Grupa 5.10 - 2023

Elektrotehnički fakultet

Univerzitet u Istočnom Sarajevu

Predmet: Paralelni računarski sistemi

Tema: Paralelizam u algoritmima sortiranja

Autori: Milan Vlaški i Fejzullah Ždralović

Grupa 5.10 - 2023

Sadržaj

1. Uvod

U prvom dijelu ćemo reći nešto generalno o čitavom ovome dokumentu. Ovdje se može naći svrha postojanja dokumenta, definicije, akronimi, skraćenice, reference, kao i pregled dokumenta.

U drugom dijelu govorimo o paralelizmu. Pored paralelizma govorit ćemo i o konkurentnosti. Definisati ćemo ta dva pojma, te ćemo govoriti o razlikama između ta dva pojma. Također, u ovom dijelu je rečeno nešto više o prednostima i nedostacima paralelizma.

U trećem dijelu će biti nešto više riječi o Java programskom jeziku, o verzijama Jave, o instalaciji Jave na različitim operativnim sistemima, te o bibliotekama koje su korištene u ovome projektu kako bi se obezbjedio paralelni rad različitih algoritama za sortiranje.

Četvrti dio govori o Java Swingu. Govori o tome kakva je to biblioteka. Spomenut ćemo karakteristike i koncepte Swinga, te ćemo govoriti i o razlikama između Java Swinga i JavaFX-a.

U petom dijelu ćemo pričati o algoritmima za sortiranje. Konkretno za algoritme Insertion sort, Selection sort, Bubble sort, Merge sort, Quick sort. Vidjeti ćemo kako rade ti algoritmi, po kojem principu vrše sortiranje, te ćemo i slikovito prikazati kako to izgleda i šta se dešava u pozadini u memoriji.

Šesti dio objašnjava paralelizam u Javi. Govorit ćemo o Fork/Join framework-u za konkurentnost, o ForkJoinPool-u, o ForkJoinTask-u, te o predavanju zadatka ForkJoinPool-u.

Sedmi dio je rezervisan za paralelizam u algoritmima sortiranja, tako da ćemo prokomentarisati te algoritme i proći kroz njihov kod. Objasnit ćemo šta se dešava u paralelizaciji sa tim algoritmima i kako smo uspjeli da primijenimo paralelizaciju na te algoritme.

Osmi dio ovoga dokumenta govori o našoj aplikaciji i daje uputstvo za upotrebu. Tu ćemo moći saznati nešto više o zapažanjima u implementaciji.

1.1 Svrha

Cilj ovog projekta je vidjeti kako paralelizacija utiče na različite algoritme sortiranja. U ovom projektu smo implementirali pet algoritama za sortiranje (Bubble Sort, Insertion Sort, Selection Sort, Merge Sort i Quick Sort) i omogućili njihovo izvršavanje sekvencijalno (bez paralelizacije) i paralelno (sa paralelizacijom). Pošto ispitujemo uspješnost pojedinih algoritama sortiranja, bitna nam je njihova brzina izvršavanja, tako da prilikom izvršavanja svakog algoritma, mjeri se njegovo vrijeme i rezultat se ispisuje na ekran. Također smo napravili metode koje generišu članove u nizu, pa je moguće unijeti veličinu niza, kao i to da li će brojevi u nizu biti cjelobrojni ili decimalni. Stavili smo da su brojevi, koji se generišu pomoću Random() metode, u rasponu od -10 000 do 10 000. Aplikacija također omogućava rad sa fajlovima, tako da je moguće učitati niz iz fajla, prikazati ga na ekran, te ga sortirati. Moguće je i spremanje rezultata u fajl .

1.2 Definicije, akronimi i skraćenice

- JVM - Java Virtuelna Mašina
- IDE - Integrated Development Environment (Integrisano razvojno okruženje)

- GUI - Graphical User Interface
- API - Application Programming Interface

1.3 Reference

- https://wiki.haskell.org/Parallelism_vs._Concurrency
- "Encyclopedia of Parallel Computing" by David Padua
- "Concurrency: State Models & Java Programs" by Jeff Magee and Jeff Kramer
- https://www.geeksforgeeks.org/introduction-to-parallel-computing
- https://www.javatpoint.com/java-versions
- https://www.baeldung.com/java-fork-join
- https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
- https://www.section.io/engineering-education/introduction-to-java-swing/

1.4 Pregled

Ovaj dokument će pružiti detaljan pregled implementacije algoritama za sortiranje, objasniti paralelizam i kako je implementiran u Javi. Također će obuhvatiti detalje o instalaciji Java okruženja i korištenju Java Swing biblioteke za kreiranje grafičkog korisničkog interfejsa. Dokument će sadržavati i primjere implementacije paralelnih algoritama, te će sadržavati objašnjenje vezano za paralelizaciju u Javi. Zaključak je posljednja stavka koja se nalazi u ovome dokumentu.

2. Paralelizam

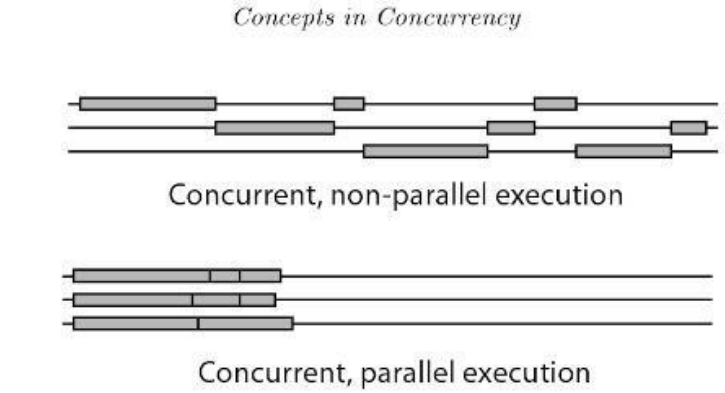
2.1 Razlika između paralelizma i konkurentnosti

Da ne bi došlo do zabune, na samom početku, napravit ćemo razliku između dva pojma, a to su paralelizam i konkurentnost. Iako su paralelizam i konkurentnost povezani, to su ipak različiti koncepti u računarstvu.

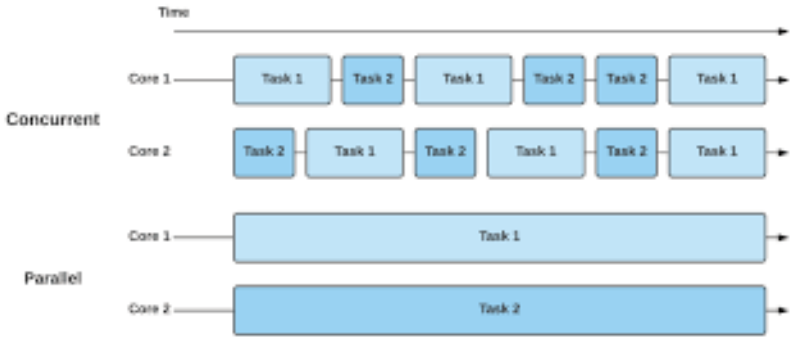
Paralelizam se odnosi na upotrebu više procesora ili procesorskih jezgara za obavljanje zadataka istovremeno kako bi se povećala računska efikasnost i smanjilo vrijeme obrade. Paralelizam uključuje podjelu većeg zadatka na manje podzadatke koje mogu paralelno obraditi različiti procesori ili jezgra za obradu. Glavni cilj paralelizma je postizanje brže i efikasnije obrade zadatka.

Konkurentnost se odnosi na sposobnost različitih dijelova sistema da se izvršavaju nezavisno i istovremeno. Konkurentnost je sposobnost programa da istovremeno obrađuje više zadataka ili zahtjeva, bez čekanja da se jedan zadatak završi prije nego što započne drugi. Glavni cilj konkurentnosti je poboljšanje ukupne odzivnosti i efikasnosti sistema. Na primjer, to može biti izvršavanje više zadataka na jednoprocesorskoj mašini.

Na slikama ispod ćemo predstaviti paralelizam i konkurentnost, kako bismo ih bolje shvatili i zapamtili.



Slika 1



Slika 2

2.2 O paralelizmu

Termin paralelizam odnosi se na tehnike za ubrzavanje programa izvođenjem nekoliko računanja u isto vrijeme. To je zapravo upotreba više procesora ili procesorskih jezgara za istovremeno obavljanje zadataka kako bi se povećala računska efikasnost i smanjilo vrijeme obrade.
Zbog toga je potreban hardver sa više procesorskih jedinica. U mnogim slučajevima, potproračuni su iste strukture. Paralelizam se postiže paralelnom obradom, što uključuje podjelu većeg zadatka na manje podzadatke koje mogu paralelno obraditi različiti procesori ili jezgra za obradu. Na primjer, Grafička izračunavanja na GPU-u (Graphics processing unit) su paralelizovana. Paralelizam se može koristiti u različitim računarskim aplikacijama, uključujući naučne simulacije, analizu podataka, obradu slika i videa i vještačku inteligenciju.

Pošto smo govorili ovdje o procesorima i jezgrima, da ne bi bilo zabune, objasnit ćemo razliku između ta dva pojma:

Procesor

Procesor, također poznat kao centralni procesorski jedinica (CPU), je glavna računarska komponenta odgovorna za izvršavanje instrukcija i obavljanje računarskih operacija. To je čip koji obavlja osnovne aritmetičke, logičke, upravljačke i ulazno/izlazne operacije. Procesor se sastoji od više komponenti, uključujući kontrolnu jedinicu (CU) i aritmetičko-logičku jedinicu (ALU).

Jezgro

Jezgro je dio procesora koji može samostalno izvršavati naredbe. Procesori se često sastoje od više jezgara, pri čemu svako jezgro može izvršavati zasebne niti instrukcija paralelno. Svako jezgro ima vlastiti set registara i može nezavisno izvršavati instrukcije. Više jezgara omogućava istovremeno izvršavanje više instrukcija i poboljšava performanse multitaskinga i paralelizma.

Razlika između procesora i jezgra je sljedeća:

- Procesor je cijeli čip ili komponenta koja obavlja računarske operacije, dok je jezgro samo jedan dio procesora.
- Procesor može sadržavati jedno ili više jezgara, ovisno o konfiguraciji. Jezgra omogućavaju istovremeno izvršavanje naredbi i mogu se koristiti za postizanje paralelizma.
- Procesor može imati i druge komponente, poput keš memorije, sabirnica i kontrolnih jedinica, dok je jezgro fokusirano na izvršavanje instrukcija.

Broj procesora i jezgara koji se nalaze u računarima zavisi od različitih faktora, uključujući namjenu računara, tip računarskog sistema i tehnološki napredak.

- desktop računari imaju obično jedan fizički procesor (CPU) sa više jezgara, obično 4 do 8, ali postoje i modeli sa više jezgara,
- laptopovi obično imaju jedan procesor sa 2 do 4 jezgra,
- Serverski računari, koji se koriste za hosting web stranica, baze podataka i druge serverske aplikacije, često imaju više procesora i više jezgara u svakom procesoru. Serverski računari mogu imati više od 10 ili čak 20 jezgara po procesoru, a neki napredniji sistemi mogu imati i stotine jezgara,
- HPC (High-Performance Computing) računari, koji se koriste za intenzivne računarske zadatke poput naučnih istraživanja, modeliranja vremenskih prilika ili simulacija, često imaju stotine, hiljade pa čak i milione jezgara. Ovi sistemi koriste se za paralelno izvršavanje velikih brojeva istovremenih zadataka.

2.3 Prednosti paralelizma

Prednosti paralelizma uključuje povećanu brzinu i efikasnost, poboljšane performanse i skalabilnost računarskih sistema, uštedu troškova, veću fleksibilnost, poboljšanu toleranciju na greške, bolje korištenje resursa, poboljšana produktivnost, brža inovacija . U nastavku ćemo nešto reći o svakoj od ovih prednosti.

Povećana brzina i efikasnost: paralelizam omogućava da više procesora ili procesorskih jezgara rade zajedno, u isto vrijeme, što može značajno povećati vrijeme izvršenja i efikasnost obrade zadataka.

Poboljšane performanse i skalabilnost: Paralelizam može poboljšati performanse i skalabilnost računarskih sistema, čineći mogućim rukovanje većim i složenijim zadacima i skupovima podataka.

Uštede troškova: Korištenjem paralelne obrade za poboljšanje efikasnosti obrade, organizacije mogu potencijalno uštediti na troškovima hardvera, energije i ukupnih troškova računarske infrastrukture.

Veća fleksibilnost: Paralelizam može pružiti veću fleksibilnost u smislu načina na koji se računski resursi dodjeljuju i koriste, omogućavajući organizacijama da prilagode resurse obrade prema potrebi kako bi zadovoljili promjenjive zahtjeve.

Poboljšana tolerancija grešaka: Korištenjem redundantnosti i mehanizama za pronalazak greške, paralelizam može poboljšati toleranciju grešaka i pouzdanost računarskih sistema, smanjujući rizik od kvarova sistema i zastoja.

Bolje korištenje resursa: Paralelizam omogućava efikasno korištenje dostupnih računarskih resursa omogućavajući istovremeno pokretanje više procesa, smanjujući vreme mirovanja i poboljšavajući ukupnu iskorišćenost sistema.

Poboljšana produktivnost: Smanjenjem vremena obrade i povećanjem efikasnosti računarskih sistema, paralelizam može omogućiti organizacijama da obavljaju zadatke brže i efikasnije, povećavajući produktivnost i propusnost.

Brža inovacija: Paralelizam može omogućiti bržu inovaciju omogućavajući istraživačima i programerima da obrađuju velike skupove podataka i izvode složene simulacije brže i efikasnije, što dovodi do bržih uvida i otkrića.

2.4 Nedostaci paralelizma

Jedan od ključnih problema sa paralelizmom je pitanje upravljanja i koordinacije različitih paralelnih procesa kako bi se osiguralo da oni rade zajedno efikasno i bez sukoba. Ovo može biti posebno izazovno u distribuiranim računarskim sistemima, gde se procesi mogu izvoditi na različitim mašinama povezanim mrežom.

Drugi ključni problem je pitanje balansiranja opterećenja, koje uključuje ravnomjernu raspodjelu radnog opterećenja na različite procesorske jedinice kako bi se osiguralo da nijedna jedinica ne miruje dok su druge preopterećene. Balansiranje opterećenja može biti posebno izazovno kada je radno opterećenje dinamično i nepredvidivo, jer može zahtijevati stalno prilagođavanje distribucije zadataka na osnovu promjenjivog radnog opterećenja.

Ostala pitanja vezana uz paralelizam uključuju potrebu za efikasnom komunikacijom i sinhronizacijom između paralelnih procesa, osiguravajući da se podaci pravilno dijele i ažuriraju između procesa i rješavanje potencijalnih uvjeta utrke, zastoja i drugih problema povezanih s konkurentnošću.

Ključni problem paralelizma je smanjenje ovisnosti podataka kako bi se mogli izvoditi proračuni na nezavisnim računskim jedinicama uz minimalnu komunikaciju između njih. U tu svrhu, čak može biti prednost da se isto izračuna dva puta na različitim jedinicama

3. Java

Java je objektno orijentisani programski jezik visokog nivoa, koji je prvi put objavio Sun Microsystems 1995. Danas je u vlasništvu Oracle kompanije. Dizajniran je tako da bude nezavisan od platforme, što znači da se Java kod može kompajlirati i pokrenuti na bilo kojoj platformi koja ima Java virtuelnu mašinu (JVM), što ga čini veoma prenosivim.

3.1 O tehnologiji

Spomenut ćemo neke od ključnih karakteristika Java programskog jezika.
Java je potpuno objektno orijentisani programski jezik, što znači da se svo programiranje vrši korištenjem objekata i klasa. Java posjeduje ugrađen Garbage Collector, koji koristi automatsko upravljanje memorijom, što olakšava upravljanje memorijom i smanjuje rizik od curenja memorije. Već smo spomenuli da je Java nezavisna od platforme, pa se Java kod kompajlira u bajt kod, koji se može pokrenuti na bilo kojoj platformi koja ima instaliranu Java virtuelnu mašinu (JVM), bez obzira na osnovni hardver ili operativni sistem. Java ima ugrađenu podršku za višenitnost, koja omogućava da se više niti izvršava istovremeno i asinhrono. Java ima snažan sigurnosni model koji uključuje funkcije kao što su provjera bajt koda i sandboxing za zaštitu od zlonamjernog koda. Java programski jezik ima veoma dobre mogućnosti rukovanja izuzecima, koje olakšavaju pisanje koda koji može rukovati neočekivanim greškama ili izuzecima na zadovoljavajući način. Java ima mnogo popularnih IDE-a (Integrisano razvojno okruženje) kao što su Eclipse, IntelliJ IDEA i NetBeans koji pružaju funkcije kao što su dovršavanje koda, otklanjanje grešaka i refaktorisanje, što olakšava razvoj Java aplikacija.

Java se široko koristi u raznim aplikacijama, uključujući web razvoj, razvoj mobilnih aplikacija, razvoj poslovnog softvera i naučno računarstvo. Ima veliku i aktivnu zajednicu programera i korisnika, što je doprinijelo razvoju bogatog sistema biblioteka, frajmvorka i alata.

3.2 Verzije Jave

Java je imala zavidan broj verzija i izdanja od svog prvog izlaska 1995. godine. U nastavku smo nabrojali neka od glavnih Javainih izdanja:

Java 1.0: Prvo stabilno izdanje Jave, predstavljeno 1996. godine.

Java 1.1: Uvedeno 1997. godine, ovo izdanje je dodalo podršku za unutrašnje klase i JavaBeans.

Java 1.2 (takođe poznat kao Java 2): Uvedeno 1998. godine, ovo izdanje je dodalo podršku za Swing GUI alat, Java imena i interfejs imenika (JNDI) i Java Foundation Classes (JFC).

Java 1.3: Objavljena 2000. godine, ova verzija je uvela Java Sound API i podršku za HotSpot JVM.

Java 1.4: Objavljena 2002. godine, ova verzija je uvela Java Native Interface (JNI), XML obradu sa JAXP-om i regularne izraze sa Java.util.regex.

Java 5 (takođe poznat kao Java 1.5): Objavljeno 2004. godine, ovo izdanje je dodalo podršku za generičke karakteristike, napomene i autoboxing.

Java 6 (takođe poznat kao Java 1.6): Objavljeno 2006. godine, ovo izdanje je uvelo poboljšanja Java virtuelne mašine (JVM), uključujući podršku za dinamičke jezike i Java Compiler API.

Java 7: Objavljena 2011. godine, ova verzija je uvela naredbu try-with-resources, naredbe višestrukog hvatanja i podršku za nizove u naredbama switch.

Java 8: Objavljeno 2014. godine, ovo izdanje je dodalo podršku za lambda izraze, Stream API i Date/Time API.

Java 9: Objavljena 2017. godine, ova verzija je uvela Java Platform Module System (JPMS), koji omogućava modularno programiranje.

Java 10: Objavljeno 2018. godine, ovo izdanje je dodalo podršku za zaključivanje tipa lokalne varijable i interfejs za prikupljanje smeća (Garbage Collector).

Java 11: Objavljena 2018. godine, ova verzija je uvela podršku za HTTP/2 i predstavila nove standardne karakteristike Java Platform Module System (JPMS).

Java 12, 13, 14, 15, 16 i 17: Izdate 2019, 2020 i 2021, ova izdanja su donijela nove funkcije, poboljšanja i poboljšanja performansi Java platforme.

Kao što možemo zaključiti, svako sljedeće izdanje Jave donosi nam nove karakteristike, poboljšanja i unapređenja performansi platforme. Zbog toga se programeri ohrabruju da ažuriraju svoje Java instalacije kako bi iskoristili prednosti najnovijih funkcija.

3.3 Instalacija Jave

Proces instalacije Jave zavisi od operativnog sistema koji koristimo. U zavisnosti od operativnog sistema, navest ćemo postupak instalacije jave.

3.3.1 Windows

1. Idite na službenu Java web stranicu: https://www.java.com/en/download/ a zatim kliknite na dugme "Download Java".
2. Prihvatite ugovor o licenci i sačuvajte instalaciju aplikacije na određenu lokaciju na vašem računaru.
3. Kada se preuzimanje završi, dvaput kliknite na instalacionu datoteku i slijedite uputstva sve dok ne dovršite instalaciju.

3.3.2 MacOS

1. Idite na službenu Java web stranicu: https://www.java.com/en/download/ a zatim kliknite na dugme "Download Java".
2. Prihvatite ugovor o licenci i sačuvajte instalaciju aplikacije na određenu lokaciju na vašem računaru.
3. Kada se preuzimanje završi, dvaput kliknite na DMG datoteku i slijedite uputstva sve dok ne dovršite instalaciju.

3.3.3 Linux

Postupak instalacije Jave na Linux operativni sistem se razlikuje u zavisnosti od toga koju distribuciju Linuxa koristite. U slučaju da koristiti Ubuntu evo koraka koje trebate slijediti:

1. Otvorite prozor terminala i unesite sljedeću komandu kako biste ažurirali listu paketa:

```
sudo apt update
```

2. Unesite sljedeću komandu kako biste instalirali Javu:

```
sudo apt install default-jdk
```

(Ovo će komanda instalirati podrazumevanu verziju Jave koja je dostupna u skladišku Ubuntu paketa.)

3. Provjerite koju verziju Jave ste upravu instalirali, tj. Koji verziju sada posjedujete:

```
java -version
```

4. Java Swing

Java Swing je grafički korisnički interfejs (GUI) toolkit koji je dio Java Standardne biblioteke - Java Foundation Classes (JFC). Koristi se za kreiranje aplikacija zasnovanih na prozorima što ga čini pogodnim za razvoj jednostavnih desktop aplikacija.

Java Swing je izgrađen na vrhu apstraktnog prozorskog alata - AWT (Abstract Windowing Toolkit) API koji je isključivo napisan u Java programskom jeziku.

Java Swing pruža lagane komponente koje su nezavisne od platforme, što ga čini pogodnim i efikasnim u dizajniranju i razvoju desktop aplikacija (sistema). Swing omogućava programerima da razvijaju desktop aplikacije sa bogatim interfejsom, jer podržava različite komponente poput dugmadi, polja za unos teksta, padajućih menija, tabela, dijaloga i drugih grafičkih elemenata.

4.1 Karakteristike i koncepti Swinga

Platformska nezavisnost: Jedna od najvažnijih karakteristika Jave Swing je platformska nezavisnost. Swing komponente se renderiraju [1]na Java Virtual Machine (JVM), što omogućava da se aplikacije izvršavaju na različitim operativnim sistemima kao što su Windows, macOS i Linux, bez potrebe za prilagođavanjem koda za svaku platformu.

MVC arhitektura: Swing se oslanja na Model-View-Controller (MVC) arhitekturu, koja razdvaja logiku aplikacije, prikaz korisničkog interfejsa i upravljanje korisničkim akcijama. Ovo

[1] Rendiranje - postupak stvaranja slike od nekog modela uz pomoć posebnog programa.

omogućava modularnost i olakšava održavanje aplikacije.

Komponente i događaji: Swing pruža širok spektar grafičkih komponenti koje programeri mogu koristiti za izgradnju korisničkog interfejsa. Svaka komponenta generiše događaje pa korisnik može da intereaguje s njima. Ti događaji su na primjer klikovi mišem ili unos teksta. Programeri mogu reagovati na ove događaje i izvršiti odgovarajuće akcije.

Jednostavan Look&Feel: Swing omogućava programerima da prilagode izgled i stil komponenti, kao što su boje, fontovi, pozadine i razmaci. Također podržava teme i pluggable izglede koji omogućavaju da se aplikacija prilagodi korisničkom stilu ili sistemskim postavkama.

Dugmad, meniji i dijalozi: Swing ima bogatu podršku za različite vrste komponenti. To uključuje dugmad (JButton), menije (JMenu, JMenuItem), padajuće liste (JComboBox), tabele (JTable), frejmove (Jframe), paneli (JPanel), dijaloge (JDialog) i još mnogo toga. Ove komponente mogu se prilagođavati i nadograđivati prema potrebama aplikacije.

Layout Manageri: Swing koristi koncept layout managera za organizaciju komponenti unutar kontejnera, kao što su JFrame ili JPanel. Layout manageri određuju kako će se komponente rasporediti i prilagoditi u zavisnosti od veličine prozora ili kontejnera. Neke od uobičajenih layout managera su BorderLayout, FlowLayout, GridLayout i BoxLayout.

Event Handling: Swing koristi model događaja i slušače događaja za reagovanje na korisničke akcije. Programeri mogu dodati slušače događaja na komponente i definirati metode koje će se pozvati kada se događaj desi. Na primjer, dodavanje ActionListenera na dugme (JButton) omogućava reagovanje na klik na to dugme.

Rendering i Double Buffering: Swing koristi dvosmjerno "double buffering" kako bi poboljšao vizuelno iskustvo korisnika. Renderiranje komponenata se vrši na prikaznom bufferu koji nije vidljiv korisniku, a zatim se ta slika kopira na ekran kako bi se izbjeglo vidljivo treperenje i poboljšalo performanse.

Internationalization i Localization: Swing podržava međunarodizaciju (Internationalization) i lokalizaciju (Localization), što omogućava da se aplikacija prilagodi različitim jezicima, regionalnim postavkama, datuma i vremenima. Komponente kao što su JLabel, JButton i JMenu podržavaju rad s višejezičnim tekstom.

Drag and Drop podrška: Swing ima ugrađenu podršku za povlačenje i ispuštanje (drag and drop) operacije. To omogućava korisnicima da prevlače komponente ili podatke između različitih dijelova aplikacije ili čak izvan aplikacije.

Dizajniranje Custom Komponenti: Swing omogućava programerima da dizajniraju i implementiraju vlastite custom komponente, proširene od Swing komponenti ili potpuno nove. Ovo omogućava kreiranje specifičnih interfejsa koji zadovoljavaju potrebe aplikacije.

4.2 Java Swing vs JavaFX

Swing je bio standardni izbor za razvoj desktop aplikacija u Javi do 2008. godine, ali od verzije Java 8, JavaFX je postao preporučeni framework za razvoj korisničkih interfejsa. Ipak, Swing je još uvijek podržan i može se koristiti za razvoj desktop aplikacija u Javi, te smo se mi odlučili koristiti baš njega. Istina je da Java Swing ima obilježja starijeg dizajna i izgleda u odnosu na moderne GUI framework-e, pa neko ko razvija modernije korisničke interfejse, vjerovatno će koristiti JavaFX-a ili neki sličan alat.

Iako je JavaFX postao popularan i prihvaćen u razvoju desktop aplikacija, Java Swing nije potpuno izašao iz upotrebe. Swing i dalje ostaje podržan i dostupan u Javi, a mnoge postojeće aplikacije zasnovane na Swing-u i dalje se održavaju i koriste širom svijeta. Međutim, JavaFX se smatra preporučenim framework-om za razvoj modernih korisničkih interfejsa u Javi, posebno za novije projekte.

5. Sortiranje

Sortiranje je proces premještanja podataka u rastućem ili opadajućem redoslijedu, kako bismo ih razvrstali i na određeni način uredili. Također možemo reći i da je sortiranje proces raspoređivanja podataka u smislenom redoslijedu kako bismo ih mogli kasnije brže, efikasnije i lakše analizirati. To je važan koncept u računarstvu i široko se koristi u mnogim aplikacijama. Sortiranje omogućava efikasno pretraživanje, filtriranje i analizu podataka, kao i poboljšanje performansi mnogih algoritama. Sortiranje su najčešća u abecednom ili numeričkom obliku (A-Z ili Z-A; 1-9 ili 9-1).

Postoji nekoliko popularnih algoritama za sortiranje kao što su: **Bubble sort, Insertion sort, Selection sort, Merge sort i Quic sort.** Svaki od ovih algoritama ima svoje karakteristike, prednosti i nedostatke.

Bubble sort je jednostavan, ali spor algoritam koji prolazi kroz skup više puta, poređujući i zamjenjujući susjedne elemente sve dok svi elementi ne budu u pravilnom redoslijedu. **Insertion sort** prolazi kroz skup i ubacuje svaki element na pravo mjesto u već sortiranu podlistu. **Selection sort** prolazi kroz skup i bira najmanji element i stavlja ga na prvo mjesto, zatim drugi najmanji element stavlja na drugo mjesto i tako dalje. **Merge sort** koristi strategiju "podijeli i vladaj" i rekurzivno dijeli skup na manje podskupove, sortira ih zasebno, a zatim ih spaja u jedan sortirani skup. **Quick sort** također koristi "podijeli i vladaj" strategiju, ali umjesto spajanja koristi pivot element kako bi podijelio skup na manje i veće elemente, a zatim ih sortira rekurzivno.

Izbor odgovarajućeg algoritma sortiranja zavisi od različitih faktora kao što su veličina skupa podataka, tip podataka, performanse i drugi specifični zahtjevi aplikacije. Razumijevanje algoritama sortiranja pomaže nam da izaberemo najbolje rješenje za određenu potrebu.

U nastavku ćemo proći kroz svaki od ovih algoritama i pomoću slika objasniti kako on radi.

5.1 Algoritmi sortiranja

Algoritmi za sortiranje su osnovni koncept računarske nauke i intenzivno se koriste u mnogim različitim aplikacijama. Možemo reći da je algoritam za sortiranje zapravo algoritam koji elemente liste postavlja u određenom redoslijedu, u zavisnosti od toga šta se traži u datom kontekstu. Postoji mnogo različitih algoritama za sortiranje, od kojih svaki od njih ima svoje prednosti i nedostatke, a koji ćemo algoritam izabrati, to ovisi o specifičnom slučaju korištenja.

Algoritmi za sortiranje se mogu podjeliti na osnovu njihove vremenske i prostorne složenosti, što određuje koliko su efikasni u pogledu upotrebe vremena i memorije. Neki algoritmi su više prikladni za određene tipove podataka ili veličine, pa je važno odabrati pravi algoritam za zadatak koji rješavamo.

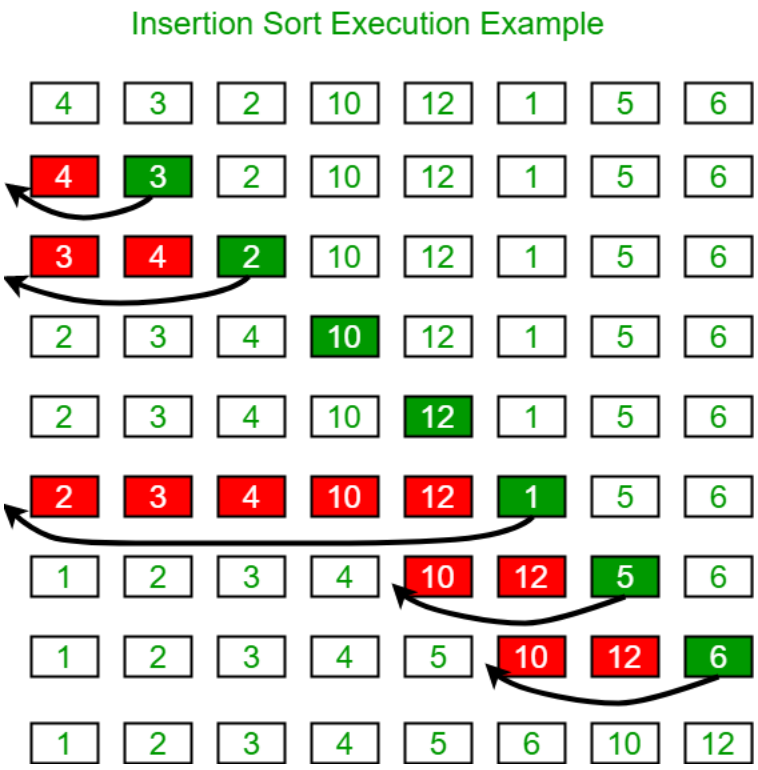
U ovome djelu, obradit ćemo pet algoritama sortiranja, a to su:

- insertion sort,
- selection sort,
- bubble sort,
- merge sort
- quick sort

5.1.1 Insertion sort

Sortiranje umetanjem (eng. Insertion sort) je jednostavan algoritam za sortiranje, koji radi krećući se kroz niz s lijeva na desno, postepeno izgrađujući sortirani dio niza. U svakoj iteraciji, algoritam uzima sljedeći nesortirani element i ubacuje ga na ispravnu poziciju u sortiranom dijelu niza. Niz je podijeljen na sortirani i nesortirani dio. Vrijednosti iz nesortiranog dijela se biraju i postavljaju na ispravan položaj u sortirani dio.

Sortiranje umetanjem ima najgori slučaj vremenske složenosti od O(n^2), ali može biti vrlo efikasno kada su upitanju mali nizovi ili već djelimično sortirani nizovi. Također jedna od njegovih prednosti je to što je ovo algoritam za sortiranje u mjestu, što znači da sortira ulazni niz, tu gdje se nalazi, bez potrebe za zauzimanjem dodatne memorije. Što se tiče prostora, ova tri algoritma, insertion sort, selection sort i bubble sort, koriste O(1) konstantnog prostora, jer sortiraju u mjestu, ali O(n^2) vremena.



Slika 3

```java
public class InsertionSort {

    public static void insertionSort(int[] niz) {

        int i, j, priv;

        for(i=1;i<niz.length;i++){

            j=i;

            while(j>0 && niz[j-1]>niz[j]){

                priv=niz[j-1];
                niz[j-1]=niz[j];
                niz[j]=priv;
                j--;
            }
        }

    }
}
```

Slika 4

5.1.2 Selection sort

Sortiranje odabirom (eng. Selection sort) je jednostavan algoritam sortiranja koji sortira niz ponavljajući postupak pronalaženja minimalnog elementa iz nerazvrstanog dijela i i zamjenjuje ga prvim nesortiranim elementom.

Algoritam održava dva podniza u danom nizu:

- Podniz koji je već sortiran
- Preostali podskup koji još nije sortiran

U svakoj iteraciji, tj. u svakom prolazu se bira minimalni element iz nerazvrstanog podniza i premješta se u razvrstani niz. Selection sort ima svojstvo minimiziranja broja zamjena, stoga je najbolji izbor kada su troškovi zamjene visoki.

Selekciono sortiranje ima vremensku složenost u najgorem slučaju od O(n^2), ali ima prednost što je algoritam za sortiranje u mjestu, što znači da sortira ulazni niz, tu gdje se nalazi, bez potrebe za zauzimanjem dodatne memorije.

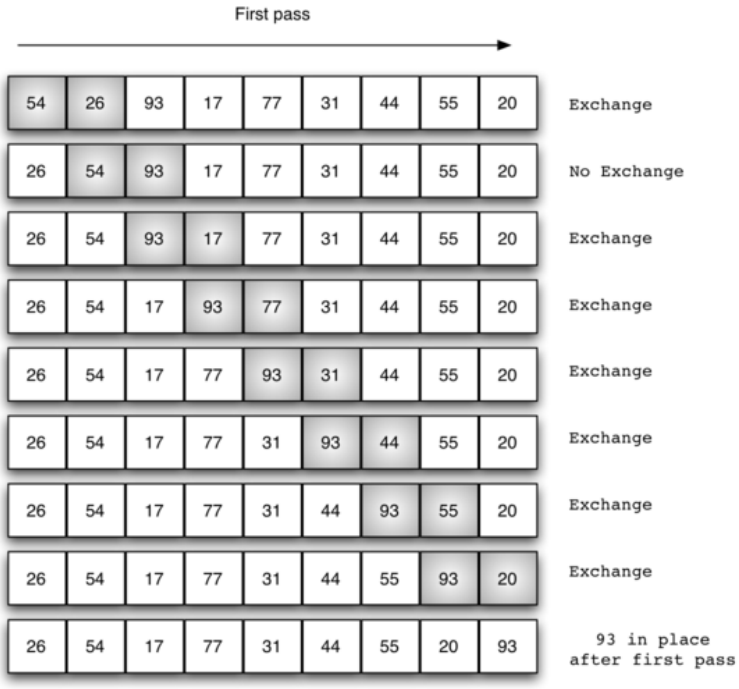
Slika 5

```
public class SelectionSort {

    public static void selectionSort(int[] niz) {

        int i, min,j,pom;

        for(i=0;i<niz.length-1;i++){

            min=i;

            for(j=i+1;j<niz.length;j++){
                if(niz[j]<niz[min]){
                        min=j;
                        }
                   }

                pom=niz[min];
                niz[min] = niz[i];
                niz[i]=pom;
        }
    }
}
```

Slika 6

5.1.3 Bubble sort

Bubble sort je algoritam sortiranja koji radi na način da mjenja susjedne elemenate ako su u pogrešnom redoslijedu. Nakon svake iteracije najveći element ide na kraj (u slučaju rastućeg redoslijeda) ili najmanji element dolazi na kraj (u slučaju opadajućeg redoslijeda). Prolazak kroz elemente se ponavlja dok se niz ne sortira. Ovaj algoritam nije prikladan za velike skupove podataka jer su njegova prosječna složenost i složenost u najgorem slučaju O(n^2), što znači da njegove performanse opadaju kako se veličina ulaznih podataka povećava, što ga čini neefikasnim.



Slika 7

```
public class BubbleSort {
    public static void bubbleSort(int[] niz) {

        int i, j, pom;
        for(i=1;i<niz.length;i++){
            for(j=0;j<niz.length-1;j++){

                if(niz[j]>niz[j+1]){

                    pom=niz[j];
                    niz[j] = niz[j+1];
                    niz[j+1]=pom;
                   }
               }
        }

    }
}
```

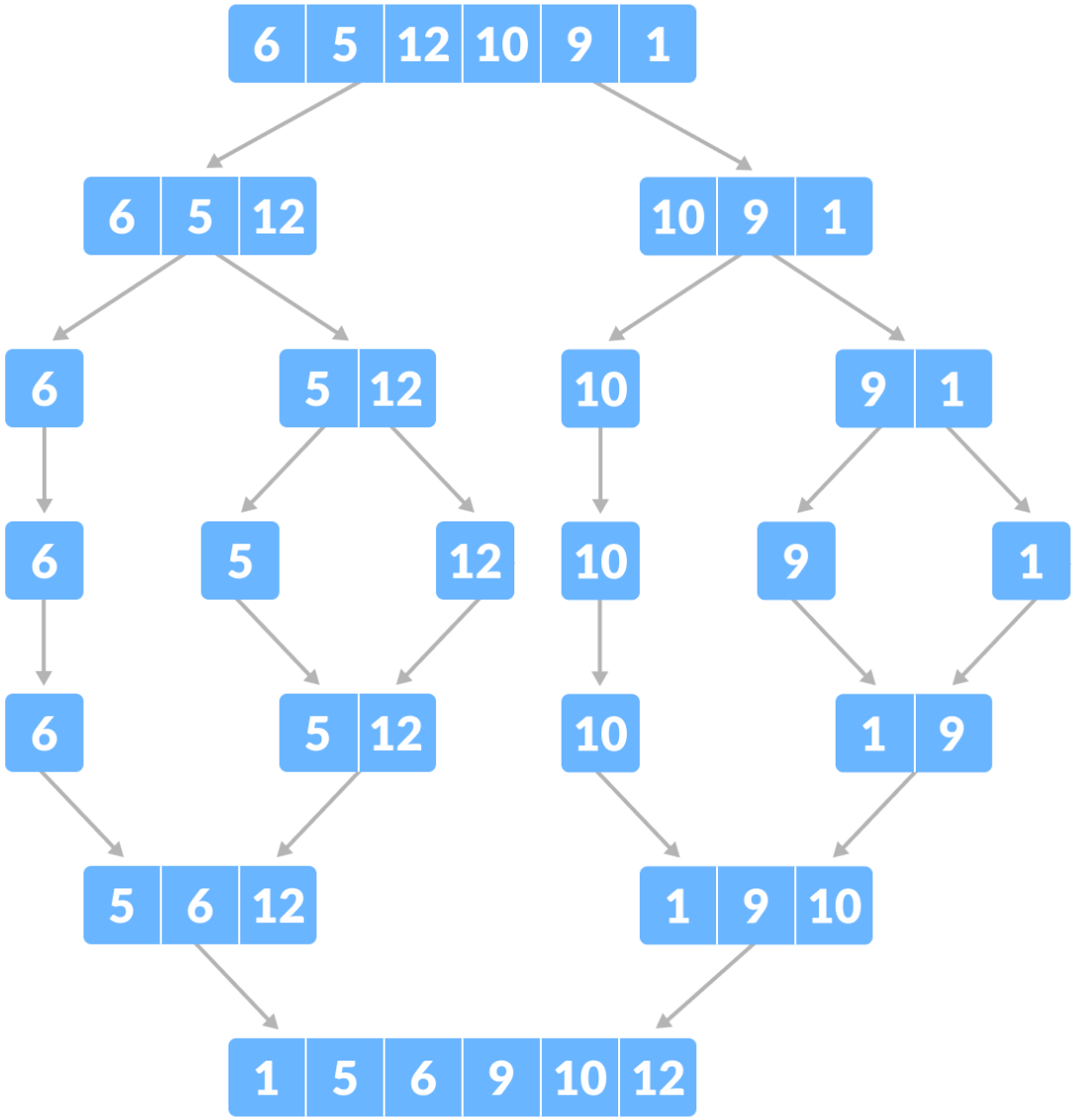
Slika 8

5.1.4 Merge sort

Merge sort algoritam je izumio John von Neumann 1945. Godine. Za razliku od gornja tri algoritma za sortiranje, ovaj se algoritam temelji na tehnici podijeli pa vladaj. To je algoritam zasnovan na uspoređivanju koji radi tako što rekurzivno dijeli ulazni niz na manje podnizove, sortira ih, a zatim ih ponovo spaja. Srce Merge Sort algoritma je *merge()* funkcija koja se koristi za spajanje dvije polovice niza. Spajanje je ključni postupak koji pretpostavlja da su dvije

polovice sortirane i spaja ih u jednu cjelinu. Ovaj algoritam možete odabrati onda kada vam je potrebno stabilno i brzo sortiranje. Ima zagarantovano vreme rada u najgorem slučaju od O(n*log(n)), što ga čini jednim od najefikasnijih algoritama za sortiranje. Što se tiče prostora, ovaj algoritam koristi linearno prostor O(n).

Sortiranje spajanjem je algoritam za sortiranje opće namjene koji se može koristiti za sortiranje bilo koje vrste podataka, uključujući brojeve, nizove i objekte. Posebno je pogodan za sortiranje velikih skupova podataka ili podataka koji su pohranjeni na vanjskim uređajima za pohranu (kao što su tvrdi diskovi) koji imaju sporo vrijeme pristupa.



Slika 9

```
public class MergeSort {

public static void mergeSort(int[] niz, int lijevi, int desni) {
        if (lijevi < desni) {

          int srednji = (lijevi + desni) / 2;

          // Sortiramo prvu i drugu polovicu
          mergeSort(niz, lijevi, srednji);
          mergeSort(niz, srednji + 1, desni);

          // Spajamo sortirane polovice
          merge(niz, lijevi, srednji, desni);
        }
      }

public static void merge(int[] niz, int lijevi, int srednji, int desni) {
        // Velicine dva podniza koji ce se spojiti
        int n1 = srednji - lijevi + 1;
        int n2 = desni - srednji;

        /* Kreiranje pomocnih nizova */
        int[] L = new int[n1];
        int[] D = new int[n2];

        /* Upisujemo podatke u nove nizove */
        for (int i = 0; i < n1; ++i)
          L[i] = niz[lijevi + i];
        for (int j = 0; j < n2; ++j)
          D[j] = niz[srednji + 1 + j];

        /* Spajamo podnizove */

        int i = 0, j = 0;
        int k = lijevi;

        while (i < n1 && j < n2) {
          if (L[i] <= D[j]) {
            niz[k] = L[i];
            i++;
          }
          else {
            niz[k] = D[j];
            j++;
          }
          k++;
        }
```

```
            /* Upisujemo ostale elemente iz L[] ako su ostali(zbog
            posljednjeg elementa koji moze ostati neupisan) */
        while (i < n1) {
            niz[k] = L[i];
          i++;
          k++;
        }

           /* Upisujemo ostale elemente iz D[] ako su ostali (zbog
           posljednjeg elementa koji moze ostati neupisan)*/
        while (j < n2) {
            niz[k] = D[j];
          j++;
          k++;
        }
      }
}
```

Slika 10

5.1.5 Quick sort

Kao i merge sort, quick sort je rekurzivni algoritam koji također radi po principu podijeli pa vladaj. Ovaj algoritam bira jedan element kao pivot, a zatim dijeli listu na dve podliste tako da svi elementi manji od pivota budu u jednoj podlisti, a svi elementi veći od pivota u drugoj podlisti, a onda se rekurzivno sortiraju ove dvije podliste. Ovaj proces se ponavlja dok se cijela lista ne sortira.

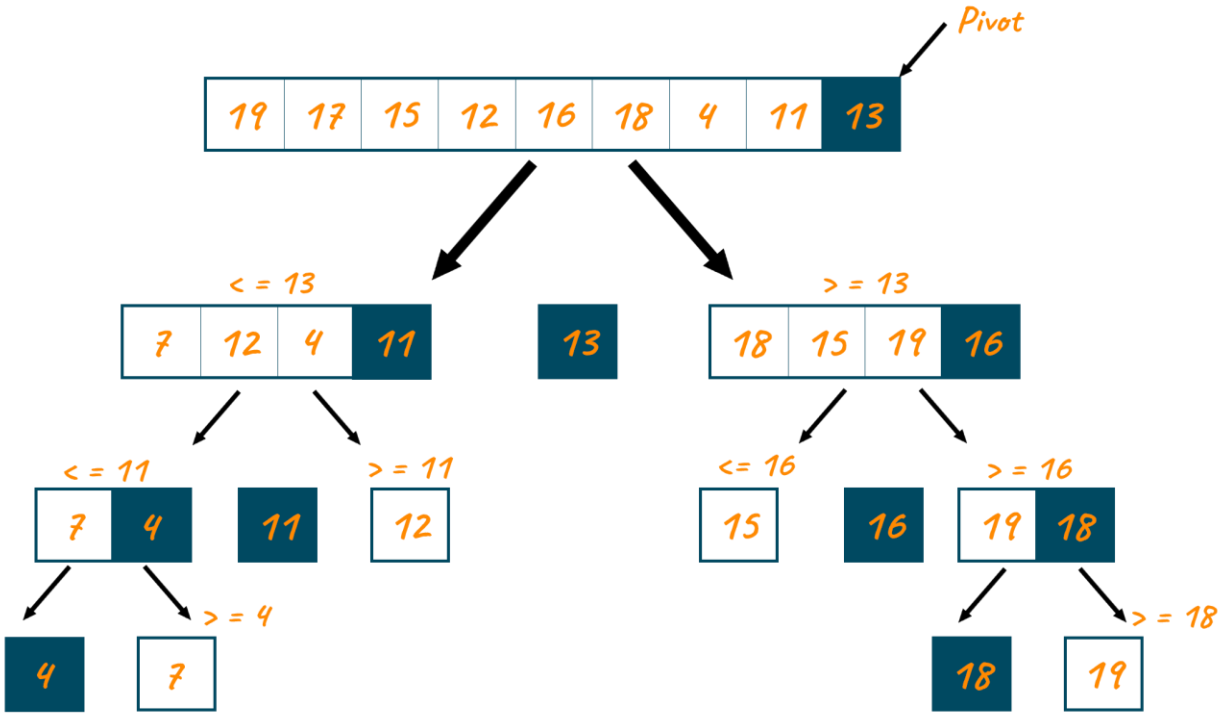
Postoji mnogo različitih verzija Quick Sort algoritma koji biraju pivot na različite načine:

- Uvijek bira prvi element kao pivot
- Uvijek bira posljednji element kao pivot
- Bira slučajni element kao pivot
- Bira srednji element kao pivot

Ključni proces u i quick sort algoritmu je „partition()“ metoda. Cilj ove metode je da stavi pivot r na njegov ispravan položaj u sortiranom nizu i da postavi sve elemente manje od r ispred r, a sve veće elemente poslije r.

Brzo sortiranje ima vrijeme rada u najgorem slučaju od $O(n^2)$, ali u praksi obično radi mnogo bolje od ovoga zbog upotrebe dobrog algoritma za odabir pivota i činjenice da podliste obično nisu podijeljene ravnomerno. Prosječno vrijeme rada brzog sortiranja je O(n*log(n)), što je jedan od najbržih dostupnih algoritama za sortiranje.

Ovo je algoritam za sortiranje u mjestu, što znači da ne zahtijeva dodatnu memoriju za pohranjivanje privremenih nizova. Lako se implementira i može se modifikovati da sortira objekte ili druge složene tipove podataka.


Pivot
19 17 15 12 16 18 4 11 13
< = 13
7 12 4 11
13
> = 13
18 15 19 16
< = 11
7 4
11
> = 11
12
<= 16
15
16
> = 16
19 18
> = 4
4
7
> = 18
18
19

Slika 11

```
public class QuickSort {

    public static void sort(int[] niz, int lijevi, int desni) {
        if (lijevi < desni) {
            int pivotIndex = partition(niz, lijevi, desni);
            sort(niz, lijevi, pivotIndex - 1);
            sort(niz, pivotIndex + 1, desni);
        }
    }

    private static int partition(int[] niz, int lijevi, int desni) {
        int pivot = niz[desni];
        int i = lijevi - 1;
        for (int j = lijevi; j < desni; j++) {
            if (niz[j] < pivot) {
                i++;
                int temp = niz[i];
                niz[i] = niz[j];
                niz[j] = temp;
            }
        }
        int temp = niz[i + 1];
        niz[i + 1] = niz[desni];
        niz[desni] = temp;
        return i + 1;
    }
}
```

Slika 12

6. Paralelizam u Javi

6.1 Fork/Join framework

Fork/Join framework je dio Java platforme koji pruža podršku za paralelno izvršavanje zadataka koji se mogu podijeliti na manje dijelove i spojiti rezultate. Ovaj framework je uveden u Java SE 7 kao dio **java.util.concurrent paketa**.

Glavni koncept Fork/Join frameworka je "podijeli i vladaj" strategija. To znači da se veliki zadatak dijeli na manje dijelove koji se mogu izvršavati nezavisno. Svaki podzadatak se naziva "fork" (razgranavanje). Nakon što su svi podzadaci završeni, rezultati se kombinuju u "join" (spajanje) koraku.

Fork/Join framework je implementacija ExecutorService interfejsa koja pomaže u iskorištenju više procesora. Dizajnirana je tako da rekurzivno razbija zadatke na manje dijelove. Cilj je iskorištenje dostupne procesorske moći za poboljšanje performansi aplikacije. Dakle, ovaj framework radi „fork“, račvanje zadatka, dok se ne dobije zadatak koji je dovoljno jednostavan da bude izvršen asinhrono. Nakon toga se radi „join“, sabiranje rezultata, ako je to potrebno. Glavne komponente ovog frameworka su ForkJoinPool i ForkJoinTask.

6.2 ForkJoinPool

ForkJoinPool je klasa u Java Fork/Join frameworku koja predstavlja bazen radnih niti (eng. worker threads) koje se koriste za paralelno izvršavanje podzadataka. ForkJoinPool je odgovoran za upravljanje izvršavanjem podzadataka u Fork/Join frameworku. On omogućava efikasno korištenje raspoloživih procesorskih jezgara i automatski raspoređuje podzadatke radnim nitima.

ForkJoinPool je specifičan tip ExecutorService za upravljanje paralelnim izvršavanjem. On organizuje radnu snagu u grupu radnih niti. Svaka radna nit ima svoj red zadataka koji čeka da se izvrši. Kada radna nit završi svoj trenutni zadatak, ona može preuzeti drugi zadatak iz svog reda ili uzeti zadatak iz reda druge radne niti ako je njen red prazan.

ForkJoinPool je srce ovog frameworka. On upravlja nitima radnicama i nudi informacije o stanju i performansama bazena. Niti radnice mogu izvršavati samo po jedan zadatak, ali to ne znači da ForkJoinPool kreira novu nit za svaki novi podzadatak. Naime, svaka nit ima svoj dvoizlazni red (double ended queue – deque, dek) koji čuva zadatke. Statički ForkJoinPool, vraćen metodom commonPool(), je pogodan za većinu aplikacija te je njime obezbjeđeno smanjeno korištenje resursa. Interesantno je što pri instanciranju novog ForkJoinPool-a možemo specificirati nivo paralelizacije, u suprotnom, nivo paralelizacije biće jednak broju dostupnih jezgara.

Ključne karakteristike ForkJoinPoola su:

- *Dinamičko korištenje radnih niti*: ForkJoinPool se prilagođava trenutnom opterećenju sistema i automatski prilagođava broj radnih niti na osnovu raspoloživih procesorskih jezgara. Novi podzadaci se dinamički dodjeljuju slobodnim radnim nitima.

- *Work-stealing algoritam*: ForkJoinPool koristi work-stealing algoritam za efikasnu raspodjelu podzadataka između radnih niti. Radne niti sa manje posla mogu "ukrasti" zadatke od drugih radnih niti koje su više opterećene, čime se postiže bolja ravnoteža opterećenja.

- *Paralelno izvršavanje*: ForkJoinPool podržava paralelno izvršavanje podzadataka, što omogućava efikasno korištenje višejezgarnih sistema. Podzadaci se dijele i izvršavaju paralelno na raspoloživim procesorskim jezgrama.

6.2.1 Work-stealing algorithm

Work-stealing algoritam je algoritam raspodjele zadataka (taskova) u Fork/Join frameworku koji omogućava radnim nitima (worker threads) da preuzimaju i izvršavaju zadatke iz reda drugih radnih niti ako su njihovi redovi prazni. Ovaj algoritam je zapravo algoritam koji krade posao. Ustvari, on pokušava „ukrasti“ zadatke od drugih niti koje su zauzete izvršavanjem nekog zadatka, ako je njegova radna niz izvršila sve zadatke i prazna je.

Kada radna nit završi izvršavanje svog trenutnog zadatka, ona pokušava preuzeti dodatni zadatak iz svog vlastitog reda. Međutim, ako je njen red prazan, ona koristi mehanizam work-stealing-a (krađe) da bi preuzela zadatak iz reda neke druge radne niti. Po defoltu, nit uzima zadatak sa kraja svog deka. Kada se dek isprazni, nit preuzima zadatak sa repa deka neke druge niti, ili sa

globalnog ulaznog reda pošto je tu najvjerovatnije najveći dio posla.

Ideja iza work-stealing algoritma je da radne niti koje su manje opterećene, tj. imaju manje zadatka u svom redu, mogu "ukrasti" zadatke od drugih radnih niti koje su više opterećene. Na taj način se povećava iskorištenost resursa i bolje raspoređuje radna snaga u sistemu. Ovaj pristup smanjuje šansu da će se niti takmičiti oko zadataka. Takođe, nit će rijetko morati da traži posao zbog toga što radi prvo na najvećim dijelovima posla.

Work-stealing algoritam pomaže u postizanju visoke efikasnosti i boljeg iskorištenja resursa u paralelnim izračunavanjima.

6.3 ForkJoinTask

ForkJoinTask je apstraktna klasa u Java Fork/Join frameworku koja predstavlja podzadatak (eng. task) koji se može izvršavati paralelno. ForkJoinTask je osnova za implementaciju podzadataka u Fork/Join frameworku. Ona pruža osnovne metode i funkcionalnosti za upravljanje paralelnim izvršavanjem i rezultatima tih podzadataka.

ForkJoinTask ima dvije glavne podklase:

- ***RecursiveAction***: Ovo je podklasa ForkJoinTask koja predstavlja podzadatak koji ne vraća rezultat. Umjesto toga, on obavlja neki posao ili vrši izmjene nad globalnim stanjem.
- ***RecursiveTask***: Ovo je podklasa ForkJoinTask koja predstavlja podzadatak koji vraća rezultat. Metoda compute() u ovoj podklasi treba vratiti rezultat izračunavanja.

Obje podklase imaju metodu **fork()** koja dijeli podzadatak na manje podzadatke i paralelno ih izvršava. Također imaju metodu **join()** koja čeka da se izvršenje podzadataka završi i vraća rezultat izračunavanja. Obje ove klase imaju metodu **compute()** koju je potrebno overrideovati.

6.4 Predavanje zadataka ForkJoinPool-u

Moguće je pokrenuti zadatak na nekoliko načina.

- *submit*() ili *execute*() metoda

```
forkJoinPool.execute(customRecursiveTask);
int result = customRecursiveTask.join();
```

- *invoke*(), izvršava fork (grananje) zadatka i vraća rezultat, te nije potrebno ručno uraditi nikakav join

```
int result = forkJoinPool.invoke(customRecursiveTask);
```

- *invokeAll*() metod predstavlja najjednostavniji način za predavanje niza zadataka u bazen. Ona prima zadatke kao parametre, forkuje ih i vraća kolekciju objekata tipa *Future* redoslijedom kojim su i napravljeni
- *fork*() i *join*() metode je moguće koristiti odvojeno. fork() metod predaje zadatak to jest, priprema ga za asinhrono izvršavanje u bazenu u kojem se trenutno nalazi. Izvršavanje zadatka okida se metodom *join*()

```
customRecursiveTaskFirst.fork();
result = customRecursiveTaskLast.join();
```

7. Paralelizam u algoritmima sortiranja

7.1 Implementacija paralelnog merge sorta

Prirodno je paralelizaciju merge sort algoritma uraditi koristeći ForkJoinPool iz tog razloga što on podrazumijeva rekurzivno polovljenje niza.

```
if (lijevi < desni) {

    int srednji = (lijevi + desni) / 2;

    // Sortiramo prvu i drugu polovicu
    mergeSort(niz, lijevi, srednji);
    mergeSort(niz, srednji + 1, desni);

    // Spajamo sortirane polovice
    merge(niz, lijevi, srednji, desni);
}
```

Slika 13 - Osnova sekvencijalnog merge sorta

Primjećujemo da sami algoritam sortiranja predstavlja instanciranje jednog SortTask objekta (koji je tipa RecursiveAction) te njegovo okidanje metodom invoke().

```
public static void parallelMergeSort(int[] niz, int lijevi, int desni) {
    SortTask task = new SortTask(niz, lijevi, desni);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(task);
}
```

Slika 14 - Početak sortiranja paralelnog mergesorta

Primjećujemo da je osnova paralelnog algoritma sortiranja skoro pa identična sekvencijalnoj. Razlika je u tome što se pozivi metode zamjenjuju instanciranjem novog objekta tipa SortTask, te okidanje svih objekata koji su kreirani metodom invokeAll(). Sam rad prepušten je implementaciji ForkJoinPool-a i nitima koje postoje unutar njega.

```
@Override
protected void compute() {

    if(lijevi < desni) {

        int srednji = (lijevi + desni) / 2;

        SortTask prvaPolovina = new SortTask(niz, lijevi, srednji);
        SortTask drugaPolovina = new SortTask(niz, srednji + 1, desni);

        invokeAll(prvaPolovina, drugaPolovina);

        MergeSort.merge(niz, lijevi, srednji, desni);
    }
}
```

Slika 15 - Osnova paralelnog merge sorta

7.2 Implementacija paralelnog quick sorta

U sekvencijalnom quicksort algoritmu zapažamo rekurziju i particioniranje elemenata na lijevu i desnu stranu. Samim tim, logično je lijevu stranu predati jednoj niti, a desnu drugoj, pa lijevu stranu lijeve strane dodijeliti nekoj novoj niti, a desnu stranu desne niti nekoj drugoj, itd.

```
@Override
public <T extends Comparable<T>> void sort(T[] niz, int lijevi, int desni) {
        if (lijevi < desni) {
            int pivotIndex = partition(niz, lijevi, desni);
            sort(niz, lijevi, pivotIndex - 1);
            sort(niz, pivotIndex + 1, desni);
        }
    }
```

Slika 16 - Osnova sekvencijalnog quicksorta

Prirodni opis ovog algoritma odgovara njegovoj stvarnoj podjeli na nove instance taskova.

```
@Override
public <T extends Comparable<T>> void sort(T[] niz, int lijevi, int desni) {
    SortTask<?> task = new SortTask<T>(niz, lijevi, desni);
    pool = new ForkJoinPool();
    pool.invoke(task);
}
```

Slika 17 - Početak sortiranja paralelnog quicksorta

Ova implementacija je slična onoj u mergesortu, s tim što nije potrebno spajati rezultate sortiranja. Sličnosti između paralelne i sekvencijalne implementacije su očigledne. Razlikuju se u tome što se u paralelnoj implementaciji mora okinuti lijeva i desna strana, korištenjem invokeAll() metode. Također, poziv metode zamijenjen je instanciranjem SortTask objekta.

```java
@Override
protected void compute() {

    if(lijevi < desni) {

        int pivot = QuickSort.partition(niz, lijevi, desni);

        SortTask<?> lijevaStrana = new SortTask<T>(niz, lijevi, pivot - 1);
        SortTask<?> desnaStrana = new SortTask<T>(niz, pivot + 1, desni);

        invokeAll(lijevaStrana, desnaStrana);
    }
}
```

Slika 18 - Osnova paralelnog quicksorta

7.3 Implementacija paralelnog selection sorta

Okidanje sortiranja identično je za svaku paralelnu izvedbu. Preostali algoritmi funkcionišu nešto drugačije od prethodnih. Ranije smo primijetili divide and conquer strategiju rekurzivnog dijeljenja nizova na taskove, i to na polovine. To rekurzivno polovljenje urađeno je do najmanje moguće mjere, te je garantovano da je krajnji niz sortiran.

```java
@Override
protected void compute() {
    if(desni - lijevi <= GRANICA) {
        //Sortiranje podniza selection sortom
         for (int i = lijevi; i < desni; i++) {
             int min = i;
               for(int j = i + 1; j < desni + 1; j++) {
                   if(niz[j].compareTo(niz[min]) < 0)
                       min = j;
               }

                T pom = niz[min];
                niz[min] = niz[i];
                niz[i] = pom;
            }
    }
    else {
        //Podjela zadatka na podzadatke
        int srednji = lijevi + (desni - lijevi) / 2;

        SortTask<?> prvaPolovina = new SortTask<T>(niz, lijevi, srednji);
        SortTask<?> drugaPolovina = new SortTask<T>(niz, srednji + 1, desni);

        invokeAll(prvaPolovina, drugaPolovina);
        //Spajanje sortiranih nizova
        MergeSort.merge(niz, lijevi, srednji, desni);
    }

}
```

Slika 19 - Osnova paralelnog selection sorta

Strategija razbijanja posla se sada primjenjuje nekom proizvoljnom *granicom*, gdje, ako je dio niza manji od te granice, taj dio se sortira sekvencijalno. Ako ne, nastavlja se razbijanje zadataka na manje, sve dok se ne dobiju dovoljno mali podnizovi. Na kraju, ovako sortirane podnizove potrebno je spojiti, te ponovo koristimo metod MergeSort.merge().
Granica za selection sort je 500, određena testiranjem nad nizom od 1,000,000 brojeva, gdje je vrijeme izvršenja u opsegu 600ms do 850ms. Ovim je i implicitno rečeno da se za nizove dužine ispod 500 ne isplati paralelizovati ovaj algoritam. To potvrđuje stavku da se paralelizacija ne isplati za manje skupove podataka.

7.4 Implementacija paralelnog insertion sorta

Granica za paralelni insertion sort određena je da bude 300.

```
@Override
protected void compute() {
    if(desni - lijevi <= GRANICA) {

        //Sortiranje podniza insertion sortom
         for (int i = lijevi + 1; i < desni + 1; i++) {

             int j = i;

             while(j > lijevi && niz[j - 1].compareTo(niz[j]) > 0) {
                   T pom = niz[j - 1];
                   niz[j - 1] = niz[j];
                   niz[j] = pom;
                   j--;
               }
            }
    }
    else {
        //Podjela zadatka na podzadatke
        int srednji = lijevi + (desni - lijevi) / 2;

        SortTask<?> prvaPolovina = new SortTask<T>(niz, lijevi, srednji);
        SortTask<?> drugaPolovina = new SortTask<T>(niz, srednji + 1, desni);

        invokeAll(prvaPolovina, drugaPolovina);
        //Spajanje sortiranih nizova
        MergeSort.merge(niz, lijevi, srednji, desni);
    }

}
```

Slika 20 - Osnova paralelnog insertion sorta

7.5 Implementacija paralelnog bubble sorta

Granica za paralelni bubble sort je određena da bude 200.

```
@Override
protected void compute() {
    if(desni - lijevi <= GRANICA) {
        //Sortiranje podniza bubble sortom
         for (int i = lijevi + 1; i < desni + 1; i++) {
                for (int j = lijevi; j < desni; j++) {

                    if (niz[j].compareTo(niz[j+1]) > 0) {
                     T temp = niz[j];
                     niz[j] = niz[j + 1];
                     niz[j + 1] = temp;
                    }
                }
            }
    }
    else {
        //Podjela zadatka na podzadatke
        int srednji = lijevi + (desni - lijevi) / 2;

        SortTask<?> prvaPolovina = new SortTask<T>(niz, lijevi, srednji);
        SortTask<?> drugaPolovina = new SortTask<T>(niz, srednji + 1, desni);

        invokeAll(prvaPolovina, drugaPolovina);
        //Spajanje sortiranih nizova
        MergeSort.merge(niz, lijevi, srednji, desni);
    }

}
```

Slika 21 - Osnova paralelnog insertion sorta

8. Aplikacija i uputstvo za upotrebu

8.1 Zapažanja o implementaciji

Zbog mogućnosti sortiranja cjelobrojnih i decimalnih brojeva jednim te istim algoritmom, postavila se potreba za generalizacijom tipa niza koji sort metode primaju. Jedna od opcija za implementaciju je bila korištenje ArrayList ili Vector klasa, predstavnice Javinog Collections interfejsa. One služe kao zamjena za nizove u Javi i pružaju mnogo pogodnosti kod korištenja. Iako pružaju mnogo opcija za manipulisanje nizovima, za našu situaciju ovo nije pogodnost, jer jedina stvar koju zahtijevamo od niza ili elemenata niza je poređenje, zamjena elemenata i slično. Apstrakcija koju ove klase pružaju dolazi po cijenu performansi. A u sortiranju nizova, jedini važan rezultat je vrijeme izvršenja (i korektnost). Takođe, operacije u algoritmima sortiranja su obično strogo određene i unaprijed poznate, te nije problem napisati ih manuelno.
Preostaje opcija korištenja običnih nizova. Pošto su jednostavniji, očekivano je da nizovi pokažu bolje performanse od Collections klasa. No odmah je očigledno ograničenje nizova u programskom jeziku Java i ostalim C-like jezicima, a to su: jaki tipovi. To jeste, tip podatka koji, u našem slučaju, prima sort metoda mora biti unaprijed poznat. Iz tog razloga koristili smo

mogućnost Jave dodanu u verziji 5 – Generics ili generični tipovi. Ukratko, moguće je proglasiti generički tip na nivou klase ili metode, i koristiti ga unutar njih, po potrebi. Po konvenciji, novi tip je označen sa velikim slovom T, i u kodu treba da predstavlja generički tip. Generički tip koji smo koristili je **<T extends Comparable<T>>**, što prevedeno znači, bilo koji Tip koji nasljeđuje Comparable klasu. Sama Comparable klasa prima T parametar, ili, ima informaciju koja vrsta objekta se poredi, te ga je potrebno ponovo tu navesti. Zaista, Integer, Double i Float klase (java wrapper klase za primitivne tipove) nasljeđuju Comparable. Čak i String. Dakle, moguće je alfabetski sortirati niz nekih riječi. Jedina posljedica ove implementacije na klasični način pisanja ovih sort algoritama je zamjena operatora više i manje, sa compareT() metodom, i ovo je uočljivo u kodu.

8.2 Uputstvo za aplikaciju

U nastavku ćemo nabrojati sve funkcionalnosti koje pruža naša aplikacija, te ćemo objasniti šta pojedini buttoni rade na interfejsu aplikacije, te koja je njihova uloga.

Naša aplikacija omogućava generisanje i sortiranje nizova brojeva, kao i praćenje vremena izvršenja različitih algoritama za sortiranje. Niz može da bude željene veličine, a brojevi se generišu random() funkcijom, u rasponu od -10 000 do 10 000. U nastavku, slijede koraci, kako biste znali koristili aplikaciju:

- *Pokrenite aplikaciju*: Aplikacija se može pokrenuti pomoću shortcut-a, koji se zove „PRS-app“ koji je .exe (Executable JAR File). Dvoklikom na shortcut otvara se aplikacija.
- *Osnovni prozor aplikacije*: Nakon što se aplikacija otvori, pre d korisnikom se nalazi osnovni prozor aplikacije. Tu može vidjeti Naslovnu traku sa nazivom aplikacije i dugmadima za minimizacija, maksimizaciju i zatvaranje aplikacije. Ispod naslovne trake nalazi se TextArea, u kojoj se prikazuju nizovi, vrijeme izvršavanja, sortirani i nesortirani niz, itd. Ispod toga se nalaze neki buttoni, koje ćemo detaljnije objasniti u nastavku.
- *Buttoni ispod TextArea*-e: Ispod textArea-e možete vidjeti četiri buttona: Odaberite file, Random niz, Upis u file i Obrisi ekran. Objasnit ćemo šta radi svaki od ovih buttona.
- *Odaberite File*: Nakon klika na ovaj button otvara se novi prozor – dijalog, koji nam omogućava da pristupimo željenoj lokaciji i da otvorimo željeni file. File-ovi koji se traže i otvaraju po defaultu su .txt fajlovi. Mada, ako korisnik stavi da želi pretraživati sve fajlove, opcijom „Files of Type / All Files“, onda je moguće da i neki drugi file prikaže na TextArea-u, kao što su npr .xml fajlovi ili .sws fileovi, ili bilo koji drugi. Naravno, ova opcija radi najbolje za .txt fajlove.
- *Random niz*: Klikom na button Random niz otvara se prozor u koji je potrebno unijeti veličinu niza. Veličina niza mora biti pozitivan cijeli broj, jer u protivnom, program će prijaviti grešku. Nakon toga, potrebno je odabrati RadioButton da li želimo da su nam brojevi u nizu cjelobrojni ili decimalni. Nakon što odaberemo sve uspješno, pritisnemo button „*Prikaži na ekran*“ gdje nam se otvara početni prozor i u textArei su prikazani svi članovi niza, koji imaju vrijednosti od -10 000 do 10 000, a ima ih onoliko koliko smo odabrali da ih bude. U slučaju da odaberemo button „*Upis u file*“, otvara nam se prozor za odabir lokacije gdje želimo da sačuvamo file. Nakon što odaberemo lokaciju i dadnemo naziv našem fajlu, uspješno smo sačuvali niz u .txt file i vraćamo se u početni prozor.
- *Upis u file*: Ovaj button nam daje mogućnost da text iz TextArea-e upišemo u fajl.

Moguće je odabrati željenu lokaciju na koju želimo spasiti fajl, kao i odabrati željeni naziv fajla.

- *Obriši ekran*: Button obriši ekran, uklanja sve šta se nalazilo na TextArea-i i ostavlja nam čistu površinu.
- *Algoritam*: U ovoj sekciji se nalazi pet radioButtona od kojih trebamo čekirati jedan kako bismo odabrali željeni algoritam. Buttoni su: InsertionSort, SelectionSort, BubbleSort, MergeSort i QuickSort. Pored tih buttona, ovdje se nalazi još čitava jedna potsekcija koja se odnosi na vrijeme izvršavanja niza. U ovoj potsekciji je potrebno odabrati željenu veličinu niza, kao i to da li su brojevi u nizu cjelobrojni ili decimalni.
- *Izvršavanje*: Ova sekcija je namjenjana za odabir radioButtona vezanog za izvršavanje algoritma. RadioButton sekvencijalno ili radioButton paralelno.
- *Vidi vrijeme izvršenja*: Ovo je button koji se nalazi u podsekciji unutar algoritam okvira, a iznad buttona *Sortiraj*. Ako smo odabrali željeni algoritam i željeni način izvršavanja, kao i veličinu niza i brojeve u nizu, onda će se na ekran ispisati vrijeme izvršenja za taj algoritam. Prvo će biti napisan naziv algoritma i da li je to paralelno ili sekvencijalno izvršenje. Zatim će pisati vrijeme izvršenja. Nakon toga piše nivo paralelizma, te na kraju broj ukradenih taskova.
- *Sortiraj*: Ovaj button se nalazi na dnu prozora. Ako smo odabrali željeni algoritam i željeni način izvršavanja, za već postojeći niz koji se nalazi ispisan na ekranu, izvršit će se sortiranje, ispisat će se sada sortirani niz na ekran i na kraju će se ispisati vrijeme koje je bilo potrebno da se to sortiranje izvrši.

8.3 Čitanje koda

8.3.1 Paketi

U našem projektu imamo pet paketa: algorithms, appsInterface, main, model, parallel. Implementacija svih sekvencijalnih algoritama nalazi se u paketu algorithms. On ima i interfejs Sort koji je zajednički za sve sort klase. Paket appsInterface, sadrži čitav izgled aplikacije. Tu se nalaze svi prozori (Frameovi), svi paneli (Panels), a na njima se nalaze svi buttonsi (RadioButtonsi i Buttonsi), te su tu implementirani svi događaji koji se dešavaju nakon što korisnik klikne na određeno dugme. Paket main sadrži main klasu, koja u sebi ima main metodu, od koje kreće izvršavanje aplikacije, a unutar main metode se poziva glavni frame aplikacije. Paket model sadrži dvije klase a to su Algoritam i SortModel. Algoritam je enum, koji sadrži samo nazive algoritama, dok klasa SortModel objedinjenje sve pozive sort metoda za različite slučajeve, ima i generisanje nasumičnih nizova i slično. Paket parallel sadrži sve paralelne implementacije naših algoritama.

9. Hipoteza

- Očekuje se da sekvencijalni algoritmi pokažu eksponencijalno povećanje vremena izvršenja što se više bude povećavala dužina niza.
- Trebali bi uočiti kvadratni rast vremena izvršenja uz rast dužine niza za algoritme bubble, insertion i selection, a nešto sporiji za quick i mergesort. Ipak, očekivano je vrijeme izvršenja nekoliko milisekundi za nizove ispod 1000.
- Paralelizacijom bi trebali postići bolje performanse u svakom slučaju za nizove od 5000 članova ili više. Trebali bi uočiti ravnanje krive odnosa dužine niza i vremena izvršenja. Pretpostavljamo da ovo izravnjavanje neće biti samo linearno za različite algoritme, jer je očekivani overhead za selection, bubble i insertion dosta veliki. Razlog tome je što se

uvodi korak sažimanja podnizova a što više puta moramo sažeti (mergati podnizove), to je veći overhead.

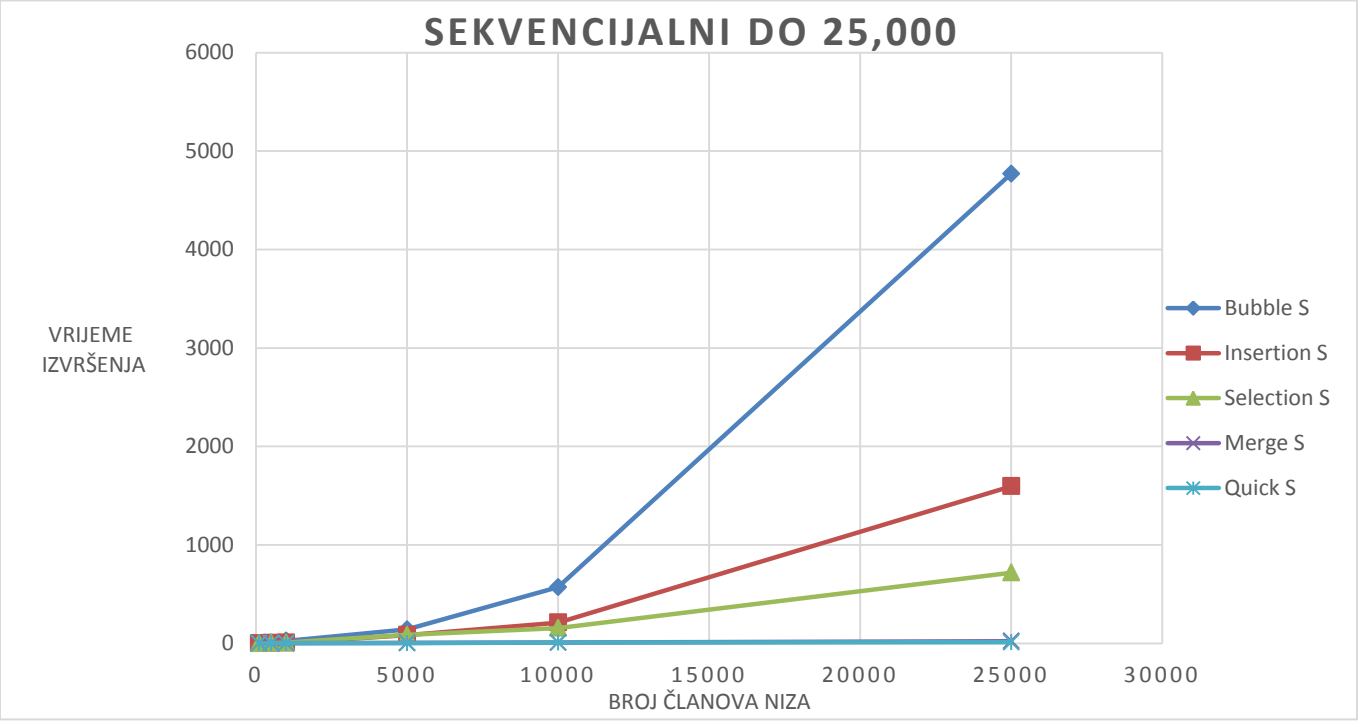
- Možemo očekivati i da bubblesort ima veći overhead, jer ima najnižu granicu od 200, te je on kandidat za najsporiji od svih paralelnih algoritama.
- Paralelni quick i mergesort, pored toga što su brži od ostalih algoritama koji se testiraju, nemaju dodatnih koraka koji već ne postoje u njihovoj implementaciji. Dakle, nemamo mnogo overheada, samim tim, izvršenje je nešto brže.

10. Rezultati

Testiranje je rađeno sa statičnim dužinama niza, od 100 do $50*10^7$. Korišteni su nizovi nasumičnih Double brojeva, sa vrijednostima u opsegu -10,000 do +10,000. Mjerenje je rađeno tako što se sortiranje nanovo pokreće iz radnog okruženja. Ovo sprječava JVM da pravi runtime optimizacije, i nadamo se, daje preciznije i realnije rezultate. *Svako vrijeme izvršenja izvršeno je do 5 puta, što ne daje preciznost individualnim mjerenjima. Ovo ne bi trebalo predstavljati problem, jer je cilj mjerenja da se uoče opšti trendovi i oblik podataka.*

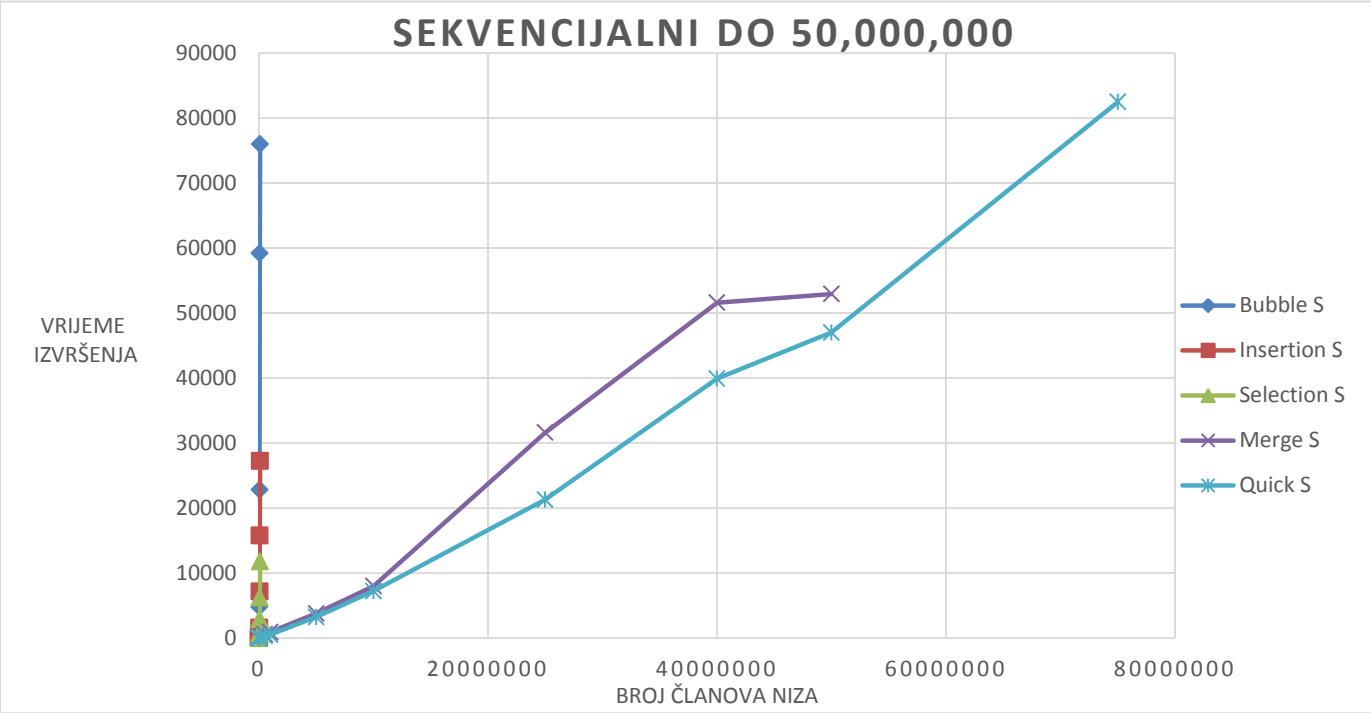
10.1 Sekvencijalna izvedba

U prvom testiranju vidjeli smo da nizovi vremenske kompleksnosti n^2 za relatvino male dužine niza postaju neefikasni. Bubblesort se pokazao najgori, jer za niz od 75000 članova njegovo vrijeme izvršenja bude skoro minut. Insertion i selection su nešto bolji, no i oni teže ka naglom povećanju vremena izvršenja za veličine reda 10^5. Iz tog razloga nisu ni mjerena vremena izvršenja koja su duža od dvije minute.



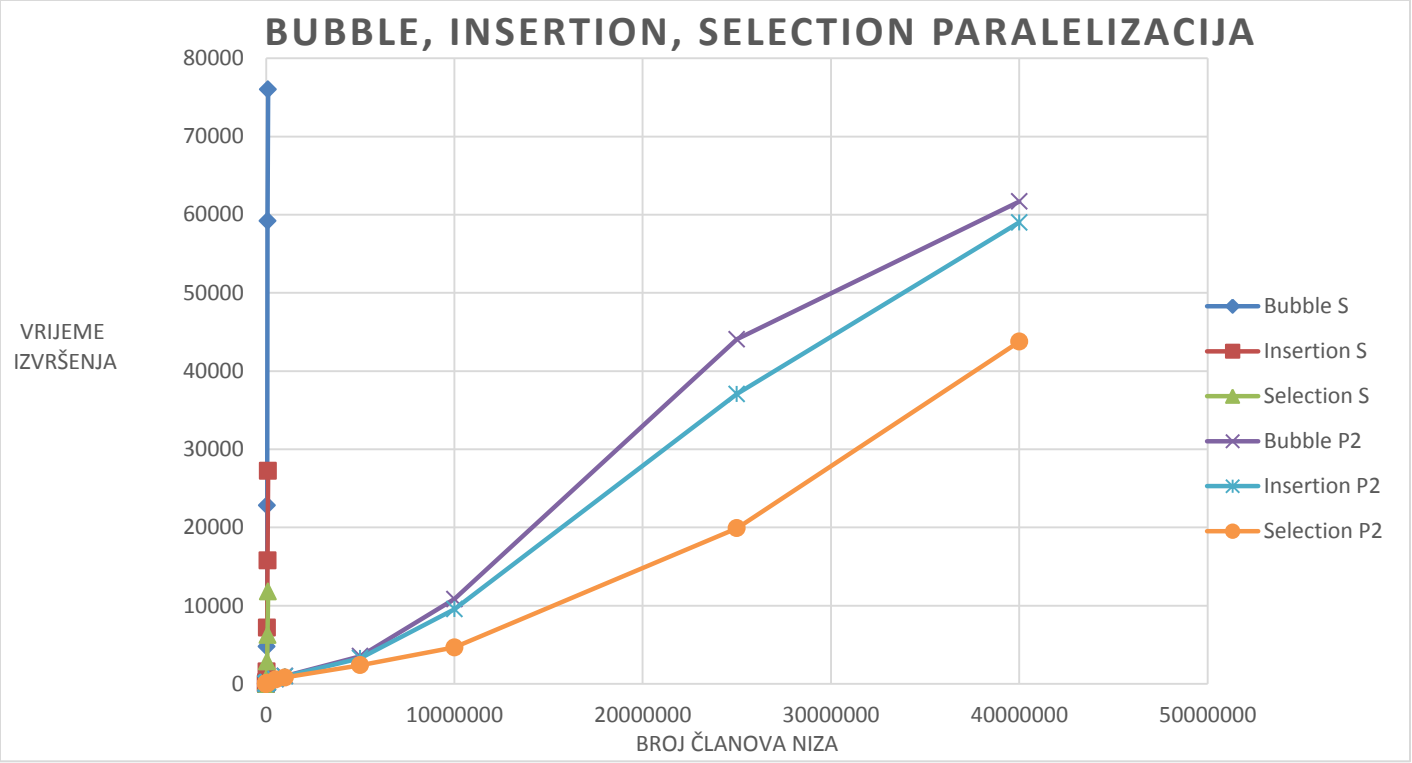
U odnosu na njih, primjećujemo velika poboljšanja kada se koriste quick i merge sort. Gdje je algoritmima reda n^2 trebalo oko minut da sortiraju niz od sto hiljada članova, quick i merge za otprilike isto vrijeme mogu sortirati niz od $5*10^7$ – što je razlika za dva reda veličine, puta pet.

Zbog toga se na donjem grafiku nizovi reda n^2 i ne vide, to jeste, predstavljeni su pravom linijom. Ovim smo htjeli pokazati za koliko male nizove ima smisla koristiti ove algoritme.

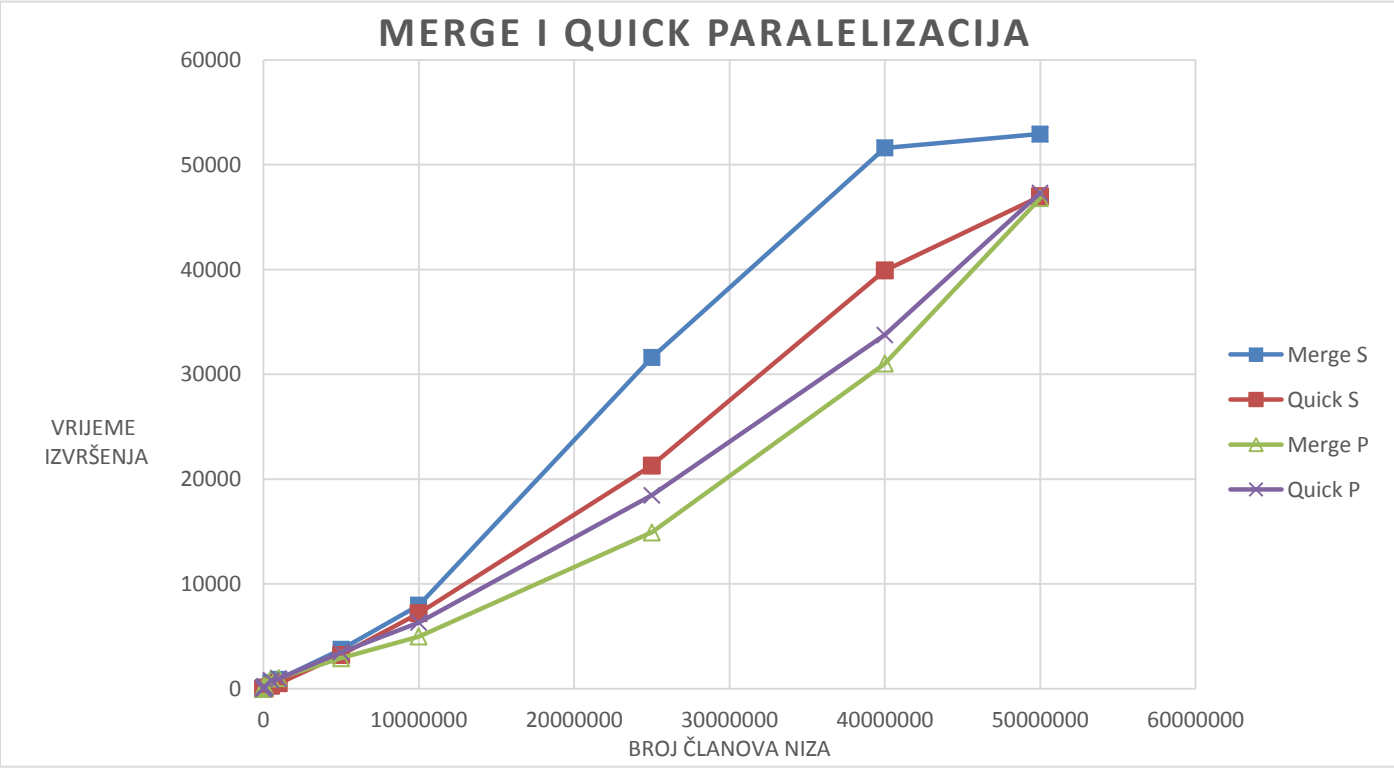


10.2 Paralelna izvedba

Paralelni algoritmi sortiranja pokazali su se efikasnim za duže nizove. I iako su bubble, insertion i selection pokazivali loše performanse, uz paralelizaciju, oni su i za milionske dužine niza pokazali zadovoljavajuće rezultate. Na grafiku, sekvencijalni sortovi predstavljaju prave linije, što znači da su izuzetno eksponencijalno rastućeg vremena izvršenja. Ovo ima smisla, jer vremenska kompleksnost ovih algoritama sigurno više nije n^2 jer se sada, umjesto cijelog niza, podnizovi od 200, 300 ili 500 sortiraju sekvencijalno, pa spajaju. Pored toga, bubblesort se zaista pokazao kao najsporiji među paralelizovanim algoritmima.

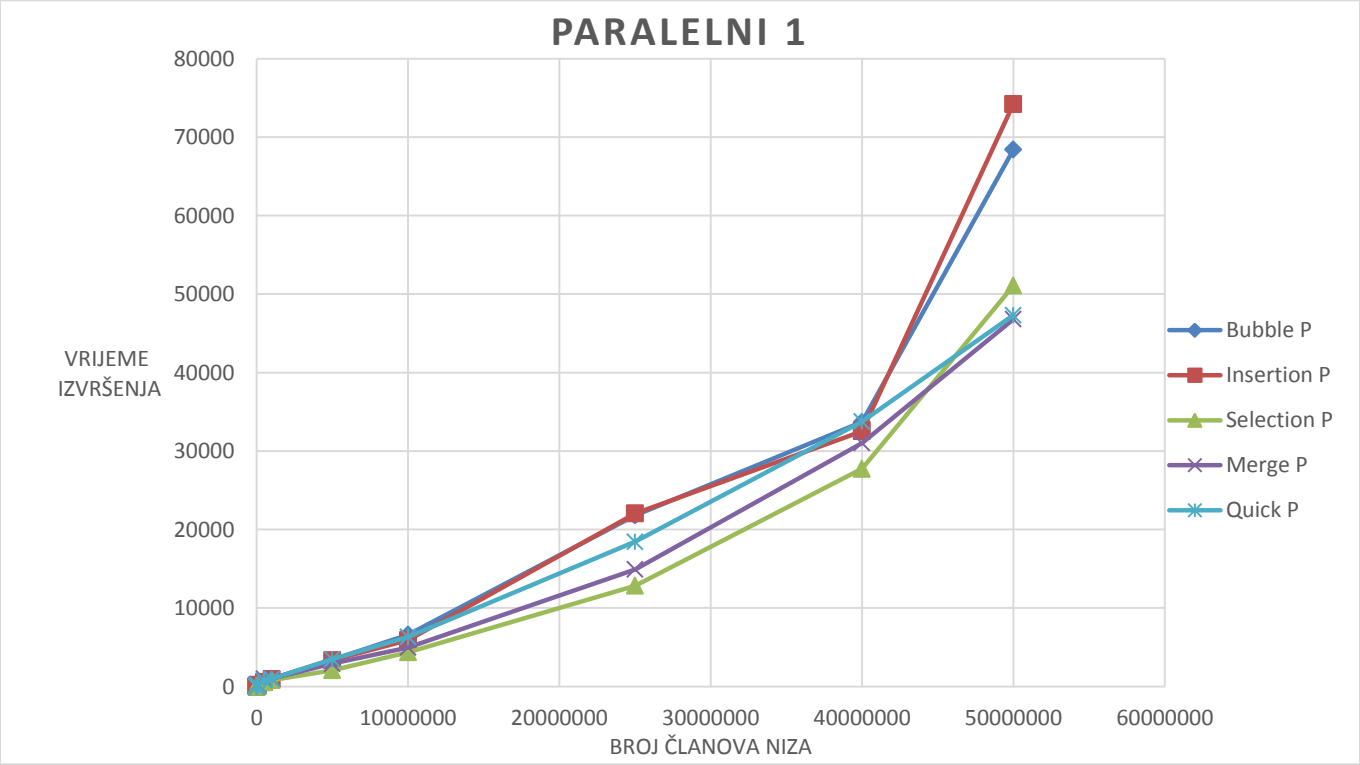


Mege i quicksort pokazali su takođe linearna poboljšanja, što je očekivano. Mergesort je pokazao manju efikasnost, možda zbog njegovog većeg overheada.

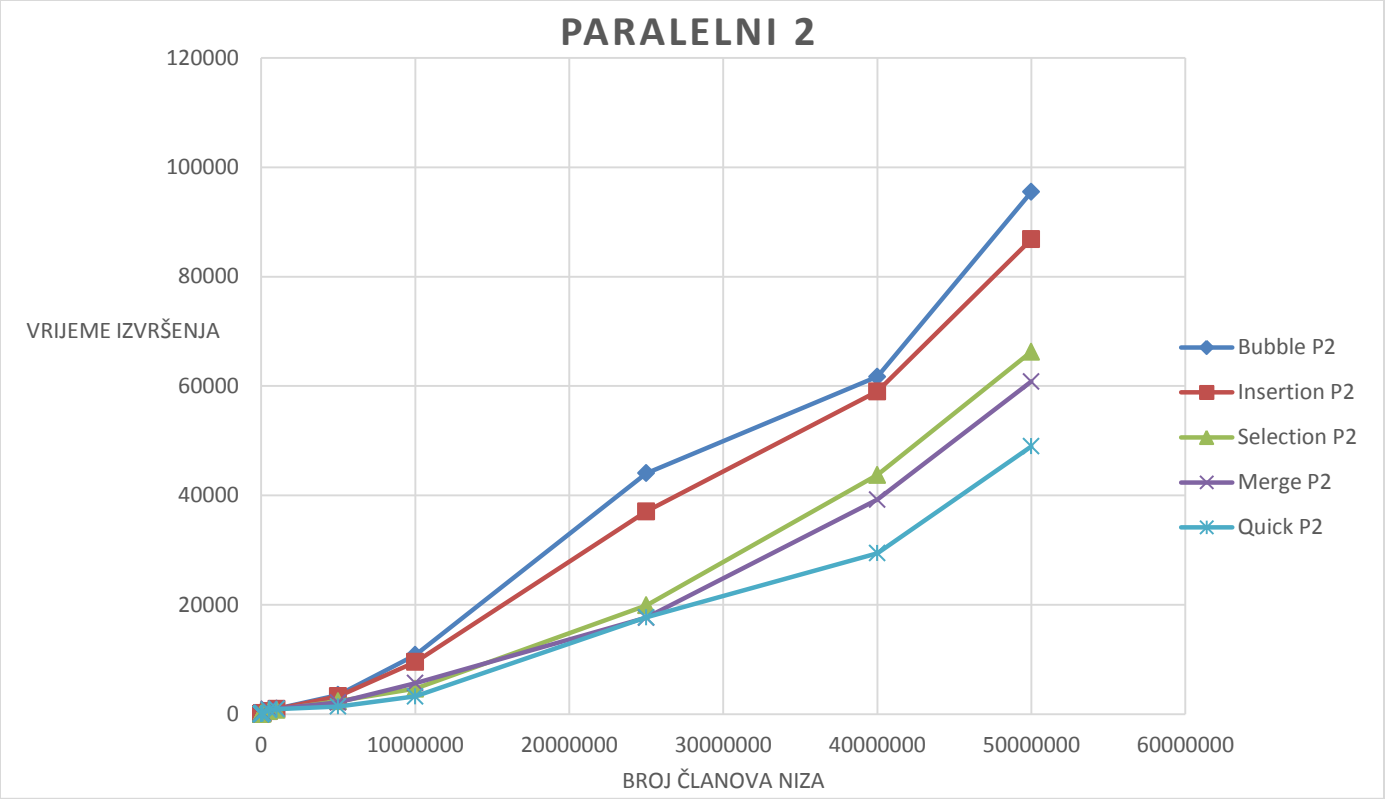


10.3 Dodatna zapažanja

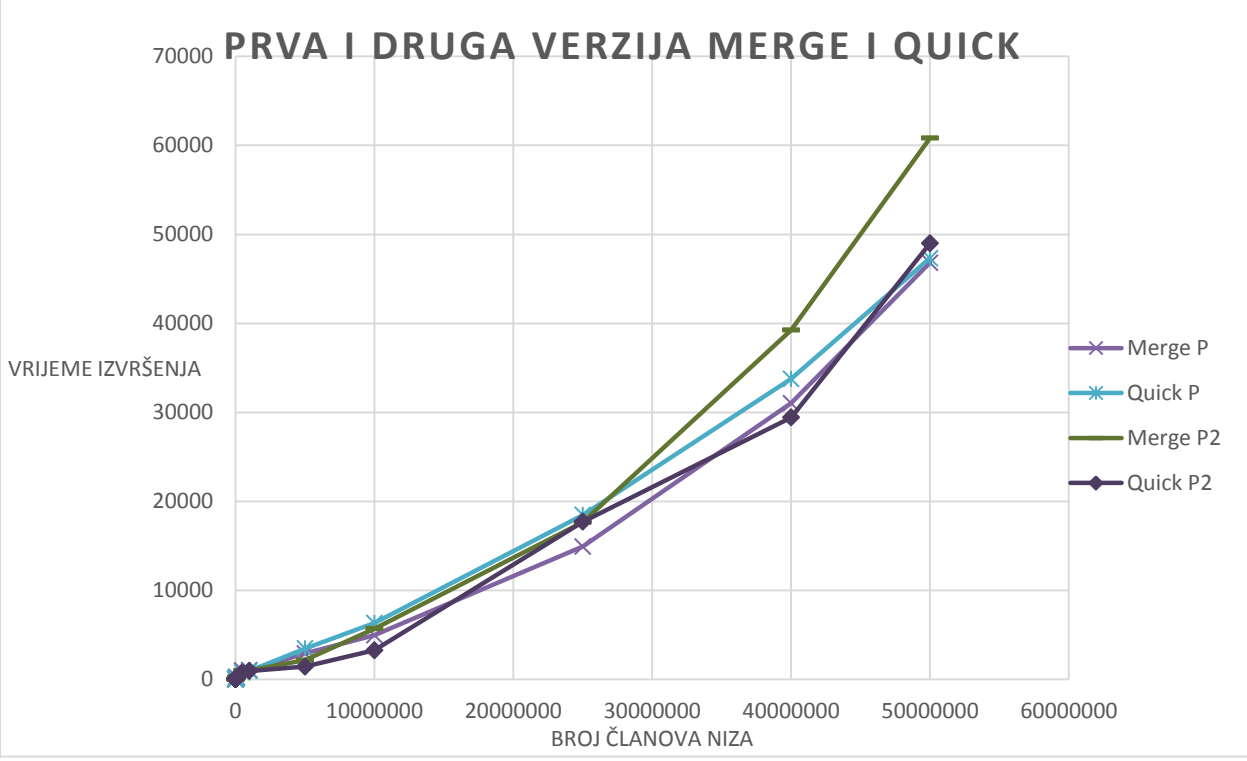
Neobična stvar kod upoređivanja svih paralelizovanih algoritama je to što je selection sort primjetno brži od ostalih. Ovo nas je natjeralo da još jednom uradimo testiranje, o čemu će biti riječi u nastavku. Ovo je neočekivan rezultat, jer merge i quicksort imaju manju vremensku kompleksnost. Jedan mogući uzrok ovog rezultata je optimizacija koju radi JVM. Naime, pri pokretanju sortiranja više puta JIT kompajler radi optimizacije ako vidi da se neke operacije stalno ponavljaju. Ovo je poznato kao „zagrijavanje“ kompajlera. No ovo je samo pretpostavka, i otvoreno pitanje.

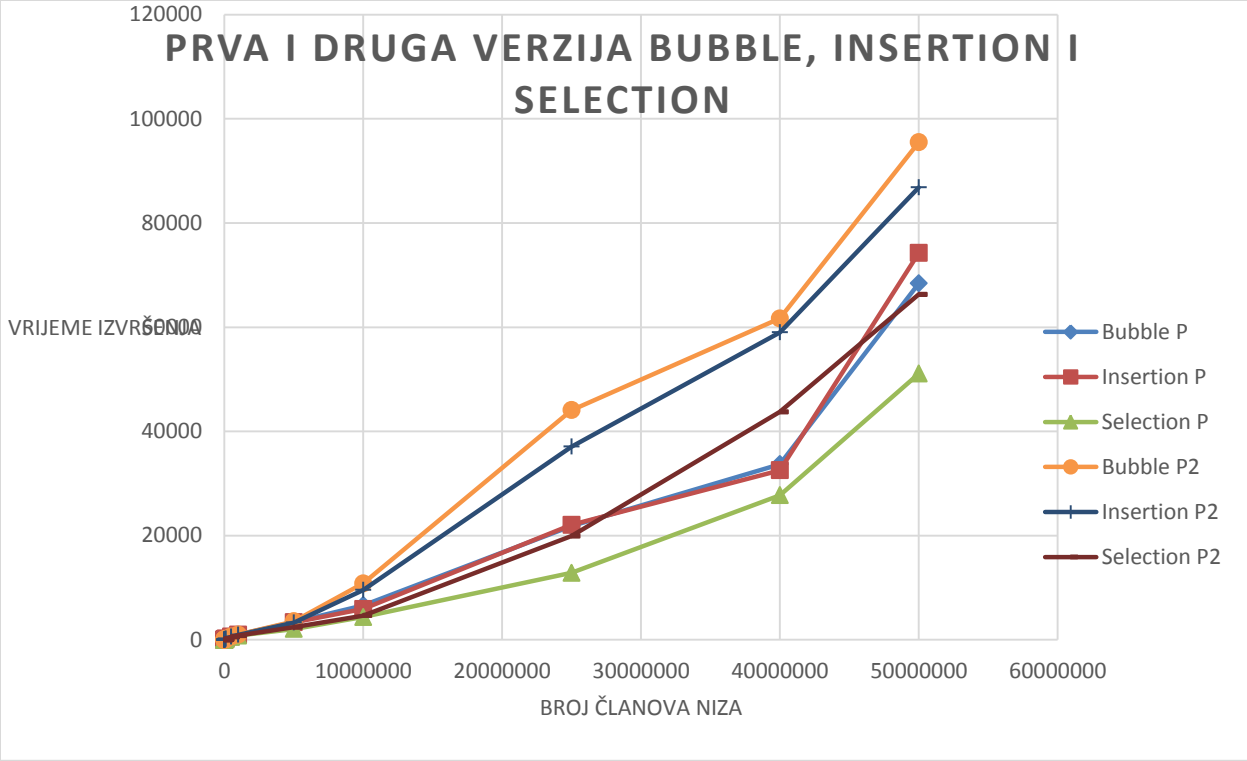


Još neobičnije je što su se rezultati promijenili u jednoj novoj sesiji, nakon restartovanja računara. Ovi rezultati su više ličili onim očekivanim, jer sada su mergesort i quicksort brži od drugih. Selectionsort se opet pokazao prilično brz, ali nikako brži od quick i merge. Ovo podržava teoriju da je do poboljšanja došlo zbog zagrijavanja. Što je još neobičnije, nakon par sati testiranja ovih algoritama, selectionsort je opet pokazao veću brzinu od ostalih, sličnu kao on iz prve ture testiranja. Zbog nedostatka vremena nismo mogli reprodukovati ove rezultate.



Vezano za ovu anomaliju, zanimljivo je što su i ostali algoritmi trpili promjene, ali na sljedeći način. U oba eksperimenta, su se merge i quick algoritmi pokazali stabilni. Dok se za ostale algoritme uočava jasna razlika.





Ne možemo sa sigurnošću reći koji je razlog ovih dešavanja. Postoji mogućnost i da ova 3 algoritma, a pogotovo selection, sadrže neke ponovljive radnje koje se lakše optimizuju.

11. Zaključak

Sveukupno, rezultati testiranja sekvencijalnih i paralelnih izvedbi ovih algoritama za sortiranje nisu bili daleko od očekivanih. Paralelizacija bubble, selection i insertion sorta učinila je ove algoritme korisnim, pored većeg overheada i nepraktičnosti njihove izvedbe. Time smo dokazali da je moguće veoma uprostiti zadatke čiji obim teži jako velikim ciframa, a koji možda nisu po prirodi rekurzivni. Iz eksperimenta sa merge i quicksortom, vidimo da je za već rekurzivne i efikasne algoritme moguće dobiti još bolju efikasnost; naravno, samo za dovoljno velike zadatke.