome računalu korištenjem više procesora, a rezultira mjerom koja daje uvid u koliko je paraleliziran kôd brži od serijskog.

$$S(n) = \frac{Vrijeme\ serijskog\ izvodjenja}{Vrijeme\ paralelnog\ izvodjenja} \tag{5.1}$$

Iduća korisna metrika koja prikazuje učinkovitost *E(n)* je omjer ubrzanja i broja korištenih procesora, odnosno je li dobiveno ubrzanje opravdava korištenje većeg broja procesora.

$$E(n) = \frac{Ubrzanje}{Broj\ procesora} \tag{5.2}$$

Te konačno slijedi zadnja mjera koja prikazuje napredak u performasni *PI*, a računa se razlikom serijskog i paralelnog vremena izvršavanja podijeljenom sa serijskim vremenom izvršavanja.

$$PI = \frac{Vrijeme\ serijskog\ izvodjenja - Vrijeme\ paralelnog\ izvodjenja}{Vrijeme\ serijskog\ izvodjenja} \tag{5.3}$$

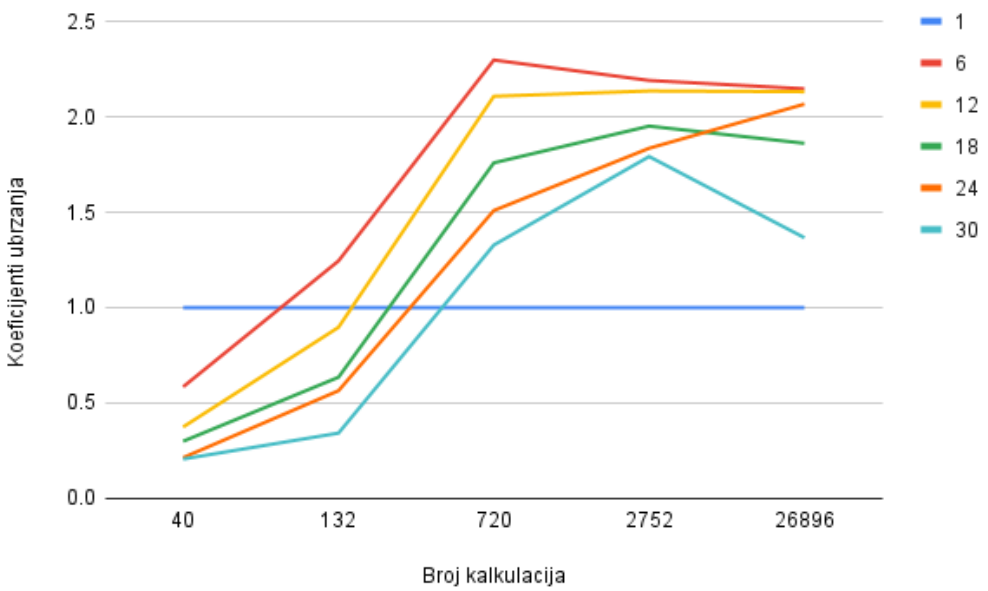
Po uzoru na navedene formule slijede rezultati sa prijenosnog računala u obliku tablica te grafova, a kako bi rezultati bili pregledniji uzeti su u obzir oni rezultati koji su koristili 1, 6, 12, 18 i 24, 30 dretvi pri računanju opcija. Ova izvršavanja prekrivaju širok obim broja dretvi. Naravno, punija informacija bi bila ona koja promatra sve dretve, no time bi se dobilo preveliki broj informacija. Također, iako bi potencijalno bilo poželjnije promatrati razlike u malom broju dretvi, posebice kod prijenosnog računala, pošto se radi o procesoru s malim brojem jezgara, promatrano je kretanje rezultata kroz širi obim broja dretvi kako bi se rezultati mogli usporediti s onima provedenima na superračunalu čiji se napredak itekako uočava korištenjem većeg broja dretvi.

Prva mjera jest ubrzanje, a za prijenosno računalo su rezultati prikazani u tabilici 5.2 te na grafu u slici 5.12. Izvršavanje na šest dretvi je u svim slučajevima najbolje, osim prvoga gdje je serijsko izvršavanje ipak prikladnije jednoj dretvi. No to je i očekivano pošto procesor ne može iskoristiti takvu prednost zbog svog malog broja jezgri. Još postoji jedna zanimljiva opservacija, a to je da sva izvršavanja osim onoga na šest dretvi imaju manji koeficijent ubrzanja na drugom slučaju, odnosno

koeficijent ubrzanja je manji od jedan što znači da je izvršavanje u tom slučaju sporije od serijskog.

Tablica 5.2 Tablica ubrzanja rezultata na procesoru AMD Ryzen™ 3 2200U

	Broj dretvi					
Broj kalkulacija	1	6	12	18	24	30
40	1	0.5854	0.3750	0.3000	0.2143	0.2069
132	1	1.2449	0.8971	0.6354	0.5648	0.3427
720	1	2.2973	2.1074	1.7586	1.5089	1.3281
2752	1	2.1903	2.1352	1.9511	1.8351	1.7916
26896	1	2.1467	2.1315	1.8610	2.0664	1.3664



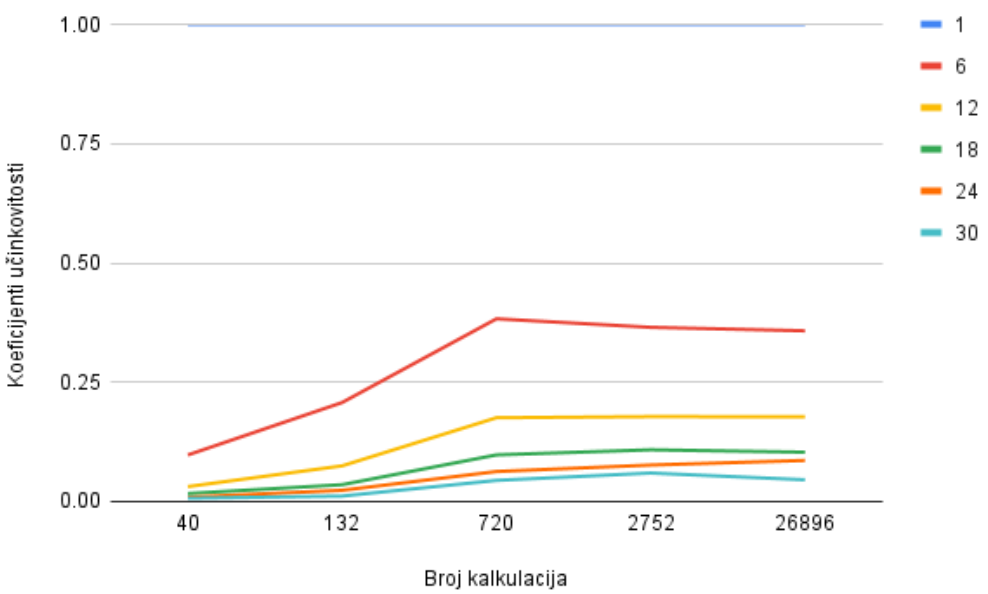
Slika 5.12 Graf ubrzanja rezultata na procesoru AMD Ryzen™ 3 2200U

Iduća mjera jest ona učinovitosti predstavljena tablicom 5.3 te grafom na slici 5.13. Ona daje vrlo sličan prikaz kao i mjera ubrzanja jedino što je u ovome slučaju serijsko izvršavanje najučinkovitije što je i logično, ali samo zato jer koristi samo jednu jezgru, a ubrzanja ostalih slučajeva nisu dovoljna kako bi opravdala korištenje

većeg broja jezgara. Ostala izvršavanja se nalaze jedno ispod drugoga kao i u grafu 5.12.

Tablica 5.3 Tablica učinkovitosti rezultata na procesoru AMD Ryzen™ 3 2200U

	Broj dretvi					
Broj kalkulacija	1	6	12	18	24	30
40	1	0.0976	0.0313	0.0167	0.0089	0.0069
132	1	0.2075	0.0748	0.0353	0.0235	0.0114
720	1	0.3829	0.1756	0.0977	0.0629	0.0443
2752	1	0.3651	0.1779	0.1084	0.0765	0.0597
26896	1	0.3578	0.1776	0.1034	0.0861	0.0455

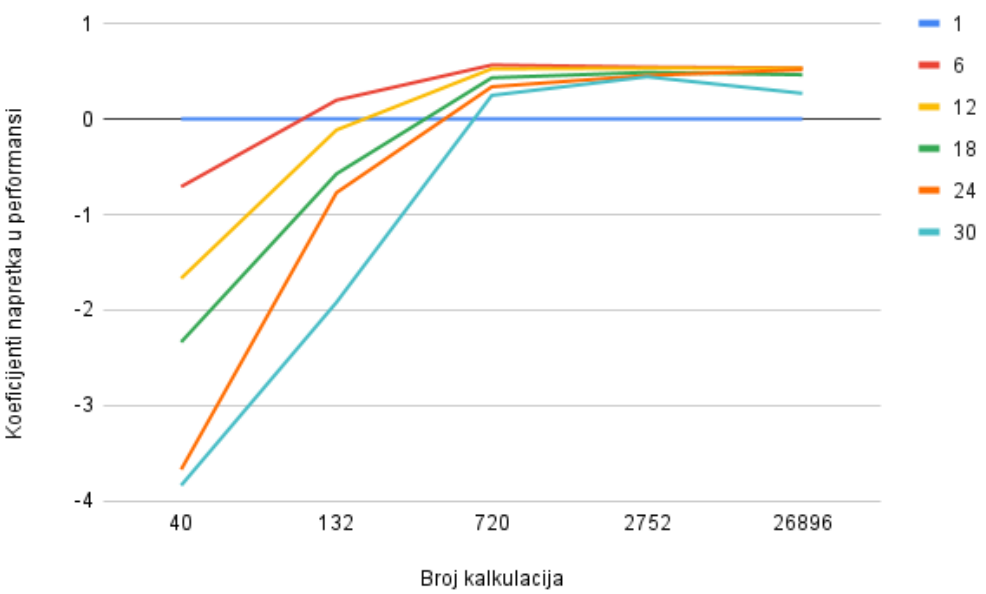


Slika 5.13 Graf učinokovitosti rezultata na procesoru AMD Ryzen™ 3 2200U

Posljednja mjera je napredak u performansi izražen u tablici 5.4 te na grafu u slici 5.14. Oni koeficijenti čija su izvršavanja bila sporija od serijskog su negativni, a njihov poredak je identičan prethodnim metrikama. Izvršavanje na šest dretvi i dalje prikazuje najveći napredak dok svako povećanje broja dretvi uzrokuje u manjim koeficijentima. To je itakako uočljivo na samom početku grafa na četrdeset kalkulacija gdje je ta razlika najveća.

Tablica 5.4 Tablica napretaka u performansama rezultata na procesoru AMD Ryzen™ 3 2200U

	Broj dretvi					
Broj kalkulacija	1	6	12	18	24	30
40	0	-0.7083	-1.6667	-2.3333	-3.6667	-3.8333
132	0	0.1967	-0.1148	-0.5738	-0.7705	-1.9180
720	0	0.5647	0.5255	0.4314	0.3373	0.2471
2752	0	0.5434	0.5317	0.4875	0.4551	0.4418
26896	0	0.5342	0.5308	0.4627	0.5161	0.2681

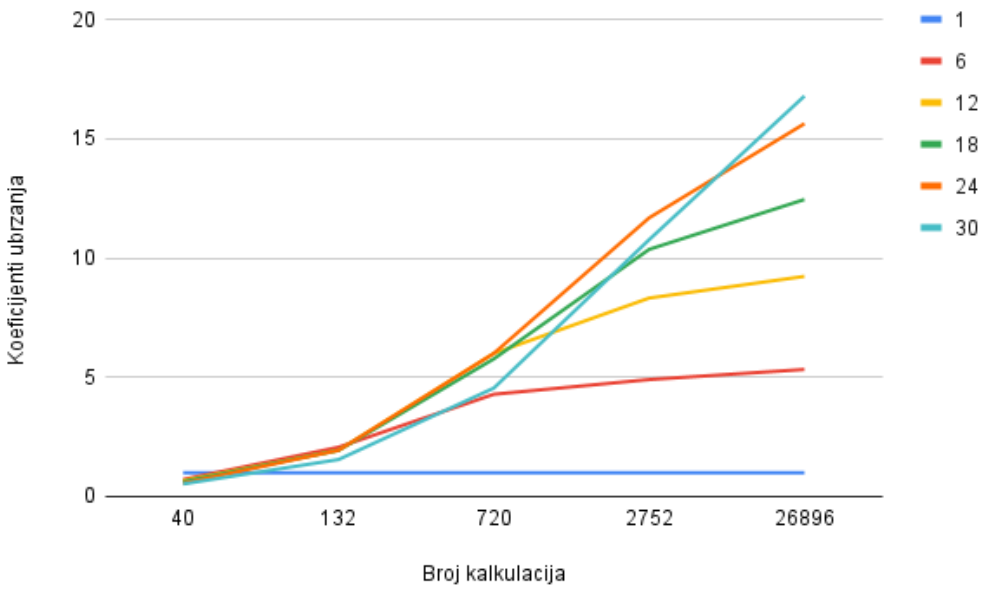


Slika 5.14 Graf napretka u performansama rezultata na procesoru AMD Ryzen™ 3 2200U

Sada je moguće analizirati podatke dobivene izvršavanjem programa na superračunalu. Prva mjera čiji su rezultati prikazani u tablici 5.5, a na grafu u slici 5.15 jest ubrzanje. U samome početku su svi koeficijenti manji od jedan, odnosno izvršavanja su sporija od serijskog. Povećanjem kompleksnosti problema raste i koeficijent ubrzanja, pa tako u zadnjem slučaju na trideset dretvi dolazi čak do 16.79, odnosno izvršavanje je gotovo 17 puta brže od serijskog što koeficijente iz tablice 5.12 čini gotovo trivijalnima. Također sam oblik grafa je vrlo zanimljiv po tome što ona izvršavanja na manjem broju dretvi u početku imaju veće koeficijente, a rastom broja kalkulacija oni nastavljaju rasti gotovo linearno, dok ona izvršavanja na velikom broju dretvi, posebice onom na tridest dretvi, rastu gotovo eksponencijalno, pa tako povećanjem broja kalkulacija prestižu ostala izvršavanja na manjem broju dretvi.

Tablica 5.5 Tablica ubrzanja rezultata na superračunalu

	Broj dretvi					
Broj kalkulacija	1	6	12	18	24	30
40	1	0.7143	0.6667	0.6250	0.5556	0.5263
132	1	2.0667	1.9375	1.9375	1.9375	1.5500
720	1	4.2857	6.0000	5.7692	6.0000	4.5455
2752	1	4.9018	8.3182	10.3585	11.6809	10.7647
26896	1	5.3265	9.2270	12.4498	15.6276	16.7871

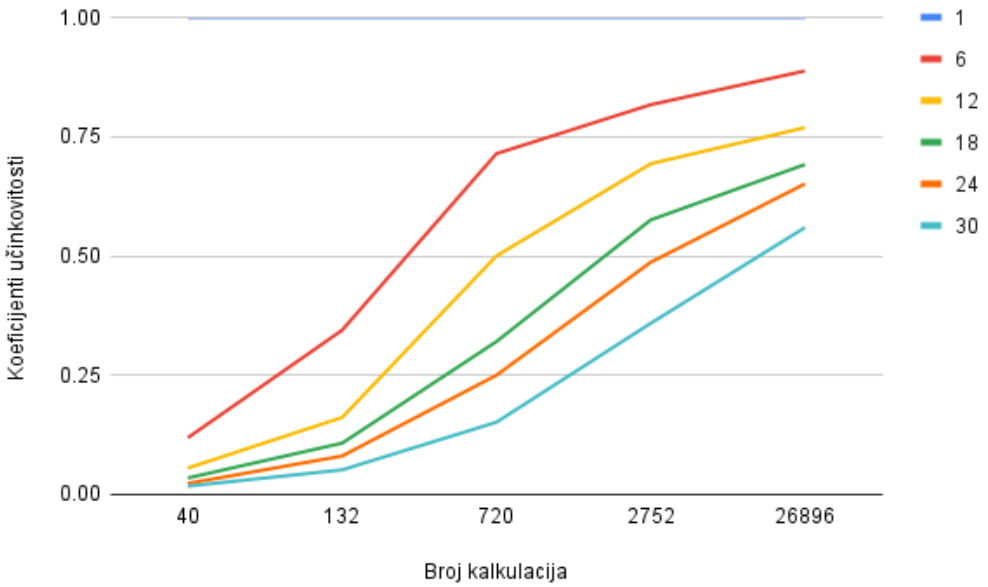


Slika 5.15 Graf ubrzanja rezultata na superračunalu

Iduća mjera učinkovitosti je prikazana u tablici 5.6 i grafu na slici 5.16. Svojim koeficijentima i izgledom grafa je vrlo slična rezultatima s prijenosnog računala, no glavna je razlika što su koeficijenti puno bliže onome serijskog izvršavanja, ali i dalje u potpunosti ne opravdaju velik broj korištenih dretvi.

Tablica 5.6 Tablica učinkovitosti rezultata na superračunalu

	Broj dretvi					
Broj kalkulacija	1	6	12	18	24	30
40	1	0.1190	0.0556	0.0347	0.0231	0.0175
132	1	0.3444	0.1615	0.1076	0.0807	0.0517
720	1	0.7143	0.5000	0.3205	0.2500	0.1515
2752	1	0.8170	0.6932	0.5755	0.4867	0.3588
26896	1	0.8878	0.7689	0.6917	0.6512	0.5596

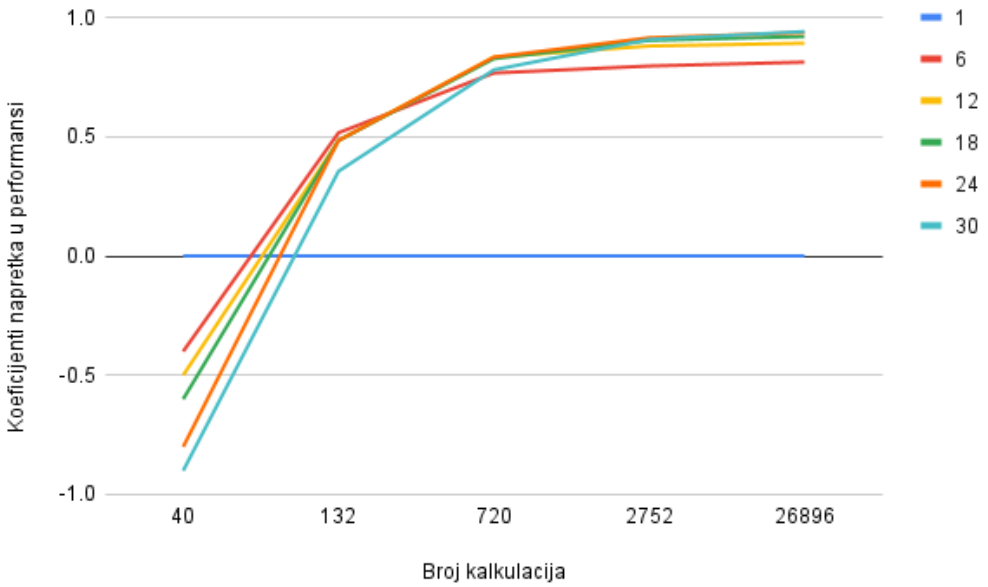


Slika 5.16 Graf učinokovitosti rezultata na superračunalu

U posljednjoj mjeri napretka u performansama prikazanim u tablici 5.7 i slici 5.17 rezultati su ponovo slični onima koji predstavljaju ubrzanje. Za razliku od opažanja da je samo izvršavanje na šest dretvi bolje od serijskog u drugome slučaju kao kod prijenosnog računala, ovdje su svi koeficijenti napredtka u performansi veći od serijskog u drugom slučaju, a negativni su samo oni u prvom slučaju. Također ono opažanje da porastom broja kalkulacija i broja dretvi dolazi do većeg napretka koje je opisano kod ubrzanja na superračunalu još uvijek vrijedi.

Tablica 5.7 Tablica napretaka u performansama rezultata na superračunalu

	Broj dretvi					
Broj kalkulacija	1	6	12	18	24	30
40	0	-0.4000	-0.5000	-0.6000	-0.8000	-0.9000
132	0	0.5161	0.4839	0.4839	0.4839	0.3548
720	0	0.7667	0.8333	0.8267	0.8333	0.7800
2752	0	0.7960	0.8798	0.9035	0.9144	0.9071
26896	0	0.8123	0.8916	0.9197	0.9360	0.9404



Slika 5.17 Graf napretka u performansama rezultata na superračunalu

Analizom rezultata moguće je zaključiti da je hipoteza bila ispravna te da se povećanjem kompleksnosti problema i dodavanjem većeg broja dretvi, vrijeme računanja smanjuje. Iako kod prijenosnog računala s malim brojem jezgara to nije toliko izraženo te su napredci minimalni, kod superračunala koje je moguće upotrijebiti gotovo svaku dodatnu dretvu na koju se proces raspodijeli, ti isti napretci postaju i do nekoliko desetaka puta veći. Također taj isti napredak u ubrzanju izvršavanja ne znači nužno da je taj način izvršavanja i učinkovitiji, već će korištenje većeg broja dretvi gotovo uvijek rezultirati manjom učinkovitošću naspram serijskog izvršavanja.

Poglavlje 6

Zaključak

Kao i sam rad koji promatra te spaja dva različita svijeta i perspektive, financijsku i računarsku, tako se i sam zaključak može podijeliti u dva dijela.

Prvi takav dio je računarski dio iz kojega je bitno napomenuti da još kada su tehnologije budućnosti poput kvantnog računanja u ranim danima svoga razvoja, jedan od glavnih načina na koji je moguće povećati brzinu izvršavanja programa jest korištenjem većeg broja jezgara, odnosno paralelizacijom kôda. Iako nije moguće bilo koji program učiniti spremnim za paralelizaciju, pažljivim planiranjem, ali isto tako i pomoću visoke razine razumijevanja problema, moguće je uvelike smanjiti vrijeme koje je potrebno da se on izvrši. Iako se ovakve male promjene u vremenu izvršavanja čine trivijalnima, prilikom kompleksnih računanja mogu uštedjeti puno vremena, a ukoliko se radi o izvršavanju koje mora biti gotovo istovremeno svaki dijelić sekunde ovisi o tome hoće li se problem na vrijeme izvršiti. Također ako se tome i pridoda potrošnja energije, poželjno je da program bude što efikasniji kako bi što manje energije bilo utrošeno na njegovo izvršavanje.

No svaka tehnologija nije uvijek na korist čovječanstvu, pa isto tako i računarstvo visokih performansi. Iako ono donosi mnoge prednosti nad izvršavanjem programa koji ne primjenjuje paralelizaciju kôda uvijek se bitno zapitati kome ovakve tehnologije najviše koriste. Svakako se koriste za razne dobrobiti primjerice u istraživanjima raznih lijekova, pa tako je i jedan od glavnih zadataka superračunala Fugaku bio otkrivanje kandidata za lijek protiv bolesti Covid-19 [13]. No s druge strane, a posebice u financijskom svijetu se pojavljuje sve više bivših programera kao što su Alexis Gol-

dstein i Dave Lauer koji su radili za velike institucije koje se bave visoko frekventnim trgovanjem, a danas se aktivno bave sprječavanjem istih institucija u zarađivanju pomoću takovih naprednih algoritama. Pametnim korištenjem visoko frekventnog trgovanja velike institucije su u velikoj prednosti naspram malih ili individualnih ulagača što je za njih poželjno, no za velik broj ostalih svakako nije.

No u konačnici računarstvo visokih performansi jest područje s puno potencijala ukoliko se ono pravilno iskoristi za dobre svrhe.

Bibliografija

[1] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, ser. Chapman & Hall/CRC Computational Science. CRC Press, 2010.

[2] High frequency trading. , s Interneta, https://thehedgefundjournal.com/high-frequency-trading/ , 17.07.2021.

[3] M. Dempster, J. Kanniainen, J. Keane, and E. Vynckier, *High-Performance Computing in Finance: Problems, Methods, and Solutions*, ser. Chapman and Hall/CRC Financial Mathematics Series. CRC Press, 2018.

[4] Top500, june 2021. , s Interneta, https://www.top500.org/lists/top500/2021/06/ , 21.07.2021.

[5] K. Lenac, "Dretve," prezentacija iz kolegija Operacijski sustavi.

[6] Multithreading (computer architecture). , s Interneta, https://en.wikipedia.org/wiki/Multithreading_(computer_architecture) , 22.07.2021.

[7] Options. , s Interneta, https://www.investopedia.com/terms/o/option.asp , 28.07.2021.

[8] Barchart amd options chain. , s Interneta, https://www.barchart.com/stocks/quotes/AMD/options , 28.07.2021.

[9] Volatility definition. , s Interneta, https://www.investopedia.com/terms/v/volatility.asp , 05.08.2021.

[10] Openmp application programming interface. , s Interneta, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf , 19.08.2021.

[11] Amd ryzen™ 3 2200u. , s Interneta, https://www.amd.com/en/products/apu/amd-ryzen-3-2200u , 23.08.2021.

[12] N. Thakur, S. Kumar, and V. Patle, "Comparison of serial and parallel searching in multicore systems," *International Conference on Parallel, Distributed and Grid Computing*, 2014.

[13] J. Dongarra, "Report on the fujitsu fugaku system," University of Tennessee, Knoxville, Tech. Rep., 2020.

Sažetak

U ovome završnome radu su povezani računarstvo visokih performansi sa svijetom financija na primjeru računanja financijskog derivata opcija korištenjem programa koji primjenjuje paralelizaciju kôda na većem broju procesora superračunala. U konačnici su rezultati izvršavanja programa uspoređeni s izvršavanjem serijskog kôda na jednom procesoru.

Ključne riječi **— Računarstvo visokih performansi, Opcije, Paralelizacija kôda, Superračunalo**

Abstract

In this bachelor's thesis high performance computing meets the world of finance on an example of computing the financial derivative options using a program that applies the parallelization of code on a larger number of processors of a supercomputer. The results of program execution are compared with serial code execution on one processor.

Keywords **— High performance computing, Options, Code parallelization, Supercomputer**

Dodatak A

Programski kôd

Čitavi programski kôd je dostupan na CDu priloženom uz završni rad, a alternativno je dostupan na stranici `www.github.com/mmatotan/option-calculator`.