|  |  |  |
| --- | --- | --- |
|  | Univerzitet u Istočnom Sarajevu  Elektrotehnički fakultet |  |

**Paralelizacija Floyd – Warshall algoritma**

Seminarski rad

Studije: Prvi ciklus

Odsjek: Računarstvo i informatika

Predmet: Paralelni računarski sistemi

Studenti: Mentor:

Kristina Knežević, 2060 Doc. dr Nikola Davidović

Tamara Elez, 2068

Istočno Sarajevo, septembar 2024

# **Sadržaj**

[Uvod 3](#_Toc176812865)

[Paralelno računarstvo 4](#_Toc176812866)

[Tipovi paralelizma 4](#_Toc176812867)

[Strategije paralelnog programiranja 5](#_Toc176812868)

[Prednosti paralelizacije 6](#_Toc176812869)

[Nedostaci paralelizacije 7](#_Toc176812870)

[Paralelni modeli 8](#_Toc176812871)

[Višenitnost 8](#_Toc176812872)

[Problemi višenitnosti 11](#_Toc176812873)

[Programsko rješenje 11](#_Toc176812874)

[Opis aplikacije 12](#_Toc176812875)

[Sekvencijalna implementacija 14](#_Toc176812876)

[Paralelna implementacija 17](#_Toc176812877)

[**Literatura** 20](#_Toc176812878)

## Uvod

Sadašnje stanje razvoja integrisanih kola ukazuje da ubrzavanje mikroprocesora jednostavnim povećanjem radne frekvencije više neće biti moguće. Tome doprinose fizička ograničenja minijaturizacije, transmisiona granica bakarnog provodnika itd. 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 što ga čini računski intenzivnim i vremenski zahtjevnim za grafove sa velikim brojem čvorova. Paralelizacija ovog algoritma predstavlja značajan pokušaj ka povećanju efikasnosti i smanjenju vremena izvršavanja.

U radu istražujemo metode i tehnike paralelizacije Floyd-Warshall algoritma implementiranog u programskom jeziku C, koristeći OpenMP tehnologiju. Rad će biti strukturisan tako da prvo razmotrimo osnovne koncepte paralelizacije i tehnologije 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, pružićemo detaljan pregled algoritma, uključujući njegovu funkcionalnost i primjenu. U nastavku rada detaljno ćemo opisati 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.

# Paralelno računarstvo

Paralelno računarstvo predstavlja brže rješavanje problema korišćenjem većeg broja procosora.

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 višeprocesorski računari imaju više elemenata obrade unutar jedne mašine, a klasteri koristite 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.

# 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. Pojedinosti o nitima će biti pojašnjene u narednim polgavljima 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 višeprocesorskom 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. Stepen 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.

# 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.

# 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 višejezgrenim 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. Konkretan 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 energetsku efikasnost 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 višejezgarni procesori su dizajnirani da budu energetski efikasniji i preuzimaju manje zahtjevne zadatke, dok se brža jezgra koriste za intenzivnije zadatke.

# 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.

# 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.

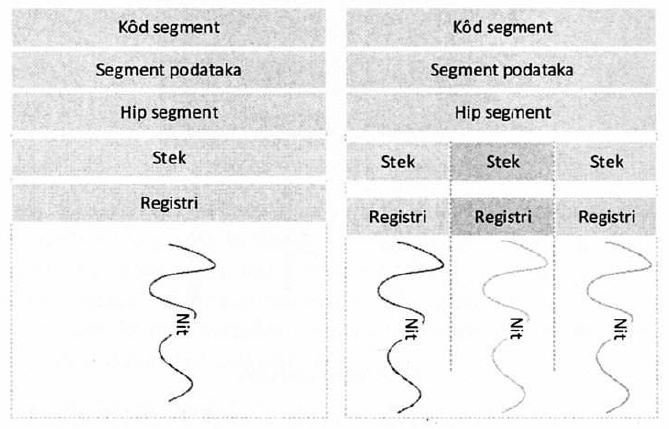
Hibridni modeli kombinuju karakteristike dijeljene i distribuirane memorije. Na primer, na nivou jednog računara (višejezgarni procesor) koristi se model sa dijeljenom memorijom, dok se između različitih računara (u klasteru) koristi model sa distribuisanom memorijom. Kombinacija MPI i OpenMP u sistemima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

# Višenitnost

Paradigma višenitne obrade je postala popularnija otkako je napredak na polju paralelizma na nivou instrukcije dostigao vrhunac krajem 1990-ih. Višenitnost 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. Višenitnost 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 promjenljive, 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 promašaja keša, druge niti mogu nastaviti, koristeći prednost neiskorišćenih 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 nastave 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 blokiran, 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 višeprocesorskih 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

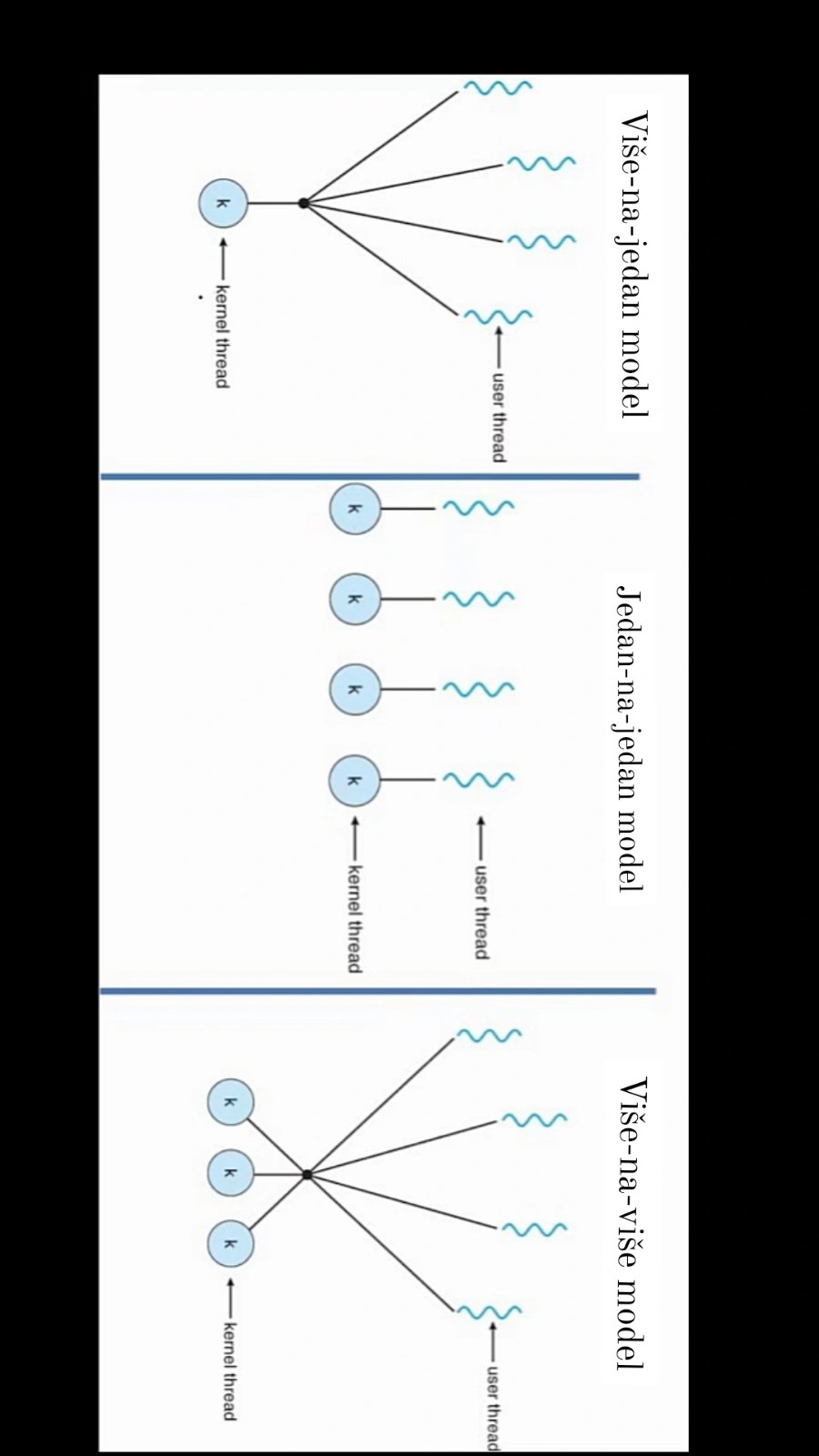
Implementacije modela više-na-jedan omogućavaju aplikaciji da kreira bilo koji broj niti koje se mogu izvršavati istovremeno. U ovakvoj implementaciji sve aktivnosti niti su ograničene na korisnički prostor. Pored toga, samo jedna po jedna nit može pristupiti jezgru, tako da je operativnom sistemu poznat 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

# 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.

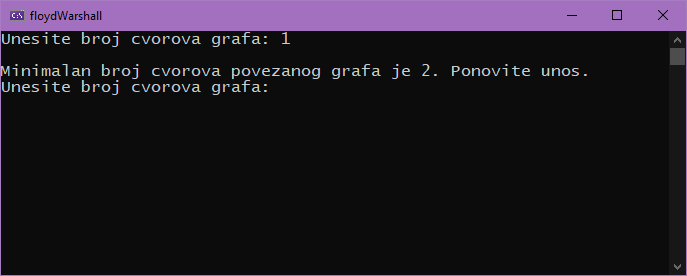
U raznim programskim jezicima i okruženjima postoje različiti načini da jedna nit „zabrani“ drugim da pristupe određenom resursu dok ta nit ne završi što ima da završi. Na primjer, u OpenMP-u biblioteci, sinhronizacija između niti i zabrana pristupa zajedničkim resursima dok jedna nit obavlja određeni 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.

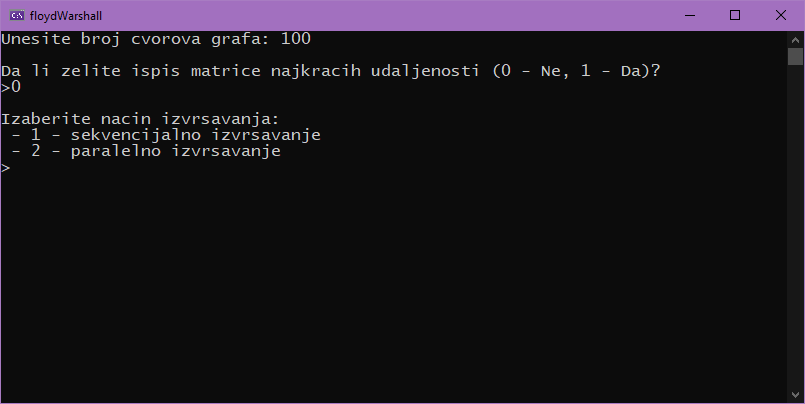
# Programsko rješenje

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. Upoterebljena je openMP tehnologija za paralelizaciju jer omogućava jednostavno uvođenje niti u postojeći kod, s fokusom na skraćenje vremena izvršenja algoritma, a poseban akcenat je stavljen na razumno korišćenje memorije i resursa.

# 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 3 - Izgled pokrenute aplikacije i slučaj kada se zahtjeva ponovni unos**

Slika 4 - Izgled pokrenute konzolne aplikacije i slučaj kada je korisnik unio odgovarajući broj

int main() {

SetConsoleTitle(L"floydWarshall");

int n, odabir, odgovor; //promjenljive za cuvanje broja cvorova grafa i korisnickih odabira

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);

//dinamicka alokacija prostora za matricu

int\* matricaUdaljenosti;

matricaUdaljenosti = (int\*)malloc((n \* n) \* sizeof(int));

//popunjavanje matrice udaljenosti brojnim vrijednostima

popuniMatricu(matricaUdaljenosti, n);

printf("\nDa li zelite ispis matrice najkracih udaljenosti (0 - Ne, 1 - Da)?\n>");

scanf("%d", &odgovor);

printf("\nIzaberite nacin izvrsavanja:\n - 1 - sekvencijalno izvrsavanje\n - 2 - paralelno izvrsavanje\n>");

scanf("%d", &odabir);

switch (odabir)

{

case 1:

//poziv funkcije za sekvencijalno izvrsavanje Floyd - Warshall algoritma

sekvencijalniAlgoritam(matricaUdaljenosti, n, odgovor);

break;

case 2:

//poziv funkcije za paralelno izvrsavanje Floyd - Warshall algoritma

paralelniAlgoritam(matricaUdaljenosti, n, odgovor);

break;

default:

break;

}

//oslobadjanje zauzete memorije

free(matricaUdaljenosti);

}

}

void popuniMatricu(int\* matrica, int n) {

int i, j, vrijednost;

//seed-ovanje pseudoslucajnog 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 - funkcija za popunjavanje matrice

Kodni listing 1 - kod koji implementira main funkciju

U priloženom kodu su prvo 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 alocira se memorija za matricu u vidu jednodimenzinalnog niza i matrica se popunjava po pozivu odgovarajuće funkcije.

Funkciji *popuniMatricu* se proslijeđuje pokazivac na matricu udaljenosti grafa i broj čvorova grafa. Glavna dijagonala se popunjava nulama jer je udaljenost izmedju nekog čvora i samog sebe 0. Ostatak matrice se popunjava random pseudoslucajnim pozitivnim vrijednostima, izimajući u obzir da su gornja i donja trougaona matrica simetrična. Pretpostavili smo da je graf potpuno povezan tj. da postoji grana izmedju 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 terstiranje. 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 dosadašanji kod aplikacije se izvršava sekvencijalno.

# Sekvencijalna implementacija

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 debagovanje 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.

Kodni listing 3 - sekvencijalna implementacija Floyd-Warshall algoritma

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

//biljezenje pocetnog vremena

pocetak = omp\_get\_wtime();

//sekvencijalna implementacija Floyd-Warshall algoritma

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.

}

//biljezenje zavrsnog 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 izvrsenja algoritma i ispis rezultata u milisekundama

printf("\n\nVrijeme izvrsenja algoritma: %.6f sekundi.\n\n", kraj - pocetak);

}

Funkcija *sekvencijalniAlgoritam* vrši pronalazak najkraćih putanja u matrici udaljenosti čiji se pokazivač proslijeđuje 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.

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.

Kodni listing 4 - kod funkcije za ispis sadržaja matrice

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");

}

}

# Paralelna implementacija

void paralelniAlgoritam(int\* matrica, int n, int odg) {

int i, j, k;

int brojNiti = omp\_get\_max\_threads(); //inicijalizacija promjenljive u kojoj se cuva ukupan broj niti procesora

double pocetnoVrijeme, krajnjeVrijeme;

printf("\nImate %d niti na raspolaganju.\n", brojNiti);

//dinamicka alokacija niza u kom ce se cuvati vrijeme aktivnosti pojedinacnih niti

double\* vrijemeNiti = (double\*)malloc(brojNiti \* sizeof(double));

//inicijalizacija niza

for (i = 0; i < brojNiti; i++) {

vrijemeNiti[i] = 0.0;

}

omp\_set\_num\_threads(brojNiti); //postavljanje broja niti koje ce se koristiti za sljedeci paralelni region

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(); //zavrsetak rada niti sa i-tim redom

vrijemeNiti[idNiti] += zavrstakRadaNiti - pocetakRadaNiti; //izracunato vrijeme koje je nit aktivno provela dodaje se ukupnom vremenu rada te niti.

}

}

}

krajnjeVrijeme = omp\_get\_wtime(); //biljezenje zavrsetka izvrsavanja algoritma

if (odg == 1)

{

printf("\n\nMatrica najkracih udaljenosti:\n\n");

ispisMatrice(matrica, n);

}

printf("\nVrijeme paralelnog izvrsenja 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 5 - 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, deklarisana 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 alocira 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 zakljucavanja, što može poboljšati ukupne performanse programa.

**#pragma omp for schedule(static)** direktiva naglašava paralelizaciju unutrašnjih petlji Floyd-Warshall algoritma. Zadatak se u *for* perlji 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.

**Literatura**

[1] Miloš Ivanović, *Paralelno programiranje – skripta sa primjerima*, Prirodno-matematički fakultet Univerziteta u Kragujevcu, Kragujevac, 2016

[2] Wikipedia, Paralelna obrada, preuzeto 6.9.2024. sa <https://sr.wikipedia.org/wiki/Паралелна_обрада>

[3] Skillicorn, David B., *Models for practical parallel computation*, International Journal of Parallel Programming, 20.2 133–158 (1991)

[4] Suzana Filipović, *Niti,* 18.10.2019. pristupano 6.9.2024. <https://racunariprogramiranje.wordpress.com/2019/10/18/нити>