Paralelni algoritam za Josipov problem u programskom jeziku C++

Kodžoman, Leon

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

PARALELAN PROGRAM ZA JOSIPOV PROBLEM U PROGRAMSKOM JEZIKU C++

Završni rad

Leon Kodžoman

Osijek, 2021.

SADRŽAJ

1. UVOD

Imenovan po Flavije Josipu (Titus Flavius Josephus**,** 37. – oko 100., rimsko-židovski povjesničar), ovaj problem potječe iz njegovog doživljaja opsade Yodfata gdje su Rimski vojnici zarobili njega i 40 drugih vojnika u pećini. Odabirući samoubojstvo umjesto zarobljeništva odlučuju da će se pogubljivati naizmjence. Josip tvrdi da ili čistom srećom ili božjom rukom, on i još jedan vojnik su ostali do kraja te su se ipak predali umjesto da počine samoubojstvo [1].

Točni detalji postupka pomoću kojeg su izveli taj pothvat su nejasni te se interpretacije uglavnom svode na krug osoba te da se od određene osobe odbrojava do nekog broja te da čim se dosegne taj broj ta osoba se uklanja iz kruga. U Josipovom slučaju se uglavnom uzima da je taj korak eliminacije ili uklanjanja jednak 3 te da na kraju moraju ostati dvije preživjele osobe. Da bi te dvije osobe preživjele moraju se odabrati dvije pozicije u krugu na kojima bi te osobe trebale stati.

U prvom poglavlju će se obraditi slučaj kada je korak eliminacije q jednak 2. U tom slučaju je moguće izvesti funkcije koje izravno povezuju broj osoba i broj osobe koja će preživjeti. Ovaj korak je nužan jer daje uvid u problem te način rješavanja.

U drugom poglavlju, dodavanjem dodatnih uvjeta, se dobije opći problem koji se rješava kasnije u radu. Predstavljen je opći problem kao i način prikazivanja kombinacija koje su opisane kasnije u radu.

Analiza problema, koja je tema trećeg poglavlja, opisuje vremensku, prostornu i ukupnu kompleksnost problema. Kompleksnost je visoka te izravno utječe na implementaciju.

Implementacija algoritma se radi u četvrtom poglavlju gdje se također optimiziraju dva dijela algoritma da bi se program izvršavao brže.

1.1. Zadatak završnog rada

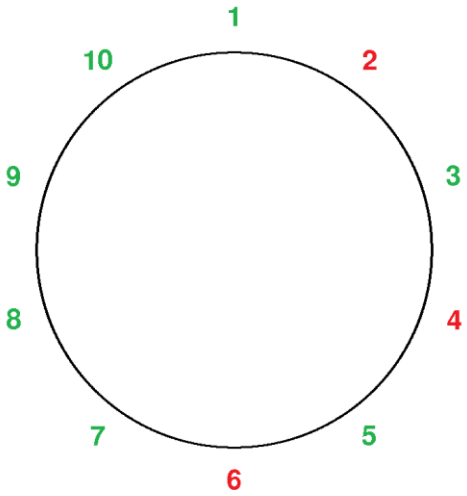
Zadatak završnog rada je matematički obraditi Josipov problem i kreirati te analizirati paralelni algoritam za njega te ga primijeniti kako bi se dobili rezultati za veće slučajeve problema. Dobiveni rezultati performansi algoritma će biti uspoređeni međusobno te djelomično između dva računala.

2. JOSIPOV PROBLEM

Francuski matematičar Theriault Nicolas se bavio Josipovim problemom u svom radu [2] iz 2001. godine te je predstavio dva algoritamska rješenja problema. Također je opisan broj koraka jednog od tih algoritama te je također i predstavljeno rješenje drukčije inačice problema. Jedna posebna verzija problema je Mačji Josipov problem u kojem svaka osoba ima nekoliko „života“ tijekom uklanjanja. Na takvoj inačici problema su radili 2018. godine autori članka [3] u kojem su tražili koja je zadnja preživjela osoba. Njihova inačica problema je posebna po tome što uklanjaju po jedan život sljedećih k vojnika te onda preskoče jednog. U određenim situacijama se pronalaze formule za preživjelu osobu j. Nadalje, Mačjim Josipovim problemom su se bavili 2010. godine autori Ruskey Frank i Williams Aaron [4]. Dokazali su dva rezultata. Kada su n i k stalni onda je j konstantan za svaki broj života l veći od n-tog Fibonaccijevog broja. Kada su n i j stalni onda postoji vrijednost za k koja omogućuje da je j preživjela osoba za bilo koji l. Za prvu tvrdnju su dali algoritam za određivanje preživjele osobe j.

Josipov problem u osnovnom obliku, opisan u ovom poglavlju, je veoma jednostavan te postoji nekoliko različitih programskih pristupa koji rješavaju problem [5]. Traženje, kasnije opisanih, ne-Josipovih skupova je složenije od osnovnog problema te za njega postoji još manje istraživanja. Rad koji se bavio upravo tim specifičnim problemom ima jedan vrlo maleni dio posvećen problemu, ali u tom dijelu nije demonstriran algoritam rada nego samo osnovna zapažanja i rezultati do $n = 26$ [6].

Za zadane prirodne brojeve n i q koji predstavljaju broj osoba u krugu i korak eliminacije, potrebno je pronaći poziciju preživjele osobe na kraju procesa uklanjanja. Jedan jednostavan slučaj koji je lako rješiv je slučaj kada je n ljudi u krugu te je korak eliminacije $q = 2$. U tom slučaju je moguće izvesti jednadžbu pomoću koje se može izračunati rješenje za svaki n.

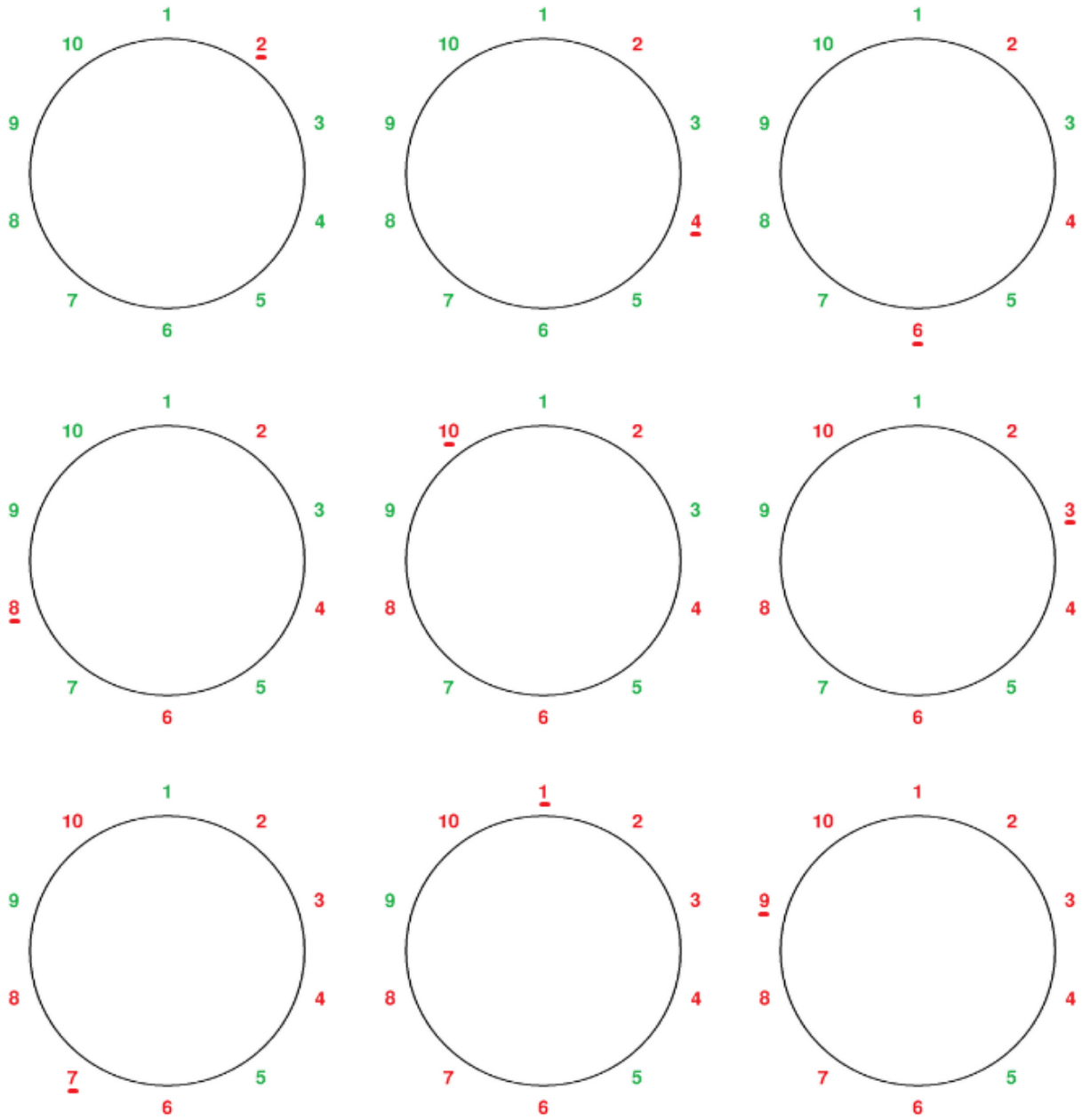


Slika 2.1.: Prve tri eliminirane osobe: 2 ,4 i 6.

Algoritam započinje na način da se započne s prvom osobom te da se kreće u smjeru kazaljke na satu. Uklanjanje se obavlja na način da se broji od 1 sve do koraka eliminacije. Kada brojanje postigne korak eliminacije, pripadna osoba se ukloni te se opet započinje brojat od 1. Na slici 2.1 se može vidjeti da su tim algoritmom prve 3 uklonjene osobe: 2, 4 i 6. Nastavljajući s algoritmom se uklanjaju osobe sve dok ne preostane jedna osoba.

Na slici 2.2 ispod se može vidjeti redoslijed uklanjanja osoba sve dok ne preostane jedna. Zeleno označene osobe su osobe koje aktivno sudjeluju u kretanju po krugu, a crveno označene osobe su one koje su uklonjene iz kruga dok su podcrtane osobe one koje su zadnje uklonjene.

Kao što se može vidjeti na slikama, uklanjanje se izvodi sve dok u krugu ne ostane osoba broj 5. Za slučaj kada imamo 10 osoba te korak 2, prvo se uklanja osoba 2, zatim osoba 4, osoba 6, osoba 8 te osoba 10. Nakon toga prva osoba je osoba 1, a druga osoba je osoba 3, koja se uklanja, jer one koje su uklonjene se ne uzimaju u obzir. Na isti način se nastavlja dalje, preskače se osoba 5 te se uklanja osoba 7. Prateći ovakav algoritam do kraja uklanjaju se osobe 1 i 9 te preostane osoba 5. Redoslijed uklanjanje za ovaj slučaj je: 2, 4, 6, 8, 10, 3, 7, 1 i 9. Iz prethodno navedenog niza uklanjanja te vizualizacije na slici 2.2 odmah ispod može se uočiti nekoliko obilježja uklanjanja.



Slika 2.2.: Prikaz redoslijeda eliminacije za n = 10, q = 2.

Jedna od očitijih pravilnosti je da se u prvom krugu uklone sve osobe na parnim mjestima. Ako je u krugu bio paran broj osoba može se zaključiti rekurzivna funkcija $f(n) = 2f\left(\frac{n}{2}\right) - 1$ gdje je $f(n)$ broj preživjele osobe ako se u krugu na početku nalazi n osoba. Ta funkcija povezuje pozicije preživjelih osoba prije i poslije svakog kruga uklanjanja. Ako je pozicija osobe u drugom krugu uklanjanja 4 onda se iz relacije može izračunati gdje je ta osoba bila prije kruga uklanjanja: $x = 2 * 4 - 1 = 7$. Osoba 4 je nakon prvog kruga zapravo bila 7 jer su parne osobe uklonjene. Pomoću slike 2.2 se može potvrditi valjanost relacije.

Slična funkcija se dobiva ako je neparan broj osoba u krugu. Funkcija za neparan broj osoba je: $f(n+1) = 2f\left(\frac{n}{2}\right) + 1$. Kada se napravi tablica na osnovi vrijednosti *n* i vrijednosti $f(n)$ dobije se sljedeće:

Tablica 2.1.: Pozicija osobe koja treba preživjeti.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$f(n)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1

Prema tablici se može uočiti uzorak da pozicija preživjele osobe je uvijek 1 kada je *n* potencija broja 2. U ostalim su slučajima pozicije sve neparne te se uvijek razlikuju za 2 te s rastom broja *n* i pozicije rastu. Način za izravno odrediti $f(n)$ je jednostavan. Prvi korak je da se pronađe najveća potencija broja 2 koja je manja od uzetog broja *n* te da se oduzme od broja *n*. Rezultat prethodnog koraka će se zvati *l*. Krajnje rješenje je da se *l* udvostruči te da se pribroji 1. Za primjer će se uzeti *n* = 14. Uklanjanjem najveće potencije broja 2 se dobije 14 - 8 = 6, množenje s 2 daje 12 te se doda 1 da se dobije rezultat 13. Ostatak brojeva ovoga niza se može pronaći na poveznici u popisu literature [7].

Opći slučaj problema gdje je *q* promjenjiv, a na kraju treba ostati jedna osoba je zanimljiviji i korisniji za rad, no jedan od problema s promjenjivim brojem *q* je što za *q* iznad 3 ne postoji gotova eksplicitna formula nego postoji samo rekurzivno rješenje koje će biti opisano i korišteno kasnije u radu. Da bi se odredilo na koji način se uklanjaju osobe treba se prvo uočiti da nije važno koliki je *q* jer ako je veći od broja ljudi onda se samo „omotava" oko kruga ljudi. Efektivno ako je pozicija osobe u krugu od 10 ljudi 7, to znači da će tu osobu jednako ukloniti koraci za *q* od 7, 17, 27 itd. Primjećuje se da ti brojevi zadovoljavaju sljedeći izraz: $i = (q \,\%\, n)$. Naravno, taj izraz radi samo za prvo uklanjanje, no ako samo preimenujemo preostale osobe može se nastaviti koristiti sve dok ne ostane jedna osoba.

Ako se uzme da je $f(n, q)$ broj osobe koja će preživjeti gdje je *n* broj osoba, a *q* je korak eliminacije onda jednadžba koje opisuje rekurzivnu prirodu općeg slučaja je sljedeća:

$$f(n, q) = \big((f(n-1, q) + q - 1)\ mod\ n\big) + 1 \qquad f(1, k) = 1$$

Operacija „mod" je oznaka za modulo, tj. ostatak nakon cjelobrojnog dijeljenja.

3. NE - JOSIPOVI SKUPOVI

U prethodnom poglavlju je opisan opći slučaj kada nije bitno koliki je q. Moguće je još više generalizirati problem ako se uzme da ne mora ostati jedna osoba na kraju nego više osoba.

Ako na kraju u krugu mora preživjeti k odabranih osoba te osobe mogu biti u više različitih pozicija te se više ne traži koja osoba treba ostati nego se iz osoba koje moraju ostati traži korak eliminacije q.

Za primjer će se uzeti $n = 5, k = 2$. Ako je 5 ljudi u krugu te 2 osobe moraju preživjeti te dvije osobe mogu biti na pozicijama {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 5} {4, 5}. Kao što se može uočiti, više nije trivijalno doći do rješenja, čak i rekurzivno jer više ne postoji samo jedno rješenje. U sljedećoj tablici je prikaz mogućih pozicija osoba u krugu. U tablici je svaka pozicija gdje osoba treba preživjeti označena s "X", a pozicije gdje nije bitno da osoba preživi su označene s "-".

Tablica 3.1.: Prikaz mogućih rasporeda dvije osobe na pet mjesta.

Slučaj	Pozicija				
	1	2	3	4	5
1	X	X	-	-	-
2	X	-	X	-	-
3	X	-	-	X	-
4	X	-	-	-	X
5	-	X	X	-	-
6	-	X	-	X	-
7	-	X	-	-	X
8	-	-	X	X	-
9	-	-	X	-	X
10	-	-	-	X	X

Uzimajući slučaj 3, skup {1, 4} za primjer, može se demonstrirati težina pronalaska rješenja. Ako se krene s $q = 2$, osobe se uklanjaju redom: 2, 4, te je nastao problem. Skup kaže da osoba 4 mora preživjeti, no odabran korak ju uklanja što znači da $q \neq 2$. Dalje, kada je $q = 3$, uklanjaju se osobe redom: 3, 1, te opet kao u prvom slučaju dođe do konflikta, ali ovaj puta s osobom 1 stoga $q \neq 3$. Za $q = 4$ može se uočiti da se odmah treba ukloniti osoba *4* što nije dozvoljeno te $q \neq 4$. Niti jedan korak sve do 8 nije rješenje te je prvo rješenje $q = 8$ što daje redoslijed uklanjanja: 3, 2, 5.

Kada se prikaže grafički u tablici može se primijetiti očiti uzorak mogućih rasporeda osoba. Naime, zbog činjenice da od *n* ljudi biramo *k* ljudi broj slučajeva se može zapisati kao broj kombinacija. Broj slučajeva je $\binom{n}{k}$. Za gore naveden primjer $\binom{5}{2} = 10$. Prethodno naveden izračun vrijedi za slučaj kada je *k* jednak 2. Za ostale slučaje se računa na isti način, no samo se mijenja *k*. Kada bi se ispisali svi slučaji za $n = 5$ te se sortirali po binarnoj reprezentaciji skupova, koja je opisana odmah ispod, onda bi se dobio zanimljiv uzorak koji je od velike važnosti kasnije. Pozicije gdje osoba treba preživjeti će se prikazati s „1”, a ostale s „0”. Svaki raspored osoba će se tretirati kao niz jedinica i nula: svaki skup će se prevesti u ekvivalentni prikaz s jedinicama i nulama koji će se nadalje zvati kombinacija. Skup {4, 5} će na mjestima 4 i 5 imati “1”, a na ostalima “0” što znači da će izgledati kao *00011*. Skup {1, 4} će ekvivalentno biti *10010*.

Tablica 3.2.: Prikaz sortiranih kombinacija i njihovih rješenja.

Kombinacija	Obrazloženje	Rješenje
00000	Svi se trebaju ukloniti	q > 0
00001	Preživljava osoba 5	1
00010	Preživljava osoba 4	3
00011	Preživljavaju osobe 4 i 5	1
00100	Preživljava osoba 3	2
...	...	...
11100	Preživljavaju osobe 1, 2 i 3	9
11101	Preživljavaju osobe 1, 2, 3 i 5	4
11110	Preživljavaju osobe 1, 2, 3 i 4	5
11111	Preživljavaju svi	Ne postoji $q > 0$

Kao što se može uočiti broj kombinacija je ovisan o n eksponencijalno, preciznije broj kombinacija je: $N = 2^n$ zato što se iz uzorka vidi da se kombinacije mogu prikazati kao binarna reprezentacija koje je potencija broja 2.

Kada se, radi jednostavnosti, da se ne traži ručno, napiše računalni program čija je zadaća pronaći prvi q koji zadovoljava svaki slučaj kada je n = 5 dobiju se rezultati u desnom stupcu tablice 3.2. Ako se program primjeni na sve većim vrijednostima od n gdje naravno raste broj mogućih kombinacija primijeti se da se program naizgled smrzne kada je n = 9. Kada se izračuna broj kombinacija koje treba provjeriti, ne bude veoma visok, bude jednak 2^9 = 512 što je za moderna računala vrlo malena brojka. Detaljnim praćenjem programa zapravo se dođe do jedne kombinacije gdje program nije u stanju pronaći q. Kada se pusti program da traži q za tu kombinaciju veoma dugo i dalje nije u stanju pronaći rješenje. Kombinacija na kojoj se program zaustavi je, primjenjujući isti način označavanja s „1" i „0", *010011110*. Nakon što se program izmijeni da obustavi pretragu kada se postigne određena gornja granica pretraživanja, program završi te za n = 9 se istaknu 3 slučaja koji nemaju rješenje: *010011110, 011110010* i *110010011*.

Za provjeru nepostojanja rješenja za q može se uzeti redoslijed eliminacije 3, 4, 6, 7. Za uklanjanje prve osobe odgovara korak $3 + 9k$. Za uklanjanje druge osobe odgovara $1 + 8m$. Za treću osobu odgovara $2 + 7r$ i za četvrtu osobu odgovara $1 + 6s$.

$$q = 3 + 9k = 1 + 8m = 2 + 7r = 1 + 6s$$

Zbog $3 + 9k$, q mora biti djeljiv s 3, no $1 + 6s$ dijeljenjem s 3 daje ostatak 1 što nije moguće. Prema tome ne postoji kombinacija koja odgovara. Isti je slučaj s ostalim redoslijedima u sva 3 slučaja.

Podskupovi skupa {1, 2, 3, ..., n} osoba koje se na početku nalaze u krugu za koje ne postoji korak eliminacije q koji će baš te osobe na kraju ostaviti u krugu se zovu ne-Josipovi skupovi te je cilj napraviti novi program čija je svrha što brže pronalaziti takve slučaje. Ne-Josipovi skupovi se odnose samo na podskupove nekog nadskupa, a ne svih nadskupova, npr. skup {2, 3, 4, 5, 8} jest ne-Josipov podskup od {1,2,3,4,5,6,7,8,9}, no taj isti podskup je podskup od skupa {1,2,3,4,5,6,7,8} što dokazuje q = 23, što nalaže da će se pod skupovi gledati odvojeno za svaki n.

Program koji će biti predstavljen je dizajniran da što brže pronalazi ne-Josipove skupove za razne vrijednosti broja n. Program će raditi na principu da zapravo pronalazi rješenja za svaki

skup. Kada se završi s pronalaženjem rješenja dovoljno je prebrojati rješenja koja nisu pronađena tijekom rada.

4. ALGORITAM ZA RAČUNANJE RASPOREDA UKLANJANJA OSOBA

Prvi korak u formiranju programa je odabiranje pogodnog algoritma. Neki algoritmi mogu biti učinkoviti teoretski, no veoma teški za implementirati, a mogu biti i jednostavni za implementirati, no spori itd. Algoritam koji će se nadalje koristiti za implementaciju će biti iterativna metoda slična rekurzivnoj za opći slučaj samo što će se pratiti rezultati nakon svakog koraka. Algoritam je jednostavan za implementirati, no ima, blago rečeno, veliku kompleksnost koja će biti opisana kasnije u radu. Odabir osnovnog algoritma samo pruža informaciju o odabiru osoba za ukloniti te ne pruža pogled u ostale popratne stvari nužne za program. Potrebno je usput pratiti koliko se skupova pojavilo te koji ne-Josipovi skupovi su se pojavili. Popratne strukture podataka su od iznimne važnosti jer njihov odabir može imati znatan učinak na performanse te skalabilnost i paralelizaciju programa.

4.1. Uklanjanje osoba

Uklanjanje će se izvoditi na način da se stvori niz osoba te će se na njima izvoditi uklanjanje. Najjednostavnija metoda uklanjanja ispadne najbrža, a to je da se osoba koja se ukloni fizički ukloni iz tog niza. Može se raditi i s preusmjerivanjem pokazivača, no ta metoda ispadne spora i neučinkovita zbog mnogo dereferenciranja pokazivača.

Implementacija niza osoba će biti jedna vrijednost duljine 64 bita gdje svaki pojedini bit predstavlja osobu na tom mjestu. Na taj način se izravno može tretirati svaki bit kao jedna pozicija u krugu. Prethodno naveden način označavanja osoba s „0“ i „1“ je prvenstveno izveden iz osnovne implementacije zbog svoje jednostavnosti. Obilježje ovakvog načina rada je da kada se zapisuju rezultati nije potrebno zapisivati kako izgleda koje rješenje jer sami niz osoba kao brojčana vrijednost (u dekadskom sustavu) govori na kojem mjestu se nalazi. Za primjer će se uzeti prvi ne-Josipov skup za $n = 9$: *010011110*. Kada se taj broj interpretira u bazi *10* dobije se vrijednost 158 što jedinstveno označava da je mjesto 158 u nizu rješenja upravo ta kombinacija.

4.2. Zapis rezultata

Zapisivanje će se izvoditi tako da se na početku stvori jedan veliki niz svih kombinacija. Jedno bitno obilježje ovog načina rada je da taj popis mora biti prostorne kompleksnosti $O(2^n)$ gdje je *n* broj osoba jer toliko ima kombinacija. Budući da su sve kombinacije jedinstvene, potrebno ih je u programu samo označavati da li su pronađene. Oznake će biti najmanji tip podatka koji je C++ jeziku cijeli broj od 8 bitova. Uzet će se taj poseban tip podataka radi jednostavnosti te radi štednje memorijskog prostora. Moguće je koristiti i bolji način gdje se vrijednosti „pakiraju" u individualne bitove, no za raspon brojeva koji algoritam mora koristiti takva metoda nije nužna za implementirati.

4.3. Računanje pozicije osobe koja će se ukloniti

Srž algoritma je način uklanjanja pozicija. Umjesto korištenja rekurzivne formule koristit će se iterativni način rada koji imitira isti način na koji bi čovjek uklanjao osobe iz kruga. Prvo svojstvo za primijetiti je da bez obzira koliki je *q*, uvijek se reducira na *q* u rasponu $[0, n\rangle$, ako uzmemo da je prva osoba na poziciji 0. Nakon prvog kruga dobijemo poziciju gdje se osoba treba ukloniti pomoću malene jednadžbe korištene ranije u radu: *i = (q % n)*. Nakon prvog kruga nedostaje jedna osoba tako da je *n* za jedan manji te se više ne počinje od prve osobe nego od osobe na poziciji *indeks*. Za sljedeći krug relacija glasi:

$$i = (i + q) \,\%\, (n - 1)$$

Opći oblik se može zapisati na sljedeći način:

$$i = (i + q) \,\%\, (n - m)$$

Za svaki sljedeći krug *i* se stalno mijenja te *n* opada. Za prvi krug se uzme da je početni indeks 0 te je funkcija ista za svaki sljedeći samo što *n – m* opada do 0. Kada vrijednost *n – m* postane 1 onda je samo jedna osoba preostala u krugu.

4.4. Kompleksnost algoritma

Kompleksnost algoritma je zbog njegove prirode znatna. Naivno rješenje korišteno u programu ranije u radu kada su se tražila rješenja za n = 5 je takvo da se za svaku kombinaciju tražio q. Takvo rješenje je vremenske složenosti $O(2^n)$ jer se mora proći kroz svaku kombinaciju. Gornja granica do koje se traže rješenja, također još jedna vremenska kompleksnost, je određena eksperimentalno te je također kompleksnosti kao i vremenska, $O(2^n)$. Postoji problem sa ovakvim pristupom jer to je samo određeno iz rezultata programa. Prava gornja granica je jedna od najlošijih kompleksnosti koja postoji. Ako se uzme da je n = 5 onda prvi krug može pogoditi jednu od 5 osoba, sljedeći krug jednu od 4 i tako sve do 1 te se iz toga može zaključiti kompleksnost od $n!$. Takva kompleksnost je vrlo loša jer već za n = 12 gornja granica postane veoma velika: 479001600, sljedeći n je već prevelik te ne stane u cijeli broj od 32 bita. Iz tog razloga će gornja granica biti bazirana na potencijama broja 2.

Glavni izvor „optimizacije“ programa je žrtvovanje memorije za vrijeme izvođenja jer ako se prođe samo jednom po vrijednostima q do gornje granice te se usput proizvede tko treba preživjeti onda se izbjegne skup proces prolaska kroz svaku kombinaciju te traženje rješenja. U srži, umjesto da se traži rješenje, počne se od rješenja te se traže kombinacije. Na taj način se kompleksnost smanji s $O(2^{2n})$ vremenski i $O(1)$ prostorno na $O(2^n)$ vremenski i $O(2^n)$ prostorno. Implementacija je bazirana na prethodno navedenoj „optimizaciji“ te je to ujedno i ograničavajuća okolnost programa. Čak i da se koriste individualni bitovi za spremanje informacija već za n = 35 je potrebno 4 GB radne memorije. Za n = 40 je potrebno 128 GB memorije, a za n = 45 je potrebno 4096 GB ili 4 TB memorije.

Još jedan čimbenik vremenske kompleksnosti je petlja koja će uklanjati osobe. Ona će biti proporcionalna broju ljudi tako da je ona $O(n)$ vremenske kompleksnosti. Ukupna teoretska kompleksnost, ako bi se uzelo sve prema teoretskim vrijednostima, je $O(n * n! * 2^n)$. Završna kompleksnost algoritma koji će se koristiti za implementaciju je $O(n * 2^{2n})$.

5. IZVEDBA ALGORITMA U PROGRAMSKOM JEZIKU C++

Izvedba algoritma započinje s osnovnim algoritmom bez ikakve paralelizacije i dodatnih ubrzavanja tako da se dobije osnova na koju se mogu dodati kompleksne stvari naknadno. Osnovna izvedba će biti jedna funkcija koja se pozove te ona izračuna sve što je potrebno. Svi glavni podatci, poput polja oznaka, se pripremaju prije pokretanja funkcije. Implementacija je izvedena u programskom jeziku C++ zbog visokih performansi te jednostavnosti korištenja višenitnog (engl. multithreaded) načina rada te ostalih stvari koje su nužne za program.

5.1. Osnovna izvedba

Da bi se moglo određivati tko je u krugu, a tko nije, napravit će se polje u kojem se spremaju pozicije osoba. Polje se sastoji od tipa podataka nazvanog uint8_t koji predstavlja cijeli broj od 8 bitova. 8 bitova je dovoljno jer kompleksnost programa svejedno ne dopušta više od 256 osoba.

Prije početka programa se definira nekoliko pretprocesorskih naredbi kao na slici 5.1.1. „N“ predstavlja *n*, a „UPPER_BOUND“ je gornja granica do koje se traže rješenja. Ona je pronađena eksperimentalno nakon mnogo testiranja.

```
#define N 24

#define UPPER_BOUND std::pow(2, 1.1 * N + 1)
```

Slika 5.1.1.: Pretprocesorske naredbe

U funkciji koja traži rješenja prvo se napravi polje koje će predstavljati pozicije osoba. Odmah nakon kreiranja polja, ono se popuni s pripadnim vrijednostima. Također se napravi i dvostruko veće polje koje će se koristiti tijekom rada programa. Dvostruko je duže jer će kasnije, kada se kopira memorija, biti brže kopirati više bajtova nego provjeravati koliko ih treba kopirati.

```
uint8_t startingIndices[64];
uint8_t positionIndices[128];
for (int64_t i = 0; i < N; i++) {
	startingIndices[i] = N - i - 1;
}
```

Slika 5.1.2.: Polja pozicija

Srž algoritma je sadržana u sljedećem dijelu koda. Prije petlje se zabilježi gornja granica te se ona koristi u petlji (maxQ). Petlja obilazi svaki *q* od početka (startQ) do gornje granice te svaki put prekopira indeks osoba (pomoću funkcije „memcpy“) što vrati krug na početno stanje. Zatim se kreira nova vrijednost koja će pratiti kombinaciju kao vrijednost od 64 bita. Na početku se pretpostavi da su sve osobe prisutne. Također se zabilježi indeks te koliko je ljudi u krugu.

```
int64_t maxQ = int64_t(UPPER_BOUND);
for (int64_t q = startQ; q <= maxQ; q++) {

	std::memcpy(positionIndices, startingIndices, 64 * sizeof(uint8_t));

	int64_t combination = (1ull << N) - 1;
	int64_t index = 0;
	int64_t friendCount = N;

	while (friendCount) {

		index = (q + index) % friendCount--;

		combination -= (1ull << positionIndices[index]);

		uint8_t* indexToDelete = positionIndices + index;
		std::memcpy(indexToDelete, indexToDelete + 1, 64 * sizeof(uint8_t));

		combinationOccurances[combination] = 1;
	}
}
```

Slika 5.1.3.: Glavna petlja

Unutarnja petlja se izvršava sve dok ima ljudi u krugu. Prvo se pomoću modulo operacije smanji gdje bi *q* trebao ukloniti osobu te se smanji broj osoba u krugu. U sljedećoj liniji se gasi odgovarajući bit u vrijednosti koja prati kombinaciju. Gašenje bitova se radi tako da se uzme indeks osobe na poziciji koju treba ukloniti te se taj bit gasi. Gašenje bitova se izvodi tako da se uzme vrijednost 1 te se pomakne određen broj bitova u lijevo te se ta vrijednost oduzme od niza koji prati gdje se tko nalazi. Oduzimanje radi točno jer se ista pozicija ne može ponoviti više puta, tj. ne može se isti bit više puta oduzeti. Nakon toga se opet kopiraju vrijednosti, ali se u ovom slučaju svi indeksi iznad onog koji se uklanja prekopiraju za jedno mjesto dolje tako da se efektivno ukloni traženi indeks. Na kraju zahvaljujući činjenici da vrijednost koja prati kombinaciju istovremeno pokazuje gdje se ona točno nalazi u nizu rezultata, može se jednostavno označiti da je upravo ta kombinacija pronađena. Varijabla „combinationOccurances“ se predaje funkciji kao argument zajedno s varijablom „startQ“. Pogodno je funkciji dati odakle da počne jer kad bude više niti onda se samo izmjeni gdje koja nit počinje računati.

Tablica 5.1.1.: Prikaz kako se uklanjaju osobe za n = 5 i q = 9.

		Varijable			
		combination	index	friendCount	positionIndices
Vrijednosti na početku		31_{10} (11111_2)	0	5	{4,3,2,1,0}
Prolazak petljom	1	30_{10} (11110_2)	4	4	{4,3,2,1}
	2	22_{10} (10110_2)	1	3	{4,2,1}
	3	18_{10} (10010_2)	1	2	{4,1}
	4	2_{10} (00010_2)	0	1	{1}
	5	0_{10} (00000_2)	0	0	{}

U prethodnoj tablici je demonstrirano kako algoritam radi za stvarni primjer te se može vizualizirati kako algoritam točno funkcionira. Na početku su prisutne sve osobe (11111) te je indeks uklanjanja jednak 0. Prvo uklanjanje promijeni indeks na 4 te se uzme vrijednost na tom indeksu iz „positionIndices“ (0). Ta vrijednost se koristi da bi se ugasio taj bit u „combination“ te nakon gašenja bita se „positionIndices“ prekopira tako da se ukloni iskorištena vrijednost. Drugo uklanjanje mijenja indeks na 1 te uzima drugu vrijednost iz polja „positionIndices“ (3). Kao u prethodnom koraku ta vrijednost se koristi za gašenje pripadnog bita te se polje opet kopira da se ukloni ta vrijednost. Postupak se ponavlja onoliko puta koliko ima bitova. Jedna stvar za primijetiti je da je ovaj primjer identičan objašnjenom u poglavlju 3 te za treći prolazak petljom skup preživjelih je identičan. Jedina razlika je što je *q* veći za jedan jer se počinje od 0, a ne od 1.

Ostatak koda koji pokreće funkciju se nalazi u glavnoj funkciji „main“. Programski kod je zadužen za kreiranje polja gdje se spremaju rezultati te praćenje koliko vremena treba da se izvrši funkcija. Broj nemogućih situacija je samo broj mjesta u polju koji su jednaki 0.

```
int64_t combinationCount = 1ull << N;
uint8_t* combinationOccurances = new uint8_t[combinationCount]();

auto t1 = std::chrono::steady_clock::now();
josephus(combinationOccurances, 0);
auto t2 = std::chrono::steady_clock::now();

int64_t impossibleCombinationCount = 0;
for (int64_t i = 0; i < combinationCount - 1; i++) {
	impossibleCombinationCount += !combinationOccurances[i];
}
delete[] combinationOccurances;

printf("n is %2d,   %5lld occurances,   %10.3f seconds\n", N,
impossibleCombinationCount, std::chrono::duration_cast<std::chrono::microseconds>(t2 -
t1).count() / 1000000.0);
```

Slika 5.1.4.: Pokretački kod u „main“ funkciji

Nakon što je napravljen kod potrebno ga je paralelizirati da bi ga se ubrzalo što više moguće. Zahvaljujući osnovnoj verziji lagano je za primijetiti da petlja po kojoj se pretražuju rješenja nema ovisnost o prethodnim rezultatima što je savršeno za paralelizaciju. Petlja ide po vrijednostima broja *q* tako da ih preskače za broj niti. Ne postoji garancija da će sve niti imati različite kombinacije tijekom rada što znači da će potencijalno biti duplikata tijekom rada. Duplikati nisu problem jer je samo bitno označiti da se našlo rješenje, a ne koje rješenje.

Za paralelizaciju je potrebno koristiti dodatne niti. Niti su poput dodatnih procesa koji rade neki zadatak. Na isti način kao što je „main“ funkcija koja se izvodi u glavnom programu, svaka nit dobije neku funkciju za izvoditi. Bitno je za spomenuti da su niti asinkrone s obzirom na glavni program što znači da rade nezavisno te u modernim procesorima se izvršavaju na drugim procesorskim jezgrama. U jeziku C++ one dolaze iz zaglavlja „thread“. Izmjena koda koji računa rješenja je samo u petlji. Petlja se izmjeni na sljedeći način:

```
for (int64_t q = startQ; q <= maxQ; q += THREAD_COUNT)
```

Slika 5.1.5.: Izmijenjena glavna petlja funkcije

U ovom načinu rada „startQ“ će biti inkrementiran od 0 za svaku nit te će nova pretprocesorska naredba „THREAD_COUNT“ biti zadužena da svaka nit „preskače“ ostale.

```
int64_t combinationCount = 1ull << N;
uint8_t* combinationOccurances = new uint8_t[combinationCount]();

std::thread* threads = new std::thread[THREAD_COUNT]();
for (int64_t i = 0; i < THREAD_COUNT; i++) {
	threads[i] = std::thread(josephus, combinationOccurances, i);
}

auto t1 = std::chrono::steady_clock::now();
for (int64_t i = 0; i < THREAD_COUNT; i++) {
	threads[i].join();
}
auto t2 = std::chrono::steady_clock::now();

int64_t impossibleCombinationCount = 0;
for (int64_t i = 0; i < combinationCount - 1; i++) {
	impossibleCombinationCount += !combinationOccurances[i];
}

delete[] threads;
delete[] combinationOccurances;

printf("n is %2d,   %5lld occurances,   %10.3f seconds\n", N,
impossibleCombinationCount, std::chrono::duration_cast<std::chrono::microseconds>(t2 -
t1).count() / 1000000.0);
```

Slika 5.1.6.: Pokretački kod s podrškom za više niti

Prethodno navedena naredba zamjenjuje izraz „q++“ s „q += THREAD_COUNT“. Time se posao ravnomjerno raspodjeli na broj niti određen pretprocesorskom naredbom. Veća, znatna, promjena koda je u funkciji „main“ gdje je potrebno upravljati nitima.

Izmijenjen kod kreira niti te ih sve pokrene sa svojim počecima. Npr. ako kod radi na 4 niti, one će dobiti sljedeće početke: 0, 1, 2, 3. Nakon što svaka nit završi sa svojim q-om, q se poveća za broj niti na: 4, 5, 6, 7, nakon toga se poveća na: 8, 9, 10, 11 i tako sve do gornje granice. Tim postupkom se zadatak podijelio te nema preskakanja kombinacija. Zatim je potrebno za svaku nit pozvati funkciju „join“ koja natjera glavni program da pričeka da ta nit završi sa svojim poslom. Na taj način prolaskom kroz sve niti će se pričekati da one i završe. Ostatak je nepromijenjen osim brisanja memorije koju su niti koristile.

Sa gotovim kodom se može za razne vrijednosti broja n izračunati koliko i gdje ima ne-Josipovih skupova te naravno, može se izmjeriti vrijeme koje je bilo potrebno da se izračuna. Računanje pomoću kojeg će se određivati učinkovitost algoritama će se izvoditi na vrijednostima $n \in [20, 27]$ jer za vrijednosti ispod 20 vrijeme izvođenja bude manje od jedne sekunde te bude suviše osjetljivo na promjene, a vrijednosti iznad 27 traju previše za obično skupljanje podataka.

Računalo na kojem su rezultati dobiveni ima procesor s 6 fizičkih jezgri i 12 virtualnih. Iz tog razloga će broj niti biti 1, 2, 4, 6, 8 i 12 jer ti brojevi niti se mogu dobro rasporediti na virtualnim jezgrama procesora. Unosi u tablici su vrijeme u sekundama potrebno da se izračunaju skupovi za određeni n i broj niti.

Tablica 5.1.2.: Vrijeme potrebno za izračunati skupove te broj ne-Josipovih skupova.

		Broj niti						Broj pronađenih ne-Josipovih skupova
		1	2	4	6	8	12	
n	20	1.14	0.89	0.80	0.61	0.51	0.45	238
	21	2.85	2.51	1.87	1.41	1.11	0.92	794
	22	28.81	6.61	4.57	3.22	2.54	2.10	0
	23	29.99	19.41	11.31	8.42	6.53	5.24	0
	24	80.08	52.64	28.52	21.465	16.87	13.13	308
	25	212.19	128.57	73.73	54.664	42.95	34.47	545
	26	700.03	284.61	178.46	132.84	106.39	87.06	0
	27	1516.38	847.66	487.32	309.19	256.50	215.50	270

5.2. Optimiziranje pristupa memoriji

Unutar glavne petlje programa jedna od najbitnijih operacija je uklanjanje indeksa osobe iz polja. Ta operacija se izvodi s „memcpy“ funkcijom, no može se optimizirati još više jer ta funkcija ima dodatnih skrivenih provjera koje usporavaju glavnu petlju. Može se djelomično optimizirati ručno tako da se napiše funkcija za to, no postoji bolje rješenje, AVX2 instrukcije [8].

AVX2 instrukcije su posebne instrukcije koje se nalaze na nekim modernim procesorima, uključujući na procesoru na kojem je program napisan. Odlika tih instrukcija je SIMD princip koji glasi da se jedna instrukcija primjenjuje istovremeno na više komadića podataka. Instrukcije u AVX2 skupu instrukcija su razne, od zbrajanja i oduzimanja, do permutacija i raznih matematičkih operacija. AVX2 instrukcije također imaju vrlo velike registre, duljina 256 bita. Normalni registri su dugački 64 bita te mogu držati u sebi jednu vrijednost. AVX2 registri su posebno dizajnirani tako da u jednom registru mogu biti 4 vrijednosti duljine 64 bita ili 8 vrijednosti od 32 bita ili 16 vrijednosti od 16 bitova ili 32 vrijednosti od 8 bitova duljine. U poboljšanoj verziji programa će se koristiti instrukcije za pomicanje podataka kada se uklanjaju indeksi iz polja. Pomicanje podataka će obavljati sljedeće dvije funkcije: „*_mm256_loadu_si256“* i „*_mm256_storeu_si256“*. Prva funkcija se koristi da bi se iz neke memorijske lokacije učitalo 256 bita informacija ili

jednostavnije 32 bajta što je u programu 32 indeksa odjednom dok je druga suprotna, ona samo spremi 32 bajta informacija na neku lokaciju. Primjenom te dvije funkcije se samo treba izmijeniti dio koda koji računa rješenja.

```
// Prije petlje
__m256i _originalIndicesPart1 = _mm256_loadu_si256((__m256i*)(startingIndices +  0));
__m256i _originalIndicesPart2 = _mm256_loadu_si256((__m256i*)(startingIndices + 32));

// Zamjenjuje prvi memcpy poziv
_mm256_storeu_si256((__m256i*)(positionIndices +  0), _originalIndicesPart1);
_mm256_storeu_si256((__m256i*)(positionIndices + 32), _originalIndicesPart2);

// Zamjenjuje drugi memcpy poziv
uint8_t* indexToDelete = positionIndices + index;
__m256i _indexDataPart1 = _mm256_loadu_si256((__m256i*)(indexToDelete +  1));
__m256i _indexDataPart2 = _mm256_loadu_si256((__m256i*)(indexToDelete + 33));
_mm256_storeu_si256((__m256i*)(indexToDelete +  0), _indexDataPart1);
_mm256_storeu_si256((__m256i*)(indexToDelete + 32), _indexDataPart2);
```

Slika 5.2.1.: Optimizirani „memcpy“ pozivi pomoći AVX2 instukcija

U novom kodu prve dvije linije se nalaze ispred prve petlje te učitavaju sve početne indekse u dvije varijable tipa „*__m256i*“ koje predstavljaju prethodno navedene registre. Nakon početka petlje umjesto prvog poziva funkcije „memcpy“ koriste se funkcije za spremanje vrijednosti u polje. Također umjesto drugog poziva funkcije „memcpy“ prvo se učitaju vrijednosti koje treba pomaknuti te se odmah spreme jedan bajt niže čime se ukloni osoba na nekom indeksu, poput „memcpy“, samo učinkovitije.

Tablica 5.2.1.: Vrijeme potrebno za optimiziran pristup memoriji.

		Broj niti						Broj skupova
		1	2	4	6	8	12	
n	20	1.04	0.68	0.79	0.57	0.47	0.39	238
	21	2.82	2.66	1.81	1.30	1.05	0.88	794
	22	8.72	6.85	4.15	2.94	2.47	1.96	0
	23	28.32	19.06	11.20	7.88	6.15	4.89	0
	24	76.69	45.46	27.59	20.08	15.77	12.52	308
	25	190.32	118.58	69.39	49.89	40.43	33.609	545
	26	575.08	281.43	167.41	124.16	100.94	83.57	0
	27	1369.70	856.39	462.11	289.46	242.31	210.41	270

Kada se uzme prosjek svih rezultata spram ne optimizirane verzije dobije se prosječno poboljšanje od 6.44 %. Poboljšanje nije suviše veliko, no nije za zanemariti.

5.3. Optimiziranje modulo operacije

Modulo operacija je od iznimne važnosti u algoritmu jer ona je zaslužna za određivanje pravih pozicija za ukloniti, ali ima vrlo velik i znatan problem. U modernim računalima kada se traži dijeljenje dva cijela broja dobije se cjelobrojni rezultat te usput se dobije i ostatak. Kada se radi modulo operacija, procesor zapravo radi dijeljenje, no samo uzme ostatak te odbaci rezultat dijeljenja. Dijeljenje dva broja je i na modernim računalima vrlo skupa operacija jer se ne može izvesti učinkovito. Zbrajanje i oduzimanje brojeva duljina 64 bita je vrlo brzo, 1 takt, množenje je nešto sporije s 3 takta, no to nadoknadi s time da se svaki takt može novo množenje pokrenuti pod uvjetom da ne ovisi o prethodnim množenjima. Međutim, dijeljenje, u slučaju računala na kojem se kod izvršava, nema stalno trajanje nego ovisi u ulazima te traje između 13 i 44 procesorskih taktova te se ne može više dijeljenja istovremeno izvršavati. S obzirom da vrijednosti za *q* postaju veće s vremenom, a broj koji ga dijeli je uvijek malen može se uzeti da je trajanje uvijek preko 35 taktova. Optimizacija modulo operacije nije uvijek moguća, no u ovom slučaju, jer se zna točno s čim će se dijeliti, moguće je optimizirati.

Optimizacija se zasniva na tome da dijeljenje dva broja se može zapisati kao množenje s recipročnom vrijednosti broja: $\frac{5}{3} = 5 * \frac{1}{3} = 5 * 0.\dot{3} = 1.\dot{6}$. Množenjem ta dva broja se dobije isti rezultat, no operacija je bila brža jer je bilo množenje, a ne dijeljenje.

Ako se uzme najveći binarni broj koji stane u cjelobrojnu varijablu od 64 bita te se taj broj tretira kao da je točka koja razdvaja cijeli i razlomački dio na lijevoj strani broja, a ne na desnoj, onda taj broj predstavlja, što preciznije moguće, broj 1. Npr. Uzimajući broj od 8 bitova: $11111111.00000000_2 = 255_{10}$ te tretirajući ga kao što je prethodno navedeno dobije se: $0000000.11111111_2 = 0.99609375_{10}$. Taj broj će se zvati M_0. Kada se M_0 podijeli s djeliteljem, gore u primjeru 3, dobije se približno $\frac{1}{3}$ te je zbog nepreciznosti potrebno dodati 1 na M_0 da se djelomično popravi preciznost. Takav „popravljen“ broj će se zvati *M*. Množenjem *M* s djeljenikom se dobije broj *L* koji predstavlja rezultat početnog dijeljenja, ali predstavlja dio s desne strane decimalne točke što je vrlo bitno. Množenje broja *L* s djeliteljem proizvodi 128 bitova

rezultata, gornjih 64 bita su dio iznad decimalne točke, a donjih 64 su ispod decimalne točke. Ostatak početnog dijeljenja je u gornjih 64 bita rezultata [9].

Kada se taj postupak primjeni na primjeru iznad dobije se sljedeće:

$$N = FFFFFFFFFFFFFFFF_{16} = 18446744073709551615_{10}$$
$$M = \frac{N}{3} + 1 = 5555555555555556_{16} = 0.3333333333333333_{10}$$
$$L = 5 * M = AAAAAAAAAAAAAAAF_{16} = 0.6666666666666666_{10}$$
$$R = (L * 3) \gg 64 = 2_{16} = 2_{10}$$

Izmjene u kodu s obzirom na operaciju nisu toliko znatne. „Magični“ brojevi (M) se računaju prije početka petlje te se spreme u polje da bi se koristili kasnije. Kod je jednostavan, prvo se odredi gdje bi trebalo ukloniti sljedeću osobu te se pomnoži s „magičnim“ brojem. Nije potrebno uklanjati gornjih 64 bita jer obično množenje ih svejedno ukloni. Nakon toga se poziva funkcija „*__umulh*“ koja izračuna samo gornja 64 bita rezultata koji predstavljaju ostatak te istovremeno sljedeći indeks.

```
// Prije petlje
int64_t magicNumbers[64];
for (int64_t i = 1; i < 64; i++) {
	magicNumbers[i] = 0xFFFFFFFFFFFFFFFF / i + 1;
}

// Ovo se zamjeni
index = (q + index) % friendCount--;

// S Ovim
int64_t lowBits = (q + index) * magicNumbers[friendCount];
index = __umulh(lowBits, friendCount);
friendCount--;
```

Slika 5.3.1.: Izmjene potrebne za optimizirani modulo

Iako bi takva metoda sama po sebi poboljšala brzinu za oko 40%, kod je također izmijenjen na način da se koriste AVX2 instrukcije za prebacivanje memorije, optimizirani modulo te „odmotavanje“ petlje tako da svaki prolazak petljom računa 4 slučaja istovremeno što sveukupno znatno ubrza program, no kod veoma je dug i nečitak zbog odmotavanja petlje te neće biti prisutan u radu, samo rezultati. Razlog zašto se petlja odmotava je zato što u uskoj unutarnjoj petlji

uzimanje magičnih brojeva iz L1 predmemorije je relativno sporo za samo jedan broj. Optimizacija je znatna s 61.68% bržim vremenom izvođenja te je vrijedna zamršenosti koda.

Tablica 5.3.1.: Vrijeme potrebno pomoću AVX2, optimizacije modulo-a i odmotavanja petlje.

		Broj niti						Broj skupova
		1	2	4	6	8	12	
n	20	0.55	0.59	0.49	0.36	0.30	0.27	238
	21	1.85	1.59	1.10	0.81	0.66	0.58	794
	22	5.57	3.41	2.52	1.90	1.50	1.34	0
	23	17.37	11.41	6.69	5.10	4.21	3.50	0
	24	47.75	29.67	17.54	13.38	11.46	10.54	308
	25	118.37	74.89	47.79	35.87	30.76	30.85	545
	26	287.62	194.16	132.95	88.41	78.89	76.09	0
	27	701.83	459.59	269.45	216.51	204.00	183.79	270

6. REZULTATI

Iz tablica rezultata se može primijetiti trend da se povećanjem broja niti smanjuje potrebno vrijeme, no nije jasno vidljivo koliko zapravo. Jedna stvar koja se može primijetiti izravno je da s povećanjem vrijednosti broja *n* raste vrijeme potrebno za izračun za neki faktor između 2 i 3. Kada se uzmu sve vrijednosti u obzir dobiju se sljedeći faktori:

Tablica 6.1: Faktori rasta implementacija te njihov prosjek.

	Implementacija 1	Implementacija 2	Implementacija 3
Faktor rasta	2.550054	2.575294	2.581692
Prosječan faktor rasta	2.569013		

Iz tablice 6.1 se može vidjeti da je faktor rasta sličan procijenjenoj kompleksnosti algoritma. Razlog zašto je oko 2.5 je činjenica da s rastom broja osoba se unutrašnja petlja koja ih uklanja izvodi više puta te ona pravi tu dodatnu razliku. Jedna od posljedica takvog rasta je da je

svaki sljedeći broj toliko puta „teži“ za izračunat te traje toliko duže. Ako se pretpostavi da je rast isti te da se koristi treća implementacija s 12 niti onda se može pretpostaviti kakva će biti vremena izvođenja za veće *n*. Ako se koristi treća implementacija s 16 niti na boljem računalu s 8 fizičkih i 8 virtualnih jezgri onda se dobije za $n = 27$ vrijeme od 102.38 sekundi spram prvog računala. Koristeći taj rezultat dobije se drugi stupac u tablici pretpostavljenih vremena izvođenja.

Tablica 6.1: Pretpostavljena vremena izvođenja.

n	Pretpostavljena vremena	
	Računalo 1	Računalo 2
30	~ 0.86 sati	~ 0.48 sati
35	~ 4 dana	~ 2.24 dana
40	~ 1.23 godina	~ 0.68 godina
45	~ 1.38 stoljeća	~ 0.77 stoljeća
50	~ 15.5 tisućljeća	~ 8.6 tisućljeća
55	~ 1.7 milijuna godina	~ 0.96 milijuna godina
60	~ 194 milijuna godina	~ 100 milijuna godina

Zahvaljujući visokoj kompleksnosti može se iz ekstrapoliranih podataka vidjeti da bi vremena izvođenja postala nepodnošljiva vrlo brzo te da isplativost algoritma znatno opadne. Već oko 40 postane nepraktično osim na superračunalima gdje bi se posao mogao podijeliti na mnogo procesorskih jezgri. Pretpostavljena vremena na drugom, bržem računalu, su bolja, ali spram kompleksnosti problema, od malenog su značaja.

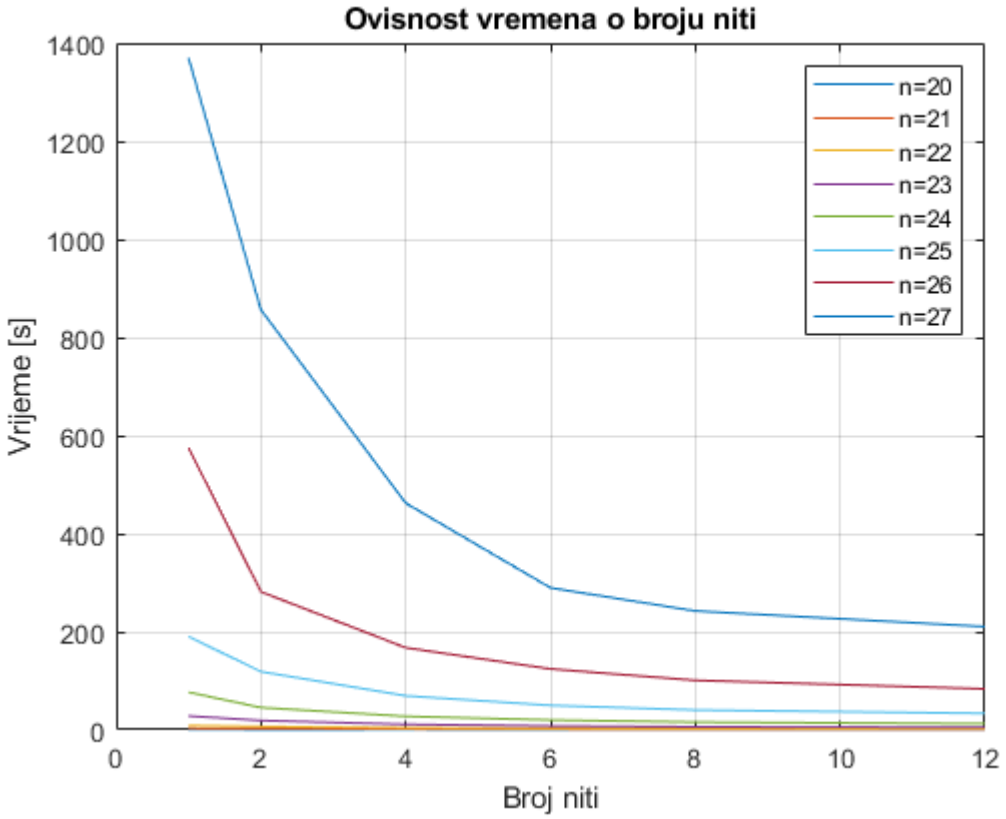
Cilj rada je bio kreirati paralelan program koji računa ne-Josipove skupove te je naglasak bio na implementaciji programa, no kada se program primijeni na više vrijednosti broja *n* onda se dobiju sljedeći rezultati za ne-Josipove skupove (redovi gdje ima 0 skupova nisu prikazani):

n	Broj ne-Josipovih skupova
9	3
12	13
15	26
18	54
20	238
21	794
24	308
25	545
27	270
28	114
30	5539
33	605

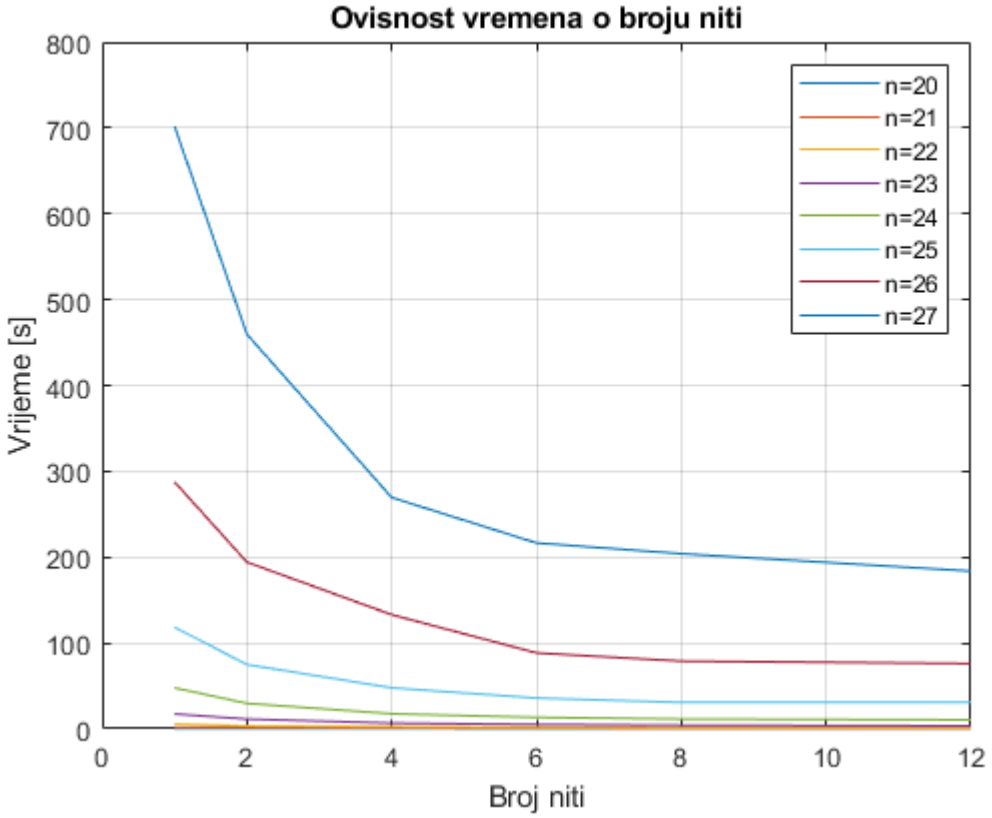
Tablica 6.2: Broj ne-Josipovih skupova do n = 33

Ako bi se zapisali rezultati uključujući one gdje je broj skupova jednak 0 onda se dobije niz brojeva, ekvivalentan nizu koji se može pronaći na poveznici u popisu izvora [10]. Niz brojeva na poveznici ide do $n = 26$ što znači da su rezultati poslije 26 potpuno novi rezultati za ovaj problem.

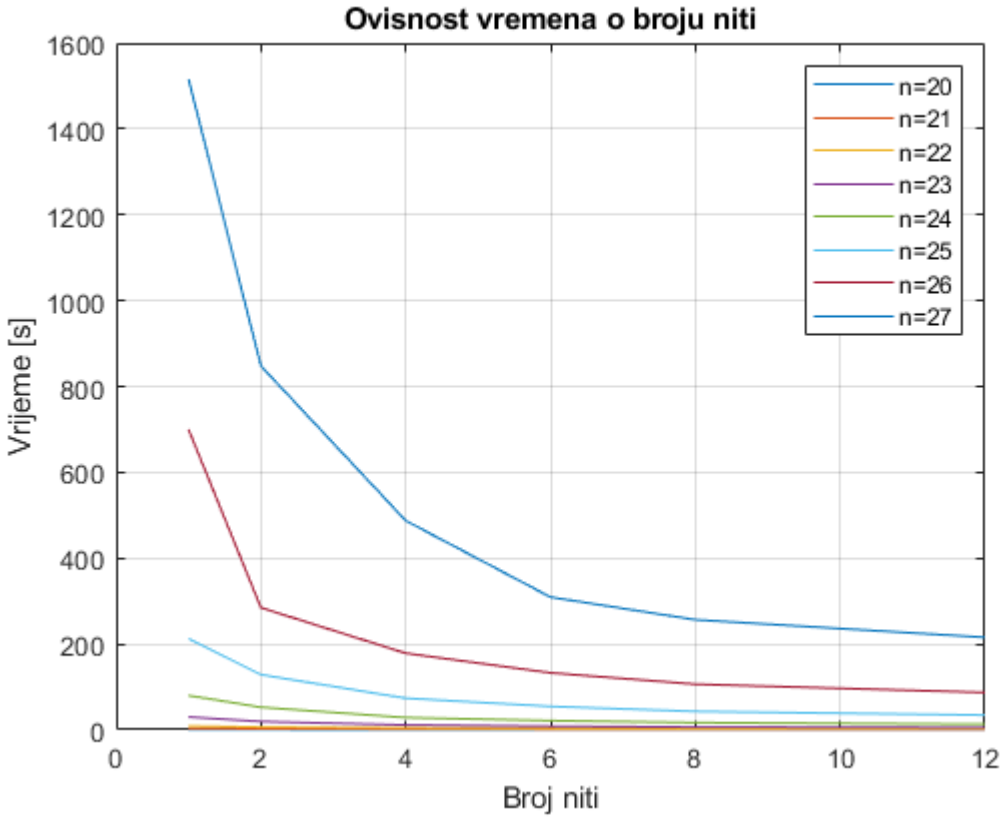
Na sljedeće dvije stranica slike 6.1, 6.2 i 6.3 su grafovi podataka iz tablica 5.1.2, 5.2.1 i 5.3,1, a slika 6.4 predstavlja slučaj kada je $n = 27$ te su sve tri implementacije izravno uspoređene. Na slikama 6.1, 6.2 i 6.3 se može primijetiti da s porastom broja niti vrijeme opada, ali s porastom broja *n* vrijeme se povećava.



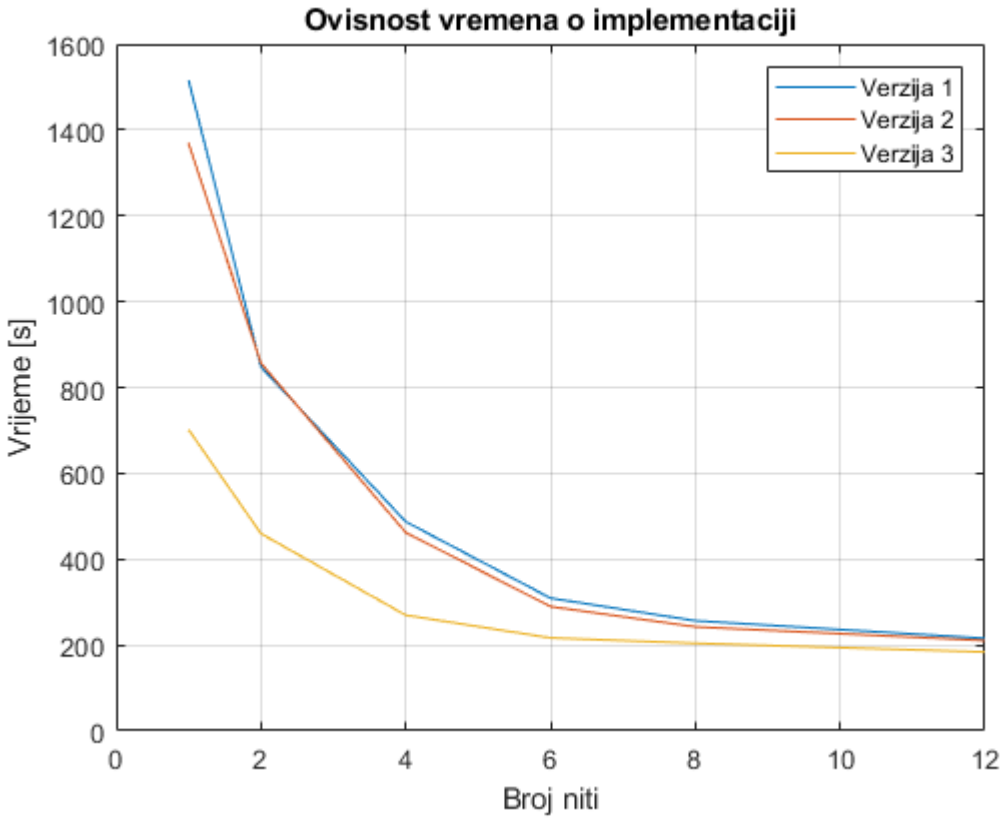
Slika 6.1: Graf rezultata prve implementacije.



Slika 6.2: Graf rezultata druge implementacije.



Slika 6.3: Graf rezultata treće implementacije.



Slika 6.4: Usporedba performansi sve tri verzije programa.

7. ZAKLJUČAK

Josipov problem je veoma zanimljiv problem iz perspektive matematike i programiranja. Problem zahtijeva modulo aritmetiku da se odredi redoslijed uklanjanja iz kruga osoba. Rješenja postaju kompleksnija što se problem više generalizira. Generalizacija problema kada se traži zadnja preživjela osoba nakon uklanjana ostalih s korakom uklanjanja q se pokaže da je jednostavna za opisati, no nema izravne jednadžbe. Povećavanjem broja preživjelih osoba se pokaže da postoje nemoguće situacije. Pronalaženje takvih situacija se pokazalo da je kompleksno i skupo u smislu vremena izvođenja.

Algoritam pronalaženja ne-Josipovih skupova je jednostavan i učinkovit, no ima vrlo visoku kompleksnost zbog prirode problema. Paralelizacija uspije poboljšati vrijeme izvođenja jer je lagano za paralelizirati. Optimizacija prebacivanja memorije pomoću AVX2 instrukcija i optimizacija modulo operacije uspiju zajedno ubrzati program za preko 60%.

Analizom podataka se vidi da je svaka sljedeća vrijednost za n 250% zahtjevnija za izračunati nego prethodna što se može pročitati iz grafova vremena. Korištenje više niti ubrza program proporcionalno broju fizičkih jezgri na računalu te omogući traženje većih rezultata.

Nije poznato da li je moguće izračunati izravno da li je neka kombinacija moguća jer svaki korak ovisi o prethodnom te uvijek ima 2^n slučajeva što ograničava brzinu algoritma. Potencijalno se može još ubrzati program na način da se koriste AVX2 paralelne instrukcije da se radi 4 puta više kombinacija odjednom. Moguće je organizirati posao na grafičkoj kartici, no i dalje je ograničenje ukupna količina memorije. Ako bi se uspjelo ukloniti ograničenje memorije bez da se poveća vrijeme, algoritam bi bio mnogo brži.

LITERATURA

1. Flavius Josephus. The Complete Works Of Flavius Josephus. Andesite Press (8 Aug. 2015)
2. Theriault, Nicolas (2001). "Generilazations of the Josephus Problem"
3. Sullivan, Shaun; Insko, Erik (2018). "A variant on the Feline Josephus Problem"
4. Ruskey, Frank; Williams, Aaron (2010). "The Feline Josephus Problem"
5. https://www.codingninjas.com/blog/2021/09/11/special-case-of-josephus-problem-when-k2 (zadnji pristup Kolovoz 2021)
6. R. Graham, D. Knuth, O. Patashnik. Concrete Mathematics. Addison Wesley; 2nd edition (1994)
7. https://oeis.org/A006257 (zadnji pristup Kolovoz 2021)
8. https://software.intel.com/sites/landingpage/IntrinsicsGuide (zadnji pristup Kolovoz 2021)
9. https://lemire.me/blog/2019/02/08/faster-remainders-when-the-divisor-is-a-constant-beating-compilers-and-libdivide (zadnji pristup Kolovoz 2021)
10. https://oeis.org/A208848 (zadnji pristup Kolovoz 2021)

SAŽETAK

Josipov problem je problem u kojem *n* osoba stoji u krugu. Cilj je prolaziti kroz krug i uklanjati svaku drugu osobu sve dok ne preostane samo jedna. Moguće je izvesti formulu pomoću koje se može iz broja *n* odrediti koja osoba treba preživjeti.

Poopćavanjem problema se dobije nova inačica problema takva da se traže *q*-ovi, a ne preživjele osobe. Traženjem rješenja poopćenog problema se dođe do spoznaje da određeni slučaji nisu mogući. Traženje tih nemogućih slučajeva je cilj implementiranog programa.

Analiza algoritma koji traži nemoguće slučaje je bitna jer implementacija algoritma ovisi o njoj. Zbog velike kompleksnosti određena žrtvovanja su potrebna po pitanju vremena izvođenja i zauzete memorije.

Implementacija je napravljena u programskom jeziku C++ zbog performansi i lakoće korištenja više niti. Radi ubrzavanja programa, implementiraju se dvije optimizacije, jedna vezana za kopiranje memorije, a druga vezana za modulo operaciju. Obje optimizacije zajedno ubrzavaju program za 60%.

Ključne riječi: C++, optimizacija, Josipov problem, višenitnost, memorija

Parallel program for the Josephus problem in C++

Abstract

The Josephus problem is a problem in which *n* people are standing in a circle. The goal is to go through the circle and eliminate every second person until only one remains. It is possible to produce a formula that is capable of calculating the position of the survivor from the number of people.

By generalizing the problem a new form of it is created such that the solutions are *q*-s and not the surviving people. Searching for solutions of the generalized problem reveals the fact that some combinations are impossible. The searching of those combinations is the goal of the implemented program.

Analysis of the algorithm which searches for the impossible cases is important because the implementation depends on it. Due to its high complexity, some sacrifices are necessary with respect to memory and execution time.

The implementation is made in C++ because of performance and the ease of use of threads. To speed up the program, 2 optimizations are made, one to speed up memory transfers and the other to optimize the modulo operation. Together the optimizations speed up the program by 60%.

Keywords: C++, optimization, Josephus problem, multithreading, memory