



Univerzitet u Istočnom Sarajevu  
Elektrotehnički fakultet



# Paralelizacija Floyd – Warshall algoritma

Seminarski rad

Studije: Prvi ciklus

Odsjek: Računarstvo i informatika

Predmet: Paralelni računarski sistemi

Studenti:

Kristina Knežević, 2060

Tamara Elez, 2068

Mentor:

Doc. dr Nikola Davidović

Istočno Sarajevo, septembar 2024

## Sadržaj

Uvod.....	4
1. Paralelno računarstvo .....	5
1.1. Tipovi paralelizma.....	5
1.2. Strategije paralelnog programiranja .....	6
1.3. Prednosti paralelizacije.....	7
1.4. Nedostaci paralelizacije.....	8
1.5. Paralelni modeli.....	8
1.6. Višenitnost.....	9
1.6.1. Problemi višenitosti .....	12
2. Korišteni alati i tehnologije .....	13
2.1. Programski jezik C .....	13
2.1.1. Identifikatori .....	14
2.1.2. Promjenljive.....	14
2.1.3. Osnovni tipovi podataka .....	15
2.1.4. Operatori .....	16
2.1.5. Upravljačke strukture.....	18
2.1.6. Nizovi.....	20
2.1.7. Pokazivači .....	20
2.1.8. Biblioteke.....	20
2.1.9. Funkcije.....	21
2.2. OpenMP.....	23
2.3 Visual Studio 2022 .....	23
2.4 Konzolna aplikacija .....	24
3. Floyd – Warshall algoritam .....	26
3.1 Matematičko objašnjenje.....	27
3.2 Praktična primjena.....	28
4. Programsko rješenje .....	29
4.1 Opis aplikacije .....	29
4.2 Sekvencijalna implementacija algoritma.....	32
4.3 Paralelna implementacija algoritma .....	34
5. Testiranje .....	37
5.1 Prvo testiranje .....	37
5.2 Drugo testiranje .....	38

5.3	Analiza rezultata testiranja .....	39
	Zaključak.....	40
	Literatura.....	41
	Dodaci .....	42
	Popis slika .....	42
	Popis tabela .....	42
	Popis kodnih listinga.....	42

## Uvod

Sadašnje stanje razvoja integrisanih kola ukazuje da ubrzavanje mikroprocesora jednostavnim povećanjem radne frekvencije više neće biti moguće. Usljed ove činjenice, danas je razvoj mikroprocesora prije svega usmjeren na uvećanje broja procesorskih jezgara, time uvećavajući značaj paralelnog programiranja i konkurentne obrade podataka uopšte. Dok se ne desi neki veći prodor u industriji mikroelektronike, paralelizacija ostaje jedini način da se pristupi bilo kakvoj masovnoj obradi.

Paralelni računski sistemi predstavljaju jedan od ključnih aspekata modernog računarskog inženjeringa, omogućavajući ubrzavanje izvođenja kompleksnih algoritama putem istovremenog izvršavanja zadataka. Jedan od takvih algoritama je Floyd-Warshall-ov, koji se koristi za pronalaženje najkraćih puteva između svih parova čvorova u grafu. Ovaj algoritam ima široku primjenu u raznim oblastima, uključujući mrežne protokole, analizu socijalnih mreža, i optimizaciju logističkih mreža.

Floyd-Warshall algoritam u svom serijskom obliku ima vremensku složenost  $O(n^3)$  što ga čini računski intenzivnim i vremenski zahtjevnim za grafove sa velikim brojem čvorova. Paralelizacija ovog algoritma predstavlja pokušaj ka povećanju efikasnosti i smanjenju vremena izvršavanja.

U radu su istražene metode i tehnike paralelizacije Floyd-Warshall algoritma implementiranog u programskom jeziku C, koristeći OpenMP tehnologiju. Rad je strukturisan tako da su prvo razmotreni osnovni pojmovi i koncepti paralelizacije. Dat je opis tehnologija koje se koriste za implementaciju paralelnih računarskih sistema, uz fokus na OpenMP tehnologiji, koja omogućava jednostavnu i efikasnu paralelizaciju programa pisanih u C jeziku. Nakon toga, pružen je detaljan pregled algoritma, uključujući njegovu funkcionalnost i praktičnu primjenu. U nastavku rada detaljno je opisan proces paralelizacije Floyd-Warshall algoritma, uključujući analizu performansi i uporednu evaluaciju sekvencijalnog i paralelnog izvršavanja algoritma.

Cilj ovog istraživanja je pokazati kako se performanse algoritma mogu značajno poboljšati korištenjem paralelizacije, te pružiti smjernice za implementaciju sličnih rješenja u praksi.

## 1. Paralelno računarstvo

Paralelno računarstvo predstavlja brže rješavanje problema korišćenjem većeg broja procesora.

Kod paralelnog računarstva u užem smislu postoji dijeljena memorija između više procesora, dok kod distribuiranog računarstva svaki procesor poseduje svoju lokalnu memoriju, tj. svoj lokalni adresni prostor. Paralelni računar je računarski sistem sa više procesora koji podržava paralelno programiranje. Distribuiran računar je sistem sa distribuiranom memorijom u kome su elementi obrada povezan sa mrežom. Višejezgrani procesor je procesor koji sadrži više jedinica obrade (jezgara) na istom čipu.

Paralelni računari se mogu grubo klasifikovati prema nivou na kojem hardver podržava paralelizam: gdje višejezgrani i višeprocorski računari imaju više elemenata obrade unutar jedne mašine, a klasteri koriste više računara da rade na istom zadatku. Specijalizovane paralelne računarske arhitekture se ponekad koriste zajedno sa tradicionalnim procesorima, za ubrzavanje specifičnih zadataka.

Paralelno programiranje je programiranje u jeziku koji dozvoljava da se eksplicitno zada kako će se razlagati delovi izračunavanja, tj. kako će biti izvršeni konkurentno na različitim procesorima. Tako da možemo reći da je to oblik programiranja u kojem se zadaci podijele na manje podzadatke koje se izvršavaju istovremeno na više procesora ili jezgara unutar jednog računarskog sistema.

Najveća motivacija za razvoj paralelnih sistema predstavljaju tzv. Grand Challenge problemi. To su fundamentalni problemi nauke i inženjerstva, koji su kompleksni i njihovo rješavanje putem numeričkih simulacija zahtjeva izuzetno brze računare.

### 1.1. Tipovi paralelizma

Paralelna obrada postoji u više oblika:

- na nivou bita
- na nivou instrukcije
- paralelizam podataka
- funkcionalni paralelizam

Paralelizam na nivou bita se odnosi na mogućnost procesora da istovremeno obradi više bitova podataka u jednoj operaciji i ova forma paralelnog izvršavanja instrukcija je bazirana na povećanju dužine procesorskih riječi. Ako procesor mora izvršiti sabiranje dvaju brojeva koji se sastoje od 32 bita na 32-bitnom procesoru, to će biti obavljeno u jednom ciklusu. Ali ako se koristi 64-bitni procesor, on može obraditi dvije takve operacije u istom vremenu, efektivno udvostručujući brzinu.

Paralelizam na nivou instrukcija se postiže kada se više operacija izvodi u jednom ciklusu, što se radi ili njihovim izvršavanjem istovremeno ili korišćenjem praznina između dve uzastopne operacije koje se stvaraju zbog kašnjenja. Postoji jedna specifična nit izvršenja procesa i razlikuje se od konkurentnosti po tome što ona uključuje dodjeljivanje više niti jezgru CPU-a u strogoj alternaciji. Pojednosti o nitima će biti pojašnjene u narednim poljavljima rada.

Paralelizam podataka je tip paralelizma u kome nezavisni procesi primjenjuju istu operaciju na različite elemente skupa podataka. Svi taskovi mogu da se izvršavaju konkurentno. U višeprocorskom sistemu koji izvršava jedan skup instrukcija, paralelizam podataka se postiže kada svaki procesor obavlja isti zadatak na različitim distribuiranim podacima. U nekim situacijama, jedna izvršna nit kontroliše operacije nad svim podacima. U drugim, različite niti kontrolišu operaciju, ali izvršavaju isti kod.

Funkcionalni paralelizam je tip paralelizma u kome nezavisni podzadaci izvršavaju funkcije nad istim ili različitim elementima podataka. Stepent konkurentnosti je limitiran brojem konkurentnih podzadataka. Uobičajeni tip ovakvog paralelizma je *pipeline*, koji se sastoji od premještanja jednog skupa podataka kroz niz odvojenih zadataka gdje svaki zadatak može da se izvrši nezavisno od drugih.

## 1.2. Strategije paralelnog programiranja

U savremenom paralelnom računarstvu, prisutno je, u manjoj ili većoj mjeri, četiri praktična pristupa, i to:

- Proširenje kompajlera u smislu dodavanja mogućnosti da sekvencijalne programe automatski prevodi u paralelne
- Proširenje postojećeg jezika dodavanjem novih paralelnih operacija
- Dodavanje novog paralelnog sloja na postojeći sekvencijalni jezik
- Uvođenje potpuno novog jezika koji prirodno podržava paralelizam

Kod proširenja kompajlera, zadatak je modifikacija postojećeg kompajlera dodavanjem mogućnosti da automatski detektuje paralelizam u sekvencijalnim programima i da odatle proizvede paralelni izvršni fajl. Prednost ovakvog pristupa je ta da programeri ne bi morali da paralelizuju svoj kod, niti da uče paralelno programiranje, već samo da nastave da koriste jednostavnije sekvencijalne jezike, a da paralelizaciju ostave kompajleru i operativnom sistemu. Mane ovog pristupa su što, ukoliko je programer zakomplikovao kod, kompajler sa velikom vjerovatnoćom neće moći ni da pronađe potencijalni paralelizam. Dakle, ovaj pristup funkcioniše isključivo kod jednostavnih konstrukcija, petlji isl. Razvijeni su mnogi eksperimentalni kompajleri ovog tipa, ali nijedno rešenje nije produkcionog kvaliteta.

Najlakši, najjeftiniji i najpopularniji pristup paralelnim programiranju je putem proširenja postojećeg jezika jer zahtjeva samo razvoj biblioteke rutina. Svodi na dodavanje paralelnih funkcija u postojeći sekvencijalni jezik, uključujući funkcije za kreiranje i terminiranje paralelnih procesa, njihovu sinhronizaciju, međusobnu komunikaciju isl. Primjeri iz prakse su MPI, PVM, POSIX niti, OpenMP idr. Međutim, mana je u tome što kompajleri ne učestvuju u generisanju paralelnog koda, niti omogućavaju hvatanje grešaka. Zbog toga je otežano otklanjati greške, čak i u jednostavnim programima.

Jedan od navedenih načina je dodavanje paralelnog sloja sekvencijalnom jeziku. Paralelni program možemo da posmatramo kao da je dvoslojan. Donji sloj je jezgro u kome proces manipuliše sopstevnom porcijom podataka da bi proizveo svoju porciju rezultata. Ovaj sloj može da se implementira u postojećem sekvencijalnom programskom jeziku. Gornji sloj kontroliše kreiranje i sinhronizaciju procesa, kao i dijeljenje podataka među procesima. Paralelni sloj može da bude isprogramiran nekim paralelnim jezikom. Razvijeno je nekoliko istraživačkih prototipova, ali nijedan komercijalni sistem ovog tipa nije zaživio.

Posljednji pristup je razvijanje paralelnog programskog jezika. Primer je jezik Occam, sa potpuno novom sintaksom, koji podržava i paralelno i sekvencijalno izvršavanje procesa. Drugi način je dodavanje paralelnih konstrukcija u već postojeći programski jezik. Primeri su High Performance Fortran i C\* te kompajlersko rešenje CUDA kompanije nVidia, koje dodaje specijalne instrukcije za programiranje grafičkih procesora. Prednost se ogleda u tome što programer predočava paralelizam samom kompajleru, što povećava vjerovatnoću da će izvršni program dostići visoke performanse. Nedostatak je da se zahtjeva razvoj novih kompajlera. Proizvođačima su potrebne godine za razvoj kvalitetnog kompajlera za svoj paralelni sistem. Drugo, novi jezici možda i neće biti standardizovani i onda proizvođači odlučuju da ne prave kompajler za te jezike na svojim mašinama.

Dok se rad na razvijanju paralelizujućih kompajlera i paralelnih programskih jezika visokog nivoa nastavlja, u praksi najčešće korišten pristup ostaje upotreba postojećeg jezika sa paralelnim konstrukcijama niskog nivoa. MPI, pthreads i OpenMP dominiraju savremenim svijetom paralelnog računarstva. Dobija se prilično visoka efikasnost, kao i portabilnost koda, ali po cijenu nešto težeg kodiranja i otklanjanja grešaka.

### **1.3. Prednosti paralelizacije**

Tehnologija paralelizacije se danas primjenjuje u većini uređaja koje svakodnevno koristimo, a to su desktop računari, laptopi i pametni mobilni uređaji. Uređaji su opremljeni višezegrenim procesorima kako bi se postigle bolje performanse i efikasnost.

Prva i ključna prednost paralelizacije računarskih procesa je povećanje brzine izvršavanja. Pošto su performanse obrnuto srazmjerne vremenu izvršavanja, kraće vrijeme doprinosi uočljivo boljim performansama. Korišćenjem više procesora ili jezgara, veliki zadaci se mogu podijeliti na manje podzadatke koji se izvršavaju istovremeno, smanjujući ukupno vrijeme obrade. Na taj način se efikasnije upotrebljavaju resursi, smanjujući neaktivno vrijeme procesora, čime je povećana propusnost kao mjera izvršenog posla u jedinici vremena. Istovremeno obavljanje više zadataka poboljšava odziv sistema i korisničko iskustvo. Ovo je naročito izraženo u aplikacijama koje zahtijevaju obradu u realnom vremenu (npr. multimedijalne aplikacije za obradu videa). Moderne video igre sa visokokvalitetnom 3D grafikom, koriste paralelizaciju putem GPU-a za obradu piksela i renderovanje složenih scena u realnom vremenu. Paralelna obrada značajno poboljšava fluidnost rada takvih aplikacija.

Za slučajeve kada se od aplikacija zahtjeva obrađivanje velike količine podataka (npr. analize podataka, simulacije), paralelizacija omogućava da se isti algoritmi izvode na više grupa podataka istovremeno. Pojedinačni rezultati se po završetku paralelne obrade kombinuju u konačni rezultat seta podataka. Konkretni primjer je aplikacija Google Maps. Kada korisnik traži rutu, Google Maps mora obraditi ogromne setove geografskih podataka, generisati rute, i ažurirati mape u realnom vremenu. Prilikom izračunavanja najbrže rute između tačaka, Google Maps koristi algoritme za pretragu grafova koji mogu biti paralelizovani kako bi se brzo obradio veliki broj mogućih ruta. Sve te operacije koriste paralelizaciju kako bi se podaci brzo obradili, jer se različiti dijelovi mape ili različiti koraci pretrage mogu istovremeno izvršavati na više servera.

Paralelni sistemi su skalabilni – veći broj procesora ili mašina može se koristiti za rješavanje većih problema ili brže izvršavanje zadataka. To omogućava da paralelizacija bude efikasna čak i u okruženjima sa velikim brojem korisnika ili zadataka. Određeni dugotrajni zadaci koji su u

sekvencijalnom okruženju trajali po nekoliko sati se nakon adekvatnog poboljšanja putem paralelizacije izvršavaju u kraćem vremenskom intervalu reda sekundi ili minuta.

Značajno je spomenuti energetske efikasnosti paralelnih sistema. Umjesto da jedno jezgro radi na visokom opterećenju, više jezgara može da obavi zadatak na nižem nivou opterećenja. Na taj način se može smanjiti potrošnja energije i produžiti životni vijek hardverskih komponenti. Takođe, neki višezegarni procesori su dizajnirani da budu energetski efikasniji i preuzimaju manje zahtjevne zadatke, dok se brža jezgra koriste za intenzivnije zadatke.

#### **1.4. Nedostaci paralelizacije**

Iako paralelizacija donosi značajne prednosti u računarstvu, neophodno je uzeti u obzir i negativne karakteristike ovog pristupa i poteškoće koje unosi.

Za programere je paralelno programiranje složenije od sekvencijalnog, obzirom da moraju obratiti pažnju na dodatne stvari kao što je pravilno dijeljenje zadatka na podzadatke, upravljanje komunikacijom među nitima ili procesima i osiguravanje sinhronizacije njihovog rada. Može se zaključiti da su paralelni programi daleko složeniji za pisanje od sekvencijalnih. Greške poput "race conditions" i zaglavljenja, koje će detaljnije biti objašnjene kasnije u radu, su češće i teže za otkrivanje, a potom i otklanjanje.

Proces sinhronizacije niti ili procesa može stvoriti dodatni "overhead", odnosno opterećenje koje stvara kontraefekat i dovodi do usporavanja rada. Zbog toga, kada više niti pristupa zajedničkim resursima, potrebno je pravilno implementirati sinhronizaciju da bi se izbjegle greške koje smanjuju pozitivno dejstvo paralelizacije.

Paralelizacija nije uvek isplativa, za manje i kratkotrajne zadatke može da izazove više problema nego što donosi koristi. Neki algoritmi ili problemi su inherentno sekvencijalni i ne mogu se efikasno podijeliti na paralelne zadatke. U ovakvim situacijama paralelna implementacija ne daje bolji rezultat. Tada se pokušava naći jedan dio problema za koji bi paralelna obrada bila korisna, obzirom je bezvrijedan pokušaj paralelizacije čitavog programa.

Implementacija sistema koji podržavaju paralelizaciju može biti skupa, u smislu inicijalne nabavke hardvera, tako i u pogledu održavanja opreme, a poznate su visoke cijene procesora naprednih karakteristika.

#### **1.5. Paralelni modeli**

Model paralelnog programiranja je apstrakcija paralelne računarske arhitekture, sa kojom je pogodno izraziti algoritme i njihov sastav u programima. Vrijednost programskog modela može se procijeniti na osnovu njegove opštosti: koliko dobro se može izraziti niz različitih problema za različite arhitekture, i njegove performanse: koliko efikasno kompajlirani programi mogu da izvrše.

Način na koji se pišu i izvršavaju paralelni programi zavisi od tipa memorijskog modela. Postoje dva glavna tipa: model sa dijeljenom memorijom i model sa distribuisanom memorijom, a pored njih i hibridni memorijski model. U modelu zajedničke memorije, paralelni procesi dijele globalni adresni prostor koji čitaju i pišu asinhrono. To znači da svi procesi mogu čitati i pisati u zajedničku memoriju, što omogućava lakšu komunikaciju između njih. Niti su zasnovane upravo na ovom modelu. Komunikacija između niti se ostvaruje putem pristupa zajedničkim



promenljivama u memoriji, nema potrebe za slanjem poruka i to smanjuje overhead komunikacije. Potrebna je sinhronizacija kako bi se sprečile greške poput "race conditions" i nekonzistentnog pristupa podacima. Tehnologije koje koriste ovaj model su OpenMP i Pthreads.

U distribuisanom memorijskom modelu, svaki proces ima svoju privatnu memoriju, i procesi međusobno komuniciraju preko mreže. Oni nemaju direktan pristup memoriji drugih procesa, pa se podaci šalju i primaju putem poruka. Potrebno je jasno definisati kada i kako procesi komuniciraju putem poruka, ali nema direktne potrebe za upravljanjem zajedničkom memorijom. Na ovaj način neće doći do sukoba u pristupu zajedničkoj memoriji, ali složenija komunikacija može povećati vrijeme obrade zbog overhead-a slanja poruka. Sistemi koji koriste ovaj model su najčešće klasteri računara i distribuisani sistemi.

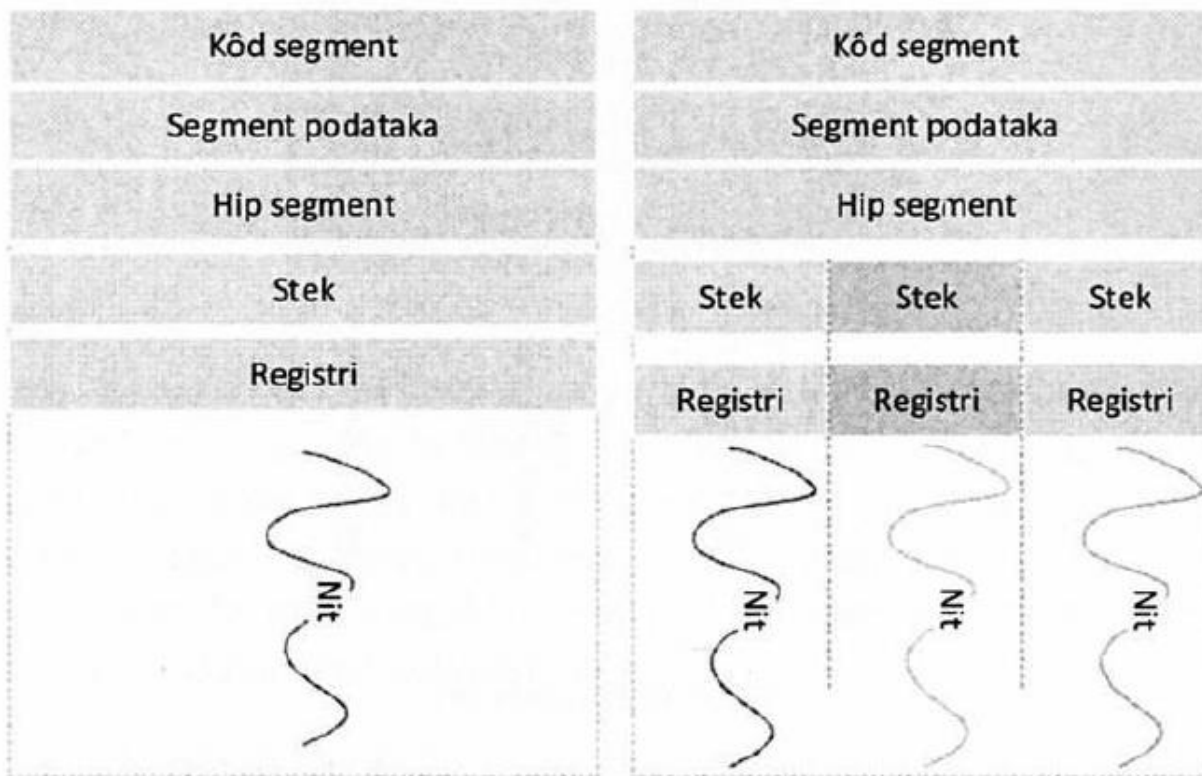
Hibridni modeli kombinuju karakteristike dijeljene i distribuirane memorije. Na primer, na nivou jednog računara (višejezgarni procesor) koristi se model sa dijeljenom memorijom, dok se između različitih računara (u klasteru) koristi model sa distribuisanom memorijom. Kombinacija MPI i OpenMP u sistemima visokih performansi je tehnologija koja koristi ovaj model.

## **1.6. Višenitost**

Paradigma višenitne obrade je postala popularnija otkako je napredak na polju paralelizma na nivou instrukcije dostigao vrhunac krajem 1990-ih. Višenitost je osobina kojom se omogućava pisanje računarskih programa koji će izvesti istovremeno dvije ili više operacija. Procesor će naizmjenično posvetiti određeni dio vremena izvođenju instrukcija u svakoj od niti, zavisno od prioriteta pojedinih niti. Neki programski jezici, kao što je Java, podržavaju višenitnost od početka, a kod C jezika to nije ugrađeni dio, nego zahtjeva upotrebu odgovarajuće biblioteke. Višenitost omogućava da zahtjevni procesi ne ometaju ostale procese u njihovom izvršavanju.

Nit (thread) je najmanji niz instrukcija koji je komponenta procesa i kojima se može nezavisno upravljati. Može da se zamisli kao tok izvođenja operacija koji se događa nezavisno od procesa ili događaja u okolini. Poput klasičnog programa koji započinje u tački A i završava u tački B, nema u sebi petlju događaja, već se izvršava ne uzimajući u obzir što se događa oko njega. Sve niti jednog procesa imaju isti imenski prostor, koriste zajedničku memoriju i tabele otvorenih datoteka. Na taj način je višenitno programiranje (eng. multithreading) znatno jednostavnije nego raspoređivanje poslova u više sistemskih procesa.

Sve niti unutar jednog procesa dijele isti imenski prostor, što znači da mogu pristupati istoj memoriji i dijeliti promenljive unutar tog procesa. Ovo se ogleda u tome kada jedna nit promijeni vrijednost globalne promenljive, sve druge niti mogu vidjeti tu promjenu. Zbog dijeljenja imenskog prostora, niti mogu lako razmjenjivati podatke bez potrebe za slanjem poruka ili kopiranjem, što olakšava paralelno programiranje. Osim resursa procesa kojem pripadaju, niti imaju i sopstvene resurse. Svaka nit poseduje svoje registre, programski brojač i stek, a razlikuje ih i jedinstveni identifikator (thread ID). Na početku izvršavanja svaki proces dobija svoj memorijski prostor i kontrolnu (inicijalnu) nit. Ova nit ima zadatak da obavi potrebne inicijalizacije i kreira ostale niti koje su potrebne za izvršavanje procesa. S obzirom da niti imaju sve karakteristike procesa, pri čemu neke resurse dijele sa drugim nitima, često se nazivaju i lakim procesima.



**Slika 1 – Jedna nit procesa i više niti istog procesa**

Neke od prednosti niti su: ako nit napravi puno promjena keša, druge niti mogu nastaviti, koristeći prednost neiskorištenih računarskih resursa, što može dovesti do bržeg izvršavanja, s obzirom da bi ti resursi bili neiskorišćeni da se samo jedna nit izvršava; ako nit ne može iskoristiti sve računarske resurse procesora (jer instrukcije zavise od međusobnih rezultata), pokretanje druge niti ih može iskoristiti i ako nekoliko niti obrađuje isti skup podataka, one mogu dijeliti keš, što dovodi to bolje iskorišćenosti keša ili sinhronizacije njihovih vrednosti.

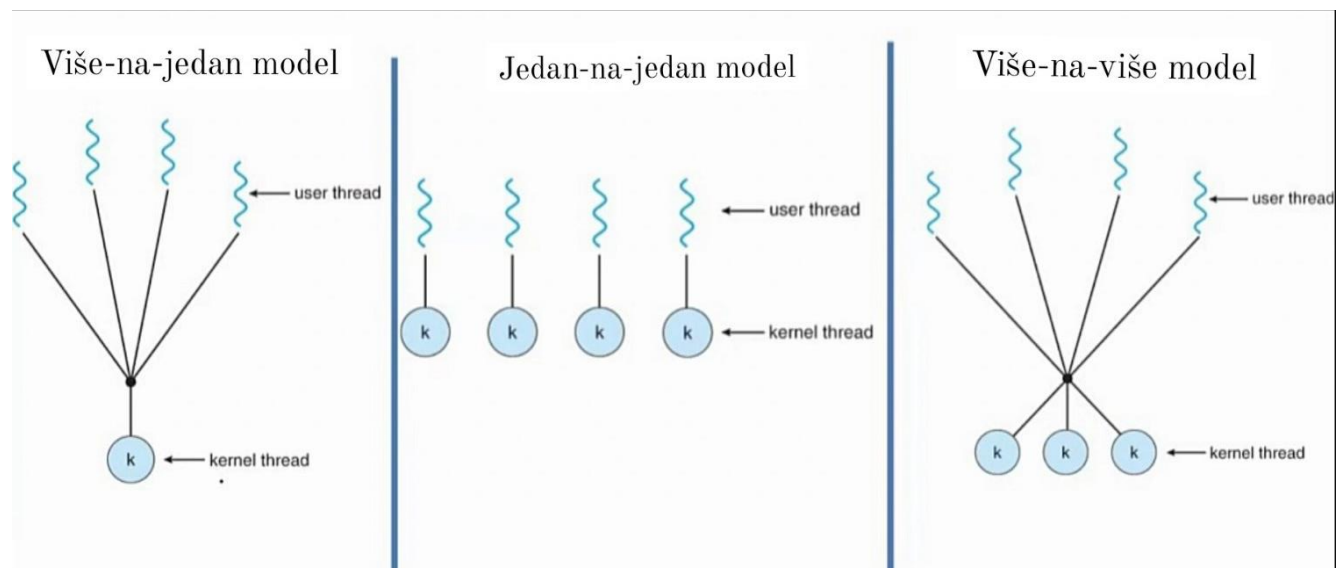
Ključna prednost upotrebe niti je značajna ušteda memorijskog prostora i vremena. Niti pružaju mogućnost aplikacijama da nastavljaju rad u situacijama kada se izvršavaju dugotrajne operacije koje bi, bez podjele poslova procesa na niti, privremeno zaustavile izvršavanje ostalih dijelova procesa. Slično, korišćenjem niti omogućava se rad procesa čiji su dijelovi potpuno blokirani, za razliku od tradicionalnog (sekvencijalnog) programiranja, kada je proces blokirani, čitav proces mora da čeka dok se blokirana operacija ne završi.

Neke od kritika višenitne obrade su: višestruke niti mogu međusobno smetati jedna drugoj pri deljenju hardvera kao što su keševi ili bafer asocijativnog prevođenja (TLBs); izvršno vrijeme jedne niti nije poboljšano već može biti smanjeno, čak i kada se samo jedna nit izvršava zbog niže frekvencije i/ili dodatnih faza protočne obrade koje su neophodne da se realizuje hardver smjene niti; hardverska podrška za višenitnu obradu je vidljivija softveru, stoga zahtjeva više promjena na nivou programa i operativnog sistema, za razliku od višeprocorskih sistema.

U zavisnosti od toga da li se nitima upravlja sa korisničkog ili sistemskog nivoa, niti se nazivaju korisničke ili niti jezgra. Pri tome, pristup procesoru i priliku da se izvršavaju imaju samo niti

jezgra, tako da je potrebno napraviti odgovarajuću korespodenciju između korisničkih i niti jezgra. Ovo se postiže preslikavanjem (mapiranjem) korisničkih u niti jezgra. Najčešće podržana preslikavanja su:

- Preslikavanje više u jednu  
Implementacije modela više-na-jedan omogućavaju aplikaciji da kreira bilo koji broj niti koje se mogu izvršavati istovremeno. U ovakvoj implementaciji sve aktivnosti niti su ograničene na korisnički prostor. Pored toga, samo jedna po jedna nit može pristupiti jezgru, tako da je operativnom sistemu poznat samo jedan entitet za planiranje. Kao rezultat, ovaj model višenitnog rada obezbeđuje ograničenu konkurentnost i ne iskorišćava multiprocesore.
- Preslikavanje jedna u jednu  
Model jedan-na-jedan je među najranijim implementacijama istinskog višenitnog rada. U ovoj implementaciji, svaka korisnička nit koju kreira aplikacija poznata je kernelu i sve niti mogu pristupiti jezgru u isto vrijeme. Glavni nedostatak ovog modela je to što svaka dodatna nit povećava opterećenja procesa.
- Preslikavanje više u više  
Model više-prema-više, koji se takođe naziva model na dva nivoa, minimizuje napor programiranja. Na ovaj način program može imati onoliko niti koliko je prikladno, a da proces ne bude previše opterećujući. U ovom modelu, biblioteka korisničkih niti obezbeđuje napredno raspoređivanje niti na nivou korisnika iznad niti kernela. Kernel treba da upravlja samo nitima koje su trenutno aktivne.



Slika 2 - Uporedni prikaz različitih višenitnih modela

### 1.6.1. Problemi višenitnosti

Uzroci najčešćih problema višenitnog programiranja leže u istim stvarima koje predstavljaju njegove prednosti. Iako korišćenje zajedničke memorije predstavlja određeno olakšanje često dolazi do sukoba više niti pri pristupanju i čitanju/mijenjanju iste memorijske lokacije u isto vrijeme. Takva pojava se naziva sukob niti (eng. *race condition*). Desi se da jedna nit počne sa čitanjem vrijednosti neke memorijske lokacije i prije nego što neka druga nit završi sa pisanjem u nju; tako prva nit dobije pola stare a pola nova podatke, što često uzrokuje netačne rezultate. U ovakvim situacijama se najčešće pribjegava rješenjima koja obuhvataju razne načine zaključavanja memorije.

U raznim programskim jezicima i okruženjima postoje različiti načini da jedna nit „zabrani“ drugim da pristupe određenom resursu dok ta nit ne završi što ima da završi. Na primjer, u OpenMP-u biblioteci, sinhronizacija između niti i zabrana pristupa zajedničkim resursima dok jedna nit obavlja određeni posao postiže se korišćenjem kritičnih sekcija i zaključavanja, korišćenjem **#pragma omp critical** direktive. Tako jedna nit može "zaključati" kritičnu sekciju koda, čime se osigurava da samo jedna nit u tom trenutku ima pristup resursu ili delu koda. Sve ostale niti moraju čekati dok ta nit ne završi i oslobodi kritičnu sekciju.

## 2. Korišteni alati i tehnologije

### 2.1. Programski jezik C

C programski jezik spada u proceduralne programske jezike opšte namjene, nastao 1972. godine. Autor jezika je Dennis Ritchie, a nastao je u istraživačkom centru Belove laboratorije u Nju Džerziju za potrebe operativnog sistema Unix. Osnovni cilj je bio sastavljanje jezika, koji će moći da zamijeni asemblerske jezike koji su zavisni od računara. Zamišljen kao jednostavan i fleksibilan, C je vrlo brzo postao jedan od najpopularnijih jezika svog vremena, što je ostao i do današnjeg dana. Posljedica široke prihvaćenosti i efikasnosti C-a je ta da su kompajleri, biblioteke i interpretatori drugih programskih jezika visokog nivoa često napisani na C-u.

Programi napisani u ovom jeziku su često bliski načinu rada hardvera, te od programera zahtijevaju da dobro razumije rad procesora, memorije, ulazno-izlaznih uređaja itd.

Programski jezik C se u određenoj mjeri mijenjao tokom godina te je u više navrata neformalno i formalno standardizovan. Kako se upotreba jezika C godinama značajno proširila na veći broj različitih platformi, javila se potreba za zvaničnom standardizacijom. Prvi zvanični standard za jezik C, takozvani ANSI C ili C89, izdao je Američki nacionalni institut za standard 1989. godine. Isti standard je naredne, 1990. godine potvrđen i od strane Međunarodne organizacije za standardizaciju (engl. ISO), te se ta ista verzija jezika nekada označava i sa C90. Verzijom C11 izvršena je nova standardizacija koja se najprije odnosi na jasan i precizan opis modela za višenitno (engl. multithreading) izračunavanje. Posljednja verzija standarda, pod zvaničnim nazivom ISO/IEC 9899:2018, objavljena je u junu 2018. godine.

Organizacione cjeline C-a su funkcije. Početna tačka izvršavanja programa je tzv. „glavna funkcija” (engl. main), a cijelokupan program se organizuje u određeni broj drugih funkcija, u zavisnosti od potrebe. Funkcije se posmatraju odvojenim, izolovanim cjelinama čime se postiže modularnost koda koji je kao takav lak za održavanje i dalje razvijanje.

Obrada programa na jeziku C sastoji se od 4 koraka:

- **Pisanje izvornog koda:** Proces kreiranja tekstualnih instrukcija u C jeziku koje definišu logiku i funkcionalnost softverskog sistema, omogućavajući kompajleru da ih konvertuje u izvršni kod koji računar može da izvrši. Izvorni kod programa je zapravo tekstualna datoteka, kojoj je pridružena odgovarajuća ekstenzija. Ekstenzija datoteka koje sadrže izvorni kod programa pisanih u programskom jeziku C je `.c`.
- **Prevođenje izvornog koda:** Proces prevođenja izvornog koda napisanog u C jeziku u mašinski kod koji računar može da izvrši, koristeći kompajler za prevođenje i linker za spajanje svih potrebnih komponenti u izvršni program. Ovaj korak obuhvata leksičku, sintaksnu i semantičku analizu.
- **Povezivanje programa:** U fazi povezivanja (eng. *linking*) se u jedan izvršni program povezuju datoteke objektnog koda koje su nastale prevođenjem izvornog koda i objektni moduli koji sadrže podatke iz biblioteka (standardnih i drugih biblioteka). U toku ove faze se pronalaze i identifikuju simboli koji se koriste u jednoj datoteci, a definisani su u nekoj drugoj.

- **Izvršavanje programa:** Ova faza predstavlja trenutak kada se program koji je prethodno kompajliran i linkovan pokreće na računaru i računar zapravo obavlja zadatke koje je program definisao, kao što su izračunavanja, prikazivanje informacija ili obrada podataka.

### 2.1.1. Identifikatori

Identifikatori su imena koja koristimo da bismo odredili (identifikovali) različite elemente programa. Služe za označavanje svih vrsta elemenata programa: podataka, simbola, konstanti, tipova podataka koji su definisani od strane programera i potprograma. U većini programskih jezika koji su danas u upotrebi postoji slična konvencija za pisanje identifikatora. U programskom jeziku C, oni mogu da se sastoje od slova, cifara i znaka podvučeno (`_`), a prvi znak u identifikatoru ne smije biti broj. . Pošto znak razmak (space) nije dozvoljen, ukoliko se identifikator sastoji od više riječi, uobičajeno je da se one razdvajaju znakom `'_'`, ili da se identifikator sastoji od malih slova, dok svaka nova riječ počinje velikim slovom. Takođe, pošto se znak `'_'` obično koristi za početni znak u nazivima sistemskih funkcija ili promjenljivih, nije preporučljivo (iako je dozvoljeno) da se taj znak koristi kao početni znak identifikatora definisanih od strane programera.

Kao identifikatori se ne mogu koristiti ni ključne (službene) ni rezervisane riječi, jer je njihovo značenje unapred definisano.

auto	extern	sizeof
break	float	static
case	for	struct
char	goto	switch
const	if	typedef
continue	int	union
default	long	unsigned
do	register	void
double	return	volatile
else	short	while
enum	signed	

Slika 3 - ključne riječi u programskom jeziku C

### 2.1.2. Promjenljive

Promjenljive su osnovni objekti koji se koriste u programiranju. Svakoj promjenljivoj je pridružen određen memorijski prostor, koji služi za čuvanje odgovarajućeg podatka, koji predstavlja vrijednost promjenljive. Tokom izvršenja programa, toj vrijednosti može da se pristupi, tj. ta vrijednost može da se pročitati a može i da se promijeni. U programskom jeziku C

svaka promjenljiva ima svoj tip koji određuje vrstu podataka koji se mogu čuvati u toj varijabli, kao i veličinu memorije koja će biti rezervisana za tu varijablu.

Korištenje promjenljivih podrazumijeva deklaraciju same promjenljive, dodjelu vrijednosti, eventualne kasnije izmjene te vrijednosti, kao i upotrebu dodijeljene vrijednosti u izrazima i funkcijama.

*Deklaracija* promjenljive uključuje određivanje tipa same promjenljive i njenog imena. Prilikom deklaracije, promjenljivoj se može odmah dodijeliti i odgovarajuća vrijednost, tj. može se izvršiti *inicijalizacija* promjenljive. U programskom jeziku C sve promjenljive moraju biti deklarirane prije upotrebe, ustaljeno je pravilo da se deklaracija promjenljivih najčešće radi na početku funkcija i prije izvršnih naredbi. Ime varijable podrazumijeva identifikator koji se koristi za referenciranje na određenu lokaciju u memoriji. Imena varijabli moraju biti jedinstvena unutar svog opsega. Tip podataka varijable definiše vrstu podataka koji se mogu čuvati u varijabli i veličinu memorije koja će biti rezervisana za tu varijablu. Vrijednost varijable je podatak koji se čuva u varijabli.

Jedna od ključnih karakteristika promjenljivih je opseg varijable, odnosi se na dio programa u kojem promjenljiva može biti korišćena. Opseg određuje vidljivost i dostupnost promjenljive u različitim dijelovima programa, kao i njeno trajanje (vrijeme životnog ciklusa). Na osnovu opsega promjenljive se dijele na:

- lokalne - promjenljive koje su definisane unutar određene funkcije ili bloka koda i dostupne su samo unutar tog bloka koda ili funkcije. One imaju ograničeni opseg, što znači da nisu vidljive izvan tog bloka ili funkcije.
- globalne - promjenljive definisane izvan svih funkcija imaju globalni opseg. Globalne promjenljive su “vidljive”, tj. mogu se koristiti u više funkcija.

### **2.1.3. Osnovni tipovi podataka**

Podaci su predmet obrade u programima. Svaki podatak ima određene osobine koje čine tip podatka. Tip podatka određen je skupom mogućih vrijednosti koje može da uzme podatak i skupom mogućih operacija koje mogu da se izvode nad podatkom. Određivanjem tipa nekog podatka, definiše se i veličina memorijskog prostora potrebnog za smještanje i način reprezentacije tog podatka. Osnovni tipovi se još nazivaju prostim ili skalarnim podacima. Ne mogu da se rastave na manje elemente koji bi se nezavisno obrađivali. U suprotnosti s njima postoje složeni ili strukturirani podaci, koji se sastoje od nekoliko elemenata koji mogu da se nezavisno obrađuju.

U programskom jeziku C osnovni tipovi podataka su:

- Cjelobrojni tipovi

Osnovni tip podatka za predstavljanje cijelih brojeva je `int`. Podrazumijeva se da pomoću ovog tipa možemo predstavljati označene cijele brojeve, tj. pozitivne i negativne brojeve kao i broj nula. Standardom nije definisana tačna veličina podatka tipa `int` (u bitovima), već je definisano samo da se za ovaj tip podatka koristi najmanje dva bajta (16 bita). Tačna veličina obično zavisi od konkretne mašine, tj. samog hardvera računara i operativnog sistema. Na današnjim računarima, podatak tipa `int` se zapisuje pomoću 4 bajta (32 bita) ili 8 bajtova (64 bita). Pored

osnovnog cjelobrojnog tipa, moguće je koristiti i “kraće” i “duže” tipove. Ako osnovnom tipu pridružimo kvantifikator short, u zapisu short int, uvešćemo podatak koji je po veličini potencijalno manji (kraći) od podatka tipa int. Ako tipu int pridružimo kvantifikator long, u zapisu long int, uvešćemo podatak koji je potencijalno veći od osnovnog podatka tipa int.

- Realni tipovi

Za predstavljanje realnih brojeva, odnosno preciznije brojeva u pokretnom zarezu koristimo tip osnovne preciznosti float, tip dvostruke preciznosti double i tip long double, koji se nekada naziva i podatak višestruke preciznosti. Slično kao i kod cjelobrojnih tipova, standardom nije tačno definisana veličina realnih tipova, ali je propisano da podatak tipa double koristi najmanje onoliko bajtova koliko i float, a da podatak tipa long double koristi barem onoliko bajtova koliko i double. Na većini današnjih računara podaci tipa float se zapisuju pomoću 4 bajta, podaci tipa double pomoću 8, a podaci tipa long double pomoću 10 bajtova. Osobine sistema brojeva sa pokretnim zarezum su definisane IEEE754 standardom, koga se u današnje vrijeme pridržava većina proizvođača hardvera.

- Znakovni tip

Znaci - karakteri (velika i mala slova, cifre, znaci interpunkcije itd.) se u programskom jeziku C predstavljaju tipom char. Tip char je zapravo cjelobrojni tip podatka, koji je veličine jedan bajt.

U verzijama programskog jezika C koje se široko koriste nema posebnog tipa podatka za predstavljanje logičkih vrijednosti tačno i netačno (true i false), već se umjesto njih najčešće koriste cjelobrojni tipovi, tako što se smatra da broj 0 ima vrijednost netačno, dok sve vrijednosti različite od nule imaju vrijednost tačno. Treba napomenuti da je standardom C99 uveden i logički tip podatka, a podaci ovog tipa mogu da uzimaju vrijednost true i false. Međutim, pri programiranju u programskom jeziku C, ustaljena je i česta praksa da se logički izrazi zasnivaju samo na brojevnim vrijednostima.

#### 2.1.4. Operatori

Operatori predstavljaju radnje koje se izvršavaju nad operandima dajući pri tome određeni rezultat. Operacije i relacije koje se mogu izvršavati na podacima osnovnih tipova predstavljamo odgovarajućim operatorima. Pored operacija koje se primjenjuju na osnovnim tipovima, podržane su i operacije na pojedinačnim bitovima. Po broju operandi koji učestvuju u operaciji, odgovarajuće operatore dijelimo na unarne (učestvuje samo jedan operand), binarne (dva operandi) ili ternarne (tri operandi). Jezik C podržava širok spektar operatora, uključujući aritmetičke, relacione, logičke, bitovske i druge operatore.

**Aritmetički operatori** - za sabiranje, oduzimanje, množenje, dijeljenje i računanje ostatka pri dijeljenju koriste se (binarni) infiksni operatori +, -, \*, / i %.

**Operatori poredenja** - nad cijelim brojevima i brojevima u pokretnom zarezu definisani su binarni relacijski operatori za poredenje. Kada neki od operatora poredenja bude primjenjen na dva operandi (dva broja) dobija se vrijednosti 0 (koja odgovara logičkoj vrijednosti netačno) ili vrijednost 1 (tačno).

To su operatori:



< operator manje

<= operator manje ili jednako

> operator veće

>= operator veće ili jednako

== operator jednakosti

!= operator nejednakosti (operator različito)

**Logički operatori** - za operande uzimaju logičke izraze (tj. izrazi koji mogu da imaju jednu od dvije vrijednosti: tačno ili netačno). S obzirom na to da su u programskom jeziku C logičke vrijednosti predstavljene brojevima, može se reći da su i logički operatori takvi da se primjenjuju na brojeve i kao rezultat daju podatak koji je broj (tipa int) i to ponovo: vrijednost 0, ukoliko je istinitosna vrijednost logičkog izraza netačna, odnosno vrijednost 1, ako je istinitosna vrijednost logičkog izraza tačna. U programskom jeziku C postoje tri logička operatora:

&& operator konjunkcije

|| operator disjunkcije

! operator negacije

**Operatori na nivou bitova** - bitovski operatori u programskom jeziku C su specijalni operatori koji omogućavaju manipulaciju pojedinačnim bitovima u brojevima. Ovi operatori se koriste za bitovske operacije koje su često efikasne i korisne u niskonivovskom programiranju i u situacijama gde je potrebna brza manipulacija podacima na nivou bita. Nad bitovima su definisani sljedeći operatori:

~ komplement

& bitovska konjunkcija

| bitovska disjunkcija

^ bitovska ekskluzivna disjunkcija

<< pomjeranje ulijevo

>> pomjeranje udesno

**Operatori dodjele** - simboli koji se koriste za dodjeljivanje ili ažuriranje vrijednosti promjenljivih. Oni ne samo da dodjeljuju vrijednost, već mogu i kombinovati dodjeljivanje sa drugim operacijama, kao što su aritmetičke ili bitovske operacije.

=: dodjeljuje direktno.

+=, -=, \*=, /=, %=: kombinovani sa aritmetičkim operacijama.

&=, |=, ^=, <<=, >>=: kombinovani sa bitovskim operacijama.

**Uslovni operator** - C ima samo jedan ternarni operator (operator koji uzima tri operanda), koji se koristi u sljedećem zapisu:

izraz1 ? izraz2 : izraz3

Izraz koji je formiran ovim operatorom se naziva uslovni izraz. Uslovni operator funkcioniše po sljedećem principu: prvo se izračunava izraz1. U slučaju da je istinitosna vrijednost rezultata tog izraza tačno (različita od nula), onda se računa izraz izraz2 i ta vrijednost je ujedno i vrijednost čitavog uslovnog izraza. Ako je istinitosna vrijednost rezultata tog izraza izraz1 netačno, izračunava se izraz3 i vrijednost tog izraza izraz3 je i vrijednost čitavog uslovnog izraza.

**Operatori inkrementiranja i dekrementiranja** - unarni operatori uvećanja za 1 odnosno umanjenja za 1. Operator uvećanja za jedan se zapisuje pomoću dva uzastopna znaka + bez razmaka između (++), dok se operator umanjenja zapisuje pomoću znaka –.

**Operator sizeof** - unarni operator sizeof daje za rezultat broj koji predstavlja veličinu memorijskog prostora (u bajtovima) potrebnu za smještanje svog operanda. Može se primjenjivati na promjenljive, tip podatka ili izraz.

#### 2.1.5. Upravljačke strukture

Upravljačke strukture predstavljaju elemente pomoću kojih se postiže izvršavanje pojedinih grupa naredbi, u zavisnosti od nekih uslova i ponavljanje pojedinih dijelova algoritma. Pošto u programskom jeziku C ne postoji logički tip promjenljive, uslov je svaki izraz čija se vrijednost može izračunati, a uslov se smatra ispunjenim (tačnim) ako je vrijednost različita od nule. U suprotnom (vrijednost je jednaka nuli), smatra se da uslov nije ispunjen, tj. netačan je. Dakle, u pitanju su uslovna grananja i petlje.

**if else (uslovno grananje)** - omogućava izvršenje različitih blokova koda na osnovu uslova. Izraz koji slijedi nakon riječi *if* je logički uslov (za koga se smatra da može da bude tačan ili netačan). Naredbe *naredba1* i *naredba2* mogu biti ili pojedinačne naredbe ili složena naredba (blok naredbi), tj. više naredbi grupisanih u okviru vitičastih zagrada. Ako je uslov u *if* izrazu tačan, izvršava se kod unutar *if* bloka; ako nije, može se izvršiti alternativni kod unutar *else* bloka. *Else* dio je opcioni i ne mora se navesti ako za tim nema potrebe.

```
if(uslov)
    naredba1;
else
    naredba2;
```

**Petlje** – koriste se u situacijama kada je potrebno da se određeni dio koda izvršava više puta. Sastavni dijelovi petlje su uslov petlje i blok petlje — dok je uslov tačan, blok petlja se ponavlja. Kada zadati uslov postane netačan, ponavljanje se prekida i program nastavlja sa izvršavanjem. U programskom jeziku C postoje sledeće vrste petlji:

- **while:** u okviru nje prvo se ispituje vrijednost izraza uslov. Ako taj izraz ima vrijednost tačno, izvršavaju se naredbe koje se ponavljaju (pojedinačna naredba ili blok naredbi). Ova petlja se koristi kada uslov treba proveriti prije svake iteracije.

```
while(uslov)
    blok naredbi koje se ponavljaju
```

- **do...while:** ključna riječ *do* označava početak petlje. Nakon što program naiđe na naredbu *do*, izvršavaju se naredbe koje slijede u vitičastim zagradama. Zatim slijedi provjera istinitosti uslova. Petlja će se izvršavati sve dok je uslov ispunjen. Sadržaj petlje će se uvijek izvršiti barem jednom jer se uslov provjerava tek na kraju. Petlja se koristi kada programer zna da se neki blok naredbi u iterativnom procesu mora izvršiti najmanje jednom.

```
do{
    blok naredbi koje se ponavljaju
}while(uslov)
```

- **for:** na početku *for* petlje vrši se inicijalizacija. Nakon toga se ispituje uslov. Ukoliko je vrijednost izraza uslov tačna, izvršavaju se naredbe koje čine tijelo petlje. Ako vrijednost uslova nije tačna, dolazi do kraja i program nastavlja sa radom od prve naredne naredbe koja slijedi poslije petlje. Ako je uslov bio tačan, nakon izvršenja naredbi iz tijela petlje, izvršava se naredba korak, nakon čega se program ponovo vraća na ispitivanje uslova. Koristi se najčešće kada je prije početka izvršenja petlje poznato koliko puta će biti ponavljene iteracije.

```
for(inicijalizacija; uslov; korak)
    naredbe koje se ponavljaju
```

**Switch naredba** - počinje riječju *switch* nakon koje slijedi *izraz*, koji se piše u malim zagradama. Nakon ovog izraza slijedi niz od nekoliko dijelova koji počinju riječju *case*. Iza svake riječi *case* slijedi konstantna vrijednost koja mora biti istog tipa kao i vrijednost izraza u malim zagradama, potom znak *:*, te, nakon toga, jedna ili više naredbi, koje su odvojene *;*. Idući od jednog do drugog slučaja, program ispituje da li je izraz jednak konstantnoj vrijednosti koja slijedi nakon neke riječi *case*. Kada program naiđe na vrijednost koja je jednaka vrijednosti *izraz*, izvršavaju se naredbe koje slijede nakon dvije tačke, sve dok se ne dođe do kraja čitavog bloka ili do naredbe *break*. Naredbu *break* nije obavezno navoditi, ali ona koristi da kada se pronađe slučaj koji je povoljan, prekida se dalje izvršavanje naredbe *switch*. Na slučaj *default* se prelazi ako se nigdje ranije ne desi da je vrijednost izraza izraz jednaka nekom od konstantnih izraza. Slučaj *default* je opcionalan, što znači da se ne mora navesti.

```
switch (odabir)
{
    case 1:
        sekvencijalniAlgoritam(matricaUdaljenosti, n, odgovor);
        break;
    case 2:
        paralelniAlgoritam(matricaUdaljenosti, n, odgovor);
        break;
    default:
        break;
}
```

Kodni listing 1 - primjer switch naredbe

**Skokovi** – koriste se za kontrolu toka izvršavanja koda, omogućavaju programu da pređe na drugi dio koda, preskoči dio koda ili se vrati na prethodni dio koda. Ključne riječi koje se koriste za skokove u jeziku C su:

- `break` za prekid petlje ili switch bloka.
- `continue` za prelazak na sljedeću iteraciju petlje.
- `return` za vraćanje vrijednosti iz funkcije i prekid izvršenja funkcije.

#### 2.1.6. Nizovi

Niz predstavlja skup podataka istog tipa, koji nisu pojedinačno imenovani, već se oni identifikuju svojom pozicijom (indeksom) u nizu. Nizovi predstavljaju veoma važne i moćne strukture pomoću kojih na jednom, dovoljno velikom mjestu, smještamo veći broj podataka istog tipa.

Sintaksa deklaracije niza je: `tip imeNiza[dimenzija];`

Tip određuje tip elemenata niza, dok dimenzija predstavlja broj elemenata niza. Dimenzija niza treba da bude konstantan cjelobrojni pozitivan izraz, na primjer pozitivan cio broj, ili cjelobrojna promjenljiva kojoj je dodijeljena neka pozitivna vrijednost. Elementima niza se pristupa pomoću njihovih pozicija indeksa u nizu. Element na početnoj poziciji ima indeks 0, dok posljednji element niza ima indeks dimenzija-1. Prilikom kreiranja niza, na osnovu tipa podatka elemenata niza i dimenzije niza u memoriji se zauzima odgovarajući prostor koji je potreban da se smjeste svi elementi niza. Susjedni elementi niza se smještaju na susjedne memorijske lokacije.

Za nizove promjenljive veličine, čije dimenzije nisu poznate prilikom kompajliranja, u jeziku C se može koristiti dinamička alokacija memorije. To se postiže korišćenjem funkcija poput funkcija `malloc`, `calloc` ili `realloc` za alokaciju memorije i funkcije `free` za oslobađanje zauzete memorije.

#### 2.1.7. Pokazivači

Pokazivači u programskom jeziku C su specijalne vrste varijabli koje čuvaju adrese drugih varijabli u memoriji. Koriste se za direktan pristup i manipulaciju podacima u memoriji, što omogućava efikasnije korišćenje resursa i rad sa dinamičkim podacima. Deklarišu se na sljedeći način: `tip_pokazivaca * ime_pokazivaca;` Inicijalizacija pokazivača je dodjeljivanje adrese promjenljive pokazivaču. Pokazivač se inicijalizuje dodjeljivanjem adrese neke promjenljive koristeći znak `&` (operator adrese). To omogućava pokazivaču da upućuje na specifičnu lokaciju u memoriji. Dereferenciranje je proces pristupanja vrijednosti na koju pokazivač pokazuje. Koristi se operator `*`. Npr: `int value = *p;`

Pokazivači se koriste za rad sa dinamički alociranom memorijom. Omogućavaju efikasan pristup i manipulaciju elementima nizova. `ime niza` je pokazivač na njegov prvi element. Mogu da se koriste za prosljeđivanje argumenata funkcijama, omogućavajući funkcijama da modifikuju vrijednosti promjenljive koje su prosljeđene kao pokazivači.

#### 2.1.8. Biblioteke

U programskom jeziku C, biblioteke su ključni dio za rad s različitim funkcijama i uslugama koje nisu dio osnovnog jezika. Standardne biblioteke pružaju funkcionalnosti za rad sa

stringovima, ulazom/izlazom, matematikom, radom sa datotekama i mnogim drugim potrebama. Rutine iz biblioteka se uključuju u program ključnom riječi: **#include <biblioteka.h>**.

Biblioteke koje su korišćene za izradu aplikacije su *omp.h*, *stdio.h*, *stdlib.h* i *windows.h*.

*stdio.h* biblioteka (Standard Input Output) u programskom jeziku C pruža funkcionalnosti za ulaz i izlaz podataka. To uključuje rad sa standardnim uređajima za ulaz/izlaz (kao što su tastatura i ekran), kao i rad sa datotekama. U ovoj biblioteci se nalaze funkcije za formatiranje, čitanje, pisanje i upravljanje datotekama. Funkcije korištene iz ove biblioteke:

- **printf()** - za ispis formatiranih podataka na standardni izlaz. Format predstavlja niz karaktera koji sadrži tekst za ispis i specijalne format specifikatore koji se koriste za određivanje kako će različite vrste podataka biti formatirane
- **scanf()** - za unos podataka sa standardnog ulaza, obično sa tastature. Ona čita podatke sa tastature u skladu sa zadatim formatom i dodjeljuje ih odgovarajućim promjenljivim

*stdlib.h* biblioteka koristi se za rad sa dinamičkom memorijom, sortiranje, pretragu i druge osnovne funkcionalnosti. Funkcije biblioteke korišćene u izradi aplikacije su:

- **malloc** - za dinamičko alociranje memorije tokom izvršenja programa. Pomaže da se dobije tačno onoliko prostora koliko treba za određene podatke. Prima veličinu memorije koju treba alocirati kao argument i vraća pokazivač na početak bloka.
- **free** - kada se koristi funkcija `malloc()` (ili slične funkcije kao što su `calloc()` ili `realloc()`), dobija se dio memorije koji je potreban za čuvanje podataka. Međutim, kada se završi s radom sa tim prostorom, potrebno ga je “vratiti” računaru kako bi mogao ponovo da ga koristi, pomoću ove funkcije.

*omp.h* biblioteka omogućava rad sa OpenMP (Open Multi-Processing), što je standard za višenitno programiranje na višejezgarnim procesorima. Funkcije biblioteke *omp.h* korišćene u izradi aplikacije su:

- **#pragma omp parallel for** - automatski raspodjeljuje iteracije petlje između dostupnih niti
- **omp\_get\_max\_threads()** - dobijanje maksimalnog broja niti koje OpenMP može koristiti za paralelno izvršavanje
- **omp\_get\_wtime()** - vraća tekuće vreme u sekundama kao *double* podatak, koje je precizno do mikrosekundi. Može se koristiti za mjerenje vremena trajanja izvršavanja dijela programa.

*windows.h* biblioteka služi pristupu funkcijama i strukturama specifičnim za Windows API, korištena za podešavanje naslova prozora pokrenute aplikacije.

### 2.1.9. Funkcije

Funkcija je zaokruženi dio programa (često se naziva i potprogramom), koji može (ali i ne mora) da uzima ulazne podatke, izvršava niz naredbi i vraća rezultat programu iz kog je ta funkcija pozvana. Jednom napisana funkcija može više puta biti iskorištena u okviru jednog

programa, ali se, takodje, može koristiti i pozivati u više različitih programa. Zbog toga se često kaže da se upotrebom funkcija dobija mogućnost “ponovnog korišćenja” (engl. re-usability) programskog koda, što najčešće dovodi do vremenskih ušeda u razvoju.

Deklaraciju funkcije shvatamo kao najavljivanje funkcije ili kreiranje prototipa funkcije, gdje navodimo tip podatka koji funkcija vraća kao rezultat, ime funkcije, te broj i tip podataka koji se funkciji predaju kao argumenti. Opšti oblik deklaracije funkcije bi izgledao ovako:

```
tipRezultata ime_funkcije(tip1 arg1, ... ,tipN argN);
```

gdje tipRezultata predstavlja tip podatka koji funkcija vraća kao rezultat svog izvršavanja. Nakon tipa rezultata, navodi se ime funkcije. Imena funkcija su identifikatori, te, stoga i za njih važe ista pravila kao i za imena promjenljivih. Poželjno je da ime funkcije ukazuje na ono šta funkcija radi.

Prilikom definisanja funkcije navodimo tip rezultata funkcije, ime funkcije i listu argumenata, nakon čega slijedi tijelo funkcije, koje se sastoji od jedne ili više naredbi. Rezultat funkcije je vrijednost koja se izračunava u okviru tijela funkcije i vraća kao rezultat na ono mjesto u programu sa kog je funkcija pozvana. Opšti oblik definicije funkcije izgleda ovako:

```
tipPodatka ime_funkcije(tip1 arg1, ... ,tipN argN)
{
Tijelo funkcije
}
```

Pri deklarisanju i definisanju funkcije unutar malih zagrada koja slijedi nakon imena funkcije, navodimo formalne argumente ili parametre funkcije. Ispred naziva svakog parametra navodi se njegov tip, a parametri se razdvajaju zarezom. Preporučljivo je da imena parametara funkcije oslikavaju njihovo značenje i ulogu u funkciji.

Tijelo funkcije se navodi unutar vitičastih zagrada koje slijede nakon zatvorene male zagrade.

Funkcija se izvršava tako što je “pozivamo” iz nekog drugog dijela programa. Prilikom poziva funkcije navodi se njen naziv, nakon čega slijedi lista stvarnih argumenata. Stvarni argumenti moraju biti navedeni u redoslijedu koji je zadan deklaracijom ili definicijom funkcije, na način da redoslijed tipova stvarnih argumenata odgovara redoslijedu formalnih argumenata.

Funkcija vraća rezultat svog izvršavanja pomoću naredbe return. Oblik u kome se naredba return koristi je prilično jednostavan: `return rezultat;`

Po izvršavanju naredbe return funkcija završava sa svojim izvršavanjem, a vrijednost rezultat se vraća onom dijelu programa iz kojeg je pozvana funkcija.

Rekurzivne funkcije su funkcije koje u okviru svoje definicije pozivaju tu istu funkciju.

## 2.2. OpenMP

OpenMP je tehnologija na strani kompajlera za kreiranje koda koji radi na više jezgara odnosno niti i može značajno poboljšati performanse programa.

OpenMP API koristi *fork-join* model paralelnog izvršavanja. Više niti obavlja zadatke definisane implicitno ili eksplicitno OpenMP direktivama. Sve OpenMP aplikacije počinju kao jedna nit izvršenja, koja se naziva početna nit. Početna nit se izvršava sekvencijalno sve dok ne naiđe na paralelnu konstrukciju. U tom trenutku, ova nit kreira grupu od sebe i nula ili više dodatnih niti i postaje glavna nit nove grupe. Svaka nit izvršava komande uključene u paralelni region, a njihovo izvršavanje se može razlikovati u skladu sa dodatnim direktivama koje daje programer. Na kraju paralelnog regiona, sve niti su sinhronizovane.

Izvršno okruženje je odgovorno za efikasno raspoređivanje niti. Svaka nit dobija jedinstveni ID, koji ga razlikuje tokom izvršavanja. Planiranje se vrši prema upotrebi memorije, opterećenju mašine i drugim faktorima i može se prilagoditi promjenom promjenljivih okruženja.

Posjeduje niz alata koji se mogu koristiti za pravilno opisivanje kako paralelni program treba da rukuje promenljivim. Ovi alati dolaze u obliku zajedničkih i privatnih klasifikatora tipa promenljivih. Privatni tipovi kreiraju kopiju promenljive za svaki proces u paralelnom sistemu, a dijeljeni tipovi sadrže jednu instancu promenljive koju svi procesi mogu da dijele.

OpenMP ima niz alata za upravljanje procesima. Jedan od oblika kontrole je barijera:

```
#pragma omp barrier i kritične sekcije: #pragma omp critical {...}
```

Direktiva o barijeri zaustavlja sve procese da ne bi prešli na sljedeću liniju koda dok svi procesi ne dostignu barijeru. Ovo omogućava programeru da sinhronizuje sekvence u paralelnom procesu. Kritična sekcija obezbjeđuje da liniju koda pokreće samo jedan proces u jednom trenutku, obezbjeđujući bezbjednost niti u tijelu koda.

Moć OpenMP-a dolazi do izražaja pri razdvajanju većeg zadatka na više manjih zadataka. Direktiva o podjeli posla omogućava jednostavno i efikasno razdvajanje zadataka koji se inače izvršavaju serijski u brze paralelne dijelove koda.

Glavna prednost upotrebe OpenMP-a je lakoća razvoja paralelizma pomoću jednostavnih konstrukcija koje se (često) ne razlikuju mnogo od originalne implementacije.

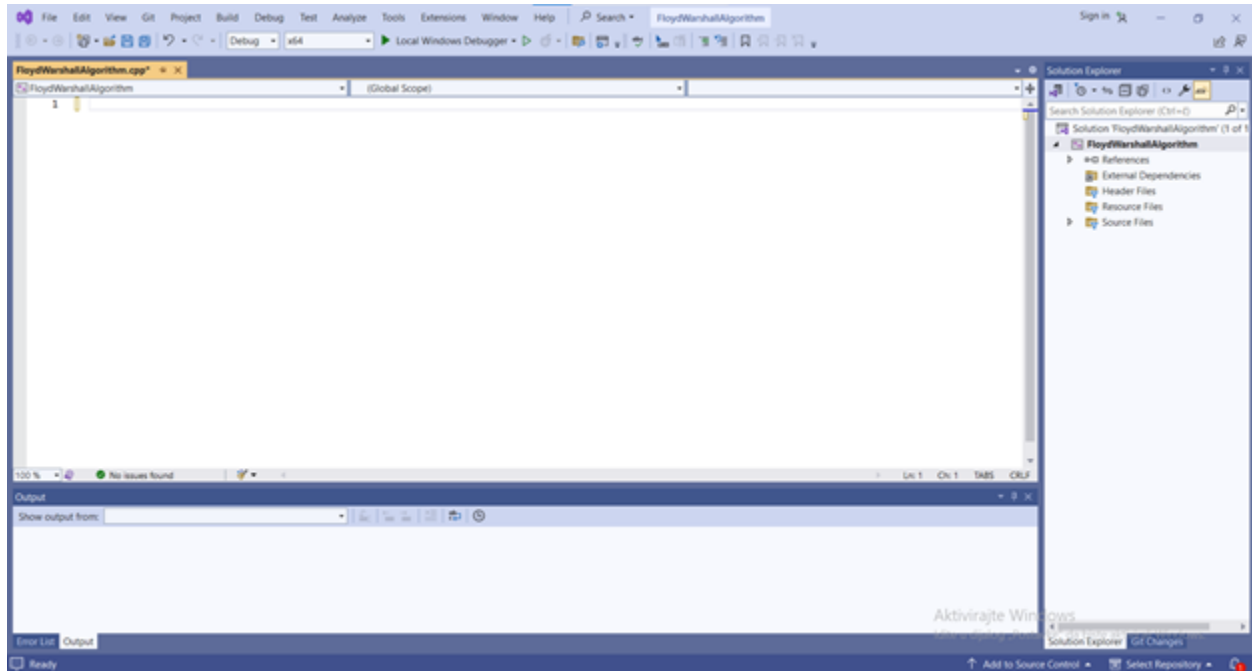
## 2.3 Visual Studio 2022

Visual Studio 2022 je integrisano razvojno okruženje razvijeno od strane kompanije Microsoft, predstavlja napredan alat za razvoj softvera koji nudi sveobuhvatan skup alata i funkcionalnosti za kreiranje, testiranje i održavanje različitih vrsta softverskih rješenja. Dizajniran je za profesionalne programere i timove, omogućavajući im da efikasno pišu, debuguju i optimizuju kod u različitim programskim jezicima i platformama.

Da bi se kreirao projekat u programskom jeziku C, potrebno je izvršiti sljedeće korake:

- Pokrenuti Visual Studio 2022 aplikaciju.
- Odabrati opciju Create a new project u glavnom meniju.

- Izabrati u prvoj padajućoj listi C++ .
- Zatim izabrati vrstu projekta, u našem slučaju Console App.
- Unijeti ime projekta i odabrati lokaciju na kojoj treba sačuvati projekat, zatim kliknuti na opciju create.



**Slika 4 - izgled kreiranog projekta u Visual Studio 2022 IDE**

Jedna od najznačajnijih promjena u Visual Studio 2022 je prelazak na 64-bitnu arhitekturu. Ovo je prvi put da je Visual Studio dostupan u 64-bitnom formatu, što donosi značajna poboljšanja u performansama. Sa ovom promjenom, Visual Studio može bolje koristiti dostupnu memoriju, omogućavajući bolju stabilnost i efikasnost pri radu sa velikim i kompleksnim projektima. Ova nadogradnja značajno smanjuje vrijeme čekanja i omogućava brže učitavanje i obradu podataka, što je ključna prednost za programere koji se bave velikim projektima.

Visual Studio 2022 posjeduje raznovrsne alate za kodiranje, olakšano debugovanje i testiranje aplikacija te podršku za više programskih jezika kao što su C#, C++, Visual Basic, JavaScript, Python i dr. Na taj način omogućava razvoj za Windows, macOS, Linux, kao i za mobilne platforme kao što su Android i iOS. Značajno je spomenuti i integraciju sa sistemima za kontrolu verzija sa Git-om i GitHub-om za upravljanje verzijama koda.

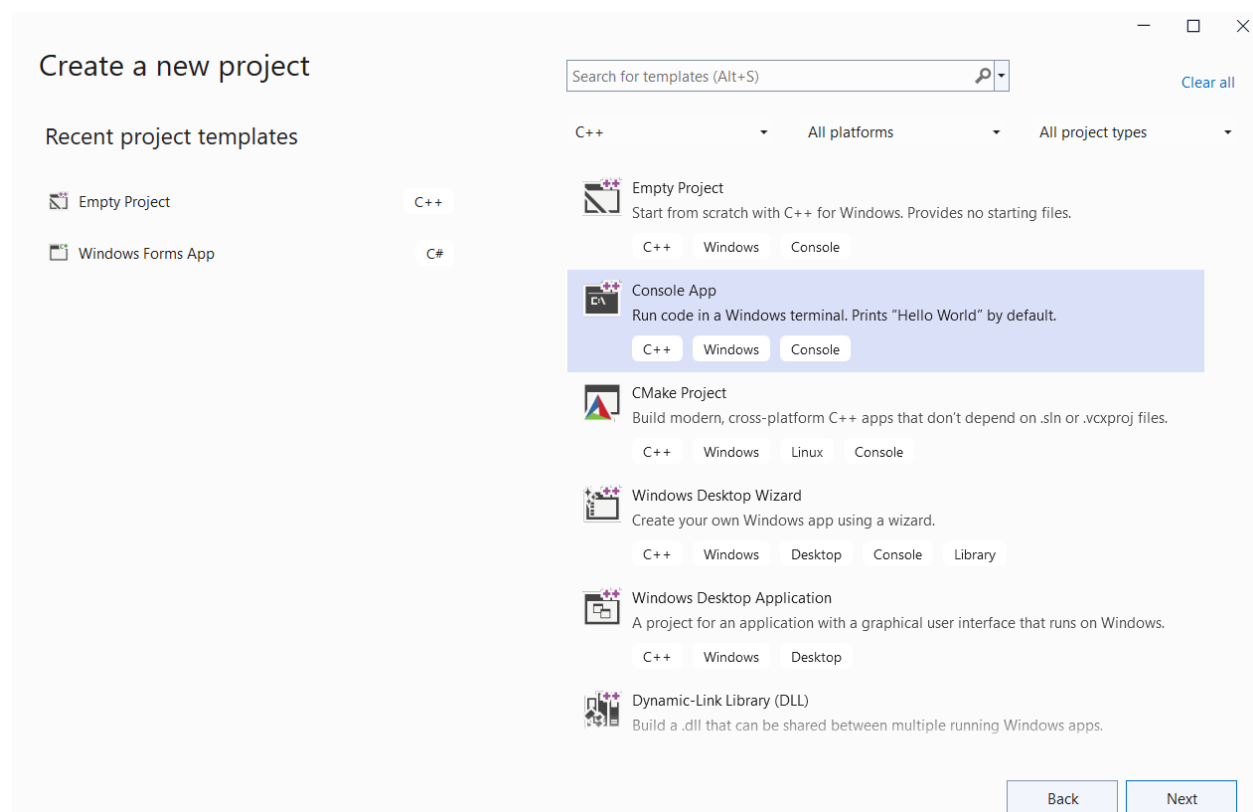
## **2.4 Konzolna aplikacija**

*Console App* ili konzolna aplikacija, je vrsta softverske aplikacije koja koristi tekstualni interfejs za komunikaciju s korisnikom. Umjesto grafičkog korisničkog interfejsa (GUI) sa prozorima, dugmadima i drugim vizuelnim elementima, konzolne aplikacije komuniciraju putem komandne linije ili terminala i obično se koriste za zadatke koji ne zahtijevaju složene grafičke prikaze ili interaktivne elemente.



Umesto grafičkog interfejsa, konzolne aplikacije koriste tekstualni interfejs, gde korisnici unose komande putem tastature i dobijaju rezultate u tekstualnom formatu. oriste se za brzi razvoj i testiranje programskih koncepata i algoritama bez potrebe za grafičkim interfejsom, kao i za kreiranje skripti koje automatski obavljaju zadatke kao što su backup podataka, generisanje izveštaja ili obrada velikih količina podataka. Idealan je za alate koji analiziraju, manipulišu i obrađuju tekstualne podatke. Komunikacija sa korisnikom obično se odvija putem unosnih komandi (input) i ispisivanja rezultata (output) na ekran.

Konzolne aplikacije se mogu napisati u različitim programskim jezicima, uključujući C#, Java, Python, C++, i mnoge druge. Obično se izvršavaju u terminalu ili komandnoj liniji operativnog sistema, kao što su Command Prompt na Windows-u, Terminal na macOS-u ili Linux-u.



**Slika 5 - kreiranje konzolne aplikacije u kroz Visual Studio 2022 IDE**

### 3. Floyd – Warshall algoritam

U računarstvu Floyd-Warshall algoritam (ponekad se naziva i Roy-Floyd algoritam ili WFI algoritam, otkad je Bernard Roy opisao algoritam 1959. godine) je algoritam analize grafova za nalaženje najkraćih putanja u težinskom, usmjerenom grafu. Jedno izvršenje algoritma će naći najkraće putanje između svih parova čvorova. Floyd-Warshall algoritam je primjer dinamičkog programiranja.

Algoritam koristi matricu udaljenosti, koja se inicijalizuje sa vrijednostima udaljenosti između čvorova. U početku, ako postoji direktna ivica između čvorova, udaljenost se postavlja na težinu te ivice; inače, udaljenost se postavlja na beskonačnost (ili neku veoma veliku vrijednost). U našoj aplikaciji pretpostavili smo da je graf potpuno povezan tj. da postoji grana između bilo koja dva čvora grafa i zato nema beskonačnih vrijednosti. Dijagonalne vrednosti (udaljenost čvora od samog sebe) se postavljaju na nulu. Za  $n$  čvorova formira se matrica dimenzija  $n \times n$ . Element matrice  $(i, j)$  predstavlja težinu ivice od čvora  $i$  do čvora  $j$ .

---

**Algorithm 1:** The Floyd-Warshall (FW) algorithm

---

```
1 for  $(u, v) \in E$  do
2   |  $M_{u,v} \leftarrow w(u, v)$ 
3 end
4 for  $v = 1 \rightarrow n$  do
5   |  $M_{v,v} \leftarrow 0$ 
6 end
7 for  $k = 1 \rightarrow n$  do
8   | for  $i = 1 \rightarrow n$  do
9     | for  $j = 1 \rightarrow n$  do
10      | if  $M_{i,j} > M_{i,k} + M_{k,j}$  then
11        | |  $M_{i,j} \leftarrow M_{i,k} + M_{k,j}$ 
12      | end
13    | end
14  | end
15 end
```

---

Slika 6 - pseudokod Floyd - Warshall algoritma

Izlaz algoritma je takođe u matričnom obliku: broj u  $i$ -toj vrsti i  $j$ -toj koloni je težina najkraće putanje između čvorova  $i$  i  $j$ .

### 3.1 Matematičko objašnjenje

FW algoritam rješava problem nalaska najkraćih udaljenosti između čvorova (eng. *all-pairs shortest path – ASPS*) u povezanom težinskom grafu  $G(V, E, w)$ , gdje  $V = \{1, \dots, n\}$  predstavlja skup čvorova grafa,  $E \subseteq V \times V$  su ivice grafa, a  $w$  je težinska funkcija  $E \rightarrow R$  koja predstavlja cijenu puta između dva čvora.

Pretpostavimo da imamo graf  $G$  sa  $V$  čvorova i težinama ivica. Težine ivica između čvorova  $i$  i  $j$  su predstavljene matricom  $W$ , gdje je  $W[i][j]$  težina ivice između čvorova  $i$  i  $j$ . Ako ne postoji direktna ivica između  $i$  i  $j$ , težina  $W[i][j]$  se postavlja na beskonačnost ( $\infty$ ).

Inicijalno, postavljamo matricu udaljenosti  $D$  na istu vrijednost kao i matrica težina  $W$ , pri čemu je  $D[i][j]=W[i][j]$ . Takođe, za sve  $i$ ,  $D[i][i]=0$ , jer je udaljenost od čvora do samog sebe nula.

Koraci u algoritmu:

#### 1. Inicijalizacija

Inicijalno, postavljamo udaljenosti između svih čvorova prema težinama ivica. Ako između dva čvora nema direktne ivice, udaljenost se postavlja na beskonačnost.

$$D^{(0)}[i][j] = \begin{cases} W[i][j] & \text{ako } i \neq j \\ 0 & \text{ako } i = j \end{cases}$$

#### 2. Iterativno ažuriranje

Floyd-Warshallov algoritam koristi pomoćnu matricu  $D$  da ažurira udaljenosti između čvorova koristeći sve moguće čvorove kao posrednike. Ovaj korak se može izraziti formulom:

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

gdje:  $D^{(k)}[i][j]$  predstavlja najkraću udaljenost između čvorova  $i$  i  $j$  koristeći samo prvih  $k$  čvorova kao posrednike.

$D^{(k-1)}[i][j]$  predstavlja udaljenost između čvorova  $i$  i  $j$  koristeći samo prvih  $k-1$  čvorova kao posrednike.

Ova formula implicira da, za svaki par čvorova  $i$  i  $j$ , možemo provjeriti da li postoji kraći put kroz čvor  $k$ . Ako je tako, ažuriramo  $D[i][j]$  na kraću udaljenost.

#### 3. Konačni rezultati

Nakon što su svi čvorovi korišćeni kao posrednici (od  $k=1$  do  $k=V$ ), matrica  $D$  sadrži najkraće udaljenosti između svih parova čvorova.

$D[i][j]$  = najkraća udaljenost između  $i$  i  $j$

### 3.2 Praktična primjena

Algoritam nalazi primjenu u realnom svijetu, npr. u bioinformatičari za formiranje klastera povezanih gena, zatim u sistemima baza podataka za optimizaciju SQL upita i u *data mining* – u. Pored toga, koristan je u sljedećim upotrebama:

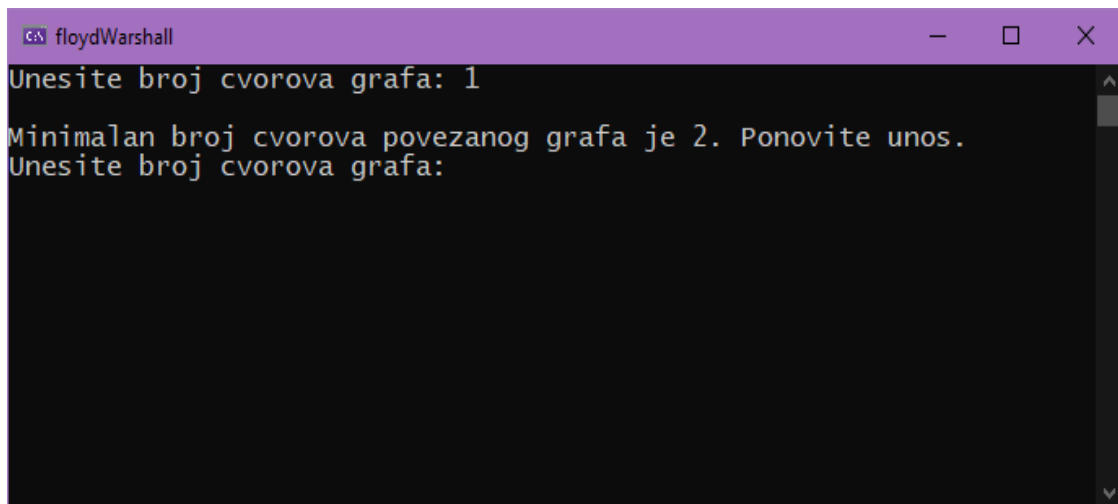
- Analiza društvenih mreža: U društvenim mrežama, gde su čvorovi korisnici, a veze među njima predstavljaju prijateljstva ili interakcije, algoritam može pomoći u pronalaženju najkraćeg puta između dva korisnika ili identifikovanju veza unutar društvenih mreža. To znači da se može koristiti za izračunavanje sličnosti ili društvene udaljenosti između korisnika u velikim mrežama kao što su Facebook ili LinkedIn.
- Telekomunikacione mreže: Telekomunikacione kompanije koriste Floyd-Warshall algoritam za optimizaciju putanja podataka unutar velikih mreža, kao što su internet ili mobilne mreže. Algoritam može pomoći u pronalaženju najefikasnijeg puta za prenos podataka između različitih čvorova mreže, čime se smanjuju kašnjenja i optimizuje korišćenje resursa.
- Navigacioni sistemi i mape: Navigacioni sistemi kao što su Google Maps ili GPS uređaji koriste algoritme za pronalaženje najkraćih puteva između lokacija. Iako se Floyd-Warshall algoritam koristi za specifične slučajeve, on može biti primjenjen u mrežama puteva, posebno kada treba da se izračunaju sve moguće najkraće rute između više tačaka.
- Mrežna analiza i optimizacija: Računarske mreže, gdje su čvorovi računari ili uređaji, a veze između njih predstavljaju komunikacione kanale, mogu koristiti FW algoritam za optimizaciju protoka podataka i pronalaženje najkraćih komunikacionih puteva između uređaja u mreži. Na taj način doprinosi optimizaciji protokola za rutiranje u mrežama.
- Bioinformatika: U oblastima kao što su genomska analiza ili istraživanje proteinskih interakcija, grafovi se koriste za modeliranje kompleksnih bioloških sistema, Floyd-Warshall algoritam može pomoći u analizi najkraćih puteva između različitih bioloških entiteta (npr. između gena ili proteina), čime se olakšava razumjevanje složenih mreža interakcija u organizmima.

## 4. Programsko rješenje

U ovom poglavlju izložena je implementacija Floyd-Warshall algoritma kroz konzolnu aplikaciju za pronalaženje matrice najkraćih udaljenosti u grafu. Aplikacija je iskodirana u C programskom jeziku, korištenjem Microsoft Visual Studio 2022 programsko okruženje. Izrada aplikacije je započeta kreiranjem novog projekta u pomenutom programskom okruženju i uključivanjem openMP biblioteke na način koji je naveden u prethodnim poglavljima rada. Upotrebljena je openMP tehnologija za paralelizaciju jer omogućava jednostavno uvođenje niti u postojeći kod, s fokusom na skraćanje vremena izvršenja algoritma, a poseban akcenat je stavljen na razumno korišćenje memorije i resursa.

### 4.1 Opis aplikacije

Konzolna aplikacija je jednostavna za upotrebu. Nakon pokretanja aplikacije od korisnika se zahtjeva da unese broj čvorova grafa. Neophodno je unijeti pozitivan cijeli broj veći od 2, jer je to minimalan broj čvorova povezanog grafa. Broj čvorova grafa određuje dimenziju kvadratne matrice udaljenosti između čvorova. U slučaju da korisnik unese neodgovarajuću vrijednost broja čvorova omogućen je ponovni unos. Potom korisnik ima mogućnost da izabere da li želi ispis matrice najkraćih udaljenosti i da li će se program izvršavati na sekvencijalni ili paralelan način.



Slika 7 - pokrenuta aplikacija i slučaj zahtjeva za ponovnim unosom

```
floydWarshall
Unesite broj cvorova grafa: 100
Da li zelite ispis matrice najkracih udaljenosti (0 - Ne, 1 - Da)?
>0
Izaberite nacin izvršavanja:
- 1 - sekvencijalno izvršavanje
- 2 - paralelno izvršavanje
>
```

Slika 8 - Izgled pokrenute konzolne aplikacije i slučaj kada je korisnik unio odgovarajući broj

U kodu koji je priložen na sljedećoj stranici implementirana je *main()* funkcija. Prvo su uključene standardne biblioteke pomoću direktive *#include*. Biblioteka *omp.h* je namjenjena planiranoj paralelnoj implementaciji, *stdio.h* je osnovna biblioteka programskog jezika C, *stdlib.h* omogućuje rad sa dinamičkom memorijom, a *windows.h* je iskorištena samo zbog postavljanja naziva aplikacije pomoću *SetConsoleTitle* funkcije iz te biblioteke. Nakon toga su navedene definicije funkcija. Ovaj dio koda se izvršava sekvencijalno. Na osnovu korisničkog unosa alokira se memorija za matricu u vidu jednodimenzinalnog niza i matrica se popunjava po pozivu odgovarajuće funkcije.

```
void popuniMatricu(int* matrica, int n) {
    int i, j, vrijednost;

    //seed-ovanje pseudoslučajnog niza brojeva
    srand(28);

    for (i = 0; i < n; i++) {
        for (j = 0; j <= i; j++)
        {
            if (i == j)
                matrica[i * n + j] = 0;
            else
            {
                vrijednost = 1 + rand() % n;
                matrica[i * n + j] = vrijednost;
                matrica[j * n + i] = vrijednost;
            }
        }
    }
}
```

Kodni listing 2 - pomoćna funkcija za popunjavanje matrice

```

int main() {
    SetConsoleTitle(L"floydWarshall");
    int n, odabir, odgovor;
    while (1)
    {
        do {
            printf("Unesite broj cvorova grafa: ");
            scanf("%d", &n);
            if (n < 2)
                printf("\nMinimalan broj cvorova povezanog grafa je
2. Ponovite unos.\n");
        } while (n < 2);
        int* matricaUdaljenosti;
        matricaUdaljenosti = (int*)malloc((n * n) * sizeof(int));
        popuniMatricu(matricaUdaljenosti, n);
        printf("\nDa li zelite ispis matrice najkracih udaljenosti (0 -
Ne, 1 - Da)?\n>");
        scanf("%d", &odgovor);
        printf("\nIzaberite nacin izvorsavanja:\n - 1 - sekvencijalno
izvorsavanje\n - 2 - paralelno izvorsavanje\n>");
        scanf("%d", &odabir);
        switch (odabir)
        {
            case 1:
                sekvencijalniAlgoritam(matricaUdaljenosti, n, odgovor);
                break;
            case 2:
                paralelniAlgoritam(matricaUdaljenosti, n, odgovor);
                break;
            default:
                break;
        }
        free(matricaUdaljenosti);
    }
}

```

Kodni listing 3 - kod implementirane main funkcije

Funkciji *popuniMatricu* se proslijeđuje pokazivac na matricu udaljenosti grafa i broj čvorova grafa. Glavna dijagonala se popunjava nulama jer je udaljenost između nekog čvora i samog sebe 0. Ostatak matrice se popunjava random pseudoslučajnim pozitivnim vrijednostima, izimajući u obzir da su gornja i donja trougaona matrica simetrična. Pretpostavili smo da je graf potpuno povezan tj. da postoji grana između bilo koja dva čvora grafa i zato nema beskonačnih vrijednosti. Za dobijanje vrijednosti udaljenosti upotrebljena je funkcija *srand()* u C programskom jeziku koja koristi se za postavljanje sjemena (eng. *seed*) za generator slučajnih brojeva koji koristi funkcija *rand()*. Pošto je ovo aplikacija koja je fokusirana na ispitivanje algoritma pronalaska najkraćih udaljenosti nije u velikoj mjeri značajno koje se vrijednosti nalaze u matrici, jer matrica služi kao pokazni primjer, a korištenje ponovljenih vrijednosti olakšava testiranje. Po potrebi se može promijeniti broj koji se proslijeđuje *srand()* funkciji i način računanja sadržaja promjenljive *vrijednost*. Generator slučajnih brojeva svaki put daje istu sekvencu brojeva kada se program pokrene. Pseudo-slučajni brojevi nisu potpuno slučajni jer se

generišu deterministički na osnovu formule i početne vrijednosti (sjemena). Sav dosadašnji kod aplikacije se izvršava sekvencijalno.

## 4.2 Sekvencijalna implementacija algoritma

U sekvencijalnom kodu kod tradicionalnih programskih jezika naredbe se izvršavaju jedna za drugom po redoslijedu kako su napisane. Pri ovakvom izvršavanju se problem razdvaja na više diskretnih serija instrukcija, a u jednom trenutku se može izvršavati samo jedna instrukcija. U suštini, svaka operacija mora biti završena pre nego što sljedeća počne.

Obzirom da su prednosti i nedostaci paralelne obrade ranije izloženi, neophodno je razmotriti još pozitivne i negativne aspekte sekvencijalne obrade. Sekvencijalni programi su lakši za pisanje i održavanje koda, jer se izvršavanje događa po redoslijedu koji je predvidljiv. S tim u vezi je olakšano debugovanje i testiranje programa. Pošto nema paralelnih niti izbjegnuti su problemi kao što je zaglavljenje i ne treba razmišljati o sinhronizaciji i zaključavanju resursa. Determinizam je bitna karakteristika sekvencijalnog programa, što znači da se uvijek izvodi na isti način i daje iste rezultate svaki put kada se pokree, dok paralelni program može varirati u ponašanju zbog različitih redoslijeda izvršavanja niti.

Najočiglednije mane sekvencijalne obrade su ograničene performanse i dugotrajno izvršavanje složenih zadataka. Neefikasno se iskorištavaju hardverski resursi, npr. čak i na modernim računarima sa više jezgara, sekvencijalni program će koristiti samo jedno jezgro, dok ostala jezgra ostaju neiskorišćena. U aplikacijama gdje se zahtjeva brza obrada podataka u realnom vremenu često nije dovoljno brza da odgovori na zahtjeve za pravovremeno izvršavanje. Iako jednostavna i pouzdana, u današnje vreme, sa sve većim zahtjevima za performansama, često postaje ograničavajući faktor.

Funkcija *sekvencijalniAlgoritam* vrši pronalazak najkraćih putanja u matrici udaljenosti čiji se pokazivač prosljeđuje kao parametar funkcije. Parametar  $n$  je broj čvorova grafa, a od vrijednosti parametra  $odg$  zavisi da li će se konačna matrica ispisivati na konzoli. Promjenljive *pocetak* i *kraj* služe bilježenju početnog i završnog trenutka izvršavanja algoritma. Njihova vrijednost je dobijena zahvaljujući funkciji *omp\_get\_wtime()* iz biblioteke *omp.h* koja vraća vrijednost dvostruke preciznosti jednaku broju sekundi od početne vrijednosti sata realnog vremena operativnog sistema.

Algoritam za računanje najkraćih putanja se sastoji od tri ugniježdene petlje koje iteriraju kroz sve parove čvorova u grafu. Spoljna petlja sa promenljivom  $k$  prolazi kroz sve čvorove, što omogućava ispitivanje da li postoji kraći put između bilo koja dva čvora  $i$  i  $j$  putem čvora  $k$ .

Unutar unutrašnjih petlji, uvijek se preskaču elementi na glavnoj dijagonali jer je rastojanje od bilo kojeg čvora do samog sebe uvijek 0. Zatim, kod provjerava da li je pronađen kraći put između čvorova  $i$  i  $j$  preko čvora  $k$ , te ako jeste, vrijednost u matrici se ažurira, odnosno postavlja se na novu, manju vrednost. Ako kraći put ne postoji, vrednost matrice u presijeku  $i$ -te vrste i  $j$ -te kolone ostaje nepromjenjena.



```

void sekvencijalniAlgoritam(int* matrica, int n, int odg)
{
    int i, j, k; //promjenljive za iteraciju kroz matricu
    double pocetak, kraj; //promjenljive za mjerenje pocetnog i krajnjeg vremena
    //bilježenje pocetnog vremena
    pocetak = omp_get_wtime();
    for(k=0; k<n; k++)
        for(i=0; i<n; i++)
            for (j = 0; j < n; j++) {
                if (i == j) //preskace elemente na glavnoj dijagonali
                    continue;
                //azurira vrijednost matrice ako se pronadje kraci put
                else if (matrica[i * n + k] + matrica[k * n + j] <
matrica[i * n + j]) {
                    matrica[i * n + j] = matrica[i * n + k] + matrica[k
* n + j];
                }
                else; //ako nema kraceg puta, nista se ne mijenja.
            }
    //bilježenje završnog vremena
    kraj = omp_get_wtime();
    //poziv funkcije koja prikazuje matricu najkracih udaljenosti
    if (odg == 1)
    {
        printf("\n\nMatrica najkracih udaljenosti:\n\n");
        ispisMatrice(matrica, n);
    }

    //racunanje vremena izvršenja algoritma i ispis rezultata u milisekundama
    printf("\n\nVrijeme izvršenja algoritma: %.6f sekundi.\n\n", kraj - pocetak);
}

```

Kodni listing 4 - sekvencijalna implementacija Floyd Warshall algoritma

Zavisno od prethodnog izbora korisnika, ispis sadržaja matrice najkraćih udaljenosti se vrši ili ne vrši, a potom se ispiše ukupno vrijeme izvršenja algoritma reda milisekundi ( $10^{-6}$  s). Pri bilježenju krajnjeg vremena uzet je trenutak kada je završena posljednja iteracija petlji Floyd-Warshall algoritma, jer je opšte poznato da u slučaju da je potrebno ispisati konačnu matricu najviše vremena bi bilo utrošeno na ispis pojedinačnih vrijednosti iz matrice najkraćih udaljenosti na konzolu.

```

void ispisMatrice(int* matrica, int n) {
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            printf("\t%d", matrica[i * n + j]);
        printf("\n");
    }
}

```

Kodni listing 5 - funkcija za ispis vrijednosti iz matrice

### 4.3 Paralelna implementacija algoritma

```
void paralelniAlgoritam(int* matrica, int n, int odg) {
    int i, j, k;
    int brojNiti = omp_get_max_threads();
    double pocetnoVrijeme, krajnjeVrijeme;

    printf("\nImate %d niti na raspolaganju.\n", brojNiti);
    double* vrijemeNiti = (double*)malloc(brojNiti * sizeof(double));
    for (i = 0; i < brojNiti; i++) {
        vrijemeNiti[i] = 0.0;
    }
    omp_set_num_threads(brojNiti);
    pocetnoVrijeme = omp_get_wtime(); //pocetak mjerenja vremena rada algoritma
#pragma omp parallel private(i,j,k)
    {
        int idNiti = omp_get_thread_num();
        double pocetakRadaNiti, zavrstakRadaNiti;

        for (k = 0; k < n; k++)
        {
#pragma omp for schedule(static)
            for (i = 0; i < n; i++)
            {
                pocetakRadaNiti = omp_get_wtime(); //nit pocinje sa radom sa i-
                tim redom
                for (j = 0; j < n; j++)
                {
                    if (i == j)
                        continue;
                    else if (matrica[i * n + k] + matrica[k * n + j] <
matrica[i * n + j])
                    {
                        matrica[i * n + j] = matrica[i * n + k] +
matrica[k * n + j];
                    }
                    else;
                }
                zavrstakRadaNiti = omp_get_wtime();
                vrijemeNiti[idNiti] += zavrstakRadaNiti - pocetakRadaNiti;
            }
        }
        krajnjeVrijeme = omp_get_wtime(); //biljezenje zavrsetka izvršavanja
        algoritma
        if (odg == 1)
        {
            printf("\n\nMatrica najkracih udaljenosti:\n\n");
            ispisMatrice(matrica, n);
        }
        printf("\nVrijeme paralelnog izvršenja algoritma: %.6f sekundi.\n",
krajnjeVrijeme-pocetnoVrijeme);
        for (i = 0; i < brojNiti; i++) {
            printf("Nit %d je bila aktivna %.6f sekundi -> (%.2f%%) ukupnog
vremena.\n", i+1, vrijemeNiti[i], (vrijemeNiti[i] / (krajnjeVrijeme-pocetnoVrijeme))
* 100.0);
        }
        free(vrijemeNiti);
    }
}
```

Kodni listing 6 - paralelna implementacija Floyd - Warshall algoritma

Funkcija *paralelniAlgoritam* implementira paralelizovanu verziju Floyd-Warshall algoritma. Pored promjenljivih za iteraciju kroz petlje i bilježenje vremena početka i završetka izvršenja kod, deklarirana je promjenljiva *brojNiti* i njena vrijednost je inicijalizovana pomoću funkcije *omp\_get\_max\_threads()*. Pomoću ove funkcije se sačuva ukupan broj niti koje posjeduje računar na kom je aplikacija pokrenuta. Informacija o broju niti se potom ispiše na konzoli.

Da bi se mogli čuvati podaci o tome koja nit je koliko bila aktivna koristi se dinamički niz *vrijemeNiti*. Niz se alokira za onoliko niti koliko računar posjeduje i za početak se svaki element niza inicijalizuje na vrijednost 0. Takođe zbog iste namjene, na početku svake niti, nit dobija svoj jedinstveni identifikator pomoću funkcije *omp\_get\_thread\_num()*. Ovaj identifikator (*idNiti*) koristi se za praćenje vremena koje svaka nit provodi na svom dijelu posla, i njihovo vrijeme aktiviranja i deaktiviranja se čuva kao vrijednost promjenljivih *pocetakRadaNiti* i *zavrsetakRadaNiti*. Upotrebom funkcije *omp\_set\_num\_threads()* iz biblioteke *omp.h* postavi se da je broj niti koji će učestvovati u narednoj paralelnoj sekciji jednak maksimalnom broju niti, a to je *brojNiti*.

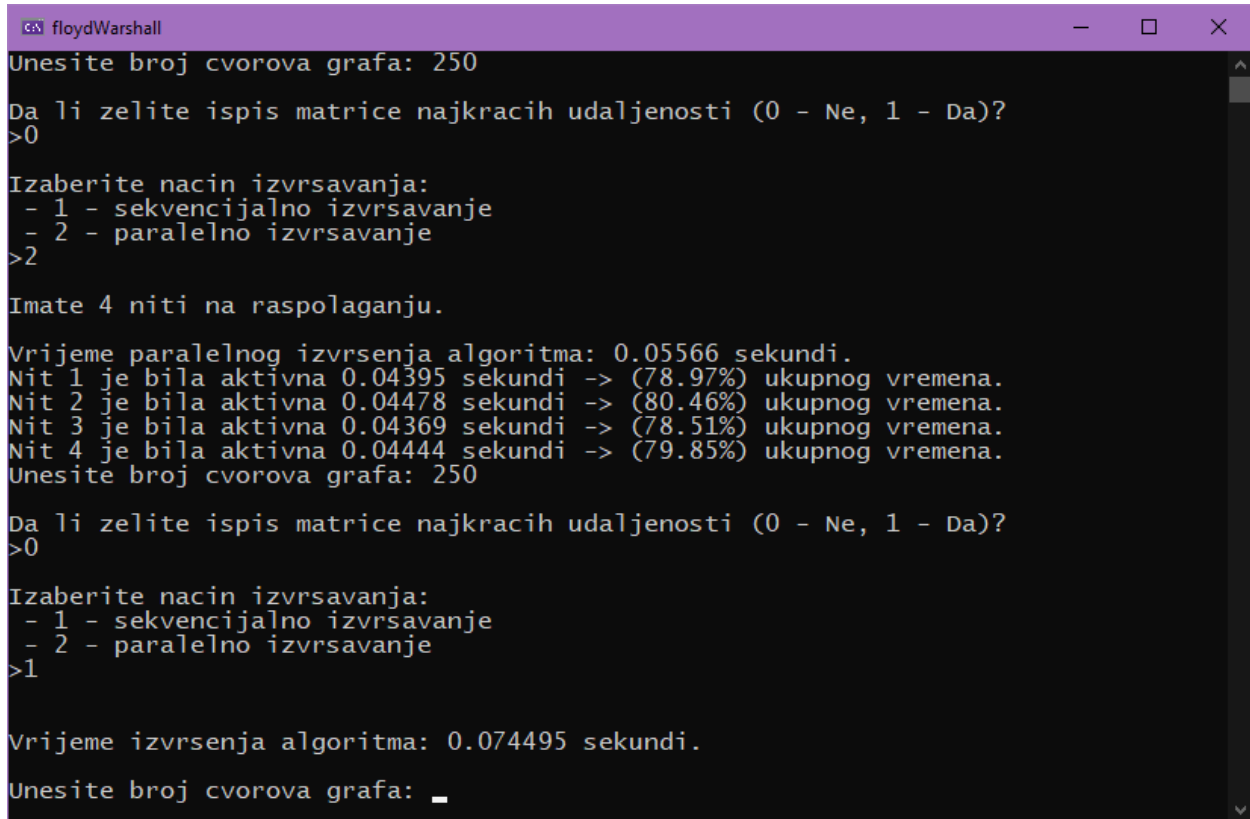
**#pragma omp parallel private(i,j,k)** direktiva označava početak paralelne sekcije, odnosno da naglasi da će se naredni kod višenitno izvršavati i svaka nit ima svoju kopiju privatnih promenljivih *i*, *j* i *k*. Deklaracija promenljivih kao privatnih (*private*) u kontekstu paralelizma donosi nekoliko prednosti. Spriječeni su sukobi koji mogu nastati kada više niti pokušava istovremeno da pristupi ili modifikuje istu promenljivu. Niti ne moraju da sinhronizuju pristup privatnim promenljivama, što znači da se smanjuje potreba za dodatnim mehanizmima sinhronizacije, poput zaključavanja, što može poboljšati ukupne performanse programa.

**#pragma omp for schedule(static)** direktiva naglašava paralelizaciju unutrašnjih petlji Floyd-Warshall algoritma. Zadatak se u *for* petlji dijeli između niti koristeći statičko raspoređivanje posla. Ova podjela se vrši jednom, prije početka izvršenja petlje, i ostaje fiksna tokom daljeg izvršavanja. Statičko raspoređivanje je efikasno kada su iteracije petlje približno jednako zahtjevne u smislu vremena izvršenja. U ovom slučaju, svaka nit će imati približno isto radno opterećenje, što vodi do uravnoteženog i efikasnog paralelnog izvršenja. Kod Floyd-Warshall algoritma svaka iteracija petlje ima sličan posao jer se u svakoj iteraciji ažurira matrica udaljenosti sa sličnim brojem operacija. Zato je statička raspodjela bolja opcija u odnosu na dinamičku, jer sve niti imaju sličan broj iteracija i obavljaju podjednako težak posao.

Svaka nit dobija određeni broj vrsta matrice na obradu. Prije nego što počne da obrađuje jednu vrstu matrice, nit bilježi vrijeme početka. Unutar svake dodijeljene iteracije spoljne petlje (za svaku vrijednost *i*), nit prolazi kroz sve vrijednosti *j* i računa najkraći put između čvorova *i* i *j*, koristeći posredni čvor *k*. Ako se nađe kraći put između čvorova *i* i *j*, nit ažurira matricu sa novom, manjom vrednošću. Proces se ponavlja za sve vrijednosti *j*, osim za one na glavnoj dijagonali (kada je *i == j*), jer se rastojanje od čvora do samog sebe ne mijenja. Na kraju svake iteracije, nit izračunava koliko je vremena provela radeći na svom zadatku, oduzimajući vrijeme početka rada od vremena završetka. Ovo vrijeme se zatim dodaje ukupnom vremenu koje je nit provela aktivno radeći, i zapisuje se u niz *vrijemeNiti[]*. Zaključujemo da su zadaci svake pojedinačne niti su da obradi dobijeni dio matrice, ažurira vrijednosti za svoje vrste, i bilježi vrijeme koje je provela u radu.

Na kraju se vrši ispis konačnih vrijednosti najkraćih udaljenosti između čvorova, ako je to korisnik odabrao, zatim ukupno trajanje izvršavanja algoritma te vrijeme rada i procentualno

učešće aktivnog vremena svake pojedinačne niti u ukupnom vremenu izvršavanja. I u ovom slučaju je pri bilježenju krajnjeg vremena uzet trenutak kada je završena posljednja iteracija petlji Floyd-Warshall algoritma, a ne vrijeme nakon ispisa sadržaja matrice najkraćih udaljenosti na konzolu.



```
floydWarshall
Unesite broj cvorova grafa: 250
Da li zelite ispis matrice najkracih udaljenosti (0 - Ne, 1 - Da)?
>0
Izaberite nacin izvorsavanja:
- 1 - sekvencijalno izvorsavanje
- 2 - paralelno izvorsavanje
>2
Imate 4 niti na raspolaganju.
Vrijeme paralelnog izvorsenja algoritma: 0.05566 sekundi.
Nit 1 je bila aktivna 0.04395 sekundi -> (78.97%) ukupnog vremena.
Nit 2 je bila aktivna 0.04478 sekundi -> (80.46%) ukupnog vremena.
Nit 3 je bila aktivna 0.04369 sekundi -> (78.51%) ukupnog vremena.
Nit 4 je bila aktivna 0.04444 sekundi -> (79.85%) ukupnog vremena.
Unesite broj cvorova grafa: 250
Da li zelite ispis matrice najkracih udaljenosti (0 - Ne, 1 - Da)?
>0
Izaberite nacin izvorsavanja:
- 1 - sekvencijalno izvorsavanje
- 2 - paralelno izvorsavanje
>1
Vrijeme izvorsenja algoritma: 0.074495 sekundi.
Unesite broj cvorova grafa: _
```

Slika 9 - primjer izvršavanja aplikacije

## 5. Testiranje

Testiranje aplikacije izvršeno je na dva različita računara. Cilj testiranja bio je da se uporede performanse aplikacije u različitim hardverskim okruženjima i da se analizira kako se ponašanje algoritma mijenja u zavisnosti od dostupnih resursa. Mjerenja su vršena na računarima različitih specifikacija kako bi se obuhvatili različiti procesorski kapaciteti i broj jezgara, što omogućava detaljnu analizu paralelne obrade i skalabilnosti rešenja. Takođe namjera testiranja je dokazivanje da paralelni način obrade skraćuje vrijeme izvršavanja zadatka u odnosu na sekvencijalno izvršavanje koda, te da veći broj niti stvara značajnu razliku u rezultatima. Na osnovu rezultata dobijenih nakon testiranja koda lakše je dobiti jasnu sliku u kojim slučajevima je bolje koristiti sekvencijalnu obradu, a kada paralelnu obradu.

Testiranje koda i mjerenje vremena izvršavanja je obavljeno za naredni broj čvorova grafa:

10, 50, 100, 250, 500, 1000, 2000 i 5000 čvorova.

### 5.1 Prvo testiranje

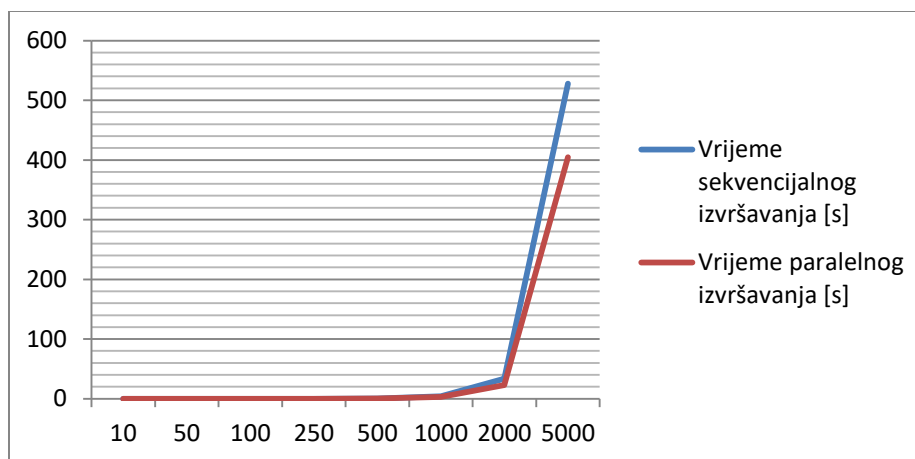
Prvo testiranje je obavljeno na laptop računaru Lenovo T440 koji posjeduje 2 procesorska jezgra i 4 niti, te sljedeće specifikacije:

- Operativni sistem: *Windows 10 Pro*
- Procesor: *Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz 2.49 GHz*
- RAM: *8,00 GB (7,69 GB upotrebljivo)*
- Grafička kartica: *Intel (R) Graphics family*

Rezultati izvršavanju su dati u narednoj tabeli:

Broj čvorova	Vrijeme sekvencijalnog izvršavanja [s]	Vrijeme paralelnog izvršavanja [s]
<b>10</b>	0.00001	0.004698
<b>50</b>	0.000911	0.001
<b>100</b>	0.005023	0.00384
<b>250</b>	0.071925	0.051428
<b>500</b>	0.60215	0.369864
<b>1000</b>	4.316286	2.9531
<b>2000</b>	33.763259	22.953227
<b>5000</b>	527.69813	404.772939

Tabela 1 - rezultati prvog testiranja vremena izvršavanja algoritma



Slika 10 - grafički prikaz rezultata prvog testiranja

## 5.2 Drugo testiranje

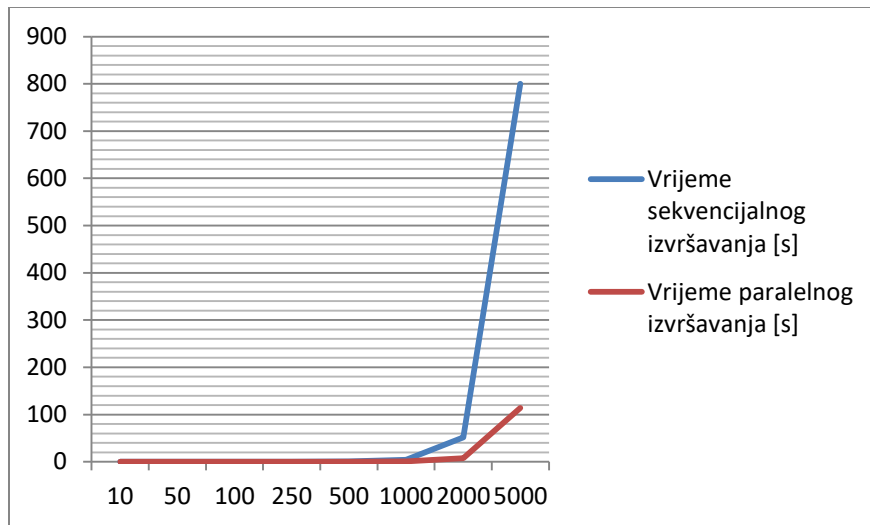
Drugo testiranje je obavljeno na laptop računaru Lenovo IdeaPad3 15ALC6 koji posjeduje 8 procesorskih jezgara i 16 niti, te sljedeće specifikacije:

- Operativni sistem: *Windows 10 Pro*
- Procesor: *AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz*
- RAM: *16,00 GB (13,8 GB upotrebljivo)*
- Grafička kartica: *AMD Radeon(TM) Graphics*

Rezultati izvršavanju su dati u narednoj tabeli:

Broj čvorova	Vrijeme sekvencijalnog izvršavanja [s]	Vrijeme paralelnog izvršavanja [s]
<b>10</b>	0.00001	0.0011021
<b>50</b>	0.000109	0.001163
<b>100</b>	0.007672	0.003888
<b>250</b>	0.112165	0.04661
<b>500</b>	0.84674	0.26574
<b>1000</b>	4.316286	1.14624
<b>2000</b>	51.715785	7.721513
<b>5000</b>	800.178412	113.971538

Tabela 2 - rezultati drugog testiranja vremena izvršavanja algoritma



Slika 11 - grafički prikaz rezultata drugog testiranja

### 5.3 Analiza rezultata testiranja

Na osnovu brojnih vrijednosti i grafičkog prikaza rezultata testiranja moguće je izvesti određene zaključke. Za manje od 10 čvorova, tj. za matricu udaljenosti grafa čija je dimenzija manja od 10 vrijeme sekvencijalnog izvršavanja je manje od 1 ms, a vrijeme paralelnog izvršenja je reda nekoliko milisekundi pri testiranju na oba uređaja. Za slučajeve u kojima je broj čvorova grafa manji od 100, u oba testiranja, bolje se pokazalo sekvencijalno izvršavanje Floyd-Warshall algoritma za pronalazak najkraćih putanja u grafu, s obzirom da su vremena izvršavanja kraća nego pri upotrebi paralelne obrade. To je vjerovatno zbog manje količine podataka koje je potrebno obraditi i više je vremena utrošeno na podjelu posla između niti nego na izračunavanja. Pri prvom testiranju je i za slučaj sa 100 čvorova vrijeme sekvencijalne obrade bilo kraće nego vrijeme paralelne obrade, međutim pri drugom testiranju vrijeme paralelne obrade je bilo već upola kraće. U svim ostalim slučajevima sa brojem čvorova većim od 100, vrijeme paralelnog izvršavanja je kraće nego vrijeme sekvencijalnog. Ovo je posebno izraženo u drugom testiranju na računaru koji posjeduje 16 niti. Očigledno je da u pojedinačnim slučajevima vrijeme paralelne obrade biva upola kraće ili čak nekoliko puta kraće u odnosu na vrijeme sekvencijalne obrade.

Zanimljivo je primjetiti da je sekvencijalno izvršavanje bilo brže pri prvom testiranju. To je vjerovatno zato što broj jezgara nije relevantan, jer sekvencijalni algoritam koristi samo jedno jezgro, ali performanse pojedinačnog jezgra su presudne, što može zavisiti od same arhitekture procesora.

Tokom testiranja pri paralelnoj obradi takođe se može primjetiti, na osnovu ispisa vremena rada pojedinačnih niti, da su sve niti približno jednako aktivne. Rezultati pokazuju relativno ujednačena vremena rada za sve niti, što je siguran znak da je opterećenje među nitima bilo prilično dobro raspoređeno.

Sve pomenuto ukazuje na to da je paralelno izvršavanje algoritma povoljniji izbor za grafove sa većim brojem čvorova.

## **Zaključak**

U seminarskom radu predstavljen je kompletan proces paralelizacije Floyd-Warshall algoritma korišćenjem OpenMP tehnologije. Kroz teorijsku analizu i kodnu implementaciju prikazana su poboljšanja u brzini izvršavanja paralelnog algoritma u poređenju sa njegovom sekvencijalnom verzijom. Testiranja su obavljena na dva različita računara kako bi se ispitale performanse u različitim hardverskim okruženjima, pri čemu su paralelne verzije algoritma pokazale značajno ubrzanje, posebno na računarima sa više jezgara.

Glavni zaključak istraživanja je da je paralelizacija Floyd-Warshall algoritma korisna za guste grafove sa velikim brojem čvorova. Paralelizacija može biti nepoželjna ili čak kontraproduktivna kada graf ima mali broj čvorova, jer se koristi relativno malo računskih resursa. Troškovi koji nastaju zbog paralelizacije, a to su kreiranje i upravljanje nitima, mogu biti veći od dobitaka u brzini izvršavanja. Tada je poželjnije koristiti sekvencijalni algoritam. Ako je graf specifično strukturiran, tako da je većina čvorova već direktno povezana, ili postoji vrlo malo putanja koje treba ažurirati, veći dio rada u unutrašnjim petljama algoritma sastoji od provjera koje ne zahtevaju puno obrade. Paralelizacija može biti nepotrebna i u tom slučaju.

Budući rad može biti fokusiran na dalju optimizaciju algoritma uz korišćenje hibridnog modela paralelizacije, gdje bi OpenMP bio kombinovan sa drugim tehnologijama, kao što je MPI. Potrebno je istražiti primenjenu optimizovanog algoritma na realnim problemima, kao što su mrežni protokoli, analiza društvenih mreža ili planiranje logističkih ruta.



## Literatura

- [1] Miloš Ivanović, *Paralelno programiranje – skripta sa primjerima*, Prirodno-matematički fakultet Univerziteta u Kragujevcu, Kragujevac, 2016
- [2] Wikipedia, Paralelna obrada, pristupano 6.9.2024.  
[https://sr.wikipedia.org/wiki/Паралелна\\_обрада](https://sr.wikipedia.org/wiki/Паралелна_обрада)
- [3] Laslo Kraus, *Programski jezik C sa rešenim zadacima*, Akademska misao, Beograd, 2006
- [4] Skillicorn, David B., *Models for practical parallel computation*, International Journal of Parallel Programming, 20.2 133–158 (1991)
- [5] Suzana Filipović, *Niti*, 18.10.2019. pristupano 6.9.2024.  
<https://racunariprogramiranje.wordpress.com/2019/10/18/нити>
- [6] Research Computing, *Using OpenMP with C*, pristupano 12.9.2024.  
<https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html>
- [7] David Bertoldi, *Parallelization of the Floyd-Warshall algorithm*, Department of Informatics, Systems and Communication, University of Milano-Bicocca

## **Dodaci**

### **Popis slika**

Slika 1 – Jedna nit procesa i više niti istog procesa .....	10
Slika 2 - Uredni prikaz različitih višenitnih modela .....	11
Slika 3 - ključne riječi u programskom jeziku C .....	14
Slika 4 - izgled kreiranog projekta u Visual Studio 2022 IDE .....	24
Slika 5 - kreiranje konzolne aplikacije u kroz Visual Studio 2022 IDE .....	25
Slika 6 - pseudokod Floyd - Warshall algoritma .....	26
Slika 7 - pokrenuta aplikacija i slučaj zahtjeva za ponovnim unosom .....	29
Slika 8 - Izgled pokrenute konzolne aplikacije i slučaj kada je korisnik unio odgovarajući broj .....	30
Slika 9 - primjer izvršavanja aplikacije .....	36
Slika 10 - grafički prikaz rezultata prvog testiranja.....	38
Slika 11 - grafički prikaz rezultata drugog testiranja.....	39

### **Popis tabela**

Tabela 1 - rezultati prvog testiranja vremena izvršavanja algoritma .....	37
Tabela 2 - rezultati drugog testiranja vremena izvršavanja algoritma .....	38

### **Popis kodnih listinga**

Kodni listing 1 - primjer switch naredbe .....	19
Kodni listing 2 - pomoćna funkcija za popunjavanje matrice .....	30
Kodni listing 3 - kod implementirane main funkcije .....	31
Kodni listing 4 - sekvencijalna implementacija Floyd Warshall algoritma.....	33
Kodni listing 5 - funkcija za ispis vrijednosti iz matrice .....	33
Kodni listing 6 - paralelna implementacija Floyd - Warshall algoritma .....	34