Paralelne izvedbe algoritama za sortiranje

Šuljug, Krešimir

Master's thesis / Diplomski rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:717944

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-13**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Diplomski studij

PARALELNE IZVEDBE ALGORITAMA ZA SORTIRANJE PODATAKA

Diplomski rad

Krešimir Šuljug

Osijek, 2016.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 08.10.2016.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Krešimir Šuljug
Studij, smjer:	Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo
Mat. br. studenta, godina upisa:	D 714 R, 14.10.2014.
OIB studenta:	39023604689
Mentor:	Doc.dr.sc. Zdravko Krpić
Sumentor:	
Predsjednik Povjerenstva:	Doc.dr.sc. Josip Balen
Član Povjerenstva:	Doc.dr.sc. Tomislav Matić
Naslov diplomskog rada:	Paralelne izvedbe algoritama za sortiranje
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Poznato je da brzina razvoja sklopovlja nadmašuje brzinu razvoja programske podrške. Samim time, potrebno je prilagoditi postojeće algoritme izvođenju na aktualnim paralelnim i raspodijeljenim računalnim sustavima. Algoritmi sortiranja jedni su od češćih koji se pojavljuju u primjeni te u mnogim situacijama iziskuju značajne računalne resurse za njihovo izvođenje. U ovom radu cilj je analizirati mogućnosti paralelizacije najpopularnijih algoritama sortiranja (Quicksort, Merge sort i algoritam po izboru). Potrebno je provesti teorijsku i praktičnu (kroz implementaciju) analizu strukture algoritama sortiranja, uočiti njihove specifičnosti i ograničenja te odrediti mjeru do koje se oni mogu paralelizirati. Korištenjem odgovarajućih mjernih veličina usporediti slijednu i paralelnu verziju svakog od implementiranih algoritama. Po potrebi izraditi korisničko sučelje u
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Vrlo dobar (4)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 Postignuti rezultati u odnosu na složenost zadatka: 2 Jasnoća pismenog izražavanja: 2 Razina samostalnosti: 2
Datum prijedloga ocjene mentora:	08.10.2016.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTIRADA

Osijek,20.12.2016.

Ime i prezime studenta:	Krešimir Šuljug
Studij:	Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo
Mat. br. studenta, godina upisa:	D 714 R, 14.10.2014.
Ephorus podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Paralelne izvedbe algoritama za sortiranje**

izrađen pod vodstvom mentora Doc.dr.sc. Zdravko Krpić

i sumentora

mojvlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osimonih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada,osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, ašto je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1
1.1. Zadatak diplomskog rada 1
2. PARALELNI I SERIJSKI ALGORITMI 2
2.1. Slijedni algoritmi 2
2.2. Paralelni algoritmi 3
2.3. Paralelne sklopovske arhitekture 3
2.4. Paralelizacija slijednih algoritama 5
2.4.1. MPI 6
2.4.2. OpenMP 6
2.5. Paralelne izvedbe algoritama 10
3. ALGORITMI ZA SORTIRANJE 12
3.1. Quick Sort 12
3.2. Merge sort 13
3.3. Radix sort 14
4. USPOREDBA SERIJSKIH I PARALELNIH IZVEDBI ALGORITAMA ZA SORTIRANJE 16
4.1. Usporedba serijske i paralelne izvedbe Quick sort-a 17
4.2. Usporedba serijske i paralelne izvedbe Merge sorta-a 19
4.3. Usporedba serijske i paralelne izvedbe Radix sort-a 22
5. ZAKLJUČAK 26
LITERATURA 27
SAŽETAK 29
ABSTRACT 30
ŽIVOTOPIS 31
PRILOZI 32
Izvorni c++ kodovi 32

1. UVOD

Sortiranje podataka je jedan od osnovnih i važnijih načina za obradu podataka. Najvažniji razlog sortiranja je olakšavanje korištenja samih podataka. Sortirani podaci se mogu lakše pretraživati, grupirati po određenoj potrebi i slično. Sortiranje se koristi za različite upotrebe, primjerice sortiranje knjiga u knjižnicama ili knjižarama, razne robe u skladištima, medicinskih kartona i sl.

Kako je s vremenom čovjek imao previše podataka za ručno sortiranje prepustio je taj posao računalu. Tako se računalo zbog svoje brzine i mogućnosti automatizacije pokazalo kao odličan alat za pohranu, obradu i rad s podacima, pa tako i za sortiranje podataka. Međutim, u današnje doba se pokušava postići što bolje tj. brže sortiranje. Pomoću paralelizacije programskih kodova ili algoritama, ljudi su shvatili da mogu još bolje iskoristiti resurse i brzinu računala. To znači da se resursi računala koriste istovremeno, a svaki resurs (bila to memorija, procesor ili grafička kartica) će obavljati dio zadatka nakon čega, kada se taj zadatak obavi, se parcijalni rezultati spoje u jedno rješenje. Paraleliziranim načinom korištenja računala, dobivaju se višestruko bolje performanse izvođenja programa, nego kad se on izvodi serijski (sekvencijalno).

Tema ovog diplomskog rada je, paralelna izvedba algoritama za sortiranje podataka. Tijekom ovog rada će biti objašnjen pojam algoritma, njegova kratka povijest, svrha i sl. Nakon toga će biti objašnjen pojam paralelizma, tj. paraleliziranje algoritama, pomoću kojih tehnologija se kodovi paraleliziraju i na koji način funkcioniraju. Također će biti opisana detaljna analiza vremenske složenosti navedenih algoritama, kako u serijskoj tako i u paralelnoj izvedbi.

Kako ne bi sve ostalo na teoriji, biti će prikazani primjeri algoritama u svojoj serijskoj i paralelnoj izvedbi. Napravit će se usporedba vremena potrebna za sortiranje tih podataka. Za sortiranje koristit će se nizovi nasumičnih brojeva koje će algoritmi morati sortirati po veličini.

1.1. Zadatak diplomskog rada

Teorijska obrada nekoliko algoritama za sortiranje (*Quick sort, Merge sort* i jednoga po izboru). Izrada konzolne aplikacije pomoću programskog jezika C++,u kojoj bi se prikazala usporedba paraleliziranog i serijskog načina sortiranja za pojedini algoritam, odnosno vrijeme potrebno za sortiranje određenog niza i omjer trajanja sortiranja serijske i paralelne izvedbe.

2. PARALELNI I SERIJSKI ALGORITMI

Pojam algoritam se koristi u računarstvu, matematici, fizici i sličnim znanstvenim disciplinama kao prikaz nekog zadatka ili programa korak po korak. Prema [1], algoritam (postupnik) je logički slijed operacija koje će izvršiti računalni program.

"Algoritam" kao riječ dolazi od latinskog prijevoda imena perzijskog matematičara Al-Hawarizmija koji je u 9. Stoljeću u Europu uveo arapski sustav decimalnih brojeva, a bavio trigonometrijom, astronomijom, geografijom i drugim znanstvenim disciplinama.

Prvo prilagođavanje nekog algoritma računalu pripada Adi Byron, koja se također smatra i prvom programerkom. Njezin algoritam je računao Bernoullijeve brojeve, a bio je napisan za analitički stroj engleza Charlesa Babbagea. Međutim ovaj stroj nikad nije proveden u djelo zbog svoje veličine i velikih mehaničkih dijelova tako da i njezin algoritam u stvarnosti nikad nije bio ni testiran. Turingov stroj je riješio većinu problema matematičarima i logičarima iz 19. i 20. stoljeća, a predstavio ga je Alan Turing. Primjenom njegovog stroja definirani su mnogi moderni problemi vezani uz analizu algoritama (iz [2]).

Algoritmi se danas mogu opisati na razne načine. Pseudo kod, dijagram tijeka ili neki od programskih jezika se koriste za pojašanjavanje algoritama. Programski jezici su prvenstveno namijenjeni kako bi se algoritam izrazio u obliku koji se može izvršiti na računalu, ali se i često koriste kao način definiranja ili dokumentiranje algoritma.

2.1. Slijedni algoritmi

Slijedni (sekvencijalni, serijski) algoritmi su reprezentativna verzija algoritama ili pseudo kodova koji se izvršavaju samo pomoću jedne procesorske niti. Kako nekim zadacima treba više vremena da se izvrše pomoću samo jedne procesorske niti, koriste se drugi načini kako bi se takav proces ubrzao. Jedan od njih je paralelizacija algoritama.

2.2. Paralelni algoritmi

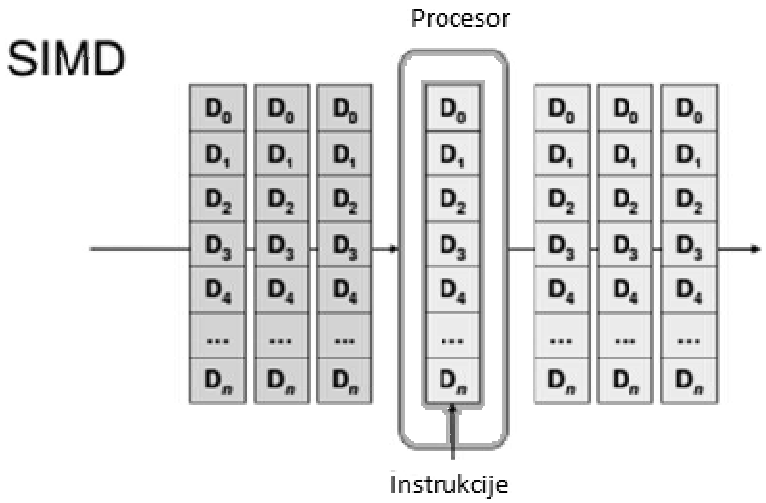
Glavni razlog paraleliziranja programskih kodova je rješavanje zadataka u kraćem vremenu nego što bi zahtijevalo slijedno računalo. Kako u današnje vrijeme dolazi do novih zahtjevnijih zadataka, tako nastaju i novi zahtjevi za sve većom računalnom moći, što znači da računala neće nikada postati dovoljno brza. Neki od takvih problema su modeliranja i simulacije, tj. rad na principu slijedne aproksimacije, više računanja, veće preciznosti (simulacije utjecaja klimatskih promjena, seizmičkih pojava, turbulencija fluida i sl.). Zbog svih tih zahtjeva potrebno je obraditi veliku količinu podataka, a neki od tih su i sortiranja u velikim bazama podataka. Da bi se iskoristila prednost paralelnog programiranja, program mora biti napisan „paralelno“, tako da se izvođenje programa može odvijati pomoću više procesa i niti, gdje je proces jedna instanca programa koji se izvodi samostalno na procesoru (prema [3]).

Kako bi se omogućila što učinkovitija paralelizacija, moraju biti zadovoljeni određeni sklopovski i programski zahtjevi, kao u [6]:

- komunikacija između procesora i memorije mora omogućavati brzu komunikaciju između individualnih procesa, kao i brzu izmjenu podataka u i iz memorije,
- mora postojati dostupan protokol za međusobnu komunikaciju između procesa,
- mora biti moguće učinkovito podijeliti potrebne računalne algoritme i ulazne podatke u odvojene procese.

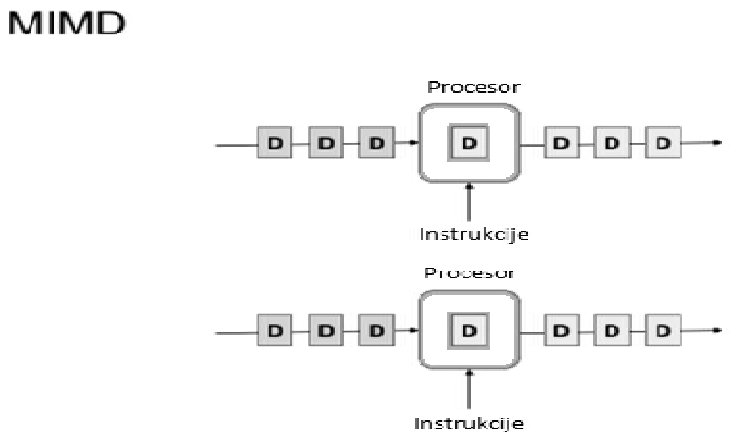
2.3. Paralelne sklopovske arhitekture

Neke od sklopovskih arhitektura koje se koriste i koje su pogodne za paralelizaciju su SIMD i MIMD. SIMD (engl. *Single Instruction Multiple Data*) se sastoji od jednog procesora koji se upotrebljava za kontrolu odvojenog skupa računalnih jedinica, koji svaki ima svoju vlastitu memoriju. Glavni nedostatak SIMD arhitekture je ta što mnoge računalne jedinice mogu ostati u stanju čekanja jer kontrolni procesor odašilje jednu instrukciju svakoj od tih jedinica, koja ili izvršava tu instrukciju ili ostaje u stanju čekanja (kako je navedeno u [3]).



Sl. 2.1. Prikaz SIMD arhitekture, iz [4]

Suprotno od SIMD arhitekture, kod MIMD arhitekture (engl. *Multiple Instruction Multiple Data*) svaki procesor ima kontrolnu i računalnu jedinicu. To znači da svaki procesor izvršava svoj vlastiti potprogram brzinom koja ne ovisi o ostalim procesorima u sustavu. Vrsta MIMD arhitekture koja se danas koristi je SMP (engl. *Shared Memory Processing*). Ona koristi kombinaciju dvije MIMD arhitekture, onu s dijeljenom memorijom i raspodijeljenom memorijom. Arhitektura sa dijeljenom ili zajedničkom memorijom funkcionira na način da više procesora rade više zadataka, ali koriste isti memorijski spremnik što znači da je čitanje i pisanje ograničeno na samo jedan procesor istovremeno. S druge strane arhitektura sa raspodijeljenom memorijom radi na način da više procesora rade neovisno sa svojim vlastitim memorijskim spremnicima. Od SMP arhitekture se očekuje najviše u daljnjem razvoju paralelizma visokih performansi. Prednosti SMP-a su te što svaki procesor ima direktan pristup dijeljenoj memoriji, ali također omogućuje procesoru u jednom čvoru (računalu) da pristupa podacima u memoriji drugih čvorova i taj dio zahtijeva korištenje paralelnog programiranje, kao što je navedeno u [3].



Sl. 2.2. Prikaz MIMD arhitekture, iz [4]

2.4. Paralelizacija slijednih algoritama

Koristi se nekoliko koraka kako se slijedni algoritam pretvara u paralelni, prema [3].

a) Pronaći dijelove slijednog programa koji se mogu izvoditi istodobno:
   - zahtjeva detaljno poznavanje rada algoritma,
   - može zahtijevati i potpuno novi algoritam.

b) Rastaviti algoritam:
   - funkcionalna dekompozicija – podjela problema na manje dijelove (podjela na dijelove koji se mogu riješiti istovremeno),
   - podatkovna dekompozicija – podjela podataka s kojima će algoritam raditi na manje dijelove (obično je ova dekompozicija jednostavnija za izvesti),
   - kombinacija funkcionalne i podatkovne dekompozicije.

c) Ostvarenje programa:
   - odabir programske paradigme i sklopovskog okruženja,
   - usklađivanje komunikacije,
   - banjska kontrola izvođenja.

d) Ispravljanje grešaka i optimizacija izvođenja.

Kako se neki algoritmi mogu zapisati na više načina, tako se i sam način paralelizacije može izvesti korištenjem različitih tehnologija. Neke od tih tehnologija su tehnlogija izmjene poruka (engl. *Message Passing Interface* – MPI), tehnologija većeg broja obrađivanja (engl. *Open Multi-Processing* – OpenMP), tehnologija hibridnog više procesorskog paralelnog programiranja (engl. *Open Hybrid Multi-Core Parallel Programming* - OpenHMPP) i drugi.

2.4.1. MPI

MPI je nastao 1992. godine kada je nastala prva verzija MPI 1.0. Verzija 1.0 je bila temeljena samo na međuprocesnoj komunikaciji od točke do točke (samo par procesa može izmjenjivati podatke). Međutim, MPI se nastavio razvijati pa tako danas postoji verzija 3.1 koja je izašla 2016. godine. Ona sadrži puno više funkcija i mogućnosti za rad u paralelnom okruženju u odnosu na svoje prijašnje verzije. MPI je podržan u nekim od najkorištenijih programskih jezika, kao što su C,C++ i FORTRAN.

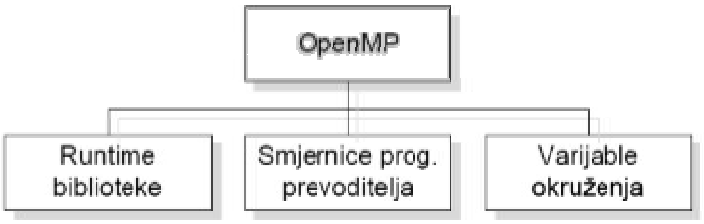
Neke od glavnih značajki MPI-ja su da svaki program u C/C++ koji koristi MPI mora uključiti predprocesorsku naredbu #include <mpi.h>. Također dvije funkcije koje se moraju naći u svakom MPI programu su *int MPI_init (int *argc, char ***argv)* i *int MPI_Finalize()*. Prva funkcija se mora nalaziti na početku bloka naredbi u kojem se nalazi MPI komunikacija, dok se druga nalazi na kraju, kako stoji u [4].

2.4.2. OpenMP

Drugi način paralelizacije je pomoću OpenMP-ja (engl. *Open Multi-Processing*). Ovaj način je korišten u izradi paralelnih izvedbi algoritama u ovom radu. OpenMP je API (engl. *Application Programming Interface*) koji se sve više koristi za paralelno programiranje u okruženju sa zajedničkom memorijom.

OpenMP se sastoji od 3 komplementarne komponente (prema [4]):

- set smjernica koje koristi programer za komunikaciju s programskim prevoditeljem vezano uz paralelizam,
- *runtime* biblioteke tj. funkcije koje omogućuju postavljanje paralelnih parametara i upite nad njima, kao što je primjerice broj niti koji će sudjelovati u obradi podataka ili instrukcija,
- ograničen broj varijabli okruženja koji se mogu koristiti za određivanje paralelnih parametara prilikom izvođenja aplikacije, kao što je npr. broj niti.



Sl. 2.3. Komponente OpenMP standarda, iz [7].

OpenMP okruženje ima tri temeljne funkcije. To su *omp_get_thread_num()* koja vraća rang niti u paralelnom dijelu koda, *omp_set_num_threads()* koji postavlja broj niti koje će se koristiti u paralelnim dijelovima koda koji slijede i *omp_get_num_threads()* koji vraća broj niti koje se koriste u paralelnom dijelu koda, kako je navedeno u [5].

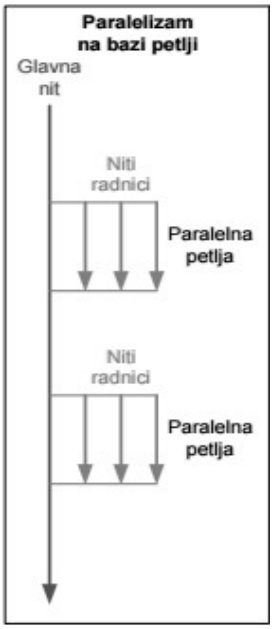
Glavni način paralelizacije pomoću OpenMP-ja jest pomoću smjernica koje se ubacuju u programski kod. Jedna od najosnovnijih i najmoćnijih smjernica je *parallel for*.

Parallel for znači da se petlja koja slijedi izvršava paralelno. Postoji nekoliko načina određivanja koliko će se niti koristiti. Jedan od njih jest taj da se varijabla okruženja OMP_NUM_THREADS postavi na potreban broj niti. Kada programski kod naiđe na smjernicu *#pragma omp parallel for*, broj iteracija petlje je podijeljen između niti. Način na koji je broj iteracija petlje podijeljen između niti zove se raspored. U statičkom rasporedu, koji je u većem broju slučajeva postavljen kao zadani, svaka nit dobiva gotovo jednak dio iteracija petlje (prema [5]).

OpenMP ima 2 glavna pristupa podijeli posla između niti (kako je navedeno u [7]):

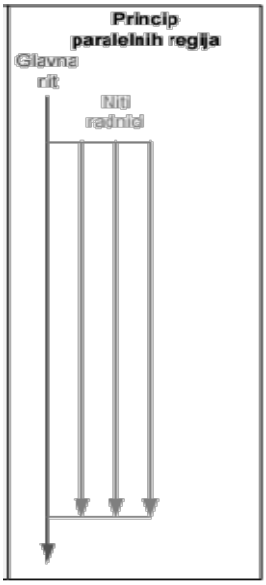
- paralelizacija na razini petlje,
- paralelizacija na principu paralelnih područja.

Kod pristupa na razini petlje, pojedinačne petlje su paralelizirane na način da je svakoj niti dodijeljen određen broj iteracija te petlje. Kod koji nije unutar paralelnih smjernica izvodi se serijski.



Sl. 2.4. Paralelizam na razini petlje, iz [7].

Kod pristupa paralelnih regija svaki dio koda se može paralelizirati, ne samo petlje. Posao unutar regija se eksplicitno dijeli među nitima koristeći identifikatore niti. U programskom kodu se najčešće koristi *if* uvjet (*if(ID==1)*), gdje je ID identifikator niti, kako bi dodijelili posao određenoj niti (iz [7]).



Slika 2.5. Paralelizam na razini regija, iz [7].

Prednost paralelizacije na razini petlje je jednostavnost implementacije istog. Dodatno pojednostavljenje paralelizacije na razini petlje dobiva se kada mali broj petlji odrađuje veliki dio posla u kodu. Pristup paralelnih regija je malo zahtjevniji, ali je fleksibilniji. Nedostatak pristupa paralelnih petlji je taj što se gubi na procesorskom vremenu prilikom stvaranja niti na početku petlje i uništavanjem istih niti na kraju. Količina utrošenog vremena ovisi o detaljima implementacije niti. Prednost principa paralelnih regija je ta što omogućuje programeru iskorištavanje podatkovnog paralelizma u većoj mjeri nego što je to moguće s principom paralelnih petlji, zato što se podaci ne trebaju ponovo sinkronizirati nakon svake petlje (prema [7]).

Kako bi se lakše izvela paralelizacija, mijenjalo ponašanje smjernica tj. kako bi bile praktičnije za korištenje, u OpenMP-u postoje odredbe (engl. *clauses*). Kako OpenMP koristi zajedničku memoriju, sve varijable u određenoj petlji dijele isti adresni prostor. U nekim situacijama to i nije najpraktičnije iz razloga što rezultat koji se dobije na kraju može biti netočan. Zbog navedenog se koristi odredba *private*. Kada programski prevoditelj naiđe na odredbu *private*, on zapravo dodjeljuje odvojene memorijske lokacije kopijama te varijable za svaku nit. Bitno je napomenuti da varijabla nije ponovo inicijalizirana nego je samo kopirana za svaku nit posebno. Ako je potrebno paralelizirati s više privatnih varijabli, jednostavnije je to odraditi s odredbom *default(private)*, samo što se u tom slučaju moraju navesti koje su varijable dijeljene, a to se radi pomoću odredbe *shared* (prema [8]).

Još neke od odredbi vezane za rad s varijablama su *firstprivate* i *lastprivate*. *Lastprivate* je specifičan po tome što na završetku izvođenja petlje, nit koja je izvela zadnju iteraciju, vlastitu vrijednost varijable kopira u glavnu nit. Važnost toga je da se na ovaj način zadržava vrijednost varijable i tako daje sličan rezultat kao serijsko izvođenje koda. Međutim *firstprivate* omogućuje da svaka nit napravi svoju instancu varijable i ta varijabla je inicijalizirana s vrijednostima varijable koju ima prije pozivanja *firstprivate* odredbe (prema [8]).

Kako je već navedeno, smjernica *parallel for* je jedana od najčešće korištenih smjernica OpenMP-a. Iz razloga što se najčešće koristi, moguće ju je razdvojiti na više dijelova. To je omogućeno pomoću smjernice *parallel* gdje se označava da će se slijedeći blok naredbi izvršiti paralelno. Na taj način će se uštediti na procesorskom vremenu jer se neće morati svaki put kreirati i uništavati niti kad se poziva petlja, već se to čini prije niza petlji i poslije njih. Smjernica *parallel for* se inače sinkronizira na barijeri pri završetku, ali ako to nije potrebno i ako slijedeća petlja nema nikakve povezanosti s trenutnom, može se koristiti odredba *nowait*. Na

taj način čim nit završi s radom na prvoj petlji može preći na slijedeću bez čekanja ostalih niti da završe svoj posao (kako je navedeno u [6]).

2.5. Paralelne izvedbe algoritama

Gotovo svi algoritmi za sortiranje mogu imati svoju paralelnu verziju. Međutim za neke algoritme se paralelizam ne isplati, tj. vrijeme izvođenja koda se neće bitno smanjiti, koliki god postotak koda da je napisan paralelno. Dobar primjer za paraleliziranje algoritma koje se ne isplati je „Kružno sortiranje“ (engl. *Bubble sort*). Paralelna izvedba *Bubble sort*-a se zove „ne parni-parni“ (engl. *Odd-even*) algoritam. Iako je kod paraleliziran njegova vremenska složenost je ista kao i kod *Bubble sort*-a, $O(n)$, za najbolji slučaj, a $O(n^2)$ za najgori slučaj, prema [10].

Iz navedenog razloga u ovom radu su obrađeni algoritmi sortiranja koji pokazuju puno bolja vremena u svojim paralelnim izvedbama. Bolje vrijeme izvedbe je moguće zbog rekurzivnog pozivanja funkcija koje služe za sortiranje, a time i rada s više niti. Međutim više niti ne znači i brži algoritam. Procesoru je potrebno neko vrijeme da napravi nit i uništi je kada završi s poslom. Iz tog razloga je potrebno na pažljiv način primijeniti paralelizaciju i staviti nekakvo ograničenje maksimalnog broja niti koje će biti korišteno u paralelnim dijelovima algoritma.

Kod paralelizacije već gotovih algoritama, vrlo je bitno obratiti pažnju kako taj algoritam funkcionira. Ako algoritam u svom rekurzivnom pozivanju koristi iste podatke, bitno je koristiti odredbu *firstprivate*, koja kopira vrijednost te varijable u svoju memoriju i na taj način omogućuje da svaka nit ima neku svoju vrijednost te varijable i ne utječe na rad druge niti, tj. kako nebi došlo do prepisivanja vrijednosti (prema [8]).

U algoritmima koji su obrađeni u radu „Brzo sortiranje“ (engl. *Quick sort*), “Sortiranje spajanje podataka“ (engl. *Merge sort*) i “Korijensko sortiranje“ (engl. *Radix sort*), korišteni su različiti primjeri paralelizacije pomoću paralelnih regija. To je omogućeno pomoću posebnih smjernica koje su moguće tek u verziji OpenMP-a 3.0 i više, a to su *task* i *sections*. Zadaci stvoreni smjernicom *task*, tj. *task*-ovi su specifični zato što omogućuju programeru da označi točno koje dijelove koda želi izvršiti paralelno. Još jedna prednost kod *task*-ova je ta što ako neka nit završi prije druge, neće čekati da završi nego će joj pomoći.

S druge strane, smjernica *sections* točno označava gdje je početak paralelne regije i koristit će samo onoliko niti koliko je puta pozvana smjernica *section*. Bitna stavka kod smjernice *sections* je ta što će program čekati na kraju regije dok svi dijelovi te regije ne završe, tj. ako neki *section* završi prije od drugog, čekat će na barijeri dok svi dijelovi ne završe, umjesto da slobodna nit

pomogne drugim nitima. Iz tog razloga je smjernica *task* puno fleksibilnija za korištenje ako se ne zna točan broj niti koji je raspoloživ. Bilo koja nit može raditi na bilo kojem *task*-u i tako niti mogu biti puno bolje iskorištene. Također nema čekanja dok drugi *task*-ovi završe, kako je navedeno u [9].

Ako se koristi samo jedna paralelna regija, programer može vrlo jednostavno kontrolirati koliko će se niti koristiti pomoću funkcije *omp_set_num_threads().*

3. ALGORITMI ZA SORTIRANJE

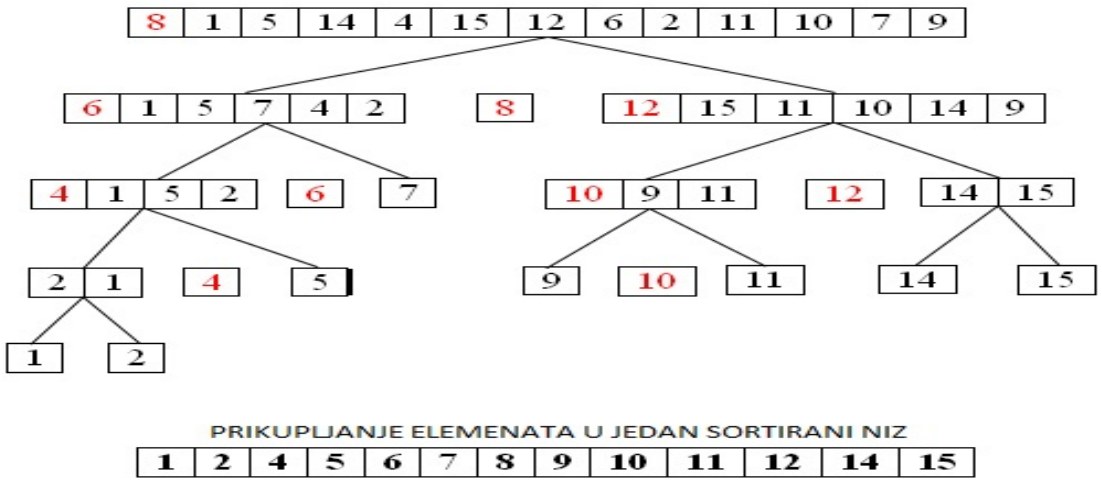
Neki od najpoznatijih i najčešće korištenih algoritama za sortiranje, a obrađeni su u ovom radu su *Quick sort*, *Merge sort* i *Radix sort*. Razlog zašto su baš oni odabrani je zbog njihovog načina, odnosno strategije sortiranja. Mogućnost podjele podataka i rekurzivnog pozivanja funkcija olakšava paralelizaciju njihovih serijskih izvedbi.

3.1. Quick sort

Quick sort je jako brz algoritam za sortiranje koji ima široku primjenu, kako u edukacijske svrhe za objašnjavanje algoritama i sortiranja, tako i u praksi za rad nad podacima. Njegova uobičajena vremenska složenost je $O(n \log n)$, što ga čini jednim od boljih (bržih) algoritama za sortiranje velike količine podataka. *Quick sort* koristi strategiju „podijeli pa vladaj“ (engl. *Divide-and-conquer*). Primjer *Quick sort*-a je prikazan na slici 3.1, a tri su koraka potrebna kako bi se pravilno izveo *Quick sort* (prema [11]):

1. izabere se jedan član niza koji se naziva pivot (crveni brojevi na slici 3.1.),
2. nakon toga se svi elementi manji od vrijednosti pivota prebace prije pivota, a veći poslije pivota,
3. rekurzivno se pozivaju 1. i 2. korak za podnizove s manjim i većim vrijednostima od pivota.

Kada rekurzija dođe do niza veličine 0 ili 1, tada taj niz ne mora biti sortiran, što znači da je glavni niz (početni) sortiran (kako je prikazano u [12]).



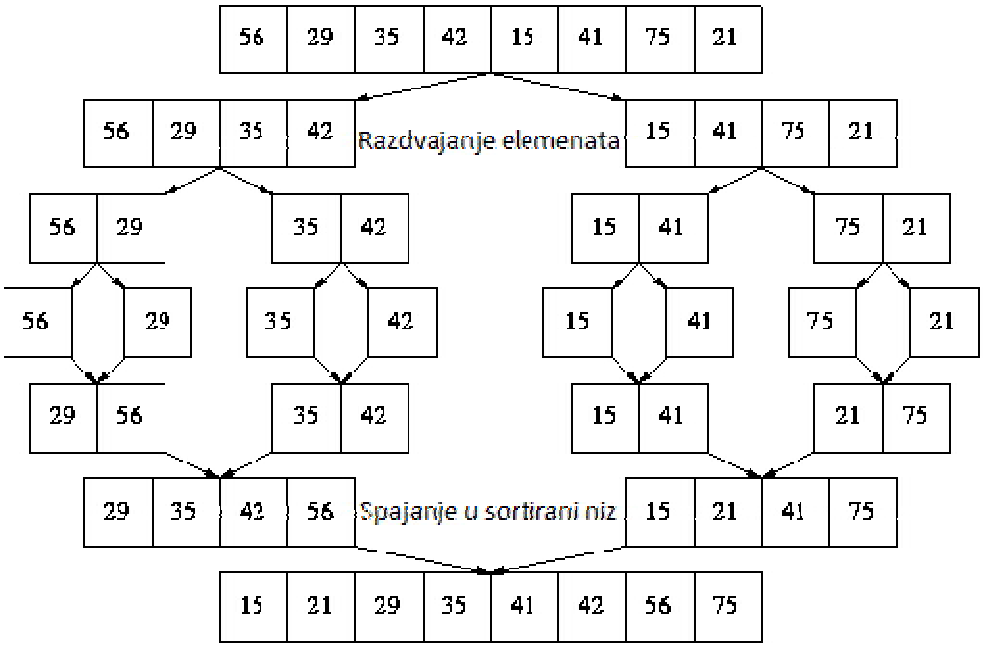
Sl. 3.1. Primjer *Quick sort*-a

Quick sort se zbog svoje strategije, „podijeli pa vladaj“, vrlo jednostavno može paralelizirati. Nakon što se niz podijeli na 2 dijela, podijeljeni nizovi mogu biti sortirani paralelno, svaka nit svoj podniz, a zbog rekurzivnog pozivanja broj niti se može povećati do maksimalnog broja niti koji se odredi u algoritmu. Na taj način se uvelike smanjuje vrijeme potrebno za sortiranje.

3.2. Merge sort

Merge sort je također jedan od bržih algoritama za sortiranje. On je iz obitelji algoritama koje koriste spajanje, a nazivaju ga još „podijeli pa vladaj uz rekurziju“ algoritam. *Merge sort* zapravo radi tako da prvo razdvoji ne sortirani niz na N podnizova, tako da svaki ima 1 element, a time se podrazumijeva da je niz od jednog elementa već sortiran. Nakon što je algoritam razdvojio sve elemente u nizove, algoritam ih uspoređuje i spaja u novi niz koji je također sortiran. To se nastavlja raditi sve dok se ne dobije niz u potpunosti kao cjelina koji je također sortiran. Prikaz toga se vidi na slici 3.2. Zbog razdvajanja i spajanja, *Merge sort* koristi dodatni niz za usporedbu i zbog toga se daje mala prednost *Quick sortu*. On ne mora praviti dodatni niz i zato zauzima manje memorije (izvedeno iz [13]).

Merge sort se također jako dobro paralelizira zbog svog načina funkcioniranja, strategije „podijeli pa vladaj“. Ako je paralelizacija algoritma dobro izvedena, vremenska složenost se može svesti na $O(log^3 n)$, ako je dovoljno procesora dostupno. Ovakav način sortiranja (podijeli pa vladaj) se odlično može iskoristiti u praksi ako je kombiniran s nekim selekcijskim sortiranjima, npr. „Sortiranje umetanjem“ (engl. *Insertion sort*). Kombiniranje s ovakvom vrstom sortiranja se radi kako bi se spajanje nizova svelo na vremensku složenost O(*1*), no nije nužno to koristiti, prema [14].

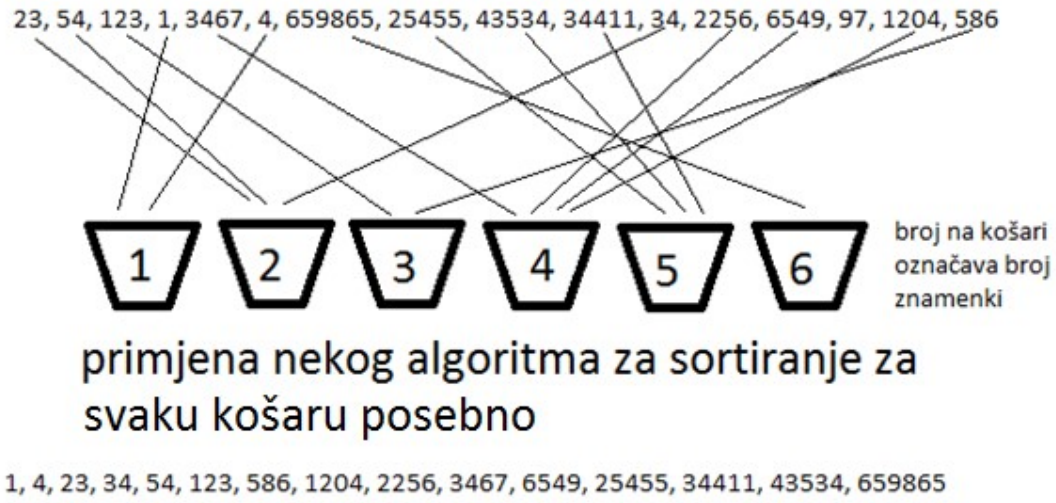


Sl.3.2. Primjer *Merge sort*-a

3.3. Radix sort

Radix sort je vrsta algoritama koji ne radi na principu usporedbe, međutim, kao i kod drugih algoritama, postoji više načina njegove implementacije. Ideja *Radix sorta* je da uspoređuje znamenku po znamenku, krenuvši od jedinica tj. najmanje važne znamenke (engl. *Least significant digit*) pa sve do najvažnije znamenke (engl. *Most significant digit*). Složenost ovakvog algoritma je $O(wn)$ gdje je n broj elemenata veličine w. Vremenska složenost je dosta dobra u odnosu na druge vrste algoritama, a ona je $O(n \log n)$ što ga svrstava u jedne od boljih i učinkovitijih algoritama (kako je navedeno u [15]). Kako kod paralelne izvedbe tako i serijska funkcionira na isti način, pomoću takozvanih košara (engl. *Basket/Bucket*) koje sadrže brojeve s jednakim brojem znamenki (iz [16]).

Što se tiče njegove paralelne izvedbe, ovaj algoritam je prvotno i napravljen za paralelno sortiranje zbog svoje funkcionalnosti. Kao što je vidljivo na slici 3.3. podatke dijeli po košarama gdje rasporedi brojeve po broju znamenaka (ovisno o znamenci postavi je u određenu košaru). Nakon što se niz raspodijeli po košarama, primjenjuje se neki od algoritama za sortiranje nad košarama i nakon toga se košare vraćaju u originalni niz sortirane po veličini, ali za određenu znamenku (prvo jednoznamenkasti, pa dvoznamenkasti itd.).



Sl. 3.3. Primjer *Radix sort*-a

4. USPOREDBA SERIJSKIH I PARALELNIH IZVEDBI ALGORITAMA ZA SORTIRANJE

Za usporedbu serijskih i paralelnih izvedbi koristili su se nizovi od *N* elemenata (gdje je *N* 100 tisuća, 500 tisuća, milijun, 5 milijuna, 10 milijuna, 50 milijuna, 100 milijuna i 200 milijuna) i mjerilo vrijeme koliko je potrebno da se takav niz sortira. Nakon svakog mjerenja izračunat je omjer ubrzanja u odnosu na serijsku izvedbu. Paralelne izvedbe imaju ograničenje na maksimalni broj niti koji je postavljen na 4 pomoću *omp_set_num_threads(4)* zbog okruženja u kojem se testiranje radilo.

Testiranje je provedeno na virtualnom stroju s Linux Ubuntu operacijskim sustavom izrađenom i pokrenutom unutar VMPlayer okruženja. Linux Ubuntu po svojim predefiniranim postavkama sadrži sve potrebne biblioteke i prevoditelje potrebne za rad s OpenMP-em. Testiranje nije rađeno na Windows OS-u zato što bi se sve biblioteke trebalo ručno dodavati u okruženje u kojem bi se radilo.

Specifikacije virtualnog stroja:

- 4 GB RAM memorije,
- dodijeljena mogućnost rada sa 4 procesorske jezgre,
- 50 GB HDD.

Bitne specifikacije računala na kojem je pokrenut virtualni stroj su:

- 8GB RAM 1600MHz memorije,
- intel Core i5-3470 3.2GHz (četverojezgreni procesor bez mogućnosti *hyper threading*-a),
- 1 TB HDD.

Programski kod se prevodio *g++* programskim prevoditeljem pokretanim iz komandne linije u Xterm terminalu kojeg sadrži svaki Ubuntu OS. Komandna linija za prevođenje je: *g++ -fopenmp ime.cpp –o ime.out*. *g++* se koristi za prevođenje C++ kodova zapisanih u nekom tekst editoru s ekstenzijom *.cpp* kao što je vidljivo na slici 4.1. Pomoću *–fopenmp* prevoditelja se upućuje da će raditi s OpenMP smjernicama i odredbama. U slučaju ako se izostavi *–fopenmp,* prevoditelj bi linije programskog koda koje sadrže *#pragma omp (smjernica)* samo preskočio i kod bi se izvršio serijski. Pomoću *-o* [*naziv_izvršne_datoteke.out*] prevoditelj zna kako će imenovati izvršnu datoteku. U nekim drugim okruženjima (npr. MinGW na Windows OS-u) bi još bilo potrebno dodati OpenMP vrijednosti okoline (koliki je maksimalni broj niti i sl.).

Pokretanje izvršnih datoteka je vrlo jednostavno na Ubuntu sustavima. U terminal se upiše komandna linija *./ime.out* gdje *ime.out* predstavlja naziv izvršne datoteke koji se navede prilikom prevođenja *.cpp* datoteke.

```
g++ -fopenmp ime.cpp –o ime.out
```

Sl. 4.1. Prikaz komandne linije za prevođenje programskog koda

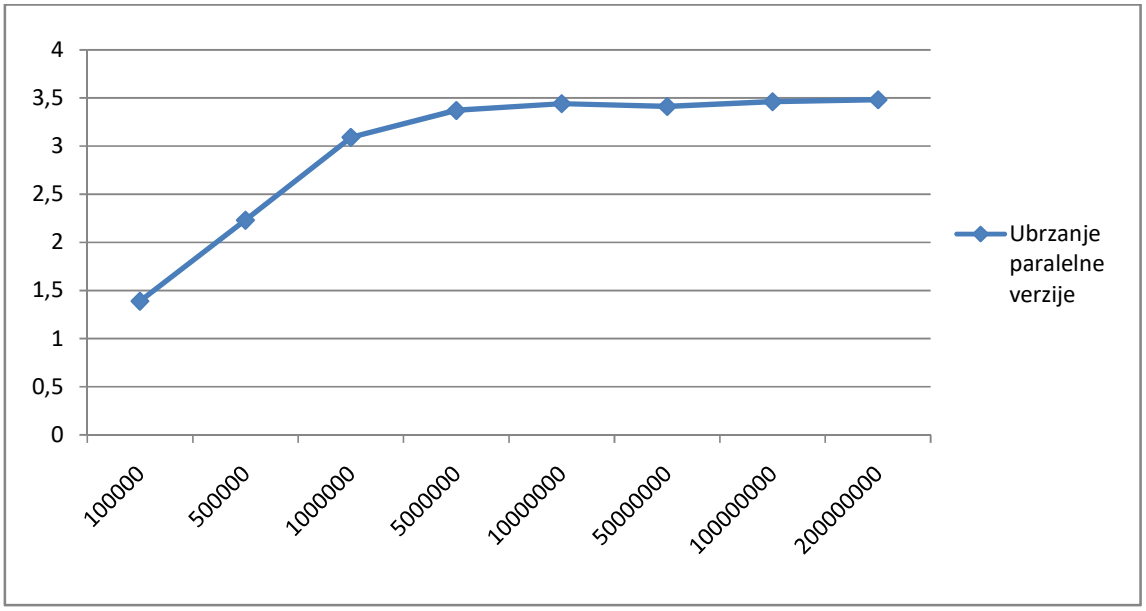
Nakon prevođenja serijskih i paralelnih verzija algoritama, izvršne datoteke su pokrenute i zapisana su vremena koja su bila potrebna za izvršavanje programskog koda. Vremena koja su se dobila su srednje vrijednosti rezultata koji su se dobivali pokretanjem istog programa više puta (5-10 puta pokrenuta je ista izvršna datoteka). Glavni razlog višestrukog ponavljanja je da se dobije što preciznije stvarno vrijeme potrebno za izvršenje zadanog programskog koda. Ubrzanje paralelne verzije u odnosu na serijsku se dobilo dijeljenjem srednje vrijednosti vremena serijske izvedbe sa srednjom vrijednosti vremena paralelne izvedbe.

4.1. Usporedba serijske i paralelne izvedbe Quick sort-a

U tablici 4.1 se nalaze vremena potrebna za izvođenje programskog koda za sve nizove kako u serijskoj tako i u paralelnoj izvedbi. Graf koji prikazuje ubrzanje paralelne verzije *Quick sort*-a u odnosu na broj elemenata niza koji se sortira vidljiv je na slici 4.2. Na grafu je vidljivo kako se ubrzanje paralelne verzije do određenog broja elemenata povećava linerarno i nakon toga ima približno jednako ubrzanje (nakon 5 milijuna elemenata ubrzanje je oko 3,45). Glavni razlog zašto se ubrzanje ne nastavlja povećavati linerano je zbog broja niti koje rade na obavljanju tog zadatka. Također kada se dobije stalno ubrzanje (oko 3,45), može se reći da je to vrijednost koliko je ubrzanje paralelne verzije u odnosu na serijsku izvedbu. U prvim mjerenjima (do milijun elemenata) ubrzanje je manje zato što je i serijska izvedba dovoljno brza, a paralelna još utroši vremena za kreiranje i uništavanje niti.

Tab 4.1. Usporedba vremena serijske i paralelne izvedbe *Quick sorta*.

Broj elemenata niza N	Vrijeme serijske izvedbe [s]	Vrijeme paralelne izvedbe [s]	Ubrzanje paralelne verzije u odnosu na serijsku
100.000	0,0126	0,0090	1,39
500.000	0,0665	0,0298	2,23
1.000.000	0,1388	0,0449	3,09
5.000.000	0,7645	0,2271	3,37
10.000.000	1,5841	0,4611	3,44
50.000.000	8,5926	2,5185	3,41
100.000.000	17,7747	5,1390	3,46
200.000.000	37,0595	10,6552	3,48



Sl. 4.2. Ubrzanje paralelne verzije *Quick sorta* u odnosu na njegovu serijsku izvedbu, gdje *x*-os predstavlja broj elemenata, a *y*-os predstavlja ubrzanje.

U paralelnoj izvedbi koda, za paraleliziranje se koriste paralelne regije u kojim se koriste smjernice *task* kao što se vidi na slici 4.4. Pomoću njih i odredbi *firstprivate*, omogućeno je da se funkcije (*quick_sort*) rekurzivno pozivaju i stvaraju niti bez ikakvih ograničenja ili mogućnosti preklapanja elemenata u nizu (npr. u slučaju da nit br. 1 zapiše nešto u varijablu „a“, a nit br. 2 zapiše nešto drugo u tu istu varijablu, dobiva se krivo rješenje).

U navedenom kodu na slici 4.3 je prikazano stvaranje paralelne regije koja poziva funkciju *quick_sort* sa samo jednom niti iako je prethodno u kodu označeno da se u paralelnim dijelovima koristi maksimalno 4. To je omogućeno pomoću smjernice *single*, a odredba *nowait* služi da ako neka nit u navedenoj regiji završi svoj posao prije druge da preuzme neki njen dio.

```
#pragma omp parallel
{
	#pragma omp single nowait
	quick_sort(0,	n-1,	a,
donjagranica);
}
```

Sl. 4.3. Dio koda koji predstavlja stvaranje paralelne regije.

Za prikazani dio koda na slici 4.4 koji se nalazi unutar funkcije *quick_sort* vidljivo je kako funkcija sama sebe poziva što znači da je rekurzivna, ali još bitniji dio je taj što je kod pozivanja funkcije navedeno da se poziva paralelno pomoću smjernice *task* (objašnjena u poglavlju 2.4.2). Također je nužno koristiti odredbu *firstprivate* s navedenim varijablama kako nebi došlo do prepisivanja ili korištenja istih. Varijabla „donjagranica“ označava samo do koje razine će se pozivati rekurzivno *quick_sort*, a kada početi *seq_quick_sort*.

```
#pragma omp task firstprivate(a, donjagranica, r, q)
	quick_sort(p, q - 1, a, donjagranica);
#pragma omp task firstprivate(a, donjagranica, r, q)
	quick_sort(q + 1, r, a, donjagranica);
```

Sl. 4.4. Dio koda koji predstavlja korištenje smjernica *task*.

Izvorni C++ programski kod za serijsku izvedbu *Quick sort*-a nalazi se u prilogu 1, a za paralelnu izvedbu istog u prilogu 2.

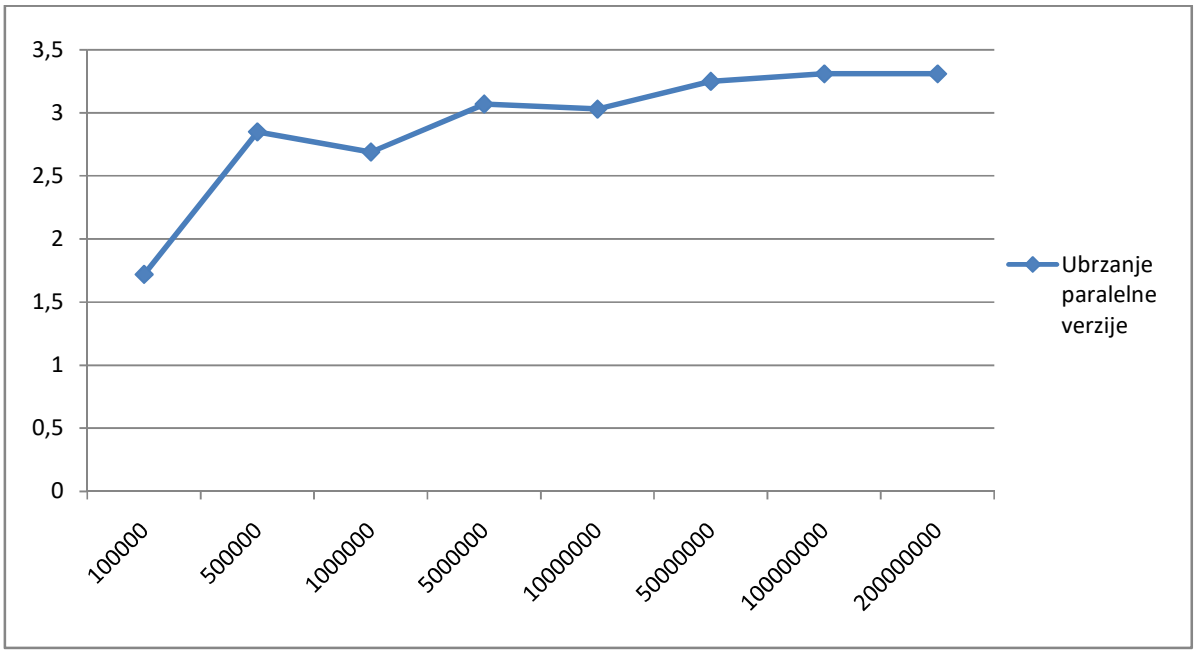
4.2. Usporedba serijske i paralelne izvedbe Merge sorta-a

U tablici 4.2 se nalaze vremena izvršavanja za sve nizove kako u serijskoj tako i u paralelnoj izvedbi. Graf koji prikazuje ubrzanje paralelne verzije *Merge sort*-a u odnosu na broj elemenata vidljiv je na slici 4.5. Na grafu je vidljivo kako se ubrzanje paralelne verzije algoritma povećava do određenog broja elemenata i nakon toga ima približno jednako ubrzanje (nakon 5 milijuna elemenata ubrzanje se ne povećava puno, do 3,3 puta). Glavni razlog zašto se ubrzanje ne nastavlja povećavati linerano je zbog broja niti koje rade na obavljanju tog zadatka. I da je više niti bilo raspoloživo vrijeme bi vrlo vjerojatno bilo identično zbog načina paralelizacije (pomoću

smjernica *section*). Na slici 4.5 je također vidljivo kako je ubrzanje na milijun elemenata manje nego na pola milijuna. Razlog toga je srednje vrijeme koje se dobilo prilikom pokretanja paralelne verzije. U nekim slučajevima je vrijeme izvedbe bilo bolje, ali zbog bolje preciznosti se koristila srednja vrijednost dobivena iz više testnih slučajeva

Tab.4.2. Usporedba vremena serijske i paralelne izvedbe *Merge sorta*.

Broj elemenata niza N	**Vrijeme serijske izvedbe [s]**	**Vrijeme paralelne izvedbe [s]**	**Ubrzanje paralelne verzije u odnosu na serijsku**
100.000	0,0185	0,0107	1,72
500.000	0,0999	0,0350	2,85
1.000.000	0,2065	0,0766	2,69
5.000.000	1,0994	0,3578	3,07
10.000.000	2,2887	0,7551	3,03
50.000.000	12,4088	3,8204	3,25
100.000.000	25,9535	7,8436	3,31
200.000.000	53,4471	16,1592	3,31



Sl. 4.5. Prikaz ubrzanja paralelne verzije *Merge sorta* u odnosu na njegovu serijsku izvedbu, gdje *x*-os predstavlja broj elemenata, a *y*-os predstavlja ubrzanje.

U paralelnoj izvedbi koda, za paraleliziranje se koriste paralelne regije u kojima se koriste smjernice *sections*, koje unutar sebe moraju imati pozivanje smjernica *section*. Pomoću njih se omogućuje paraleliziranje koda, ali bitna stavka kod *section*-a je da blok naredbi unutar navedene smjernice obavlja samo jedna nit. Iz tog razloga rad sa *section*-ima je dosta ograničen, tj. potrebno je unaprijed znati koliko niti imamo na raspolaganju ako se želi što bolje iskoristiti paralelizam.

U navedenom kodu na slici 4.6 se poziva paralelna regija koja koristi smjernicu *sections* koja unutar sebe ima četiri smjernice *section*, što znači da će raditi sa četiri niti. Ovo je bilo moguće napraviti zato što je unaprijed bilo poznato koliki je maksimalni broj niti moguće koristiti. Da je primjera radi maksimalni broj niti bio tri, kod bi puno duže trajao zato što bi prva nit koja završi tek tada počela raditi na četvrtom sectionu, a ostale tri bi čekale na barijeri, bez posla.

```
#pragma omp parallel sections
            {
                  #pragma omp section
                  {
                  seq_mergesort(a,b,low, lijevipivot);
                  }
                  #pragma omp section
                  {
                  seq_mergesort(a,b,lijevipivot+1, pivot);
                  }
                  #pragma omp section
                  {
                  seq_mergesort(a,b,pivot+1, gornjipivot);
                  }
                  #pragma omp section
                  {
                  seq_mergesort(a,b,gornjipivot+1, high);

                  }
            }
```

Sl. 4.6. Dio koda koji predstavlja stvaranje paralelne regije pomoću smjernice *sections*.

Navedeni kod prikazan na slici 4.7 koristi samo dvije niti iako ima još dvije slobodne koje ništa ne rade. Na ovom mjestu se može još bolje realizirati paralelizacija pomoću nekih drugih smjernica. Zadnji poziv funkcije merge radi samo jedna nit zbog prirode same funkcije (rad s cijelim nizom i ako bi radilo više niti došlo bi do prepisivanja).

```
#pragma omp parallel sections
            {
                  #pragma omp section
                  merge(a,b,low,lijevipivot,pivot);
                  #pragma omp section
                  merge(a,b,pivot+1,gornjipivot,high);
            }
            merge(a, b, low, pivot, high);
```

Sl. 4.7. Dio koda za pozivanje funkcije *merge* gdje se sortirani nizovi spajaju.

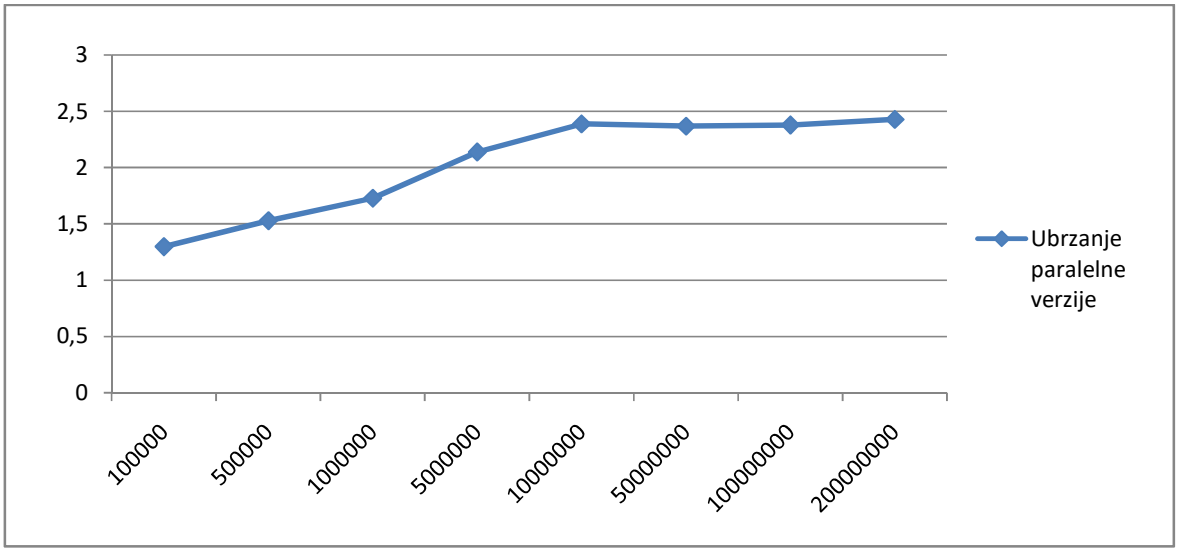
Izvorni C++ programski kod za serijsku izvedbu *Merge sort*-a nalazi se u prilogu 3, a za paralelnu izvedbu istog u prilogu 4.

4.3. Usporedba serijske i paralelne izvedbe Radix sort-a

U tablici 4.3 se nalaze vremena za sve nizove kako u serijskoj tako i u paralelnoj izvedbi. Graf koji prikazuje ubrzanje paralelne verzije u odnosu na broj elemenata vidljiv je na slici 4.8. Na grafu je vidljivo kako se ubrzanje paralelne verzije do određenog broja elemenata povećava linerarno i nakon toga ima približno jednako ubrzanje (nakon 10 milijuna elemenata ubrzanje je oko 2,4). Glavni razlog zašto se ubrzanje ne nastavlja povećavati linerano je zbog broja niti koje rade na obavljanju tog zadatka i zbog načina paralelizacije samog koda. Da je bilo moguće raditi sa većim brojem niti ubrzanje bi bilo bolje, ali bi se programski kod morao dodatno prilagoditi tome, jer je paralelizacija izvršena pomoću *sections* smjernice.

Tab.4.3. Usporedba vremena serijske i paralelne izvedbe *Radix sort*-a.

Broj elemenata niza N	Vrijeme serijske izvedbe [s]	Vrijeme paralelne izvedbe [s]	Ubrzanje paralelne verzije u odnosu na serijsku
100.000	0,0124	0,0095	1,3
500.000	0,0653	0,0428	1,53
1.000.000	0,1337	0,0774	1,73
5.000.000	0,7090	0,3309	2,14
10.000.000	1,4743	0,6166	2,39
50.000.000	7,6699	3,1352	2,37
100.000.000	15,8451	6,6608	2,38
200.000.000	32,7275	13,4672	2,43



Sl. 4.8. Prikaz ubrzanja paralelne verzije *Radix sort*-a u odnosu na njegovu serijsku izvedbu, gdje *x*-os predstavlja broj elemenata, a *y*-os predstavlja ubrzanje.

U paralelnoj izvedbi koda, za paraleliziranje se koriste paralelne regije u kojima se koriste smjernice *sections*, koje unutar sebe moraju imati pozivanje smjernica *section*. Pomoću njih se omogućuje paraleliziranje koda, ali bitna stavka kod *section*-a je da blok naredbi unutar navedene smjernice obavlja samo jedna nit.

Kako *Radix sort* funkcionira s košarama, a nasumično generirani brojevi su sezali do 1.073.676.289 (najveći broj), košare su bile ispunjene s tim brojevima i svaka košara zasebno je bila sortirana pomoću *Quick sort*-a. Zbog tehničkih mogućnosti računala na kojem se testiranje radilo, bilo je potrebno prilagoditi pozivanje funkcija koje rade sortiranje na poseban način kao što je vidljivo na slici 4.9. Razlog zašto se funkcije *Quick sort*-a pozivaju baš na ovakav način je zbog određenog broja elemenata koji se pojavljuje s kojim brojem znamenaka. Baš ovakav redoslijed pozivanja je određen zbog izvršenih testova prije paralelizacije, odnosno utvrđen je približan (ne može se znati točan broj pošto su brojevi nasumično generirani) broj elemenata koji se pojavljuje s kojim brojem znamenaka.

```
#pragma omp parallel sections
{
	#pragma omp section //1. thread
	{
	quicksort(b, 0, j - 1, 0);
	quicksort(b, br_elem * 2 / 5, t - 1, 0);
	quicksort(b, br_elem * 5 / 5, se - 1, 0);
	}
	#pragma omp section //2. thread

	{
	quicksort(b, br_elem * 4 / 5, p - 1, 0);
	quicksort(b, br_elem * 6 / 5, sd - 1, 0);
	quicksort(b, br_elem * 7 / 5, o - 1, 0);
	}
	#pragma omp section //3. thread
	{
	quicksort(b, br_elem / 5, d - 1, 0);
	quicksort(b, br_elem * 8 / 5, dev - 1, 0);
	}
	#pragma omp section //4. thread
	{
	quicksort(b, br_elem * 3 / 5, c - 1, 0);
	quicksort(b, br_elem * 9 / 5, des - 1, 0);
	}
}
```

Sl. 4.9. Dio koda koji predstavlja stvaranje paralelne regije pomoću smjernice *sections*.

Nakon što se sve košare sortiraju poziva se funkcija spajanja kao što je vidljivo na slici 4.10. Kod na slici 4.10 se također može paralelizirati, međutim nije isplativ. Zato odredba *parallel for* stoji kao komentar u kodu, jer stvaranje novih niti za ovakav rad uzima više procesorskog vremena i duže bi trajalo nego serijska izvedba. Da je bilo moguće raditi s većim brojem elemenata (npr. 1 milijarda elemenata), možda bi se isplatilo koristiti paralelizam i na tom dijelu, ali zbog tehničkih mogućnosti virtualnog računala to nije bilo moguće (program se prekine zbog nedostatka memorije, tzv. „stack overflow“). Zato je bolje ovakav tip zadatka napraviti serijski.

```
void spajanje(  int *a, int *b,
       intj, intd, intt, intc, intp, intse, intsd, into, intdev, intdes)
{
//#pragma omp parallel for
       for (int i = 0; i <br_elem/5; i++)
       {
              if(b[i]!=-1)
              a[i] = b[i];
              if (b[i + br_elem /5]!=-1)
              a[i+j] = b[i + br_elem / 5];
              if (b[i + br_elem * 2 / 5]!=-1)
              a[i+j+d] = b[i + br_elem * 2 / 5];
              if (b[i + br_elem * 3 / 5]!=-1)
              a[i+j+d+t] = b[i + br_elem * 3 / 5];
              if (b[i + br_elem * 4 / 5]!=-1)
              a[i+j+d+t+c] = b[i + br_elem * 4 / 5];
              if (b[i + br_elem * 5 / 5]!=-1)
              a[i + j + d + t + c+p] = b[i + br_elem * 5 / 5];
              if (b[i + br_elem * 6 / 5]!=-1)
              a[i + j + d + t + c + p+se] = b[i + br_elem * 6 / 5];
              if (b[i + br_elem * 7 / 5]!=-1)
              a[i + j + d + t + c + p + se+ sd] = b[i + br_elem * 7 / 5];
              if (b[i + br_elem * 8 / 5]!=-1)
              a[i + j + d + t + c + p + se + sd+o] = b[i + br_elem * 8 / 5];
              if (b[i + br_elem * 9 / 5]!=-1 )
              a[i + j + d + t + c + p + se + sd + o+dev] = b[i + br_elem * 9 / 5];

       }
}
```

Sl. 4.10. Funkcija za spajanje košara u glavni niz po redu.

Izvorni C++ programski kod za serijsku izvedbu *Radix sort*-a nalazi se u prilogu 5, a za paralelnu izvedbu istog u prilogu 6.

5. ZAKLJUČAK

Algoritmi za sortiranje se koriste za sortiranje podataka za različite potrebe. Postoje više vrsta algoritama za sortiranje, neki su brži, neki sporiji, ovisno o načinu sortiranja. Obrađeni algoritmi *Quick sort, Merge sort* i *Radix sort* su brzi algoritmi koji se mogu paralelizirati na jednostavan način zbog načina njihovog funkcioniranja, a opet prikazuju dobro ubrzanje u odnosu na serijsku izvedbu.

Iz grafova i tablica vidljivo je da su paralelne izvedbe puno brže, u nekim situacijama i do 3,5 puta u odnosu na serijsku izvedbu. Međutim, bitno je napomenuti kako se ponekad paralelizacija i ne isplati. Iako nije navedeno u tablicama, algoritmi koji su testirani, s manjim brojem elemenata pokazivali su veće vrijeme izvedbe u paralelnoj izvedbi nego u serijskoj. Iako je to sve bilo brzo (i serijska i paralelna verzija), paralelnoj je trebalo nekoliko milisekundi više nego serijskoj. Razlog tome je što je procesoru potrebno neko vrijeme da kreira niti i uništi ih. Kad se gleda uspoređivanje ubrzanja samih algoritama, *Radix sort*, iako najbrži u serijskoj izvedbi, ima najmanje ubrzanje u odnosu na svoju serijsku verziju (najbolji slučaj 2,43 puta brže). Glavni razlog toga je barijera na koju nailazi kada završava s paralelnom regijom. Na neki drugi način bi se to ubrzanje još moglo povećati, primjerice pomoću OpenMP smjernica *task*. S druge strane, *Merge sort*, iako isto koristi OpenMP smjernicu *sections* kao i *Radix sort*, zbog prirode svog algoritma puno je bolje optimiziraniji za korištenje i svaka nit obavlja otprilike istu količinu posla, tako da nema puno čekanja na barijeri. Vremenski gledano najsporiji, ali ubrzanje pomoću paralelizacije dovodi i do 3,31 puta brže u odnosu na svoju serijsku izvedbu. Paralelna izvedba *Quick sort*-a je izrađena pomoću smjernica *task* koje nemaju spomenutu barijeru, niti nikakvih čekanja i zbog toga paralelna izvedba *Quick sort*-a u odnosu na svoju serijsku verziju ima najbolje ubrzanje koje ide čak do 3,48 puta.

Iako je puno jednostavnije koristiti smjernice *sections* za paralelizaciju, zapravo se ograničava samo paraleliziranje i radilo se s četiri ili deset niti, vrijeme će biti isto. Dok kod paralelizacije uz pomoć smjernica *task*, poboljšava se fleksibilnost samog algoritma i ako bi se pokretao na nekom drugom računalu koji ima mogućnost za rad s više niti, imao bi puno bolje vrijeme.

OpenMP se sve više unaprjeđuje. Za obradu podataka se ne moraju koristiti samo procesori, tu su i drugi resursi računala poput grafičkih kartica (koriste OpenMP ubrzivače) koje se mogu koristiti za obradu podataka od verzije OpenMP 4.0. Današnje grafičke kartice sadrže veliki broj jezgri koje su sposobne za obradu podataka isto kao i procesorske.

LITERATURA

[1] B. Klajić: „Novi rječnik stranih riječi“, Školska knjiga, Zagreb, 2012.

[2] K. Šuljug: Usporedba algoritama za sortiranje podataka, ETF Osijek, 2014

[3] D. Jakobović: Paralelno programiranje – predavanja, FER Zagreb, 2015 [online]. Dostupno na: https://www.fer.unizg.hr/_download/repository/Paralelno_programiranje_predavanja%5B8%5D.pdf [24.lipnja 2016]

[4] G. Martinović: Paralelno programiranje nakupine i spleta računala, Raspodijeljeni računalni sustavi – predavanja, ETF Osijek, 2015. [24.lipnja 2016]

[5] G. Martinović: Paralelno programiranje u C-u s OpenMP, Raspodijeljeni računalni sustavi – predavanja, ETF Osijek, 2015. [24.lipnja 2016]

[6] Z. Krpić: Distribuirani računalni sustavi, Raspodijeljeni računalni sustavi – Laboratorijska vježba 1, ETF Osijek, 2015. [24.lipnja 2016]

[7] Z. Krpić: OpenMP, dio II, Raspodijeljeni računalni sustavi – Laboratorijska vježba 7, ETF Osijek, 2015. [24.lipnja 2016]

[8] „OpenMP in Visual C++“, Microsoft, 2015 [online]. Dostupno na: https://msdn.microsoft.com/en-us/library/2kwb957d.aspx [26. rujna 2016]

[9] „OpenMP Application Program Interface“, OpenMP Architecture Review Board, 2008 [online]. Dostupno na: http://www.openmp.org/mp-documents/spec30.pdf [26. rujna 2016]

[10] „Odd-even sort“, Wikipedia, 2007 [online]. Dostupno na: https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort [24.lipnja 2016]

[11] „Quick sort“, Geeksforgeeks, 2013 [online]. Dostupno na: http://quiz.geeksforgeeks.org/quick-sort/ [24.lipnja 2016]

[12] „Quick sort“, Algolist, 2008 [online]. Dostupno na: http://www.algolist.net/Algorithms/Sorting/Quicksort [24.lipnja 2016]

[13] „Merge sort“, Geeksforgeeks, 2012 [online]. Dostupno na: http://quiz.geeksforgeeks.org/merge-sort/ [24.lipnja 2016]

[14] „Parallel merge“, Wikipedia, 2009 [online]. Dostupno na: https://en.wikipedia.org/wiki/Merge_algorithm#Parallel_merge [24.lipnja 2016]

[15] „Radix sort“, Wikipedia , 2001 [online]. Dostupno na: https://en.wikipedia.org/wiki/Radix_sort [24.lipnja 2016]

[16] „Radix sort“, Geeksforgeeks, 2013 [online].
Dostupno na: http://www.geeksforgeeks.org/radix-sort [24.lipnja 2016]

SAŽETAK

Postoji više načina za paraleliziranje kodova ili algoritama. Neki od njih su MPI (engl. *Message Passing Interface*) i OpenMP (engl. *Open Multi-Processing*). U radu se koristio OpenMP zbog jednostavnosti svoje upotrebe. OpenMP je zapravo API (engl. *Application Programming Interface*) koji podržava multiprocesorsko programiranje s dijeljenom memorijom.

U radu je prikazano koliko se paralelizam isplati na velikom broju podataka te je testirano s različitim metodama paralelizacije da bi utvrdili koja se više isplati, kada, i zašto. Utvrđeno je da su *Radix sort* i *Merge sort*, koji su paralelizirani s OpenMP smjernicom *sections* imali i do 3,31 puta brže izvođenje koda nego njihova serijska verzija.

S druge strane, *Quick sort* je paraleliziran pomoću smjernice *task* čije je ubrzanje paralelne verzije u odnosu na serijsku bilo čak 3,48 puta brže. Glavna karakteristika kod *task*-ova je što će se broj niti podijeliti po taskovima i ako neka nit završi prije, pomoći će drugoj koja nije završila.

Ključne riječi: MPI, OpenMP, smjernica, ubrzanje, paralelizam.

ABSTRACT

PARALLELIZATION OF SORTING ALGORITHMS

There are several ways to paralelize a program code or an algorithm. Some of them are MPI (Message passing interface) and OpenMP (Open Multi-Processing). In this thesis OpenMP is used due to the simplicity of its use. OpenMP is actually an API (Application Programming Interface) that supports multiprocessing programming with shared memory.

The thesis describes how much parallelism is worth on a large number of data and has been tested with different methods of parallelization in order to give the best (fastest) solution for each of the test cases. The conclusion is that Radix sort and Merge sort, which were parallelized with the OpenMP directives *sections*, had up to 3.31 times faster execution of the program code than their standard version.

On the other hand, Quick sort is parallelized using the directive called *task*, bringing even faster speedups of up to 3.48. The main characteristics of the *tasks* is that the number of threads is shared among *tasks* and if one thread finishes before the other, it will help another which has not finished.

Keywords: MPI, OpenMP, directive, speedup, parallelism.

ŽIVOTOPIS

Krešimir Šuljug je rođen 25. svibnja 1992. godine u Osijeku. Osnovnu školu je pohađao u Čepinu, a srednju školu, Prirodoslovno-matematičku gimnaziju, u Osijeku. U trećem razredu gimnazije sudjelovao je na Županijskom natjecanju iz informatike i osvojio 3. mjesto. Kao student Procesnogračunarstva na FERIT-u naučio se koristiti raznim programskim alatima koji su mu pomogli u izradi ovog rada. Kroz ovaj diplomski rad, unaprijedio je svoje znanje u C++-u koje će mu koristiti u njegovom daljnjem napretku u IT svijetu. Zbog velike zainteresiranosti za tehnologijama budućnosti odlučio se za ovu temu rada, a kako je za završni rad imao temu „Algoritmi za sortiranje podataka“ ovo je bilo idealno rješenje kako se s novim načinom programiranja mogu ubrzati izvršenja takvih zadataka.

_____

PRILOZI

Izvorni c++ kodovi

Prilog 1 Quick sort (serijska izvedba)

```
#include<stdio.h>
#include<time.h>
#include<iostream>
#include<omp.h>
#include<stdlib.h>
#include<stdint.h>
usingnamespace std;
#define br_elem 200000000
#define granica 100
int partition(int p, int r, int *a)
{
	int x = a[p];
	int k = p;
	int l = r + 1;
	int t;
	while (1)
	{
		do
			k++;
		while ((a[k] <= x) && (k < r));
		do
			l--;
		while (a[l] > x);
		while (k < l)
		{
			t = a[k];
			a[k] = a[l];
			a[l] = t;
			do
				k++;
			while (a[k] <= x);
			do
				l--;
			while (a[l] > x);
		}
		t = a[p];
		a[p] = a[l];
		a[l] = t;
		return l;
	  }
}
void seq_quick_sort(int p, int r, int * a)
{
	if (p < r)
	{
		int q = partition(p, r, a);
		seq_quick_sort(p, q - 1, a);
		seq_quick_sort(q + 1, r, a);
	}
}
void quick_sort(int p, int r, int * a, int donjagranica)
{
	if (p < r)
	{
```

```
			if ((r - p) < donjagranica)
			{
				seq_quick_sort(p, r, a);
			}
			else
			{
				int q = partition(p, r, a);
				quick_sort(p, q - 1, a, donjagranica);
				quick_sort(q + 1, r, a, donjagranica);
			}
		}
}
void provjera( int *a)
{
		int br=0;
		for (int i = 0; i < br_elem-1; i++)
		{
			if (a[i] > a[i + 1])
				br++;
		}
		if(br>0)
		cout <<"krivo sortiran "<<br <<" puta"<<endl;
		else
		cout <<"dobro sortiran"<< endl;//dobro sortiran
}
int main()
{
		double t1,t2;
		int *a = newint[br_elem];
		t1 = omp_get_wtime();
		srand((int)time(NULL));
		for (int i = 0; i < br_elem; i++)
		{
			a[i] = rand();
		}
		t2 = omp_get_wtime();
		printf("generiranje niza: %g s\n", (t2 - t1));
		printf("niz se sortira \n \n ");
		t1 = omp_get_wtime();
		seq_quick_sort(0, br_elem-1, a);
		t2 = omp_get_wtime();
		printf("niz se sortirao za: %g s\n", (t2 - t1) );
		provjera(a);
		free(a);
		return 0;
}
```

Prilog 2 – Quick sort (paralelna izvedba)

```
#include<stdio.h>
#include<time.h>
#include<iostream>
#include<omp.h>
#include<stdlib.h>
#include<stdint.h>
usingnamespace std;
#define br_elem 500000000
#define granica 100
int partition(int p, int r, int *a)
{
int x = a[p];
```

```
int k = p;
int l = r + 1;
int t;
while (1)
  {
do
      k++;
while ((a[k] <= x) && (k < r));
do
      l--;
while (a[l] > x);
while (k < l)
    {
      t = a[k];
      a[k] = a[l];
      a[l] = t;
do
        k++;
while (a[k] <= x);
do
        l--;
while (a[l] > x);
    }
    t = a[p];
    a[p] = a[l];
    a[l] = t;
return l;
  }
}
void seq_quick_sort(int p, int r, int * a)
{
if (p < r)
  {
int q = partition(p, r, a);
    seq_quick_sort(p, q - 1, a);
    seq_quick_sort(q + 1, r, a);
  }
}
void quick_sort(int p, int r, int * a, int donjagranica)
{
if (p < r)
  {
if ((r - p) < donjagranica)
    {
      seq_quick_sort(p, r, a);
       }
else
     {
int q = partition(p, r, a);
#pragma omp task firstprivate(a, donjagranica, r, q)
      quick_sort(p, q - 1, a, donjagranica);
#pragma omp task firstprivate(a, donjagranica, r, q)
      quick_sort(q + 1, r, a, donjagranica);
     }
  }
}
void par_quick_sort(int n, int *a, int donjagranica)
{
       #pragma omp parallel
       {
       #pragma omp single nowait
    quick_sort(0, n-1, a, donjagranica);
```

```
  }
}

void provjera( int *a)
{
	int br=0;
	for (int i = 0; i < br_elem-1; i++)
	{
		if (a[i] > a[i + 1])
			br++;
	}
	if(br>0)
	cout <<"krivo sortiran "<<br <<" puta"<<endl;
	else
	cout <<"dobro sortiran"<< endl;//dobro sortiran
}
int main()
{
double t1, t2;
int *a = newint[br_elem];
  t1 = omp_get_wtime();
  srand((int)time(NULL));
for (int i = 0; i < br_elem; i++)
  {
    a[i] = rand();
  }
  t2 = omp_get_wtime();
  printf("generiranje niza: %g s\n", (t2 - t1));
  printf("niz se sortira \n \n ");
  t1 = omp_get_wtime();
  omp_set_num_threads(4);
  par_quick_sort(br_elem, a, granica);
  t2 = omp_get_wtime();
  printf("niz se sortirao za: %g s \n\n", (t2 - t1));
  provjera(a);
  free(a);
return 0;
}
```

Prilog 3 – Merge sort (serijska izvedba)

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<iostream>
#include<omp.h>
usingnamespace std;
#define br_elem 200000000
void merge(int *a, int *b, int low, int pivot, int high)
{
	int h, i, j, k;
	h = low;
	i = low;
	j = pivot + 1;
	while ((h <= pivot) && (j <= high))
	{
		if (a[h] <= a[j])
		{
			b[i] = a[h];
			h++;
```

```
			}
			else
			{
				b[i] = a[j];
				j++;
			}
			i++;
		}
		if (h>pivot)
		{
			for (k = j; k <= high; k++)
			{
				b[i] = a[k];
				i++;
			}
		}
		else
		{
			for (k = h; k <= pivot; k++)
			{
				b[i] = a[k];
				i++;
			}
		}
		for (k = low; k <= high; k++)
			a[k] = b[k];
}
void mergesort(int *a, int *b, int low, int high)
{
		int pivot;
		if (low<high)
		{
			pivot = (low + high) / 2;
			mergesort(a, b, low, pivot);
			mergesort(a, b, pivot + 1, high);
			merge(a, b, low, pivot, high);
		}
}
void provjera( int *a)
{
		for (int i = 0; i < br_elem-1; i++)
		{
			if (a[i] > a[i + 1])
				cout<<"nije dobro sortiran "<< a[i-1] <<" "<< a[i] <<" "<<a[i+1]<<"
"<<a[i+2]<<endl;//nije dobro sortiran

		}
		cout <<"dobro sortiran"<< endl;//dobro sortiran
}
int main()
{
		double t1, t2;
		int *a = newint[br_elem];
		t1 = omp_get_wtime();
		srand((int)time(NULL));
		for (int i = 0; i<br_elem; i++)
		{
			a[i] = rand();
			if(a[i]<0)
			a[i]=a[i]*-1;
		}
		int *b = newint[br_elem];
```

```
	t2 = omp_get_wtime();
	cout <<"vrijeme potrebno za kreiranje niza"<< t2 - t1 << endl;
	t1 = omp_get_wtime();
	mergesort(a, b, 0, br_elem - 1);
	t2 = omp_get_wtime();
	cout <<"vrijeme sortiranja je: "<< (t2 - t1) <<" s"<< endl;
	provjera(a);

	return 0;
}
```

Prilog 4 – Merge sort (paralelna izvedba)

```
#include<stdio.h>
#include<time.h>
#include<iostream>
#include<omp.h>
#include<stdlib.h>
#include<stdint.h>
usingnamespace std;
#define br_elem 200000000

void provjera( int *a)
{
	int br=0;
	for (int i = 0; i < br_elem-1; i++)
	{
		if (a[i] > a[i + 1])
			//cout<<"nije dobro sortiran " << a[i-1] << " " << a[i] << " "
<<a[i+1]<<" "<<a[i+2]<<endl;//nije dobro sortiran
	br++;
	}
	if(br>0)
	cout <<"nije dobro sortiran... krivih: "<<br << endl;
	else
	cout <<"dobro sortiran"<< endl;//dobro sortiran
}
void merge(int *a, int *b, int low, int pivot, int high)
{
	int h, i, j, k;
	h = low;
	i = low;
	j = pivot+1;
	while ((h <= pivot) && (j <= high))
	{
		if (a[h] <= a[j])
		{
			b[i] = a[h];
			h++;
		}
		else
		{
			b[i] = a[j];
			j++;
		}
		i++;
	}
	if (h>pivot)
	{
		for (k = j; k <= high; k++)
		{
			b[i] = a[k];
			i++;
```

```
		}
	}
	else
	{
		for (k = h; k <= pivot; k++)
		{
			b[i] = a[k];
			i++;
		}
	}
	for (k = low; k <= high; k++)
	{
		a[k] = b[k];

	}
	//cout<<"  nit broj: "<<omp_get_thread_num()<<endl;
}
void seq_mergesort(int *a, int *b, int low, int high)
{
	int pivot;
	if (low<high)
	{
		pivot = (high+low)/2;

		seq_mergesort(a, b, low, pivot);
		seq_mergesort(a, b, pivot + 1, high);
		merge(a, b, low, pivot, high);
		}
}

void mergesort(int *a, int *b, int low, int high)
{
	int pivot;
	pivot = (low + high) / 2;
	int lijevipivot= pivot / 2;
	int gornjipivot= lijevipivot+pivot;

		#pragma omp parallel sections
		{
			#pragma omp section
			{
			//	printf("low,	lijevipivot-1	ovo	radi	nit:	%d
\n",omp_get_thread_num());
				seq_mergesort(a,b,low, lijevipivot);
			}
			#pragma omp section
			{
			//	printf("lijevipivot,	pivot	ovo	radi	nit:	%d
\n",omp_get_thread_num());
				seq_mergesort(a,b,lijevipivot+1, pivot);

			}

			#pragma omp section
			{
			//	printf("pivot,	gornjipivot-1	ovo	radi	nit:	%d
\n",omp_get_thread_num());
				seq_mergesort(a,b,pivot+1, gornjipivot);
			}
			#pragma omp section
			{
```

```
                        //      printf("gornjipivot,      high      ovo      radi      nit:      %d
\n",omp_get_thread_num());
                                seq_mergesort(a,b,gornjipivot+1, high);
                        }
                }
                #pragma omp parallel sections
                {
                        #pragma omp section
                        merge(a,b,low,lijevipivot,pivot);
                        #pragma omp section
                        merge(a,b,pivot+1,gornjipivot,high);
                }
                merge(a, b, low, pivot, high);

}
int main()
{
        double t1, t2;
        int i, n;
        t1 = omp_get_wtime();
        int *a = newint[br_elem];
        int *b = newint[br_elem];
        srand((int)time(NULL));
        for (int i = 0; i<br_elem; i++)
        {
                a[i] = rand();
                if(a[i]<0)
                a[i]=a[i]*-1;
        }
        t2 = omp_get_wtime();
        printf("vrijeme stvaranja niza je: %g s\n",(t2-t1));
        printf("niz se sortira \n \n ");
        t1 = omp_get_wtime();
        mergesort(a, b, 0, br_elem-1);
        t2 = omp_get_wtime();
        printf("vrijeme sortiranja je: %g s\n",(t2-t1));
        provjera(a);
        free(a);
        free(b);
        return 0;
}
```

Prilog 5 – Radix sort (serijska izvedba)

```
#include<stdio.h>
#include<time.h>
#include<iostream>
#include<omp.h>
#include<stdlib.h>
#include<stdint.h>
usingnamespace std;
#definebr_elem 200000000

void provjera(  inta[])
{
        for (int i = 0; i <br_elem-1; i++)
        {
                if (a[i] >a[i + 1])
                        cout<<"nije  dobro  sortiran  "<<a[i-1]  <<"  "<<a[i]  <<"  "<<a[i+1]<<"
"<<a[i+2]<<endl;//nije dobro sortiran

        }
```

```
	cout <<"dobro sortiran"<< endl;//dobro sortiran
}
void zamjena(inti, intj,  inta[]) {
	int temp = a[i];
	a[i] = a[j];
	a[j] = temp;
}
void quicksort(  inta[],intindex , intdesni, intlijevi)
{
	int temp;
	int mid = (lijevi + desni) / 2;
	int i = lijevi;
	int j = desni;
	int pivot = a[mid+index];
	while (lijevi<j || i<desni)
	{
		while (a[i+index]<pivot)
			i++;
		while (a[j+index]>pivot)
			j--;

		if (i <= j) {
			temp = a[i+index];
			a[i+index] = a[j+index];
			a[j+index] = temp;
			i++;
			j--;
		}
		else {
			if (lijevi<j)
				quicksort(a,index, j , lijevi);
			if (i<desni)
				quicksort(a, index, desni, i);
			return;
		}
	}
}
void spajanje(  int *a, int *b,
				intj,  intd,  intt,  intc,  intp,  intse,  intsd,  into,  intdev,
intdes)
{
	int k = 0;
	int max = j;
	if (max <d) max = d;
	if (max <t) max = t;
	if (max <c) max = c;
	if (max <p) max = p;
	if (max <se) max = se;
	if (max <sd) max = sd;
	if (max <o) max = o;
	if (max <dev) max = dev;
	if (max <des) max = des;

	for (int i = 0; i <br_elem/5; i++)
	{
		if(b[i]!=-1)
		a[i] = b[i];
		if (b[i + br_elem /5]!=-1)
		a[i+j] = b[i + br_elem / 5];
		if (b[i + br_elem * 2 / 5]!=-1)
		a[i+j+d] = b[i + br_elem * 2 / 5];
		if (b[i + br_elem * 3 / 5]!=-1)
```

```
            a[i+j+d+t] = b[i + br_elem * 3 / 5];
            if (b[i + br_elem * 4 / 5]!=-1)
            a[i+j+d+t+c] = b[i + br_elem * 4 / 5];
            if (b[i + br_elem * 5 / 5]!=-1)
            a[i + j + d + t + c+p] = b[i + br_elem * 5 / 5];
            if (b[i + br_elem * 6 / 5]!=-1)
            a[i + j + d + t + c + p+se] = b[i + br_elem * 6 / 5];
            if (b[i + br_elem * 7 / 5]!=-1)
            a[i + j + d + t + c + p + se+ sd] = b[i + br_elem * 7 / 5];
            if (b[i + br_elem * 8 / 5]!=-1)
            a[i + j + d + t + c + p + se + sd+o] = b[i + br_elem * 8 / 5];
            if (b[i + br_elem * 9 / 5]!=-1 )
            a[i + j + d + t + c + p + se + sd + o+dev] = b[i + br_elem * 9 / 5];

      }
}
void ispuniniz(int *a)
{
      srand((unsigned)time(NULL));

      for (int i = 0; i <br_elem/10; i++)
      {
            a[i] =INT16_MAX * rand() % 10;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
      for (int i = br_elem/10; i <br_elem / 10*2; i++)
      {
            a[i] = INT16_MAX * rand() % 1000;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
      for (int i = br_elem / 10*2; i <br_elem / 10 * 3; i++)
      {
            a[i] = INT16_MAX * rand() % 10000;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
      for (int i = br_elem / 10 * 3; i <br_elem / 10 * 4; i++)
      {
            a[i] = INT16_MAX * rand() % 10000;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
      for (int i = br_elem / 10 * 4; i <br_elem / 10 * 5; i++)
      {
            a[i] = INT16_MAX * rand() % 100000;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
      for (int i = br_elem / 10 * 5; i <br_elem / 10 * 6; i++)
      {
            a[i] = INT16_MAX * rand() % 1000000;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
      for (int i = br_elem / 10 * 6; i <br_elem / 10 * 7; i++)
      {
            a[i] = INT16_MAX * rand() % 10000000;
            if(a[i]<0)
            a[i]=a[i]*-1;
      }
```

```
	for (int i = br_elem / 10 * 7; i <br_elem / 10 * 8; i++)
	{
		a[i] = INT16_MAX * rand() % 100000000;
		if(a[i]<0)
		a[i]=a[i]*-1;
	}
	for (int i = br_elem / 10 * 8; i <br_elem /10*9; i++)
	{
		a[i] = INT16_MAX * rand() % 1000000000;
		if(a[i]<0)
		a[i]=a[i]*-1;
	}
	for (int i = br_elem / 10 * 9; i <br_elem; i++)
	{
		a[i] = INT16_MAX * rand() % 10000000000;
		if(a[i]<0)
		a[i]=a[i]*-1;
	}

}
void radixsort(int *a, int *b)
{
	int j = 0, d = 0, t = 0, c = 0, p = 0, se = 0, sd = 0, o = 0, dev = 0, des = 0;
	// razdvajanje po bucketima
	for (int i = 0; i <br_elem; i++)
	{
		if (a[i] < 10)
		{
			b[j] = a[i];
			j++;
		}
		if (a[i] >= 10 &&a[i] < 100)
		{
			b[d+br_elem/5] = a[i];
			d++;
		}
		if (a[i] >= 100 &&a[i] < 1000)
		{
			b[t+br_elem*2 / 5] = a[i];
			t++;
		}
		if (a[i] >= 1000 &&a[i] < 10000)
		{
			b[c + br_elem * 3 / 5] = a[i];
			c++;
		}
		if (a[i] >= 10000 &&a[i] < 100000)
		{
			b[p + br_elem * 4 / 5] = a[i];
			p++;
		}
		if (a[i] >= 100000 &&a[i] < 1000000)
		{
			b[se + br_elem * 5 / 5] = a[i];
			se++;
		}
		if (a[i] >= 1000000 &&a[i] < 10000000)
		{
			b[sd + br_elem * 6 / 5] = a[i];
			sd++;
		}
		if (a[i] >= 10000000 &&a[i] < 100000000)
```

```
			{
				b[o + br_elem * 7 / 5] = a[i];
				o++;
			}
			if (a[i] >= 100000000 &&a[i] < 1000000000)
			{
				b[dev + br_elem * 8 / 5] = a[i];
				dev++;
			}
			if (a[i] >= 1000000000)
			{
				b[des + br_elem * 9 / 5] = a[i];
				des++;
			}
		}
		quicksort(b, 0, j - 1, 0); //1. thread
		quicksort(b, br_elem * 2 / 5, t - 1, 0);
		quicksort(b, br_elem * 5 / 5, se - 1, 0);

		quicksort(b, br_elem * 4 / 5, p - 1, 0);//2. thread
		quicksort(b, br_elem * 6 / 5, sd - 1, 0);
		quicksort(b, br_elem * 7 / 5, o - 1, 0);

		quicksort(b, br_elem / 5, d - 1, 0); //3. thread
		quicksort(b, br_elem * 8 / 5, dev - 1, 0);

		quicksort(b, br_elem * 3 / 5, c - 1, 0); //4. thread
		quicksort(b, br_elem * 9 / 5, des - 1, 0);
		spajanje(a, b, j, d, t, c, p, se, sd, o, dev, des);
}
void ispunimale(int *b)
{
		for (int i = 0; i <br_elem*2; i++)
		{
			b[i] = -1;
		}
}
int main()
{

		double t1, t2;
		t1 = omp_get_wtime();
		int *b = newint[br_elem*2];
		int *a = newint[br_elem];
		ispunimale(b);
		ispuniniz(a);
		t2 = omp_get_wtime();
		printf("vrijeme pravljenja niza je: %g s\n",(t2-t1));
		t1 = omp_get_wtime();
		radixsort(a, b);
		t2 = omp_get_wtime();
		printf("vrijeme sortiranja je: %g s\n",(t2-t1));
		provjera(a);
		free(a);
		free(b);

		return 0;
}
```

Prilog 6 – Radix sort (paralelna izvedba)

```
#include<stdio.h>
#include<time.h>
#include<iostream>
#include<omp.h>
#include<stdlib.h>
#include<stdint.h>
usingnamespace std;
#definebr_elem 200000000

void provjera( inta[])
{
	for (int i = 0; i <br_elem-1; i++)
	{
		if (a[i] >a[i + 1])
			cout<<"nije dobro sortiran "<<a[i-1] <<" "<<a[i] <<" "<<a[i+1]<<"
"<<a[i+2]<<endl;//nije dobro sortiran

	}
	cout <<"dobro sortiran"<< endl;//dobro sortiran
}
void zamjena(inti, intj, inta[]) {
	int temp = a[i];
	a[i] = a[j];
	a[j] = temp;
}
void quicksort( inta[],intindex , intdesni, intlijevi)
{
	int temp;
	int mid = (lijevi + desni) / 2;
	int i = lijevi;
	int j = desni;
	int pivot = a[mid+index];
	while (lijevi<j || i<desni)
	{
		while (a[i+index]<pivot)
			i++;
		while (a[j+index]>pivot)
			j--;

		if (i <= j) {
			temp = a[i+index];
			a[i+index] = a[j+index];
			a[j+index] = temp;
			i++;
			j--;
		}
		else {
			if (lijevi<j)
				quicksort(a,index, j , lijevi);
			if (i<desni)
				quicksort(a, index, desni, i);
			return;
		}
	}
}
void spajanje(  int *a, int *b,
			intj,  intd,  intt,  intc,  intp,  intse,  intsd,  into,  intdev,
intdes)
{
	int max = j;
	if (max <d) max = d;
	if (max <t) max = t;
```

```
        if (max <c) max = c;
        if (max <p) max = p;
        if (max <se) max = se;
        if (max <sd) max = sd;
        if (max <o) max = o;
        if (max <dev) max = dev;
        if (max <des) max = des;
        for (int i = 0; i <br_elem/5; i++)
        {
                if(b[i]!=-1)
                a[i] = b[i];
                if (b[i + br_elem /5]!=-1)
                a[i+j] = b[i + br_elem / 5];
                if (b[i + br_elem * 2 / 5]!=-1)
                a[i+j+d] = b[i + br_elem * 2 / 5];
                if (b[i + br_elem * 3 / 5]!=-1)
                a[i+j+d+t] = b[i + br_elem * 3 / 5];
                if (b[i + br_elem * 4 / 5]!=-1)
                a[i+j+d+t+c] = b[i + br_elem * 4 / 5];
                if (b[i + br_elem * 5 / 5]!=-1)
                a[i + j + d + t + c+p] = b[i + br_elem * 5 / 5];
                if (b[i + br_elem * 6 / 5]!=-1)
                a[i + j + d + t + c + p+se] = b[i + br_elem * 6 / 5];
                if (b[i + br_elem * 7 / 5]!=-1)
                a[i + j + d + t + c + p + se+ sd] = b[i + br_elem * 7 / 5];
                if (b[i + br_elem * 8 / 5]!=-1)
                a[i + j + d + t + c + p + se + sd+o] = b[i + br_elem * 8 / 5];
                if (b[i + br_elem * 9 / 5]!=-1 )
                a[i + j + d + t + c + p + se + sd + o+dev] = b[i + br_elem * 9 / 5];
        }
}
void ispuniniz( int *a)
{
        srand((unsigned)time(NULL));

        for (int i = 0; i <br_elem/10; i++)
        {
                a[i] =INT16_MAX * rand() % 10;
                if(a[i]<0)
                a[i]=a[i]*-1;
        }
        for (int i = br_elem/10; i <br_elem / 10*2; i++)
        {
                a[i] = INT16_MAX * rand() % 1000;
                if(a[i]<0)
                a[i]=a[i]*-1;
        }
        for (int i = br_elem / 10*2; i <br_elem / 10 * 3; i++)
        {
                a[i] = INT16_MAX * rand() % 10000;
                if(a[i]<0)
                a[i]=a[i]*-1;
        }
        for (int i = br_elem / 10 * 3; i <br_elem / 10 * 4; i++)
        {
                a[i] = INT16_MAX * rand() % 10000;
                if(a[i]<0)
                a[i]=a[i]*-1;
        }
        for (int i = br_elem / 10 * 4; i <br_elem / 10 * 5; i++)
        {
                a[i] = INT16_MAX * rand() % 100000;
```

```
			if(a[i]<0)
			a[i]=a[i]*-1;
		}
		for (int i = br_elem / 10 * 5; i <br_elem / 10 * 6; i++)
		{
			a[i] = INT16_MAX * rand() % 1000000;
			if(a[i]<0)
			a[i]=a[i]*-1;
		}
		for (int i = br_elem / 10 * 6; i <br_elem / 10 * 7; i++)
		{
			a[i] = INT16_MAX * rand() % 10000000;
			if(a[i]<0)
			a[i]=a[i]*-1;
		}
		for (int i = br_elem / 10 * 7; i <br_elem / 10 * 8; i++)
		{
			a[i] = INT16_MAX * rand() % 100000000;
			if(a[i]<0)
			a[i]=a[i]*-1;
		}
		for (int i = br_elem / 10 * 8; i <br_elem /10*9; i++)
		{
			a[i] = INT16_MAX * rand() % 1000000000;
			if(a[i]<0)
			a[i]=a[i]*-1;
		}
		for (int i = br_elem / 10 * 9; i <br_elem; i++)
		{
			a[i] = INT16_MAX * rand() % 10000000000;
			if(a[i]<0)
			a[i]=a[i]*-1;
		}

}

void radixsort( int *a, int *b)
{
	int j = 0, d = 0, t = 0, c = 0, p = 0, se = 0, sd = 0, o = 0, dev = 0, des = 0;
	// razdvajanje po bucketima
	for (int i = 0; i <br_elem; i++)
	{
		if (a[i] < 10)
		{
			b[j] = a[i];
			j++;
		}
		if (a[i] >= 10 &&a[i] < 100)
		{
			b[d+br_elem/5] = a[i];
			d++;
		}
		if (a[i] >= 100 &&a[i] < 1000)
		{
			b[t+br_elem*2 / 5] = a[i];
			t++;
		}
		if (a[i] >= 1000 &&a[i] < 10000)
		{
			b[c + br_elem * 3 / 5] = a[i];
			c++;
		}
```

```
			if (a[i] >= 10000 &&a[i] < 100000)
			{
				b[p + br_elem * 4 / 5] = a[i];
				p++;
			}
			if (a[i] >= 100000 &&a[i] < 1000000)
			{
				b[se + br_elem * 5 / 5] = a[i];
				se++;
			}
			if (a[i] >= 1000000 &&a[i] < 10000000)
			{
				b[sd + br_elem * 6 / 5] = a[i];
				sd++;
			}
			if (a[i] >= 10000000 &&a[i] < 100000000)
			{
				b[o + br_elem * 7 / 5] = a[i];
				o++;
			}
			if (a[i] >= 100000000 &&a[i] < 1000000000)
			{
				b[dev + br_elem * 8 / 5] = a[i];
				dev++;
			}
			if (a[i] >= 1000000000)
			{
				b[des + br_elem * 9 / 5] = a[i];
				des++;
			}
		}
#pragma omp parallel sections
{
		#pragma omp section
		{
		quicksort(b, 0, j - 1, 0); //1. thread
		quicksort(b, br_elem * 2 / 5, t - 1, 0);
		quicksort(b, br_elem * 5 / 5, se - 1, 0);
		}
		#pragma omp section
		{
		quicksort(b, br_elem * 4 / 5, p - 1, 0);//2. thread
		quicksort(b, br_elem * 6 / 5, sd - 1, 0);
		quicksort(b, br_elem * 7 / 5, o - 1, 0);
		}
		#pragma omp section
		{
		quicksort(b, br_elem / 5, d - 1, 0); //3. thread
		quicksort(b, br_elem * 8 / 5, dev - 1, 0);
		}
		#pragma omp section
		{
		quicksort(b, br_elem * 3 / 5, c - 1, 0); //4. thread
		quicksort(b, br_elem * 9 / 5, des - 1, 0);
		}
}


		double t1,t2;
		t1 = omp_get_wtime();
		spajanje(a, b, j, d, t, c, p, se, sd, o, dev, des);
		t2 = omp_get_wtime();
```

```
	printf("vrijeme spajanja je: %g s\n",(t2-t1));
}
void ispunimale( int *b)
{
	for (int i = 0; i <br_elem*2; i++)
	{
		b[i] = -1;
	}

}
int main()
{
	double t1, t2;
	t1 = omp_get_wtime();
	int *b = newint[br_elem*2];
	int *a = newint[br_elem];
	ispunimale(b);
	ispuniniz(a);
	t2 = omp_get_wtime();
	printf("vrijeme pravljenja niyova %g s\n",(t2-t1));
	omp_set_num_threads(4);
	t1 = omp_get_wtime();
	radixsort(a, b);
	cout << endl;
	t2 = omp_get_wtime();
	printf("vrijeme sortiranja je: %g s\n",(t2-t1));
	provjera(a);
	free(a);
	free(b);
	return 0;
}
```