Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Darko Božidar

Vzporedni algoritmi za urejanje podatkov

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Darko Božidar sem avtor magistrskega dela z naslovom:

Vzporedni algoritmi za urejanje podatkov

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 26. junija 2015 Podpis avtorja:

Za mentorstvo, pomoč, podporo in strokovne nasvete se zahvaljujem docentu dr. Tomažu Dobravcu.

Iskreno se zahvaljujem tudi svojim staršem in sestri Vesni za podporo in potrpežljivost skozi vsa leta študija.

Kazalo

Povzetek

V magistrskem delu smo preučili, implementirali in medsebojno primerjali zaporedne in vzporedne algoritme za urejanje podatkov. Implementirali smo sedem algoritmov, in sicer bitono urejanje, večkoračno bitono urejanje, prilagodljivo bitono urejanje, urejanje z zlivanjem, hitro urejanje, urejanje po delih in urejanje z vzorci. Zaporedne algoritme smo implementirali na centralno procesni enoti s pomočjo jezika C++, vzporedne pa na grafično procesni enoti s pomočjo arhitekture CUDA. Naštete implementacije vzporednih algoritmov smo delno tudi izboljšali. Poleg tega smo tudi zagotovili, da lahko urejajo zaporedja poljubne dolžine. Algoritme smo primerjali na zaporedjih števil različnih dolžin, porazdeljenih po šestih različnih porazdelitvah, ki so bila sestavljena iz 32-bitnih števil, 32-bitnih parov ključ-vrednost, 64-bitnih števil in 64-bitnih parov ključ-vrednost. Ugotovili smo, da je med zaporednimi algoritmi najhitrejše urejanje po delih, medtem ko je pri vzporednih algoritmih najhitrejše urejanje po delih ali urejanje z zlivanjem (odvisno od vhodne porazdelitve). Z vzporednimi implementacijami smo dosegli tudi do 157-kratno pohitritev v primerjavi z zaporednimi implementacijami.

Ključne besede

vzporedni algoritmi, primerjava algoritmov, urejanje podatkov, grafična kartica, CUDA

Abstract

In this master's thesis we studied, implemented and compared sequential and parallel sorting algorithms. We implemented seven algorithms: bitonic sort, multistep bitonic sort, adaptive bitonic sort, merge sort, quicksort, radix sort and sample sort. Sequential algorithms were implemented on a central processing unit using C++, whereas parallel algorithms were implemented on a graphics processing unit using CUDA architecture. We improved the above mentioned implementations and adopted them to be able to sort input sequences of arbitrary length. We compared algorithms on six different input distributions, which consist of 32-bit numbers, 32-bit key-value pairs, 64-bit numbers and 64-bit key-value pairs. The results show that radix sort is the fastest sequential sorting algorithm, whereas radix sort and merge sort are the fastest parallel algorithms (depending on the input distribution). With parallel implementations we achieved speedups of up to 157-times in comparison to sequential implementations.

Keywords

parallel algorithms, algorithm comparison, sorting, graphics card, CUDA

Poglavje 1

Uvod

Algoritmi za urejanje podatkov se uporabljajo v številnih problemskih domenah, zato je njihova hitra izvedba ključnega pomena. Obstajajo številne zaporedne implementacije algoritmov za urejanje podatkov, ki jih lahko pohitrimo s pomočjo naprednejše strojne opreme. Slabost omenjenega pristopa je povečanje stroškov. Poleg tega ni zagotovljeno, da bomo z boljšo strojno opremo dosegli želeno stopnjo pohitritve. S pojavom cenovno dostopnih vzporednih arhitektur (na primer GPE), se trend algoritmov za urejanje podatkov seli k vzporednim implementacijam, saj lahko z njimi dosežemo večjo učinkovitost.

Področje vzporednih algoritmov za urejanje podatkov je zelo aktivno. Sprva so bili algoritmi na GPE implementirani s pomočjo grafičnih programskih vmesnikov kot sta na primer *OpenGL* in *DirectX*. Omenjeni programski vmesniki so namenjeni upodabljanju grafike, kar otežuje implementacijo splošno namenskih vzporednih algoritmov. Poleg tega opisani programi niso učinkoviti. S pojavom arhitekture CUDA leta 2007 se implementiranje splošno namenskih vzporednih algoritmov v veliki meri poenostavi [9, 30].

V magistrskem delu smo se osredotočili na najbolj raziskovane vzporedne algoritme za urejanje podatkov v literaturi. To so hitro urejanje [9], bitono urejanje [28], prilagodljivo bitono urejanje [29], urejanje po delih [30], urejanje z zlivanjem [30] in urejanje z vzorci [13]. Vsi našteti algoritmi, z

izjemo urejanja po delih, temeljijo na primerjavah. Za te algoritme smo se odločili, ker njihove implementacije na arhitekturi CUDA še niso nikjer primerjane. Poleg tega se v literaturi redko pojavi primerjava med zaporedno implementacijo na CPE in vzporedno na GPE. Kar nekaj člankov opisuje primerjavo vzporednih algoritmov na eni ali več nitih, kar ni ekvivalentno primerjavi med optimizirano zaporedno in vzporedno implementacijo. Naša domneva je, da bo za kratke ključe najhitrejši algoritem urejanja po delih. Pričakujemo tudi, da bodo nekateri algoritmi občutljivi na določene porazdelitve vhodnih podatkov. Domnevamo, da bo urejanje po delih počasnejše pri daljših ključih, hitro urejanje bo počasnejše pri majhnem številu ključev, urejanje z zlivanjem bo hitrejše pri delno urejenih podatkih, itd.

Opisane implementacije vzporednih algoritmov smo delno tudi izboljšali. Za večkoračno bitono urejanje smo izdelali postopek za izgradnjo d-koračnega dela, na podlagi katerega smo implementirali tudi d-koračne ščepce. D-koračne ščepce smo tudi optimizirali z rekurzivnimi funkcijami za *branje*, *pisanje* in *primerjanje* elementov. Z naštetimi funkcijami smo zagotovili, da se elementi vedno nahajajo v registrih niti, kar omogoča hitrejše izvajanje urejanja. S pomočjo optimizirane redukcije [17] in optimizirane komulativne vsote [33] smo pohitrili delovanje hitrega urejanja [9]. Izboljšali smo tudi postopek iskanja pivotov v hitrem urejanju. Določili smo jih kot povprečje starega pivota in maksimalne oziroma minimalne vrednosti levega oziroma desnega podzaporedja, ki ju dobimo po opravljeni delitvi. Zato smo lahko iz ščepca za globalno hitro urejanje odstranili redukcijo za iskanje minimuma in maksimuma. S pomočjo komulativne vsote za predikate [18] smo pohitrili urejanje po delih [30]. Kot smo omenili, je Peters s sodelavci [29] implementiral vzporedno prilagodljivo bitono urejanje v kombinaciji z večkoračnim bitonim urejanjem [28]. V delu nismo implementirali opisanega hibrida, ampak samostojno prilagodljivo bitono urejanje. S tem smo želeli doseči pravičnejšo primerjavo med zaporednim in vzporednim prilagodljivim bitonim urejanjem. Pri urejanju z vzorci [13] smo izdelali tudi preprost in učinkovit postopek za shranjevanje elementov v sektorje. Za vse algoritme smo tudi zagotovili,

da lahko urejajo zaporedja poljubne dolžine (ne samo večkratnikov potence števila 2).

1.1 Sorodna dela

Pred pojavom arhitekture CUDA je za najhitrejši algoritem na GPE veljal algoritem bitonega urejanja (ang. *bitonic sort*) [14, 15]. Cederman in Tsigas [9] sta predlagala prvo konkurenčno implementacijo hitrega urejanja (ang. *quicksort*) na arhitekturi CUDA, ki se je lahko kosala s takratnimi najhitrejšimi algoritmi na GPE. Baraglia s sodelavci [4] predlaga bitono urejanje, ki je hitrejše od prej omenjenega hitrega urejanja za tabele večje od 8MB. Satish s sodelavci [30] nato implementira za tisti čas najhitrejše urejanje s primerjavami, tj. urejanje z zlivanjem (ang. *merge sort*) in najhitrejše urejanje brez primerjav, tj. urejanje po delih (ang. *radix sort*). Avtorji navajajo svojo implementacijo urejanja po delih kot najhitrejšo implementacijo vzporednega urejanja za tisti čas. Leischner s sodelavci [23] predlaga implementacijo urejanja z vzorci (ang. *sample sort*), ki je v povprečju za 30 % hitrejša od prej omenjenega urejanja z zlivanjem [30]. Satish s sodelavci [31] izboljša implementacijo urejanja z zlivanjem, ki je do dvakrat hitrejša od prej omenjenega urejanja z vzorci [23]. Dehne in Zaboli [13] predlagata robustnejše urejanje z vzorci v primerjavi s [23], katerega učinkovitost je neodvisna od porazdelitve vhodnih podatkov. Merrill in Grimshaw [25] implementirata urejanje po delih, ki je hitrejše od prej omenjenega Satishevega [30] in velja za najhitrejše urejanje brez primerjav na GPE. Peters s sodelavci [28] implementira večkoračno bitono urejanje (ang. *multistep bitonic sort*), ki velja za takratno najhitrejše urejanje s primerjavami. Je približno enako učinkovito kot prej omenjeno Satishevo urejanje z zlivanjem [31]. Kasneje implementacijo tudi nadgradijo in ustvarijo hibrid med prej omenjenim večkoračnim [28] in prilagodljivim bitonim urejanjem (ang. *adaptive bitonic sort*) [29]. Omenjena implementacija velja za najhitrejše vzporedno urejanje s primerjavami na GPE.

Poleg urejanja števil, je aktualno tudi področje urejanja velikih količin podatkov, ki se jih ne da v celoti shraniti v pomnilnik GPE zaradi njihove obsežnosti. Zaradi omenjenega razloga je potrebno podatke razdeliti na več delov, dele posamično urediti na GPE in jih ponovno združiti na gostitelju. Primer takega algoritma predlaga Amirul s sodelavci [3]. Algoritem je namenjen urejanju velikih količin nizov dolžine 20 znakov.

Izpostaviti velja tudi delo Mišića in Tomaševića [26]. Avtorja sta izvedla primerjavo med Cedermanovim hitrim urejanjem [9], urejanjem z zlivanjem knjižnice *Thrust* [2] in urejanjem po delih knjižnice *CUDPP* [1]. Algoritma urejanja z zlivanjem in urejanja po delih temeljita na delu [30], ki je predstavljalo osnovo tudi za naši implementaciji omenjenih urejanj. Ugotovila sta, da je algoritem urejanje po delih za približno 45 % hitrejši od ostalih dveh algoritmov. Urejanje z zlivanjem in hitro urejanje sta podobno hitra, pri čemer je hitro urejanje bolj občutljivo na porazdelitev vhodnih podatkov.

Na podlagi rezultatov magistrskega dela smo napisali tudi članek z naslovom *Comparison of parallel sorting algorithms* in ga poslali v objavo v revijo *Concurrency and Computation: Practice and Experience* [7]. Za razliko od ostalih del, predstavljenih v tem poglavju, smo v našem članku izvedli primerjavo med sedmimi implementacijami vzporednih algoritmov za urejanje podatkov na štirih tipih vhodnih podatkov.

Poglavje 2

Vzporedno računanje

2.1 Arhitektura CUDA

CUDA (*Compute Unified Device Architecture*) je programski vmesnik, ki močno poenostavi implementacijo programov za GPE podjetja *Nvidia* in predstavlja razširitev jezika *C*. Aplikacije so razdeljene na zaporedni program, izveden na *gostitelju* (ang. *host*) in enega ali več vzporednih *ščepcev* (ang. *kernel*), izvedenih na *napravi* (ang. *device*). Program na gostitelju je običajno izveden na *CPE*, medtem ko so programi na napravi običajno izvedeni na eni ali več *GPE* [11, 30].

Ščepec je program, ki ga izvajajo niti na napravi. Sestavljen je iz *mreže niti* (ang. *grid*), ki je razdeljena na enega ali več *blokov niti* (ang. *thread block*). Ob zagonu ščepca moramo podati dimenzijo mreže (število blokov niti) in dimenzijo blokov (število niti v blokih) [30].

Na strojnem nivoju so GPE sestavljene iz množice *tokovnih procesorskih gruč* (ang. *streaming multiprocesor*). Te so sestavljeni iz več *tokovnih procesorjev* (ang. *streaming processor*). Procesorske gruče ne izvedejo vseh niti v bloku naenkrat, ampak jih najprej razdelijo v skupine velikosti 32, imenovane *snopi* (ang. *warp*), ki jih nato izvedejo. Niti v istem snopu si delijo isto ukazno enoto, zaradi česar lahko sočasno izvajajo ukaze. Opisan način izvajanja se imenuje *SIMT* (ang. *single instruction, multiple thread* oziroma

slo. *en ukaz, več niti*). Slabost opisanega načina izvajanja je, da lahko posamezen snop izvaja samo en ukaz naenkrat. Za dober izkoristek strojne opreme GPE moramo poskrbeti, da niti v snopu izvajajo iste ukaze. Kadar niti istega snopa vsebujejo različne ukaze, se ti izvedejo zaporedno, kar imenujemo *razhajanje niti* (ang. *control divergence*). Pri tem je potrebno poudariti, da se snopi v istem bloku niti ne izvajajo sočasno, kar predstavlja težavo. V številnih programih je potrebno zagotoviti, da vse niti istega bloka dosežejo določeno točko programa, preden nadaljujejo z izvajanjem. V ta namen nam arhitektura CUDA omogoča *sinhronizacijo na pregradi* (ang. *barrier synchronization*). Bloke niti ne moremo medsebojno sinhronizirati, saj se ti lahko izvedejo v poljubnem vrstnem redu. Globalno sinhronizacijo med bloki lahko zagotovimo samo z več zaporednimi klici ščepcev, zaradi česar moramo vzporedne algoritme razdeliti na več ščepcev [11, 27, 30].

Niti lahko dostopajo do več vrst pomnilnikov. Vsaka nit ima svoje registre, ki imajo najmanjšo zakasnitev dostopa izmed vseh opisanih pomnilnikov. Težava je, da ima vsaka nit le nekaj 32-bitnih registrov (od 64 do 255 na trenutno najnovejših GPE). Vsaka procesorska gruča vsebuje *deljen pomnilnik* (ang. *shared memory*), ki ima prav tako zelo majhno zakasnitev. Ta pomnilnik je zelo majhen (nekaj 10 KB). Niti znotraj istega bloka dostopajo do istega deljenega pomnilnika, zato lahko z njegovo pomočjo medsebojno komunicirajo, kar je ključno za izvajanje vzporednih programov. Tu je potrebno izpostaviti, da lahko prevajalnik optimizira program. Na strojnem nivoju lahko uporabi registre namesto deljenega pomnilnika, s čimer lahko prepreči komunikacijo med nitmi. Kadar eksplicitno želimo, da prevajalnik ne izvede omenjene optimizacije, moramo deljen pomnilnik deklarirati s ključno besedo `volatile`. *Globalni pomnilnik* je skupen tako gostitelju kot tudi napravi in je tipično zelo velik (nekaj GB na današnjih GPE). Z njegovo pomočjo lahko niti komunicirajo v različnih blokih oziroma v različnih ščepcih. Njegova slabost je počasnost. Dostope lahko v veliki meri pohitrimo, če poskrbimo, da niti dostopajo na zaporedne pomnilniške naslove. Tak način dostopa imenujemo *zaporedni* (ang. *coalasced*) dostop. V tem

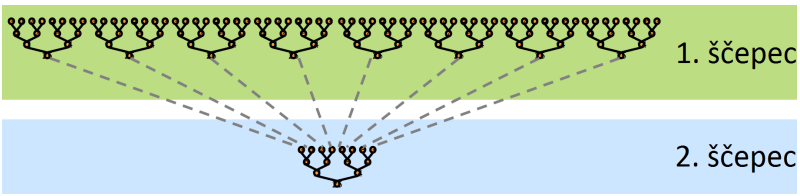
primeru lahko strojna oprema združi več dostopov do globalnega pomnilnika v kratko transakcijo. Vse niti imajo na voljo tudi skupni *konstantni pomnilnik*, iz katerega lahko samo berejo. Konstantni pomnilnik je veliko manjši od globalnega pomnilnika (64 KB), vendar je veliko hitrejši [27, 30].

Zaradi sočasnega izvajanja niti lahko cikli *preberi-spremeni-zapiši* (ang. *read-modify-write*) predstavljajo težavo. To je cikel, v katerem nit prebere neko vrednost iz določene pomnilniške lokacije, jo spremeni in zapiše nazaj na isto lokacijo. Kadar želi več niti sočasno izvesti omenjeni cikel nad isto pomnilniško lokacijo, lahko pride do *nedoločenega sinhronizacijskega stanja* (ang. *race condition*) niti. V ta namen nam arhitektura CUDA omogoča *atomarne operacije* (ang. *atomic operation*). Te operacije zagotavljajo, da lahko trenutna nit v celoti izvede cikel *preberi-spremeni-zapiši*. Pri tem ne bo smela nobena druga nit izvajati tega cikla nad isto pomnilniško lokacijo, dokler trenutna nit ne zaključi cikla [27].

Potrebno je upoštevati, da imajo novejše GPE zmogljivejšo in naprednejšo strojno in programsko opremo, zato imajo posledično tudi novejše in naprednejše funkcionalnosti. Za lažje sledenje funkcionalnosti različnih generacij Nvidia GPE se uporablja notacija *računske zmožnosti* (ang. *compute capability*). Različica računske zmožnosti GPE je predstavljena s številko (na primer 1.1, 3.0, itd.). Vse GPE z enako različico računske zmožnosti imajo enake funkcionalnosti. Večja kot je različica računske zmožnosti, bolj napredna je GPE [27].

2.2 Vzporedna redukcija

Vzporedna redukcija (ang. *parallel reduction*) velja za enega temeljnih gradnikov številnih vzporednih algoritmov. Na vhodu prejme tabelo vrednosti in operacijo redukcije. Deluje tako, da najprej inicializira rezultat z identiteto podane operacije redukcije (na primer seštevanje - 0, množenje - 1). Nato opravi prehod čez vse elemente tabele ter hkrati izvede operacijo redukcije med rezultatom in trenutnim elementom tabele. Operacija redukcije



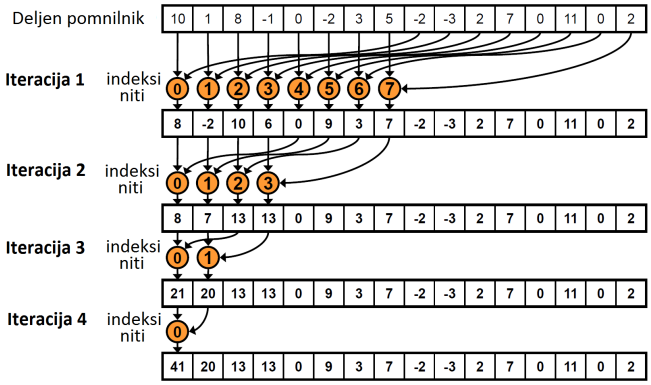
Slika 2.1: Redukcija z dvema klicema ščepcev (prirejeno po [17]).

mora biti komutativna in asociativna, saj lahko prehod po tabeli poteka v poljubnem vrstnem redu. Poleg tega mora imeti operacija definirano tudi identiteto. Primeri opisanih operacij so seštevanje, množenje, minimum in maksimum [22].

Redukcijo lahko izvedemo tudi vzporedno. Dober primer predstavlja športni turnir, na katerem sodeluje n ekip. Vse tekme enega kroga turnirja se lahko odigrajo sočasno. V naslednji krog se uvrsti $\frac{n}{2}$ ekip, v krog za tem $\frac{n}{4}$ itd., dokler na koncu ne dobimo zmagovalca [22].

Slika 2.1 prikazuje primer redukcije nad tabelo dolžine 64. Pri tem predpostavimo, da lahko vsak blok niti opravi redukcijo nad največ 8 elementi. Ugotovimo, da je potrebnih 8 blokov. Vsak blok izvede redukcijo nad svojim pripadajočim kosom vhodnih podatkov dolžine 8. Nato mora eden izmed omenjenih blokov izvesti še redukcijo nad dobljenimi rezultati. Težava je v tem, da je potrebno sinhronizirati vmesne rezultate posameznih blokov niti, preden nad njimi ponovimo redukcijo za izračun končnega rezultata. Vrstni red izvajanja blokov niti ni vnaprej določen. Poleg tega CUDA ne zagotavlja nobenega mehanizma za sinhronizacijo blokov niti znotraj istega ščepca. V ta namen opravimo več klicev ščepcev, saj se ti izvajajo zaporedno. V prvem ščepcu vsak blok niti shrani rezultat redukcije v tabelo vmesnih rezultatov, nakar nov ščepec prebere rezultate in ponovi redukcijo. V splošnem opisan postopek rekurzivno ponavljamo, dokler ne ostane en sam blok niti, ki izračuna končni rezultat [17].

Število iteracij, potrebnih za izvedbo redukcije tabele dolžine n, je enako $O(\log n)$. V prvi oziroma delovno najbolj zahtevni iteraciji je potrebno opraviti $O(n)$ operacij redukcije. S p procesorji lahko to izvedemo vzporedno v



Slika 2.2: Osnovna vzporedna redukcija (prirejeno po [17]).

času $O(\frac{n}{p})$. Časovna zahtevnost vzporedne redukcije je zato enaka:

$$T_p = O\left(\frac{n}{p} \cdot \log n\right). \tag{2.1}$$

Pri dovolj velikem številu procesorjev (tj. $O(n)$) lahko vsako iteracijo izvedemo v času *O(1)*. V tem primeru je časovna zahtevnost enaka:

$$T_n = O(\log n). \tag{2.2}$$

2.2.1 Osnovna vzporedna redukcija

Slika 2.2 prikazuje primer osnovne vzporedne redukcije seštevanja. Kot vidimo, se v vsaki iteraciji razpolovi število aktivnih niti. Prednost opisanega pristopa je v zelo majhnem razhajanju niti v snopih. Vzemimo za primer tabelo dolžine 128 in blok, ki vsebuje 64 niti. V prvi iteraciji bo aktivnih vseh 64 niti oziroma dva snopa. V naslednji iteraciji bo en snop aktiven in en snop neaktiven. Šele v tretji iteraciji bo prišlo do razhajanja niti v prvem snopu, saj bo 16 niti izvajajo seštevanje, medtem ko ostalih 16 niti seštevanja ne bo izvajalo. Ugotovimo, da je razhajanje prisotno samo v prvem snopu, in sicer v zadnjih petih iteracijah redukcije. Poleg tega med samim izvajanjem redukcije ne pride do konfliktov pomnilniških bank. Opisan algoritem

```
1  // vhodnaTab:  vhodna tabela,
2  // izhodnaTab: izhodna tabela.
3  scepecOsnovnaRedukcija(vhodnaTab, izhodnaTab):
4    extern __shared__ sTab[]
5    indeks = blockIdx.x *  2 * blockDim.x + threadIdx.x
6
7    sTab[threadIdx.x] = vhodnaTab[indeks]
8    sTab[threadIdx.x + blockDim.x] = vhodnaTab[indeks + blockDim.x]
9    __syncthreads()
10
11   for (korak = blockDim.x; korak > 0; korak /= 2):
12     if threadIdx.x < korak:
13       sTab[threadIdx.x] += sTab[threadIdx.x + korak]
14     __syncthreads()
15
16   if threadIdx.x == 0: izhodnaTab[blockIdx.x] = sTab[0]
```

Psevdokoda 2.1: Ščepec za osnovno vzporedno redukcijo (prirejeno po [17]).

je prikazan v psevdokodi 2.1. Slabost algoritma je v slabi izkoriščenosti virov GPE, saj se z vsako iteracijo za polovico zmanjša število aktivnih niti [17, 22].

2.2.2 Vzporedna redukcija z razvitjem zanke

Psevdokoda 2.2 prikazuje vzporedno redukcijo z razvitjem zanke. V osnovni implementaciji vzporedne redukcije smo ugotovili, da ta slabo izkorišča vire GPE. Za odpravo omenjene težave lahko združimo zaporedno in vzporedno implementacijo. Medtem ko niti berejo elemente iz globalnega pomnilnika, lahko sočasno izvajajo operacijo redukcije (vrstice 18 - 21). S tem dosežemo, da so vse niti aktivne večino časa izvajanja ščepca. Upoštevati moramo tudi, da zanka *for* zahteva dodatno procesiranje. V prid nam služi ugotovitev iz algoritma osnovne redukcije, kjer zadnjih šest iteracij redukcije izvede samo en snop niti. To lahko izkoristimo, kajti v zadnjih šestih iteracij ne potrebujemo sinhronizacije niti, saj se niti v istem snopu izvajajo sinhrono. Namesto zanke *for* lahko v zadnjih šestih iteracijah redukcije uporabimo razvitje zanke (ang. *loop unroll*), kot je prikazano v vrsticah 1 - 7. Opisana optimizacija prihrani delo tudi ostalim snopom, ker jim ni potrebno izvesti zadnjih šest iteracij zanke. Algoritem bi lahko še dodatno pohitrili, če bi izvedli popolno razvitje zanke oziroma če bi zanko *for* osnovnega algoritma redukcije popolnoma odstranili. V ta namen upoštevajmo, da je število niti v bloku

```
1  template <velikostBloka> redukcijaSnop(volatile sTab, tid):
2    if velikostBloka >= 64: sTab[tid] += sTab[tid + 32]
3    if velikostBloka >= 32: sTab[tid] += sTab[tid + 16]
4    if velikostBloka >= 16: sTab[tid] += sTabtid + 8]
5    if velikostBloka >= 8: sTab[tid] += sTab[tid + 4]
6    if velikostBloka >= 4: sTab[tid] += sTab[tid + 2]
7    if velikostBloka >= 2: sTab[tid] += sTab[tid + 1]
8
9  // vhodnaTab:  vhodna tabela,
10 // izhodnaTab: izhodna tabela,
11 // n:          dolžina vhodne tabele.
12 template <velikostBloka> scepecRedukcijaRZ(vhodnaTab, izhodnaTab, n):
13   extern __shared__ sTab[]
14   vsota, tid = 0, threadIdx.x
15   indeks = blockIdx.x * (2 * velikostBloka) + tid
16   velikostMrezeNiti = velikostBloka * 2 * gridDim.x
17
18   for (; indeks < n; i += velikostMrezeNiti):
19     vsota += vhodnaTab[indeks]
20     if indeks + velikostBloka < n:
21       vsota += vhodnaTab[indeks + velikostBloka]
22
23   sTab[tid] = vsota
24   __syncthreads()
25
26   // Podobno kodo uporabimo za večje bloke (512, 1024, 2048, ...)
27   if velikostBloka >= 256 && tid < 128:
28     sTab[tid] += sTab[tid + 128]
29     __syncthreads()
30   if velikostBloka >= 128 && tid < 64:
31     sTab[tid] += sTab[tid + 64]
32     __syncthreads()
33
34   if tid < 32: redukcijaSnop<velikostBloka>(sTab, threadIdx.x)
35   if tid == 0: izhodnaTab[blockIdx.x] = sTab[0]
```

Psevdokoda 2.2: Ščepec za vzporedno redukcijo z razvitjem zanke (prirejeno po [17]).

```
1 if velBloka == 256: scepecRedukcijaRZ<256><<<⋆parametri⋆/>>>(tab1, tab2)
2 if velBloka == 128: scepecRedukcijaRZ<128><<<⋆parametri⋆/>>>(tab1, tab2)
3 /⋆ ... zmanjšujemo velikost bloka za faktor 2 ⋆/
4 if velBloka == 1: scepecRedukcijaRZ<1><<<⋆parametri⋆/>>>(tab1, tab2)
```

Psevdokoda 2.3: Klic ščepca za vzporedno redukcijo z razvitjem zanke (prirejeno po [17]).

navzgor omejeno. Zaradi tega je navzgor omejena tudi velikost kosa tabele, ki ga lahko naenkrat obdela blok niti. Zaradi opisane omejitve lahko vnaprej določimo maksimalno število iteracij redukcije in s tem posledično dosežemo popolno razvitje zanke (vrstice 27 - 32). Omembe vredno je tudi dejstvo, da velikost bloka niti podamo v parametrih konstrukta `template` (vrstici 1 in

```
1  // vhodTab:  vhodna tabela v deljenem pomnilniku,
2  // izhodTab: izhodna tabela v globalnem pomnilniku.
3  template <velikostBloka> optimiziranaRedukcija(volatile vhodTab, izhodTab):
4    indeksSnop = threadIdx.x / warpSize
5    indeksNitiSnop = threadIdx.x & (warpSize - 1)
6    tid = (indeksSnop * 2 * warpSize) + indeksNitiSnop
7
8    redukcijaSnop<velikostBloka>(vhodTab, tid)
9    __syncthreads()
10
11   if indeksNitiSnop == 0: vhodTab[indeksSnop] = vhodTab[tid]
12   __syncthreads()
13
14   if indeksSnop == 0:
15     redukcijaSnop<velikostBloka / 32>(velikostBloka, threadIdx.x)
16     if indeksNitiSnop == 0: izhodTab[blockIdx.x] = vhodTab[0]
```

Psevdokoda 2.4: Optimizirana vzporedna redukcija (prirejeno po [17]).

12). S pomočjo omenjenega konstrukta jezika *C++* lahko podajamo parametre v času prevajanja, s čimer dosežemo dodatno optimizacijo. Čeprav je velikost bloka poznana že v času prevajanja, jo lahko dinamično izberemo tudi v času izvajanja, kot to prikazuje psevdokoda 2.3 [17].

2.2.3 Optimizirana vzporedna redukcija

Slabost algoritma 2.2 je v tem, da se z vsako iteracijo razpolovi število aktivnih niti (vrstice 27 - 32). To pomeni, da algoritem slabo izkorišča vire GPE. Poleg tega je potrebno med vsako iteracijo izvesti sinhronizacijo niti. Naštete slabosti odpravi algoritem 2.4. Na začetku vsak snop niti izvede redukcijo na svojem pripadajočem kosu tabele dolžine 64 (vrstica 8), za kar uporabi funkcijo `redukcijaSnop` iz psevdokode 2.2. Ta funkcija izkorišča dejstvo sinhronega izvajanja kode v snopih in zaradi tega ni potrebe po sinhronizaciji niti na pregradi. Prva nit vsakega snopa shrani rezultat redukcije svojega pripadajočega snopa (vrstica 11). Nato niti prvega snopa izvedejo redukcijo nad prej shranjenimi rezultati (vrstica 15). Na koncu prva nit v bloku shrani rezultat redukcije v izhodno tabelo. Opozoriti velja, da lahko niti izvajajo redukcijo pred izvedbo prikazane funkcije, kot je prikazano v psevdokodi 2.2 v vrsticah 18 - 21 [17].

2.3 Vzporedna komulativna vsota

Komulativna vsota (ang. *comulative sum, prefix sum, scan*) je ena izmed najpogosteje uporabljenih vzporednih operacij. Deluje tako, da na vhodu prejme tabelo $[a_0,\ a_1,\ ...,\ a_{n-1}]$ in asociativni operator $\oplus$. Kot rezultat vrne tabelo dolžine n:

$$[a_0,\ (a_0 \oplus a_1),\ ...,\ (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]. \tag{2.3}$$

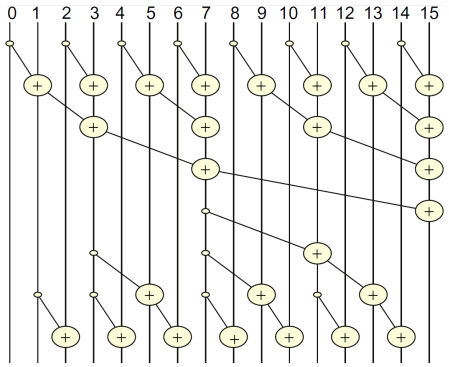
Vzemimo za primer tabelo [4, 1, 1, 7, 2, 3]. Komulativna vsota za operacijo seštevanja je enaka [4, 5, 6, 13, 15, 18]. Kot vidimo, vsak element i vsebuje vsoto vseh elementov pred i, vključno z njegovo vrednostjo, kar imenujemo *vključujoča komulativna vsota* (ang. *inclusive scan*) [19].

Velikokrat je zaželeno, da komulativna vsota na mestu i vsebuje samo vsoto vseh elementov pred i. Takšno vsoto imenujemo *izključujoča komulativna vsota* (ang. *exclusive scan*). Poleg tabele in operatorja $\oplus$ na vhodu prejme še identiteto I podanega operatorja. Kot rezultat vrne tabelo:

$$[I,\ a_0,\ (a_0 \oplus a_1),\ ...,\ (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]. \tag{2.4}$$

Iz vključujoče komulativne vsote lahko preprosto izpeljemo izključujočo in obratno. V preostanku poglavja bomo opisovali algoritme izključujoče komulativne vsote, ker se ta v praksi veliko bolj pogosto uporablja. Osredotočili se bomo na komulativne vsote za kratke tabele, ki jih lahko obdela en blok niti. Za daljše tabele smo uporabili knjižnico *CUDPP (CUDA Data-Parallel Primitives Library)* [1, 19].

Za nadaljnjo analizo je potrebno definicija *delovne zahtevnosti* (ang. *work complexity*). To je število operacij oziroma korakov potrebnih za izvedbo algoritma. Pri zaporednih algoritmih sta časovna in delovna zahtevnost vedno enaki. Zaporedna implementacija ima časovno, prostorsko in delovno zahtevnost enako $O(n)$.



Slika 2.3: Prikaz delovno učinkovite vzporedne komulativne vsote [22].

Za izvedbo vzporedne komulativne vsote tabele dolžine n je potrebnih $O(\log n)$ iteracij. V prvi oziroma delovno najbolj zahtevni iteraciji je potrebno izvesti $O(n)$ operacij. S p procesorji lahko to izvedemo vzporedno v času $O(\frac{n}{p})$. Časovna zahtevnost vzporedne komulativne vsote je zato enaka:

$$T_p = O\left(\frac{n}{p} \cdot \log n\right). \tag{2.5}$$

To je tudi časovna zahtevnost vseh algoritmov komulativnih vsot, opisanih v naslednjih poglavjih. Pri dovolj velikem številu procesorjev (tj. $O(n)$) je časovna zahtevnost enaka:

$$T_n = O(\log n). \tag{2.6}$$

2.3.1 Delovno učinkovita vzporedna komulativna vsota

Za delovno učinkovito vzporedno komulativno vsoto mora veljati, da ima enako delovno zahtevnost kot zaporedna implementacija. To pomeni, da mora biti komulativna vsota opravljena z $O(n)$ operacijami $\oplus$. V ta namen uporabimo vzorec *uravnoteženih binarnih dreves*. V dejanskem algoritmu tabele ne pretvorimo v binarno drevo. Uporabimo samo koncept drevesa, na podlagi katerega za vsako nit določimo delo, ki ga mora opraviti. Algoritem je sestavljen iz dveh faz (slika 2.3) in sicer *faza prehoda navzgor* (ang. *up-*

```
1  // vhodnaTab:  vhodna tabela,
2  // izhodnaTab: izhodna tabela.
3  scepecDelovnoUcikovitaKomVsota(vhodnaTab, izhodnaTab):
4    extern __shared__ sTab[]
5    sTab[threadIdx.x] = vhodnaTab[threadIdx.x]
6    sTab[threadIdx.x + blockDim.x] = vhodnaTab[threadIdx.x + blockDim.x]
7
8    // Faza prehoda navzgor
9    for (korak = 1; korak <= blockDim.x; korak *= 2):
10     indeks = (threadIdx.x + 1) * 2 * korak - 1
11     if indeks < 2 * blockDim.x:
12       sTab[indeks] += sTab[indeks - korak]
13     __syncthreads()
14
15   // Faza prehoda navzdol
16   for (korak = blockDim.x / 2; korak > 0; korak /= 2):
17     __syncthreads()
18     indeks = (threadIdx.x + 1) * 2 * korak - 1
19     if (indeks + korak < 2 * BLOCK_SIZE):
20       sTab[indeks + korak] += sTab[indeks]
21
22   __syncthreads()
23   izhodnaTab[threadIdx.x] = sTab[threadIdx.x]
24   izhodnaTab[threadIdx.x + blockDim.x] = sTab[threadIdx.x + blockDim.x]
```

Psevdokoda 2.5: Ščepec delovno učinkovite kom. vsote (prirejeno po [22]).

sweep phase) in *faza prehoda navzdol* (ang. *down-sweep phase*). V prvi fazi algoritma izvedemo prehod iz listov drevesa do korenskega vozlišča in ob tem izračunamo delne vsote (zgornja polovica slike 2.3). V fazi prehoda navzdol izvedemo ravno obraten prehod iz korenskega vozlišča do listov drevesa, pri čemer na podlagi prej dobljenih delnih vsot izračunamo komulativno vsoto (spodnja polovica slike 2.3). Psevdokoda 2.5 prikazuje implementacijo opisanega algoritma. Algoritem opravi $2 \cdot (n-1)$ operacij $\oplus$, torej ima delavno zahtevnost enako $O(n)$ [19, 22].

Čeprav je opisan algoritem delovno učinkovit, ni povsem primeren za arhitekturo CUDA. Algoritem slabo izkorišča vire GPE, saj večino časa deluje le majhen delež razpoložljivih niti. Poleg tega zahteva $2 \cdot \log_2 n$ sinhronizacij niti, kar v veliki meri upočasni izvajanje. Opisane slabosti odpravi algoritem *optimizirane komulativne vsote*, ki je opisan v naslednjem poglavju [33].

2.3.2 Optimizirana komulativna vsota

Kot smo omenili v poglavju 2.1, se koda na GPE izvaja v skupinah 32 niti in jo imenujemo snop. Niti si medsebojno delijo ukazno enoto, zato lahko naen-

```
1  // sTab:     vhodna tabela v deljenem pomnilniku,
2  // vrednost: element tabele, ki ga vsebuje trenutna nit.
3  template <velBloka> komulativnaVsotaSnop(volatile sTab, vrednost):
4    indeks = 2 * threadIdx.x - (threadIdx.x & (warpSize - 1))
5    sTab[indeks] = 0
6    indeks += warpSize
7    sTab[indeks] = vrednost
8
9    if velBloka >= 2: sTab[indeks] += sTab[indeks - 1]
10   if velBloka >= 4: sTab[indeks] += sTab[indeks - 2]
11   if velBloka >= 8: sTab[indeks] += sTab[indeks - 4]
12   if velBloka >= 16: sTab[indeks] += sTab[indeks - 8]
13   if velBloka >= 32: sTab[indeks] += sTab[indeks - 16]
14
15   // Pretvori vključujočo komulativno vsoto v izključujočo
16   return sTab[indeks] - vrednost
```

Psevdokoda 2.6: Komulativna vsota v snopu (prirejeno po [18, 33]).

krat izvajajo samo en ukaz. Algoritem komulativne vsote lahko prilagodimo opisanem vzorcu izvajanja.

V ta namen uporabimo algoritem *komulativne vsote v snopu* (ang. *intra-warp scan*), ki je prikazan v psevdokodi 2.6. Spremenljivko `warpSize` zagotovi arhitektura CUDA. Njena vrednost je število niti v snopu oziroma 32 (vrstici 4 in 6). Prednost implementacije je dejstvo, da se niti v snopu izvajajo sinhrono in pri tem ne potrebujejo sinhronizacije na pregradi. Nadalje vsak snop niti obdela podzaporedje dolžine 32, zato je potrebnih največ pet iteracij za izvedbo komulativne vsote. To pomeni, da lahko uporabimo *razvitje zanke* (ang. *loop unroll*, vrstice 9 - 13). Algoritem zahteva polovico manj korakov kot algoritem *delovno učinkovite komulativne vsote*, opisan v prejšnjem poglavju 2.3.1. Velikost bloka niti podamo s pomočjo konstrukta `template` jezika *C++* (vrstica 3), zaradi česar je ta poznana že v času prevajanja, kar predstavlja dodatno optimizacijo. Za velikost snopa $s = 32$ ima prikazan algoritem delovno zahtevnost $O(s \cdot \log s)$, kar je več od optimalne delovne zahtevnosti komulativne vsote $O(s)$. Na samo učinkovitost algoritma to ne vpliva, ker vse niti v snopu izvajajo isti ukaz. Vsak ukaz v snopu zahteva enak čas za izvedbo, ne glede na to, ali ga izvede ena sama nit ali vse niti snopa [33].

Na podlagi *komulativne vsote v snopu* lahko izračunamo komulativno vsoto celotnega zaporedja. Algoritem prikazuje psevdokoda 2.7. Na začetku

```
1  // vrednost: element tabele, ki ga vsebuje trenutna nit.
2  template <velBloka> optKomulativnaVsota(vrednost):
3    extern __shared__ sTab[]
4    indeksSnop = threadIdx.x / warpSize
5    indeksNitiSnop = threadIdx.x & (warpSize - 1)
6
7    rezultat = komulativnaVsotaSnop<velBloka>(sTab, vrednost)
8    __syncthreads()
9
10   if indeksNitiSnop == warpSize - 1: sTab[indeksSnop] = rezultat + vrednost
11   __syncthreads()
12
13   if indeksSnop == 0:
14     sTab[threadIdx.x] = komulativnaVsotaSnop<velBloka / 32>(
15       sTab, sTab[threadIdx.x]
16     )
17   __syncthreads()
18
19   return rezultat + sTab[indeksSnop]
```

Psevdokoda 2.7: Optimizirana komulativna vsota (prirejeno po [33]).

vsak snop izvede komulativno vsoto svojega pripadajočega odseka tabele, pri čemer vsaka nit shrani svoj rezultat v svoje registre (vrstica 7). Nato vse zadnje niti v snopih shranijo končni rezultat svoje komulativne vsote (vrstica 10). Za tem niti prvega snopa izvedejo komulativno vsoto nad prej shranjenimi vmesnimi rezultati (vrstice 14 - 16). Na koncu vsaka nit sešteje rezultat komulativne vsote svojega pripadajočega snopa in vsoto vseh prejšnjih snopov (vrstica 19) [33].

2.3.3 Komulativna vsota za binarne predikate

Nvidia GPE z računsko zmožnostjo 2.0 ali več vsebujejo dva ukaza, ki v veliki meri pohitrita izračun komulativne vsote nad tabelo predikatov z vrednostjo 0 ali 1. Ukaza sta naslednja:

- `int __popc(int x)` - v 32-bitnem številu x prešteje število bitov z vrednostjo 1,
- `int __ballot(int p)` - vrne 32-bitno število, v katerem ima bit k vrednost 1 samo v primeru, če je podani predikat p zaporedne niti k znotraj istega snopa različen od 0.

```
1 // vrednost: element tabele, ki ga vsebuje trenutna nit.
2 tvoriMasko():
3   asm("mov.u32 %0, %lanemask_lt;" : "=r"(maska))
4   return maska
5
6 komVsotaSnopPredikat(predikat):
7   maska = tvoriMasko()
8   glasovi = __ballot(predikat)
9   return __popc(glasovi & maska)
```

Psevdokoda 2.8: Komulativna vsota v snopu za predikate (prirejeno po [18]).

S pomočjo naštetih ukazov lahko vsaka nit ugotovi koliko niti z nižjim identifikatorjem v njenem snopu vsebuje predikat z vrednostjo 1. Vsaka nit pokliče ukaz `__ballot`, v katerega vstavi svoj predikat. Nato nad dobljenim rezultatom oziroma glasovi niti pokliče ukaz `__popc`, s čimer prešteje število bitov z vrednostjo 1. S tem pravzaprav dosežemo vzporedno redukcijo, kjer vse niti vsebujejo rezultat redukcije. Izračun komulativne vsote predikatov v snopu zahteva majhno spremembo prikazano v psevdokodi 2.8. Vsaka nit s pomočjo ukaza `__ballot` prejme predikatne bite oziroma glasove vseh niti v snopu, medtem ko mora prešteti samo predikate niti z nižjim identifikatorjem. V ta namen moramo za vsako zaporedno nit k v snopu zgraditi masko (vrstice 2 - 4). Ta mora vsebovati vrednost 1 na bitih manjših ali enakih k in na ostalih bitih vrednost 0. Če je zaporedni indeks niti enak $k = 18$ (začenši z 0), potem moramo dobiti masko *11111111 11111111 11000000 00000000*. Masko zgradimo s pomočjo ukaza v zbirniku (vrstica 3), s čimer dosežemo dodatno optimizacijo. Z logično operacijo *IN* med prej omenjenimi glasovi in opisano masko (vrstica 9) omejimo glasove samo na niti, ki imajo nižji identifikator kot zaporedna nit k [18].

Na podlagi opisanega algoritma lahko izračunamo tudi komulativno vsoto nad kosom tabele predikatov, ki pripada trenutnemu bloku niti. Potrebna je le majhna sprememba v psevdokodi 2.7. V vrstici 7 moramo opraviti klic funkcije `komVsotaSnopPredikat` iz psevdokode 2.8, kar je tudi edina potrebna sprememba v algoritmu [18].

Poglavje 3

Bitono urejanje

Bitono urejanje (ang. *bitonic sort*) je algoritem za urejanje podatkov, ki ga je zasnoval Ken E. Batcher [5]. Algoritem spada v skupino t.i. *mrež za urejanje podatkov* (ang. *sorting network*). To so algoritmi za urejanje podatkov, ki izvedejo vnaprej določeno zaporedje primerjav. Vrstni red, smer in število primerjav je neodvisno od vhodnih podatkov. To pomeni, da lahko algoritem predstavimo kot statično mrežo primerjalnikov, povezanih s podatkovnimi linijami. Primerjalnik dobi na vhodu dve števili (oziroma dve podatkovni liniji), kateri uredi v vnaprej določenem vrstnem redu. Mreže za urejanje podatkov so zaradi opisanih razlogov primerne za implementacije na strojni opremi in vzporednih arhitekturah (npr. CUDA) [5, 28].

Osnova za delovanje bitonega urejanja je *bitono zaporedje*. Zaporedje števil imenujemo bitono, če je sestavljeno iz dveh obratno urejenih monotono naraščajočih oziroma padajočih podzaporedij. Med bitona zaporedja štejemo tudi *ciklične zamike* omenjenih zaporedij (definicija 3.1).

Definicija 3.1 *Naj bo* $E = (e_0,\ e_1,\ ...,\ e_{n-1})$ *bitono zaporedje dolžine* n*. Za vsak* $s \in \mathbb{N}$ *velja, da je* $S_s(E)$ *ciklični zamik zaporedja* E *za* s *elementov:*

$$S_s(E) = (e_{((-s \bmod n)+n) \bmod n},\ e_{(((-s+1) \bmod n)+n) \bmod n},\ ...,\ e_{(((-s+(n-1)) \bmod n)+n) \bmod n}). \tag{3.1}$$

Definicija 3.2 *Zaporedje E je bitono, če obstaja tak $s \in \mathbb{N}$, da je zaporedje $S_s(E)$ sestavljeno iz monotono naraščajočega in monotono padajočega podzaporedja.*

Primer bitonega zaporedja je $E = (1, 3, 5, 7, 6, 4, 2, 0)$. Sestavljeno je iz nepadajočega pozdaporedja $(1, 3, 5, 7)$ in nenaraščujočega podzaporedja $(6, 4, 2, 0)$. Primer bitonega zaporedja sestavljenega iz nenaraščujočega in nato nepadajočega zaporedja je $F = (6, 4, 2, 0, 1, 3, 5, 7)$. Za zaporedji E in F pravzaprav velja $E = S_4(F)$ oziroma $F = S_4(E)$. Kot smo omenili, so ciklični zamiki bitonih zaporedij za katerokoli vrednost prav tako bitona zaporedja. Tako je na primer $S_2(E) = (2, 0, 1, 3, 5, 7, 6, 4)$ tudi bitono zaporedje.

3.1 Zaporedni algoritem

3.1.1 Bitono zlivanje

Algoritem urejanja bitonega zaporedja se imenuje *bitono zlivanje* (ang. *bitonic merge*). Za njegovo delovanje na bitonem zaporedju E dolžine n definiramo operacijo *biton korak* [6, 29]:

$$L(E) = (\min(e_0,\ e_{\frac{n}{2}}),\ \min(e_1,\ e_{(\frac{n}{2})+1}),\ ...,\ \min(e_{(\frac{n}{2})-1},\ e_{n-1})), \tag{3.2}$$

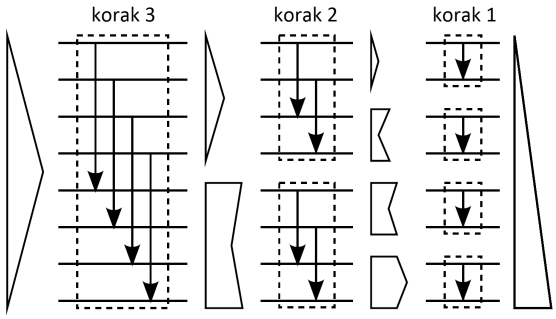
$$U(E) = (\max(e_0,\ e_{\frac{n}{2}}),\ \max(e_1,\ e_{(\frac{n}{2})+1}),\ ...,\ \max(e_{(\frac{n}{2})-1},\ e_{n-1})). \tag{3.3}$$

Izrek 3.1 *Za dobljeni podzaporedji L(E) in U(E) veljata naslednji lastnosti:*

1. *podzaporedji L(E) in U(E) sta bitoni,*
2. *za vsak element $l \in L(E)$ in $u \in U(E)$ velja:*

$$l \leq u. \tag{3.4}$$

Bitono zaporedje dolžine $n = 2^r$ lahko zlijemo v r oziroma $\log_2 n$ korakih. Kot vidimo iz operacij (3.2) in (3.3) je vsakem koraku potrebnih $\frac{n}{2}$ operacij primerjav oziroma zamenjav. Slika 3.1 prikazuje mrežo za bitono zlivanje zaporedja dolžine 8. Vodoravne črte predstavljajo števila v zaporedju, navpične



Slika 3.1: Mreža za bitono zlivanje (prirejeno po [29]).

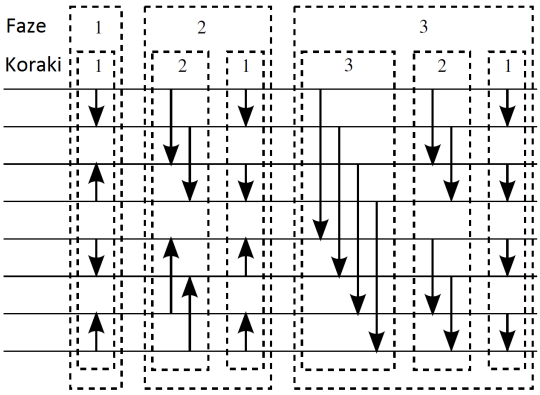
puščice pa primerjave oziroma zamenjave. Na sliki vidimo, da v začetnem koraku $k = 3$ dobimo iz enega bitonega zaporedja dve bitoni podzaporedji (koraki na sliki so indeksirani v obratnem vrstnem redu zaradi konsistentnosti s psevdokodo 3.1, ki prikazuje algoritem bitonega urejanja). V naslednjem koraku $k = 2$ dobimo iz dveh podzaporedij štiri podzaporedja. V zadnjem koraku $k = 1$ dobimo 8 podzaporedij dolžine enega elementa oziroma urejeno zaporedje. Iz opisa algoritma ugotovimo, da v vsakem koraku k dobimo $2^{(r-k+1)}$ podzaporedij dolžine 2^{k-1}. Vidimo tudi, da so v vsakem koraku vsi elementi podzaporedja p manjši od vseh elementov podzaporedja $p + 1$ (izrek 3.1). Sklepamo, da za bitono zlivanje velja naslednje rekurzivno razmerje:

$$\text{bitono\_zlivanje(E)} = (\text{bitono\_zlivanje(L(E))}, \text{bitono\_zlivanje(U(E))}).$$

Pri tem je potrebno omeniti, da je smer primerjav pri bitonem zlivanju vedno enaka, kar ne velja za algoritem bitonega urejanja [28, 29].

3.1.2 Bitono urejanje

Z algoritmom bitonega zlivanja lahko uredimo poljubno bitono zaporedje. Težava v opisanem pristopu je dejstvo, da mora biti vhodno zaporedje bitono. V ta namen je potreben algoritem *bitonega urejanja*. Ta deluje tako, da zaporedje dolžine $n = 2^r$ uredi v $r = \log_2 n$ *fazah*, pri čemer v vsaki fazi izvede algoritem bitonega zlivanja. Slika 3.2 prikazuje mrežo za bitono urejanje



Slika 3.2: Mreža za bitono urejanje (prirejeno po [29]).

zaporedja dolžine 8. Na sliki vidimo, da mrežo sestavljajo $\log_2 8 = 3$ faze oziroma 3 bitona zlivanja [28].

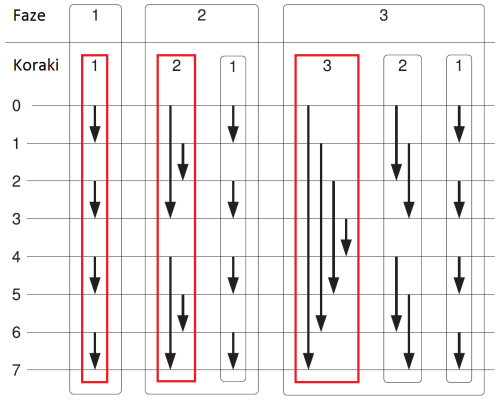
V poljubnem vhodnem zaporedju $E = (e_0,\ e_1,\ ...,\ e_{n-1})$ oziroma v fazi 0 so vsa podzaporedja dolžine 2 pravzaprav bitona zaporedja:

$$(e_0,\ e_1),\ (e_2,\ e_3),\ ...,\ (e_{n-2},\ e_{n-1}).$$

Omenjena podzaporedja lahko uredimo z bitonim zlivanjem (*1. faza* bitonega urejanja). Zagotoviti moramo, da so soda oziroma liha podzaporedja urejena v obratnem vrstnem redu, s čimer dobimo bitona podzaporedja dolžine 4 [28]:

$$(e_0,\ e_1,\ e_2,\ e_3),\ (e_4,\ e_5,\ e_6,\ e_7),\ ...,\ (e_{n-4},\ e_{n-3},\ e_{n-2},\ e_{n-1}).$$

Enak postopek nadaljujemo v *2. fazi*, kjer s pomočjo bitonega zlivanja uredimo prej dobljena bitona podzaporedja dolžine 4. Pri tem zopet upoštevamo, da morajo biti soda oziroma liha podzaporedja urejena v obratnem vrstnem redu. S tem dobimo bitona podzaporedja dolžine 8. Iz opisa algoritma ugotovimo, da v vsaki fazi f dobimo $\frac{n}{2^f}$ bitonih podzaporedij dolžine 2^f. Ugotovimo tudi, da je v fazi f potrebnih f korakov za zlitje vseh $\frac{n}{2^f}$ bitonih podzaporedij. Iz opisanega lahko sklepamo, da je zaporedje dolžine $n = 2^r$ po r - *1* fazah bitono in po r fazah urejeno [28, 29].



Slika 3.3: Mreža za normalizirano bitono urejanje (prirejeno po [28]).

3.1.3 Normalizirano bitono urejanje

Navadno bitono urejanje nam omogoča urejanje zaporedij, katerih dolžina n je potenca števila 2 ($n = 2^r;\ r \in \mathbb{N}_0$). To v praksi predstavlja težavo, saj imajo zaporedja redko omenjeno dolžino. V ta namen lahko razširimo tabelo do naslednjega večkratnika potence števila 2. Razširjen del tabele nato zapolnimo (ang. *padding*) z maksimalnimi oziroma minimalnimi vrednostmi (odvisno od smeri urejanja), vendar s tem dodamo nepotrebno delo algoritmu urejanja. Vzemimo za primer tabelo z dolžino $2^n + 1$. Prej opisani algoritem bo najprej zapolnil tabelo do dolžine 2^{n+1} in jo nato uredil. To pomeni, da bo algoritem uredil $2^n - 1$ odvečnih zapolnjenih elementov.

Da bi se izognili nepotrebnemu podaljšanju časa izvajanja, smo v naši implementaciji uporabili *normalizirano bitono urejanje* (ang. *normalized bitonic sort*). Njegova prednost je v tem, da nam omogoča urejanje zaporedij poljubne dolžine. Poleg tega olajša implementacijo na vzporednih arhitekturah, saj je smer primerjav vedno enaka (pri navadnem bitonem urejanju morajo biti monotona podzaporedja urejena v obratni smeri). Največja razlika med implementacijo navadnega in normaliziranega bitonega urejanja je v spremenjenem prvem koraku zlivanja v vseh fazah urejanja. Kot prikazuje slika 3.3 v rdečih okvirjih, so indeksi primerjav oziroma zamenjav zrcaljeni po sredini podzaporedja [28].

```
1  // tabela: vhodna tabela,
2  // n:      dolžina vhodne tabele,
3  // s:      smer urejanja (0: naraščajoče, 1: padajoče).
4  zaporednoBitonoUrejanje(tabela, n, s):
5    // Izračuna število faz potrebnih za bitono urejanje
6    steviloVsehFaz = log2(nasledjaPotencaStevila2(n))
7
8    // Faze bitonega urejanja
9    for (faza = 1; faza <= steviloVsehFaz; faza++):
10     // Koraki bitonega zlivanja
11     for (korak = faza; korak > 0; korak--):
12       // Velikost bitonih podzaporedij / 2
13       velP = 1 << (korak - 1)
14
15       // Števec primerjav, ki jih je potrebno izvesti v enem koraku zlivanja
16       for (p = 0; p < n / 2; p++):
17         // V prvem koraku vsake faze normaliziranega bitonega urejanja je
18         // potrebno drugačno indeksiranje elementov kot v ostalih korakih
19         if faza == korak:
20           indeksEl = (p / velP) * velP + ((velP - 1) - (p % velP))
21           odmik = ((p & (velP - 1)) << 1) + 1
22         else:
23           indeksEl = p
24           odmik = velP
25
26         // Indeksa elementov, katera bosta medsebojno primerjana
27         indeksLevo = (indeksEl << 1) - (indeksEl & (velP - 1))
28         indeksDesno = indeksLevo + odmik
29         if indeksDesno >= n:
30           break
31
32         // Izvede operacijo primerjave oziroma zamenjave
33         primerjajInZamenjaj(tabela[indeksLevo], tabela[indeksDesno], s)
```

Psevdokoda 3.1: Zaporedno normalizirano bitono urejanje.

3.1.4 Implementacija

Bitono urejanje je rekurziven algoritem. Preizkusili smo rekurzivno in iterativno implementacijo ter se odločili za iterativno, ker se je izkazala za hitrejšo. Psevdokoda 3.1 prikazuje implementacijo iterativnega normaliziranega bitonega urejanja. Iz psevdokode vidimo, da se v vsaki fazi (vrstice 9 - 33) bitonega urejanja izvaja bitono zlivanje (vrstice 11 - 33). Pomembni sta tudi vrstici 20 in 21, v katerih se indeksi primerjanih elementov spremenijo tako, da se zrcalijo po sredini bitonih podzaporedij (rdeči okviri na sliki 3.3). Pogoj v vrstici 29 je potreben v primeru, ko dolžina tabele ni enaka večkratniku potence števila 2. Pri tem velja izpostaviti, da je smer primerjav vedno enaka (spremenljivka `s` v vrstici 33).

3.1.5 Časovna zahtevnost

Za izračun časovne zahtevnosti bitonega urejanja zaporedja dolžine n definiramo funkcijo $C_{korak}(m)$. Ta predstavlja število operacij primerjav oziroma zamenjav, potrebnih za izvedbo enega koraka zlivanja bitonega podzaporedja dolžine m. Število vseh operacij, potrebnih za bitono urejanje zaporedja dolžine n, je zato enako [29]:

$$\sum_{f=1}^{\log n} \sum_{k=1}^{f} \frac{n}{2^k} \cdot C_{korak}(2^k). \tag{3.5}$$

Prva vsota predstavlja število vseh faz bitonega urejanja, druga vsota pa število vseh korakov zlivanja v fazi f. Izraz $\frac{n}{2^k}$ predstavlja število vseh bitonih podzaporedij v koraku k faze f. V vsakem koraku zlivanja bitonega podzaporedja dolžine 2^k dobimo podzaporedji L(E) (3.2) in U(E) (3.3), za kar je potrebnih $C_{korak}(2^k) = \frac{2^k}{2}$ operacij primerjav oziroma zamenjav. Število operacij potrebnih za obdelavo vseh $\frac{n}{2^k}$ bitonih podzaporedij v koraku k je zato enako [29]:

$$\frac{n}{2^k} \cdot C_{korak}(2^k) = \frac{n}{2^k} \cdot \frac{2^k}{2} = \frac{n}{2}. \tag{3.6}$$

Število vseh operacij primerjav in zamenjav oziroma časovna zahtevnost bitonega urejanja je enaka [29]:

$$T_1 = \sum_{f=1}^{\log n} \sum_{k=1}^{f} \frac{n}{2^k} \cdot C_{korak}(2^k) = \sum_{f=1}^{\log n} \sum_{k=1}^{f} \frac{n}{2} = \sum_{f=1}^{\log n} f \cdot \frac{n}{2} = O(n \cdot \log^2 n). \tag{3.7}$$

Časovna zahtevnost je enaka $O(n \cdot \log^2 n)$, kar ni optimalna časovna zahtevnost za urejanje s primerjavami (tj. $O(n \cdot \log n)$). Prednost bitonega urejanja je to, da deluje brez dodatnih pomožnih tabel oziroma *na mestu* (ang. *in-place*). Poleg tega zahteva zelo malo komunikacije med procesorji, kar olajša implementacijo na vzporednih arhitekturah (npr. CUDA) [28].

3.2 Vzporedni algoritem

3.2.1 Trivialna implementacija

Za trivialno implementacijo vzporednega bitonega urejanja na arhitekturi CUDA, bi lahko uporabili preprost ščepec za bitono zlivanje, ki bi izvedel vse operacije primerjav oziroma zamenjav enega koraka bitonega zlivanja. Celotno bitono urejanje bi bilo sestavljeno iz zaporedja klicev omenjenega ščepca, kjer bi vsak klic predstavljal en korak ene faze bitonega urejanja. Vsak ščepec bi vseboval $\frac{n}{2}$ niti (za tabelo dolžine n), pri čemer bi vsaka nit izvedla eno operacijo primerjave oziroma zamenjave. Opisana implementacija zelo dobro izkoristi vire GPE. Vse niti v ščepcu se lahko izvajajo povsem vzporedno, saj ni potrebe po zaporednem izvajanju. Poleg tega so pomnilniški dostopi enakomerno porazdeljeni med nitmi. Posamezne niti so tudi enakomerno delovno obremenjene. Slabost opisanega ščepca je prekomerno dostopanje do globalnega pomnilnika, kar občutno podaljša čas izvajanja algoritma. Vsaka operacija primerjave zahteva dve operaciji branja iz globalnega pomnilnika (vsaka nit prebere in primerja 2 elementa). V primeru, da je potrebna zamenjava elementov, zahteva tudi dve operaciji pisanja v globalni pomnilnik (ob predpostavki, da nit prebrani vrednosti shrani v svoje registre, s pomočjo katerih izvede zamenjavo vrednosti). Urejanje tabele dolžine $n = 2^f$ zahteva f faz, pri čemer vsaka faza vsebuje f korakov bitonega zlivanja. Vsak korak vsebuje $\frac{n}{2} = 2^{f-1}$ operacij primerjav oziroma zamenjav. To pomeni, da trivialna implementacija ob vsakem klicu ščepca zahteva [28]:

$$2 \cdot 2^{f-1} = 2^f = n \tag{3.8}$$

bralnih dostopov globalnega pomnilnika. Skupno število vseh bralnih dostopov potrebnih za celotno urejanje je tako enako:

$$n \cdot \sum_{k=1}^{f} k = n \cdot \frac{f \cdot (1+f)}{2} = n \cdot \frac{(\log n) \cdot (1 + \log n)}{2}. \tag{3.9}$$

```
1  // tabela:   vhodna tabela,
2  // n:        dolžina vhodne tabele,
3  // fazeBU:   število faz izvedenih v ščepcu za bitono urejanje,
4  // korakiLZ: število korakov izvedenih v ščepcu za lokalno zlivanje,
5  // s:        smer urejanja (0: naraščajoče, 1: padajoče).
6  vzporednoBitonoUrejanje(tabela, n, fazeBU, korakiLZ, s):
7    // Izračuna število faz potrebnih za bitono urejanje
8    steviloVsehFaz = log2(nasledjaPotencaStevila2(n))
9
10   // Izvede "fazeBU" faz bitonega urejanja v deljenem pomnilniku
11   scepecBitonoUrejanje(tabela, n, fazeBU, s)
12
13   // Izvedba preostalih faz bitonega urejanja
14   for (faza = fazeBU + 1; faza <= steviloVsehFaz; faza++):
15     // Klici ščepca za globalno bitono zlivanje (poglavje 3.2.1)
16     for (korak = faza; korak > korakiLZ; korak--):
17       scepecGlobalnoBitonoZlivanje(tabela, n, faza, korak, s)
18
19     // Izvede "korakiLZ" korakov zlivanja v deljenem pomnilniku
20     scepecLokalnoBitonoZlivanje(tabela, n, faza, korakiLZ, s)
```

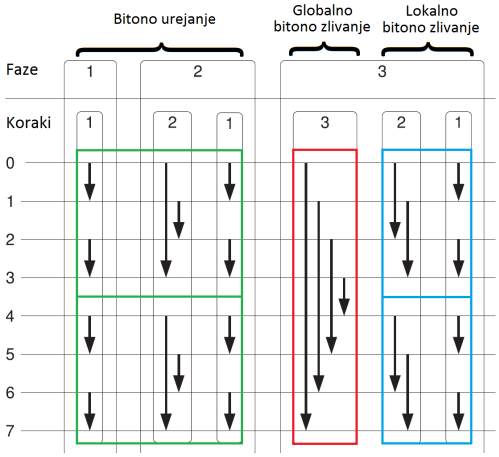
Psevdokoda 3.2: Vzporedno normalizirano bitono urejanje na gostitelju.

Število pisanj je lahko manjše ali enako (ob predpostavki, da vse niti poleg primerjave izvedejo tudi zamenjavo) [28].

3.2.2 Optimizirana implementacija

V nasprotju s trivialno implementacijo je potrebno doseči, da za vsak dostop do globalnega pomnilnika izvedemo več korakov bitonega zlivanja. V ta namen potrebujemo veliko hitrejši deljeni pomnilnik. Ob začetku izvajanja ščepca morajo niti prenesti podatke iz globalnega v deljeni pomnilnik, nad katerim morajo nato izvajati operacije primerjav in zamenjav.

Psevdokoda 3.2 prikazuje optimiziran algoritem vzporednega bitonega urejanja na gostitelju. Kot vidimo iz psevdokode, algoritem na začetku izvede ščepec za bitono urejanje v deljenem pomnilniku (vrstica 11). V ščepcu vsak blok niti izvede `fazeBU` faz bitonega urejanja, s čimer uredi svoj pripadajoč podblok velikosti $2^{\texttt{fazeBU}}$. Slika 3.4 prikazuje primer, v katerem omenjen ščepec izvede dve fazi urejanja. Kot vidimo vsak blok niti (zelena okvirja) uredi svoj pripadajoč podblok dolžine štirih elementov. Po izvedbi ščepca so urejeni podbloki preveliki, da bi jih lahko shranili v deljen pomnilnik, zato jih je potrebno zliti na drugačen način.



Slika 3.4: Klici ščepcev v optimiziranem bitonem urejanju (prirejeno po [28]).

Kot vidimo iz psevdokode 3.2, se po izvedbi začetnega bitonega urejanja izvajajo klici ščepca za globalno bitono zlivanje (vrstici 16 in 17). To je pravzaprav ščepec, ki ga opisuje prejšnje poglavje 3.2.1. Kot smo omenili, ščepec izvaja preveliko število dostopov do globalnega pomnilnika. Iz tega razloga je potrebno zmanjšati število njegovih klicev. Ob podrobnejšemu pregledu slike 3.4 ugotovimo, da se v zadnjih dveh korakih faze *3* (modra okvirja) zlivata povsem neodvisna podbloka velikosti štiri. Omenjeno dejstvo lahko izkoristimo, saj lahko take podbloke zlijemo v deljenem pomnilniku s pomočjo ščepca za lokalno bitono zlivanje. Klice ščepca za globalno bitono zlivanje izvajamo toliko časa, dokler niso neodvisni podbloki dovolj majhni, da jih lahko zlijemo v deljenem pomnilniku s pomočjo lokalnega bitonega zlivanja (vrstica 20). Na sliki 3.4 vidimo, da v ščepcu za lokalno bitono zlivanje vsak blok niti (modra okvirja) izvede 2 koraka zlivanja, s čimer zlije svoj pripadajoč podblok velikosti štiri. Na ta način v veliki meri zmanjšamo število dostopov do globalnega pomnilnika. Prednost opisanega algoritma je tudi v zmanjšanju števila klicev ščepcev, saj vsak klic zahteva zakasnitev.

Implementacija vseh treh ščepcev v psevdokodi 3.2 je zelo podobna zaporedni implementaciji bitonega urejanja (psevdokoda 3.1). Za izvedbo ščepcev bitonega urejanja in lokalnega bitonega zlivanja je potrebno kopirati podatke iz globalnega v deljeni pomnilnik (oziroma obratno po izvedbi ščepcev). V

omenjenih ščepcih je potrebna tudi sinhronizirati niti med vsakim korakom zlivanja, ker si morajo koraki slediti zaporedno. V ščepcu za globalno bitono zlivanje nam ni potrebno izvajati sinhronizacije niti, ker gostitelj izvaja posamične zaporedne klice ščepca za vsak korak zlivanja. Implementacija ščepcev je podobna tudi za urejanje parov ključ-vrednost. Razlika je le, da ob vsaki menjavi ključev zamenjamo tudi vrednosti.

Denimo, da začetno bitono urejanje in lokalno bitono zlivanje izvedeta b faz v deljenem pomnilniku. Pri urejanju tabele dolžine n je število dostopov do globalnega pomnilnika enako (ob upoštevanju (3.8)):

$$n \cdot \left(1 + \sum_{f=b+1}^{\log n} (f - b + 1)\right) = n \cdot \frac{(\log n - b + 1) \cdot (\log n - b + 2)}{2}. \quad (3.10)$$

Število 1 pred vsoto predstavlja klic ščepca za začetno bitono urejanje. Vsota predstavlja faze bitonega zlivanja po izvedbi začetnega urejanja. Iz vsote vidimo, da v vsaki fazi zlivanja izvedemo f - b klicev globalnega in en klic lokalnega zlivanja. Iz opisanega lahko sklepamo, da se ob daljših tabelah algoritem upočasni, kajti število faz lokalnega bitonega zlivanja (v deljenem pomnilniku) je vedno fiksno tekom celotnega algoritma. To pomeni, da je potrebno pri daljših tabelah izvesti več klicev globalnega bitonega zlivanja. S podaljševanjem tabele se čas izvajanja optimizirane implementacije (3.10) približuje k času izvajanja trivialne implementacije (3.9).

3.2.3 Časovna zahtevnost

Kot smo ugotovili iz opisa vzporednega algoritma, lahko niti povsem neodvisno izvajajo primerjave v posameznih korakih bitonega zlivanja. To pomeni, da jih lahko izvedejo vzporedno. Število operacij primerjav oziroma zamenjav v enem koraku bitonega zlivanja je enako $\frac{n}{2}$ (3.6). Število zaporednih primerjav lahko s p procesorji zmanjšamo na:

$$\frac{n}{2 \cdot p}, \quad (3.11)$$

pri čemer je n dolžina tabele. Nasprotno velja za korake, katerih ne moremo izvajati vzporedno. Najprej moramo dokončati vse primerjave oziroma zamenjave v trenutnem koraku, šele nato lahko izvedemo naslednji korak. Število korakov oziroma faz v urejanju tako ostane enako. Časovna zahtevnost vzporednega bitonega urejanja za p procesorjev je enaka (izpeljano iz časovne zahtevnosti zaporedne implementacije (3.7)):

$$T_p = \sum_{f=1}^{\log n} \sum_{k=1}^{f} \frac{n}{2 \cdot p} = \sum_{f=1}^{\log n} f \cdot \frac{n}{2 \cdot p} = O\left(\frac{n}{p} \cdot \log^2 n\right). \tag{3.12}$$

Ob uporabi $O(n)$ procesorjev dosežemo optimalno časovno zahtevnost:

$$T_n = O(log^2 n), \tag{3.13}$$

Izraz 3.13 pravzaprav predstavlja število vseh korakov zlivanja v vseh fazah urejanja. Pohitritev vzporedne implementacije za p procesorjev je enaka:

$$S_p = \frac{T_1}{T_p} = O\left(\frac{n \cdot \log^2 n}{\frac{n}{p} \cdot \log^2 n}\right) = O(p). \tag{3.14}$$

Z vzporedno implementacijo dosežemo pohitritev za faktor p, kar je idealna pohitritev. Pohitritev pri optimalnem številu procesorjev $O(n)$ je enaka:

$$S_n = \frac{T_1}{T_n} = O\left(\frac{n \cdot \log^2 n}{\log^2 n}\right) = O(n). \tag{3.15}$$

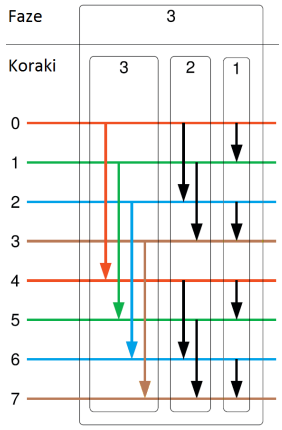
Poglavje 4

Večkoračno bitono urejanje

Optimizirana vzporedna implementacija bitonega urejanja (poglavje 3.2.2) je zelo učinkovita za krajša zaporedja. Težava je v tem, da je deljen pomnilnik majhen oziroma je njegova velikost omejena. Posledično je omejeno tudi število faz algoritma, ki se izvedejo v ščepcu za bitono urejanje in ščepcu za lokalno bitono zlivanje. Zaradi omenjenega razloga je potrebno pri daljših zaporedjih opravljati več klicev globalnega bitonega zlivanja, kar predstavlja ozko grlo algoritma. Za pohitritev algoritma bi bilo potrebno z enim klicem omenjenega ščepca izvesti več korakov zlivanja. S tem bi zmanjšali število klicev ščepca in posledično tudi število dostopov do globalnega pomnilnika. Da bi dosegli želeno, potrebujemo algoritem *večkoračnega bitonega urejanja* (ang. *multistep bitonic sort*) [28].

4.1 Zaporedni algoritem

Večkoračno bitono urejanje predstavlja kodno optimizacijo za arhitekturo CUDA. Z njegovo pomočjo zmanjšamo število klicev ščepca za globalno bitono zlivanje, in s tem posledično število dostopov do globalnega pomnilnika. Zaradi omenjenega razloga ta algoritem ne predstavlja izboljšave za zaporedno urejanje. Implementacija in njegova časovna zahtevnost ostaneta enaki kot v poglavju 3.1.



Slika 4.1: Delitev stopnje 1 v koraku 3 faze 3 (prirejeno po [28]).

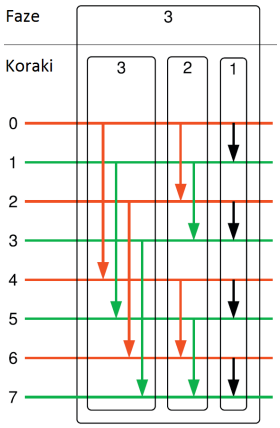
4.2 Vzporedni algoritem

4.2.1 Večkoračno bitono zlivanje

Za izvedbo več korakov zlivanja ob enem dostopu do globalnega pomnilnika moramo operacije primerjav in zamenjav (s tem posledično tudi elemente zaporedja) razdeliti v medsebojno neodvisne podmnožice. Vsaka nit mora izvesti svojo pripadajočo podmnožico primerjav. Zaradi opisane neodvisnosti niti lahko izvedejo primerjave za več korakov zlivanja naenkrat. Pri tem tudi ni potrebe po sinhronizaciji, saj vsaka nit izvaja svojo podmnožico primerjav, neodvisno od ostalih. Za opisan način delovanja morajo niti prebrati svoje pripadajoče vrednosti v registre. S tem dosežemo, da je za vsak klic ščepca potrebno samo eno branje oziroma pisanje v globalni pomnilnik.

Slika 4.1 prikazuje delitev (ang. *partition*) stopnje 1 v koraku 3 faze 3. Vsaka podmnožica operacij primerjav je označena z drugo barvo. Vidimo, da vsaka nit prebere in primerja le 2 elementa, kar je enako kot v trivialni implementaciji ob klicih ščepca za globalno bitono zlivanje (poglavje 3.2.1).

Slika 4.2 prikazuje delitev stopnje 2 v korakih 3 in 2 faze 3. Na sliki vidimo, da je delitev sestavljena iz dveh podmnožic elementov oziroma primerjav, označenih z rdečo in zeleno barvo. Vsaka podmnožica zajema 4



Slika 4.2: Delitev stopnje 2 v korakih 3 in 2 faze 3 (prirejeno po [28]).

elemente, ki jih obdela ena nit. Posamezna nit lahko znotraj istega klica ščepca izvede primerjave za dva koraka, ker so primerjave v koraku 2 odvisne le od primerjav iste podmnožice v koraku 3. Pri tem je potrebno izpostaviti, da lahko delitev stopnje 1 na sliki 4.1 vzporedno obdelajo 4 niti, delitev stopnje 2 na sliki 4.2 pa samo 2 niti [28].

4.2.2 Operacija delitve

Naj bo E zaporedje dolžine $n = 2^r$. Množico vseh indeksov zaporedja E definiramo kot $I = \{0,\ 1,\ ...,\ n-1\}$. Definiramo tudi množico $COEX_{f,k}$, ki vsebuje vse operacije primerjav oziroma zamenjav v koraku k faze f bitonega zlivanja. Upoštevati je potrebno, da je $coex_{f,k}(a, b)$ operacija, kjer sta a in b komutativna. Iz omenjenega sledi, da je množica vseh operacij primerjav v bitonem urejanju zaporedja E je enaka:

$$COEX = \bigcup_{f=1}^{r} \left(\bigcup_{k=1}^{f} COEX_{f,k} \right). \tag{4.1}$$

Definirali bomo tudi operator $K(M)$, ki preslika množico operacij primerjav oziroma zamenjav M v množico indeksov zaporedja, ki jih operacije v množici M primerjajo. Obenem definiramo tudi obraten operator $N_{f,k}$, ki preslika

množico indeksov zaporedja J v množico operacij primerjav koraka k v fazi f. Vse operacije v $COEX_{f,k}$ lahko izvedemo vzporedno, ker je vsak element zaporedja E v koraku k faze f primerjan natanko enkrat. Pri tem moramo pred izvedbo vsake operacije $c \in COEX_{f,k}$ zagotoviti, da sta obe njeni pripadajoči operaciji v predhodnem koraku $N_{f,k+1}(K(\{c\}))$ že opravljeni [28].

Definicija 4.1 *Delitev P indeksov zaporedja I definiramo kot:*
$P = \{I_1, ..., I_m\}$ *je delitev* $I \Leftrightarrow \bigcup_{i=1}^{m} I_i = I \forall i, j \in \{1, ..., m\}, i \neq j : I_i \cap I_j = \emptyset$.

Delitev P vsebuje disjunktne podmnožice indeksov elementov I, kar lahko vidimo z definicije 4.1. To pomeni, da lahko elemente posamezne podmnožice primerjamo neodvisno od elementov v ostalih podmnožicah. Potrebno je izpostaviti, da delitev P zagotavlja enakomerno porazdelitev pomnilniških dostopov med niti pri urejanju zaporedja dolžine n:

$$\forall J \in P : |J| = \frac{n}{|P|}. \tag{4.2}$$

Prav tako zagotavlja enakomerno porazdelitev dela (število primerjav):

$$\forall J \in P : |N_{f,k}(J)| = \frac{1}{2} \cdot |J|. \tag{4.3}$$

Iz enačbe (4.3) ugotovimo, da je velikost podmnožic operacij primerjav v koraku k faze f za polovico manjša kot velikost podmnožic indeksov elementov, potrebnih za primerjave. Zaradi enačb (4.2) in (4.3) velja:

$$\{N_{f,k}(J) \mid J \in P\} \text{ je delitev } COEX_{f,k}. \tag{4.4}$$

Iz enačbe (4.4) lahko sklepamo, da vse primerjave na indeksih $J \in P$ ne vplivajo na elemente, ki niso v J. V nasprotnem primeru bi bilo število operacij primerjav v delitvi premajhno [28].

Pomemben konstrukt predstavlja tudi stopnja d delitve P v koraku k faze f. Definiramo jo kot maksimalno število zaporednih korakov, pri katerih enačbe (4.2), (4.3) in (4.4) še veljajo:

$$\forall a \in \{1, 2, ..., d\} : \{N_{f,k-a+1}(J) \mid J \in P\} \textit{ je delitev } COEX_{f,k-a+1}. \quad (4.5)$$

Splošna definicija delitve P stopnje d v koraku k faze f je tako enaka:

$$P^d_{f,k} := \bigcup_{i \in I} K(N_{f,k}(K(N_{f,k-1}(...K(N_{f,k-(d-1)}(\{i\}))...)))) \ ; \ d \leq k. \quad (4.6)$$

Na sliki 4.2 smo v koraku 3 uporabili delitev $\{\{0, 2, 4, 6\}, \{1, 3, 5, 7\}\}$. Ta nam omogoča izvedbo operacij v korakih 3 in 2 faze 3, zato je njena stopnja enaka 2. Ugotovimo, da niti s pomočjo delitve P stopnje d, preberejo $\frac{n}{|P|}$ elementov iz globalnega pomnilnika v svoje registre. Nato nad prebranimi elementi izvajajo operacije primerjav za d zaporednih korakov. Zaradi medsebojne neodvisnosti med elementi delitve P ni potrebe po medsebojni sinhronizaciji niti. Z opisanim pristopom zmanjšamo število dostopov do globalnega pomnilnika in število klicev ščepcev za faktor d [28].

4.2.3 Izgradnja delitve

Pred podrobnejšim prikazom izgradnje delitve definiramo dva konstrukta:

- **d-koračni ščepec** (ang. *d-multistep*) - ščepec, ki z enim klicem obdela vse operacije primerjav in zamenjav delitve stopnje d,
- **d-koračno delo** (ang. *d-job*) - predstavlja en element (indekse in pripadajoče operacije) delitve stopnje d oziroma delo, ki ga opravi ena nit v *d-koračnem ščepcu*.

Na sliki 4.1 vidimo, da vsaka nit z enokoračnim delom obdela dva elementa. Podoben vzorec vidimo tudi na sliki 4.2, kjer niti z dvokoračnim delom obdelajo štiri elemente. Ugotovimo, da je število elementov v d-koračnem delu enako 2^d, pri čemer je število potrebnih primerjav v vsakem koraku enako

2^{d-1}. Na podlagi ugotovljenega sklepamo, da zaporedje dolžine 2^r vsebuje 2^{r-d} d-koračnih del [28].

Za izgradnjo d-koračnega dela smo izračunali funkcijo, ki nam za *e-to* zaporedno delo v koraku k vrne indeks prvega elementa znotraj dela:

$$id(e,k,d) = \left(\left\lfloor \frac{e}{2^{k-d}} \right\rfloor \cdot 2^k\right) + e \mod 2^{k-d};\ d \leq k. \tag{4.7}$$

Na podlagi funkcije (4.7) smo definirali *e-to* d-koračno delo:

$$delo(e,k,d) = \bigcup_{i=0}^{2^{d-1}-1} \{id(e,k,d) + i \cdot 2^{k-d}, id(e,k,d) + i \cdot 2^{k-d} + 2^{k-1}\}. \tag{4.8}$$

Celotno delitev stopnje d oziroma celotno delo v d-koračnem ščepcu na zaporedju dolžine 2^r v koraku k smo definirali kot:

$$delitev(r,k,d) = \{delo(e,k,d) \mid e \in [0,1,...,2^{r-d}-1]\}. \tag{4.9}$$

Ugotovimo, da definiciji d-koračnega dela in ščepca nista odvisni od trenutne faze bitonega urejanja. Ob gradnji delitve moramo prav tako upoštevati, da stopnja delitve ni večja od trenutnega koraka bitonega zlivanja [28].

Za primer izračunajmo delitev stopnje $d = 2$ v koraku $k = 3$ za zaporedje dolžine $2^3 = 8$ ($r = 3$), ki je prikazana na sliki 4.2. Kot vidimo dobimo delitev sestavljeno iz dveh dvokoračnih del {0, 2, 4, 6} in {1, 3, 5, 7}:

$$\begin{aligned}
&delitev(r=3,k=3,d=2)\\
&= \{delo(e,k=3,d=2) \mid e \in [0,1]\}\\
&= \left\{\bigcup_{i=0}^{1}\{id(e,k=3,d=2) + i \cdot 2, id(e,k=3,d=2) + i \cdot 2 + 2^2\} \mid e \in [0,1]\right\}\\
&= \{\{0,0+4\} \cup \{0+2,0+6\}, \{1,1+4\} \cup \{1+2,1+6\}\}\\
&= \{\{0,2,4,6\},\{1,3,5,7\}\}
\end{aligned}$$

4.2.4 Implementacija na gostitelju

Psevdokoda 4.1 prikazuje algoritem večkoračnega bitonega urejanja stopnje 5 (petkoračno bitono urejanje) na gostitelju. Kot vidimo, je implementacija podobna optimiziranemu bitonemu urejanju (psevdokoda 3.2). Najprej izvedemo klic ščepca za bitono urejanje (vrstica 11). Nato v prvem koraku vsake faze izvedemo klic ščepca za globalno bitono zlivanje (vrstica 16, opisano v poglavju 3.2.1). To storimo zaradi normaliziranega bitonega urejanja (poglavje 3.1.3), ki v prvem koraku vsake faze uporablja drugačen vzorec dostopanja do elementov zaporedja kot v ostalih korakih. Za tem pričnemo izvajati klice večkoračnih ščepcev (vrstice 20 - 29), pri čemer vedno poskušamo klicati ščepec najvišje možne stopnje. Na koncu vsake faze zlivanja zopet izvedemo klic ščepca za lokalno bitono zlivanje (vrstica 32) [28].

Maksimalna stopnja delitve predstavlja zelo pomemben parameter algoritma (v psevdokodi 4.1 je ta enaka 5). Večja kot je stopnja delitve, več korakov zlivanja lahko izvede večkoračni ščepec, s čimer zmanjšamo število klicev ščepcev in posledično število dostopov do globalnega pomnilnika. Ob tem je potrebno upoštevati, da stopnja delitve ne sme biti prevelika, kajti število registrov na nit je omejeno (odvisno od strojne opreme GPE). S tem je omejeno tudi število elementov, ki jih lahko nit shrani. S povečevanjem stopnje delitve povečamo tudi delovno obremenitev niti. S tem zmanjšamo velikost bloka niti v ščepcu, kar posledično pomeni zmanjšanje vzporednosti. V naši implementaciji je bila maksimalna stopnja delitve enaka 5 [28].

Vzemimo za primer, da začetno bitono urejanje in lokalno bitono zlivanje izvedeta b faz v deljenem pomnilniku. Ob upoštevanju (3.8), je pri urejanju tabele dolžine n z večkoračnim algoritmom stopnje d število dostopov do globalnega pomnilnika enako:

$$n \cdot \left(1 + \sum_{f=b+1}^{\log n} \left(\left\lceil \frac{f-b}{d} \right\rceil + 1\right)\right). \tag{4.10}$$

Ugotovimo, da je število dostopov v primerjavi z optimizirano implementacijo (3.10) približno za faktor d manjše. Večji kot je d, manj dostopov do

```
1  // tabela:   vhodna tabela,
2  // n:        dolžina vhodne tabele,
3  // fazeBU:   število faz izvedenih v ščepcu za bitono urejanje,
4  // korakiLZ: število korakov izvedenih v ščepcu za lokalno zlivanje,
5  // s:        smer urejanja (0: naraščajoče, 1: padajoče).
6  vzporednoVeckoracnoUrejanje(tabela, n, fazeBU, korakiLZ, s):
7    // Izračuna število faz potrebnih za bitono urejanje
8    steviloVsehFaz = log2(nasledjaPotencaStevila2(n))
9
10   // Izvede "fazeBU" faz bitonega urejanja v deljenem pomnilniku
11   scepecBitonoUrejanje(tabela, n, fazeBU, s)
12
13   // Izvedba preostalih faz bitonega urejanja
14   for (faza = fazeBU + 1; faza < steviloVsehFaz; faza++):
15     // Zaradi prve faze normaliziranega bitonega urejanja
16     scepecGlobalnoBitonoZlivanje(tabela, n, faza, s)
17     korak = faza - 1
18
19     // Večkoračni ščepci do stopnje 5
20     for (; korak >= korakiLZ + 5; korak -= 5):
21       petkoracniScepec(tabela, n, korak, s)
22     for (; korak >= korakiLZ + 4; korak -= 4):
23       stirikoracniScepec(tabela, n, korak, s)
24     for (; korak >= korakiLZ + 3; korak -= 3):
25       trikoracniScepec(tabela, n, korak, s)
26     for (; korak >= korakiLZ + 2; korak -= 2):
27       dvokoracniScepec(tabela, n, korak, s)
28     for (; korak >= korakiLZ + 1; korak -= 1):
29       enokoracniScepec(tabela, n, korak, s)
30
31     // Izvede "korakiLZ" korakov zlivanja v deljenem pomnilniku
32     scepecLokalnoBitonoZlivanje(tabela, n, faza, korakiLZ, s)
```

Psevdokoda 4.1: Vzporedno večkoračno bitono urejanje na gostitelju (prirejeno po [28]).

globalnega pomnilnika je potrebnih. Podobno kot pri optimizirani implementaciji, je število faz v deljenem pomnilniku b fiksno med izvajanjem celotnega algoritma. To pomeni, da je pri daljših tabelah potrebno opraviti več klicev večkoračnih ščepcev. Posledično lahko algoritem postane počasnejši pri zelo velikih tabelah [28].

4.2.5 Implementacija na napravi

Opisali bomo samo ščepce za večkoračno zlivanje, saj smo delovanje ostalih ščepcev že opisali v poglavju o optimiziranem bitonem urejanje 3.2.2. Na podlagi funkcije (4.9) smo izdelali n-koračne ščepce. Psevdokoda 4.2 prikazuje primer dvokoračnega ščepca (ščepci za ostale stopnje delitve so im-

```
1  // tabela: vhodna tabela,
2  // n:      dolžina vhodne tabele,
3  // s:      smer urejanja (0: naraščajoče, 1: padajoče).
4  dvokoračniScepec(tabela, n, korak, s):
5    // Spremenljivke za elemente, katere bo ščepec urejal
6    el1, el2, el3, el4
7    stopnjaDelitve = 2
8
9    // Globalni indeks niti
10   indeksNiti = blockIdx.x * blockDim.x + threadIdx.x
11   // Indeks prvega elementa, ki ga obdela nit. Izračunamo po (4.7).
12   zacetniIndeks = izracunajZacetniIndeks(indeksNiti, korak, stopnjaDelitve)
13   // Število niti, ki obdelajo en podblok
14   stNitiNaPodblok = 1 << (korak - stopnjaDelitve)
15   // Dolžina enega bitonega podbloka znotraj zaporedja
16   dolzinaPodbloka = 1 << (korak - 1)
17
18   preberi4(
19     tabela + zacetniIndeks, tabela + n, stNitiNaPodblok, dolzinaPodbloka, s,
20     el1, el2, el3, el4
21   )
22   primerjajInZamenjaj4(el1, el2, el3, el4)
23   shrani4(
24     tabela + zacetniIndeks, tabela + n, stNitiNaPodblok, dolzinaPodbloka,
25     el1, el2, el3, el4
26   )
```

Psevdokoda 4.2: Dvokoračni ščepec.

plementirana na podoben način). Kot vidimo najprej na začetku definiramo spremenljivke za vse elemente, katere bo ščepec urejal (vrstica 6). S tem zagotovimo, da bodo prebrani elementi shranjeni v registrih niti in ne v globalnem pomnilniku. Nato izračunamo potrebne parametre za branje oziroma shranjevanje elementov (vrstice 10 - 16). Elemente preberemo v prej definirane spremenljivke, jih uredimo in shranimo nazaj v globalni pomnilnik. Na podoben način uredimo tudi pare ključ-vrednost. Potrebno je definirati dvakrat več spremenljivk, kamor poleg ključev shranimo tudi pripadajoče vrednosti. Nato ob vsaki menjavi ključev zamenjamo tudi vrednosti.

Zaradi uporabe spremenljivk za shranjevanje elementov morajo biti funkcije za *branje*, *primerjanje* in *shranjevanje* definirane za različno število elementov. Naštete funkcije smo definirali za različne potence števila 2. S tem smo dosegli, da smo lahko funkcijo, definirano za 2^n elementov, uporabili pri definiciji funkcije za 2^{n+1} elementov. Psevdokoda 4.3 prikazuje funkciji za branje dveh in štirih elementov (funkcije za shranjevanje smo implementirali na zelo podoben način). Kot vidimo, smo v funkciji za branje štirih ele-

```
1  // tab:             vhodna tabela,
2  // konecTab:        kazalec na konec tabele,
3  // dolzinaPodbloka: dolžina enega bitonega podbloka znotraj zaporedja,
4  // elN:             spremenljivka, kamor se shrani N-ti element,
5  // s:               smer urejanja (0: naraščajoče, 1: padajoče).
6  preberi2(tab, konecTab, dolzinaPodbloka, el1, el2, s):
7    if (tab < konecTab):
8      el1 = tab[0]
9    else:
10     el1 = s == NARASCAJOCE ? MAKSIMUM : MINIMUM
11
12   if (tab + dolzinaPodbloka < konecTab):
13     el2 = tab[dolzinaPodbloka]
14   else:
15     el2 = s == NARASCAJOCE ? MAKSIMUM : MINIMUM
16
17 // Enaki parametri kot v funkciji preberi2,
18 // stNitiNaPodblok: število niti, ki obdelajo en podblok.
19 preberi4(
20   tab, konecTab, stNitiNaPodblok, dolzinaPodbloka, el1, el2, el3, el4, s
21 ):
22   preberi2(tab, konecTab, dolzinaPodbloka, el1, el2)
23   // Za branje 8 elementov "2 * stNitiNaPodblok", za branje 16 elementov
24   // "4 * stNitiNaPodblok", itd.
25   preberi2(
26     tab + 1 * stNitiNaPodblok, konecTab, dolzinaPodbloka, el3, el4
27   )
```

Psevdokoda 4.3: Branje dveh in štirih elementov za večkoračno urejanje.

mentov izkoristili implementacijo funkcije za branje dveh elementov (vrstice 22 - 27). Funkcije za 8, 16 in več elementov smo implementirali po enakem rekurzivnem postopku. V funkciji za branje dveh elementov smo tudi poskrbeli za primere, ko dolžina tabele ni potenca števila 2. V takšnih primerih bodo nekatere niti prebrale elemente, ki presegajo dolžino tabele. Takrat ne opravimo branja, ampak v spremenljivko shranimo maksimalno oziroma minimalno vrednost. Vrednost določimo glede na smer urejanja. Podobno storimo tudi v funkcijah shranjevanja, v katerih ne shranimo opisanih elementov. To je tudi edina sprememba, povezana z večkoračnimi ščepci, ki je potrebna za urejanje tabel, katerih dolžina ni večkratnik potence števila 2.

Izpostaviti velja, da smo implementirali več različic prej opisanega algoritma. Elemente smo shranili v deljen pomnilnik namesto v registre in jih nato urediti, kjer je vsaka nit urejala 2^d elementov (d - stopnja delitve). Poleg tega smo urejanje v deljenem pomnilniku prilagodili tako, da je vsaka nit urejala samo dva elementa. Obe opisani različici sta se izkazali za počasnejši.

Poskušali smo tudi z definicijo lokalne tabele, kjer smo za vsako nit uporabili drugo instanco tabele. Težava pri opisanem pristopu je odvisnost od prevajalnika. Ta na podlagi različnih parametrov določi ali se bo definirana tabela nahajala v registrih niti ali v globalnem pomnilniku [27]. Zaradi omenjenega razloga ne moremo zagotoviti, da se bo tabela nahajala v registrih. Opisana implementacija z lokalno tabelo je prav tako delovala počasneje. Pri tem je potrebno upoštevati, da je hitrost izvedbe opisanih različic urejanja odvisna od strojne opreme GPE, in sicer od velikosti deljenega pomnilnika, števila registrov na nit, hitrosti globalnega pomnilnika, itd.

4.2.6 Časovna zahtevnost

Večkoračno bitono urejanje predstavlja kodno optimizacijo za arhitekturo CUDA. Sam algoritem bitonega urejanja ostane enak. Posledično ostanejo enake tudi časovne zahtevnosti in pohitritve navedene v poglavju 3.2.3.

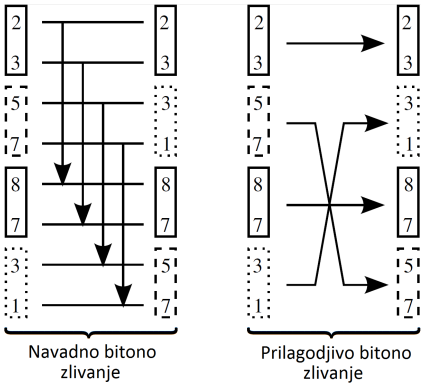
Poglavje 5

Prilagodljivo bitono urejanje

Navadno bitono zlivanje je sestavljeno iz $\log n$ korakov (poglavje 3.1.1). Iz enačbe (3.6) vidimo, da je v vsakem koraku potrebnih $\frac{n}{2}$ operacij primerjav oziroma zamenjav. Ugotovimo, da je časovna zahtevnost algoritma enaka $O(n \cdot \log n)$. Bitono zlivanje ni časovno optimalno, saj je optimalna časovna zahtevnost zlivanja dveh zaporedij enaka $O(n)$. Dosegli bi jo, če bi zmanjšali časovno zahtevnost enega koraka zlivanja bitonega podzaporedja dolžine m ($m \leq n$) iz $O(m)$ na $O(\log m)$. V ta namen potrebujemo algoritem *prilagodljivega bitonega urejanja* (ang. *adaptive bitonic sort*), ki lahko doseže omenjene lastnosti na podlagi dveh principov:

- obstaja podmnožica $2n$ primerjav bitonega zlivanja, na podlagi katerih lahko določimo rezultat vseh ostalih primerjav,
- ponavljajoč vzorec primerjav in zamenjav bitonega zlivanja lahko izkoristimo z uporabo *bitonega drevesa* (ang. *bitonic tree*), s pomočjo katerega nadomestimo veliko število zamenjav elementov v tabeli z majhnim številom zamenjav poddreves.

Na podlagi naštetih principov ugotovimo, da operacije primerjav in zamenjav niso vnaprej določene, ampak so odvisne od porazdelitve vhodnega zaporedja. To pomeni, da prilagodljivo bitono urejanje ne spada v skupino mrež za urejanje podatkov [6, 29].



Slika 5.1: Razlika med navadnim in prilagodljivim bitonim zlivanjem [29].

5.1 Zaporedni algoritem

5.1.1 Prilagodljivo bitono zlivanje

V poglavju 3.1.1 smo omenili, da bitono zlivanje zaporedja E temelji na določitvi podzaporedij *L(E)* (3.2) in *U(E)* (3.3), za kateri velja izrek 3.1. Definirali smo tudi operacijo cikličnega zamika (3.1). Omenjene lastnosti in operacije predstavljajo osnovo za prilagodljivo bitono urejanje. Za bitono zaporedje E dolžine $n = 2^r,\ r \in \mathbb{N}$ veljata naslednja izreka [6].

Izrek 5.1 *Obstaja tako število $q \in \mathbb{Z}$, da za ciklični zamik bitonega zaporedja $S_q(E) = (s_0,\ s_1,\ ...,\ s_{n-1})$ velja:*

$$L(E) = S_{((-q \bmod \frac{n}{2})+\frac{n}{2}) \bmod \frac{n}{2}}(s_0,\ s_1,\ ...,\ s_{\frac{n}{2}-1}), \tag{5.1}$$

$$U(E) = S_{((-q \bmod \frac{n}{2})+\frac{n}{2}) \bmod \frac{n}{2}}(s_{\frac{n}{2}},\ s_{\frac{n}{2}+1},\ ...,\ s_{n-1}). \tag{5.2}$$

Izrek 5.2 *Naj bo q tako število $q \in \mathbb{Z}$, da veljata enakosti* (5.1) *in* (5.2) *ter $t = q \bmod \frac{n}{2}$. Poleg tega naj velja $E = (E_0,\ E_1,\ E_2,\ E_3)$, pri čemer je dolžina E_0 in E_2 enaka t ter dolžina E_1 in E_3 enaka $\frac{n}{2} - t$. Takrat velja:*

$$(L(E),\ U(E)) = (E_0,\ E_3,\ E_2,\ E_1);\ q < \frac{n}{2}, \tag{5.3}$$

$$(L(E),\ U(E)) = (E_2,\ E_1,\ E_0,\ E_3);\ q \geq \frac{n}{2}. \tag{5.4}$$

Iz izreka 5.1 vidimo, da lahko na podlagi izračunane vrednost q določimo *L(E)* in *U(E)* (opis algoritma za iskanje q sledi v poglavju 5.1.2). To izvedemo z dvema cikličnima zamikoma za q in -q oziroma z menjavo podzaporedij E_1 in E_3 (5.3) ali E_0 in E_2 (5.4) (odvisno od vrednosti q). Po izračunu vrednosti q ni več potrebno izvajati nobenih primerjav med elementi. Dokaz za izreka 5.1 in 5.2 se nahaja v [6]. Pri tem je potrebno upoštevati, da smo uporabili definicijo za ciklični zamik iz [29], ki je drugačna kot v [6]. To pomeni, da sta pogoja v enačbah (5.3) in (5.4) ravno obratna kot v [6]. Za omenjeno definicijo cikličnega zamika smo se odločili, ker vse nadaljnje enačbe temeljijo na njej.

Na sliki 5.1 vidimo primer razlike med navadnim in prilagodljivim bitonim zlivanjem. Kot vidimo, navadno bitono zlivanje izvede primerjave med vsemi elementi, pri čemer zamenja samo podbloka *5, 7* in *3, 1*. Nasprotno velja za prilagodljivo bitono zlivanje, ki najprej poišče vrednost $q = 2$. Nato na podlagi dobljene vrednosti ugotovi, da mora zamenjati podbloka E_1 in E_3 oziroma *5, 7* in *3, 1* [29].

5.1.2 Algoritem iskanja vrednosti q

Omenili smo, da je časovna zahtevnost koraka prilagodljivega bitonega zlivanja enega bitonega podzaporedja dolžine m enaka $C_{korak} = O(\log m)$. To implicitno določa, da mora algoritem najti vrednost q z enako ali manjšo časovno zahtevnostjo. V ta namen podrobneje preučimo bitono zaporedje enoličnih elementov E dolžine $n = 2^r$, za katerega zaradi (3.4) velja:

- če je $e_{\frac{n}{2}-1} < e_{n-1}$, potem velja (5.4) (zamenjamo E_0 in E_2),
- če je $e_{\frac{n}{2}-1} > e_{n-1}$, potem velja (5.3) (zamenjamo E_1 in E_3).

Bitono zlivanje v vsakem koraku izvaja primerjave elementov e_i in $e_{\frac{n}{2}+i}$ bitonega zaporedja (ob pogoju $0 \leq i < \frac{n}{2}$). Analizirajmo primer, ko drži postopek (5.4) (za (5.3) veljata ravno obratna pogoja). V tem primeru E_0 vsebuje elemente, za katere velja $e_i > e_{\frac{n}{2}+i}$. Nasprotno E_1 vsebuje elemente, za katere velja $e_i < e_{\frac{n}{2}+i}$. Z E_0 in E_1 sta implicitno določena tudi E_2 in

```
1  // tabela: tabela, ki vsebuje bitono zaporedje z ENOLIČNIMI elementi,
2  // n:      dolžina bitonega zaporedja.
3  najdiQ(tabela, n):
4    // Določi, katera podbloka bosta zamenjana
5    menjava = tabela[n / 2 - 1] < tabela[n - 1] ? E0_E2 : E1_E3
6    q = n / 4 - 1
7
8    for(inkrement = n / 8; inkrement > 0; inkrement /= 2):
9      if ((menjava == E0_E2) XOR (tabela[i] < tabela[n / 2 + i])):
10       q += inkrement
11     else:
12       q -= inkrement
13
14   return q
```

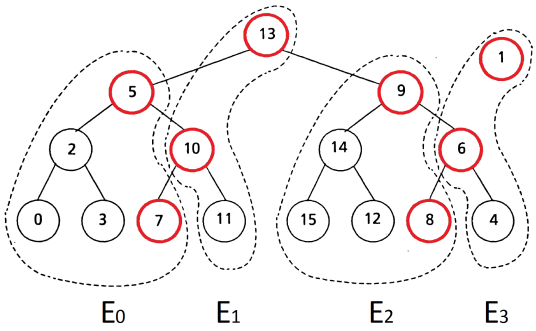
Psevdokoda 5.1: Algoritem za iskanje q (prirejeno po [6]).

E_3, ki se nahajata $\frac{n}{2}$ elementov desno. Sklepamo, da vrednost q pravzaprav predstavlja indeks meje med E_0 in E_1. Iz opisanega ugotovimo, da lahko q najdemo s preprostim binarnim iskanjem, ki ga prikazuje psevdokoda 5.1. Kot vidimo se algoritem zaključi v $\log n - 1$ korakih. Pri tem mora veljati, da so elementi zaporedja enolični. V primeru ponavljajočih se elementov so potrebne majhne spremembe v funkciji primerjanja. Kadar sta elementa enaka, lahko primerjamo (na primer) njuni začetni poziciji v zaporedju [6].

Ugotovili smo, da lahko najdemo vrednost q v času $O(\log n)$. V naslednjem koraku je potrebno zamenjati elemente E_0 in E_2 oziroma E_1 in E_3. Elementov ne moremo kopirati, saj bi s tem dobili časovno zahtevnost $O(n)$. V ta namen potrebujemo podatkovno strukturo, ki nam omogoča menjave podblokov v času $O(\log n)$ [6, 29].

5.1.3 Podatkovna struktura bitono drevo

Slika 5.2 prikazuje primer bitonega drevesa. Gre za binarno drevo, v katerem so elementi v vozliščih razporejeni v enakem vrstnem redu kot v zaporedju. Vsako vozlišče ima vrednost ter kazalca na levo in desno podvozlišče. Prvih $n - 1$ elementov je shranjenih v drevesu višine $\log n$. Zadnji element (na sliki je to 1) je shranjen v rezervnem vozlišču. Pozdaporedja E_0, E_1, E_2, E_3 so prikazana s poddrevesi na sliki 5.2. Zamenjave podzaporedij, opisane v izreku 5.2, lahko dosežemo z $C_{korak} = O(\log n)$ menjavami vrednosti in kazalcev v bitonem drevesu [6].



Slika 5.2: Bitono drevo za bitono zaporedje E = (0, 2, 3, 5, 7, 10, 11, 13, 15, 14, 12, 9, 8, 6, 4, 1). Prikazana so tudi podzaporedja uporabljena za določitev L(E) in U(E). Vozlišča, označena z rdečo, so medsebojno primerjana med izvedbo binarnega iskanja (prirejeno po [6]).

5.1.4 Zaporedna implementacija

Algoritem prilagodljivega bitonega zlivanja deluje zelo podobno kot iskanje vrednosti q v psevdokodi 5.1. Glavna razlika je v tem, da deluje na strukturi bitonega drevesa. Koraki premikanja po drevesu so povsem enaki korakom premikanja po tabeli ob iskanju q, kjer se hkrati izvajajo še menjave vrednosti vozlišč in poddreves.

Psevdokoda 5.2 prikazuje algoritem prilagodljivega bitonega zlivanja. Korensko vozlišče na začetku vsebuje element $\frac{n}{2}-1$, medtem ko rezervno vozlišče vsebuje element $n-1$. Pred začetkom iskanja v drevesu inicializiramo kazalca na elementa $\frac{n}{4}-1$ (enako kot pri iskanju q) in $\frac{3n}{4}-1$ (vrstica 10). Nato poiščemo mejo med E_0 in E_1 ter sočasno izvajamo zamenjave (vrstice 23 - 25). Po izteku zanke dobimo v drevesu podzaporedji *L(E)* in *U(E)*, ki ju rekurzivno uredimo (vrstice 31 - 33). Izpostaviti velja, da mora bitono drevo vsebovati enolične elemente. Poleg tega mora biti dolžina tabele enaka večkratniku potence števila 2. V primeru ponavljajočih se elementov lahko enakosti razrešimo s primerjanjem začetne pozicije elementov, ki jih shranimo v vozlišče poleg vrednosti elementov. Zaradi omenjenega postopka urejanje parov ključ-vrednost ne zahteva nobenih sprememb v kodi [6].

Algoritem prilagodljivega bitonega urejanja je praktično enak algoritmu

```
1  // koren:   korensko vozlišče bitonega drevesa z enoličnimi elementi,
2  // rezerva: rezervno vozlišče, katero ni del bitonega drevesa,
3  // s:       smer urejanja (0: naraščajoče, 1: padajoče).
4  prilagodljivoBitonoZlivanje(koren, rezerva, s):
5    // Določi, katera izmed podblokov E0, E1, E2 in E3 morata biti zamenjana
6    menjava = (koren.v > rezerva.v) XOR (s == NARASCAJOCE) ? E0_E2 : E1_E3
7    if (menjava == E1_E3):
8      zamenjajVrednost(koren, rezerva)
9
10   levoVoz, desnoVoz = koren.levo, koren.desno
11   while (levoVoz != NULL):
12     zamenjajElementa = (levoVoz.v > desnoVoz.v) XOR (s == NARASCAJOCE)
13     // Določi, v katero smer se bo binarno iskanje nadaljevalo
14     pot = (menjava == E0_E2) XOR zamenjajElementa ? LEVO : DESNO
15
16     // Določi funkcijo za menjavo ustreznega poddrevesa
17     zamenjajKazalec = (
18       pot == LEVO ? zamenjajKazalecDesno : zamenjajKazalecLevo
19     )
20     // Določi ustrezno funkcijo za nadaljnje iskanje
21     nadaljujPot = pot == LEVO ? nadaljujPotLevo : nadaljujPotDesno
22
23     if (zamenjajElementa):
24       zamenjajVrednost(levoVoz, desnoVoz)
25       zamenjajKazalec(levoVoz, desnoVoz)
26
27     nadaljujPot(levoVoz)
28     nadaljujPot(desnoVoz)
29
30   // Rekurzivno uredi podzaporedji L(E) in U(E)
31   if (koren.levo != NULL):
32     prilagodljivoBitonoZlivanje(koren.levo, koren, s)
33     prilagodljivoBitonoZlivanje(koren.desno, rezerva, s)
```

Psevdokoda 5.2: Prilagodljivo bitono zlivanje (prirejeno po [6]).

navadnega bitonega urejanja. Razlika je v tem, da deluje na bitonem drevesu in uporablja prilagodljivo bitono zlivanje. Prikazan je v psevdokodi 5.3. Kot vidimo, algoritem rekurzivno izvaja klice funkcije za urejanje, dokler ne pride do listov drevesa. Takrat izvede menjave vrednosti. Nato rekurzivno kliče funkcijo bitonega zlivanja na vedno večjih poddrevesih, dokler na koncu ne pokliče zlivanja nad celotnim drevesom. Pri tem je potrebno upoštevati, da moramo pred začetkom urejanja zgraditi bitono drevo in ga ob koncu urejanja spremeniti nazaj v tabelo [6].

Kadar dolžina tabele ni enaka potenci števila 2, je potrebno zgraditi obrezano (ang. *pruned*) bitono drevo. To so drevesa, v katerih so poddrevesa brez elementov, predstavljena z enim samim *navideznim* (ang. *dummy*) vozliščem, katerega levi in desni kazalec imata vrednost *NULL*. Poleg tega so

potrebne naslednje spremembe v algoritmu [6]:

- klic funkcije `prilagodljivoBitonoUrejanje` izvedemo samo, ko korensko vozlišče ni navidezno,
- klic funkcije `prilagodljivoBitonoZlivanje` izvedemo samo, ko korensko vozlišče ni navidezno,
- kadar je potrebna zamenjava vrednosti navideznega in veljavnega vozlišča, zamenjamo tudi njuna kazalca *levo* in *desno*.

5.1.5 Časovna zahtevnost

V poglavjih 5.1.2 in 5.1.3 smo pokazali, da je časovna zahtevnost izvedbe enega koraka zlivanja bitonega zaporedja dolžine m enaka $C_{korak} = O(\log m)$. Pri tem upoštevajmo, da je število faz urejanja f in korakov zlivanja k enako kot pri navadnem bitonem urejanju (3.5). Časovno zahtevnost prilagodljivega bitonega urejanja zaporedja dolžine n je enaka [29]:

$$\begin{aligned}
T_1 &= \sum_{f=1}^{\log n} \sum_{k=1}^{f} \frac{n}{2^k} \cdot C_{korak}(2^k) \\
&\leq \log n \cdot \left(\sum_{k=1}^{\log n} \frac{n}{2^k} \cdot \log 2^k \right) \\
&= n \cdot \log n \cdot \left(\sum_{k=1}^{\log n} \left(\frac{1}{2}\right)^k \cdot k \right) \\
&= n \cdot \log n \cdot \left(\frac{(-1 - \frac{1}{2} \cdot \log n) \cdot (\frac{1}{2})^{1+\log n} + \frac{1}{2}}{(\frac{1}{2})^2} \right) \\
&= n \cdot \log n \cdot \left((-2 - \log n) \cdot \left(\frac{1}{2}\right)^{\log n} + 2 \right) \\
&= n \cdot \log n \cdot \left((-2 - \log n) \cdot \frac{1}{n} + 2 \right) \\
&= O(n \cdot \log n). \qquad (5.5)
\end{aligned}$$

```
1  // koren:   korensko vozlišče bitonega drevesa z enoličnimi elementi,
2  // rezerva: rezervno vozlišče, katero ni del bitonega drevesa,
3  // s:       smer urejanja (0: naraščajoče, 1: padajoče).
4  prilagodljivoBitonoUrejanje(koren, rezerva, s):
5    if ((koren.levo == NULL)):
6      if ((s == NARASCAJOCE) XOR (koren.v < rezerva.v)):
7        zamenjajVrednost(koren, ostanek)
8      return
9
10   prilagodljivoBitonoUrejanje(koren.levo, koren, s)
11   prilagodljivoBitonoUrejanje(koren.desno, rezerva, !s)
12   prilagodljivoBitonoZlivanje(koren, ostanek, s)
```

Psevdokoda 5.3: Prilagodljivo bitono urejanje (prirejeno po [6]).

Kot vidimo, ima prilagodljivo bitono urejanje optimalno časovno zahtevnost za urejanje s primerjavami.

5.2 Vzporedni algoritem

V prejšnjem poglavju 5.1 smo ugotovili, da prilagodljivo bitono urejanje deluje na podlagi bitonih dreves. Vzporedni algoritmi na arhitekturi CUDA so pogosto implementirani kot hibridni algoritmi. Običajno uporabljajo drugačen algoritem za krajša in za daljša zaporedja. Iz tega razloga bi bilo potrebno pretvarjanje bitonega drevesa v tabelo in obratno, zato algoritem ni primeren za arhitekturo CUDA. Potrebno ga je prilagoditi do te mere, da deluje brez bitonih dreves. V ta namen potrebujemo algoritem *bitonega urejanja s preureditvijo intervalov* (ang. *Interval based Rearrangement bitonic sort*) [29].

5.2.1 Bitono zlivanje s preureditvijo intervalov

Algoritem bitonega zlivanja s preureditvijo intervalov deluje podobno kot prilagodljivo bitono zlivanje. Oba algoritma v vsakem koraku zlivanja poiščeta vrednost q za vsa bitona podzaporedja. Kot smo omenili v poglavju 5.1.4, algoritem prilagodljivega bitonega zlivanja med iskanjem vrednosti q vedno premesti elemente v pomnilniku (oziroma v bitonem drevesu), s čimer dobimo podzaporedji *L(E)* (3.2) in *U(E)* (3.3). Nasprotno pa algoritem bito-

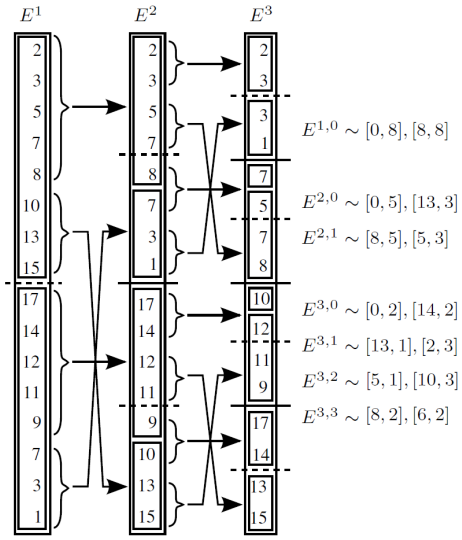
```
1  // tabela:    tabela elementov,
2  // intervalN: N-ti interval bitonega zaporedja ("o": odmik, "l": dolžina),
3  // indeks:    zaporedni indeks elementa za podane intervale.
4  vrniElement(tabela, interval0, interval1, indeks):
5    uporabiInterval0 = indeks >= interval0.l
6    odmik = uporabiInterval0 ? interval1.o : interval0.o
7    indeks -= uporabiInterval0 ? interval0.l : 0
8    indeks -= uporabiInterval0 && indeks >= interval1.l ? interval1.l : 0
9
10   return tabela[odmik + indeks]
```

Psevdokoda 5.4: Funkcija, ki na podlagi zaporednega indeksa elementa in obeh intervalov bitonega zaporedja vrne ustrezen element v tabeli (prirejeno po [29]).

nega zlivanja s preureditvijo intervalov ne izvede nobenih zamenjav v tabeli elementov, ampak na podlagi vrednosti q določi intervale tabele elementov, ki sestavljajo podzaporedji *L(E)* in *U(E)*. Omenjeni intervali so sestavljeni iz odmika v tabeli elementov o in dolžine l, torej $[o,\ l]$ [29].

Ne glede na trenutni korak v algoritmu zlivanja lahko vsako bitono zaporedje predstavimo z *dvema* intervaloma. Eden izmed intervalov predstavlja monotono naraščajoče in drugi monotono padajoče zaporedje. Za dostop do elementov potrebujemo funkcijo, ki preslika intervale v indekse tabele. Implementacija omenjene funkcije je prikazana v psevdokodi 5.4. Kot vidimo funkcija na podlagi obeh intervalov bitonega zaporedja in zaporednega indeksa iskanega elementa vrne iskan element v tabeli. Iz psevdokode ugotovimo, da se funkcija izvede v konstantnem času oziroma v času $O(1)$ [29].

Slika 5.3 prikazuje primer zlivanja bitonega zaporedja dolžine $n = 16$. Začetno bitono zaporedje $E^{1,0}$ predstavimo z intervaloma $[0,\ \frac{n}{2}]$ in $[\frac{n}{2},\ \frac{n}{2}]$ oziroma v našem primeru [0, 8] in [8, 8]. Nato poiščemo vrednost $q = 5$. Na podlagi prej omenjenih intervalov in vrednosti q izračunamo intervala podzaporedij $E^{2,0}$ (oziroma *L(E)*) in $E^{2,1}$ (oziroma *U(E)*) (opis algoritma za izračun intervalov sledi v naslednjem poglavju 5.2.3). Dobljene intervale uporabimo v naslednjem koraku, v katerem za obe zaporedji $E^{2,0}$ in $E^{2,1}$ zopet poiščemo vrednost q, na podlagi katere izračunamo intervale bitonih podzaporedij $E^{3,0}$, $E^{3,1}$, $E^{3,2}$ in $E^{3,3}$. Na enak način izračunamo še intervale za bitona podzaporedja dolžine 4 in 2 (ni prikazano na sliki 5.3). Izpostaviti



Slika 5.3: Bitono zlivanje s preureditvijo intervalov (prirejeno po [29]).

je potrebno, da bitono zaporedje $E^{1,0}$ ostane nespremenjeno med izvajanjem vseh korakov zlivanja, saj algoritem ne izvaja menjav elementov, ampak samo izračunava intervale. Na koncu na podlagi dobljenih intervalov in s pomočjo funkcije `vrniElement` preuredimo oziroma zlijemo zaporedje $E^{1,0}$ [29].

5.2.2 Algoritem iskanja vrednosti q z intervali

Za iskanje vrednosti q v bitonih zaporedjih uporabimo prirejeno različico prej opisane funkcije `najdiQ` (psevdokoda 5.1), ki jo prikazuje psevdokoda 5.5. Intervala oziroma monotoni pozdaporedji imata skoraj vedno različno dolžino, kar nam lahko predstavlja težavo. Da bi se temu izognili, prilagodimo indeksa za začetek in konec iskanja (vrstici 7 in 8) glede na krajše monotono podzaporedje. S tem dosežemo, da smo ob izvajanju iskanja vedno znotraj intervalov podzaporedij. Za dostop do elementov med binarnim iskanjem uporabljamo funkcijo `vrniElement` 5.4. Pri tem moramo vedno podati tudi smer urejanja, ker je zaradi ponavljajočih se elementov ne moremo določiti (opisano v poglavju 5.1.4). Časovna zahtevnost iskanja q z intervali je enaka $O(\log n)$. To pomeni, da je tudi časovna zahtevnost koraka

```
1  // tabela:    tabela elementov,
2  // intervalN: N-ti interval bitonega zaporedja,
3  // dPolovica: polovica dolžine bitonega zaporedja, v katerem iščemo q,
4  // s:         smer urejanja (0: naraščajoče, 1: padajoče).
5  najdiQzIntervali(tabela, interval0, interval1, dPolovica, s):
6    // Glede na dolžino intervalov določi začetni in končni indeks iskanja.
7    indeksZacetek = interval0.l <= interval1.l ? 0 : dPolovica - interval1.l
8    indeksKonec = interval0.l <= interval1.l ? interval0.l : dPolovica
9
10   while (indeksZacetek < indeksKonec):
11     indeks = indeksZacetek + (indeksKonec - indeksZacetek) / 2
12
13     // Do elementov dostopamo s funkcijo "vrniElement" (psevdokoda 5.4)
14     el0 = vrniElement(tabela, interval0, interval1, indeks)
15     el1 = vrniElement(tabela, interval0, interval1, indeks + dPolovica)
16
17     if ((el0 > el1) XOR s):
18       indeksZacetek = indeks + 1
19     else:
20       indeksZacetek = indeks
21
22   return indeksZacetek
```

Psevdokoda 5.5: Algoritem za iskanje q z intervali [29].

zlivanja bitonega zaporedja enaka $O(\log n)$. Omenjeni trditvi veljata zaradi prej izpostavljene predpostavke, da lahko vsako bitono zaporedje predstavimo z dvema intervaloma [29].

5.2.3 Tvorjenje intervalov bitonih zaporedij

Bitono zaporedje E dolžine $n = 2^k,\ k \in \mathbb{N}$ lahko predstavimo z intervaloma $[o_0,\ l_0]$ in $[o_1,\ l_1]$. S pomočjo prej opisane funkcije `najdiQzIntervali` 5.5 lahko v času $O(\log n)$ poiščemo vrednost q, na podlagi katere lahko za zaporedje E izračunamo podzaporedji:

$$L_E \simeq [o_{0,0},\ l_{0,0}],\ [o_{0,1},\ l_{0,1}], \tag{5.6}$$

$$U_E \simeq [o_{1,0},\ l_{1,0}],\ [o_{1,1},\ l_{1,1}]. \tag{5.7}$$

Za zaporedji (5.6) in (5.7) mora veljati, da je njun cikličnim zamik za $g, h \in \mathbb{N}$ enak *L(E)* (3.2) oziroma *U(E)* (3.3):

$$L(E) = S_g(L_E), \tag{5.8}$$

$$U(E) = S_h(U_E). \tag{5.9}$$

Za zaporedji L_E in U_E mora prav tako veljati, da ju sestavlja interval z monotono naraščajočimi in interval z monotono padajočimi elementi [29].

Obstaja 6 različnih tipov bitonih zaporedij dolžine n. Urejena zaporedja v naraščajočem ali padajočem vrstnem redu so najbolj trivialna za določitev L_E in U_E. Podzaporedje L_E (5.6) enostavno predstavimo z intervalom $[o_0, \frac{n}{2}]$ in podzaporedje U_E (5.7) z intervalom $[o_0 + \frac{n}{2}, \frac{n}{2}]$, kjer upoštevamo, da sta ciklična zamika g (5.8) in h (5.9) enaka 0.

Ostali štirje primeri so sestavljeni iz naraščajočih in padajočih podzaporedij, pri čemer sta podzaporedji različno dolgi. Podrobneje analizirajmo primer bitonega zaporedja, prikazanega na sliki 5.4, kjer je prvo podzaporeje (oziroma interval) monotono naraščajoče in drugo podzaporedje (oziroma interval) monotono padajoče, pri čemer velja $l_o \leq \frac{n}{2}$ (dokaz za ostale primere je zelo podoben). Kot vemo, mora za vrednost q v opisanem tipu zaporedij veljati naslednje [29]:

$$\min(E_i, E_{i+\frac{n}{2}}) = E_i;\ \forall i \in \{0, 1, ..., q-1\}, \tag{5.10}$$

$$\min(E_i, E_{i+\frac{n}{2}}) = E_{i+\frac{n}{2}};\ \forall i \in \{q, q+1, ..., \frac{n}{2}-1\}. \tag{5.11}$$

Dokaz za enačbi (5.10) in (5.11) se nahaja v [29]. Na podlagi omenjenih enačb in slike 5.4 lahko zaporedji *L(E)* in *U(E)* zapišemo kot:

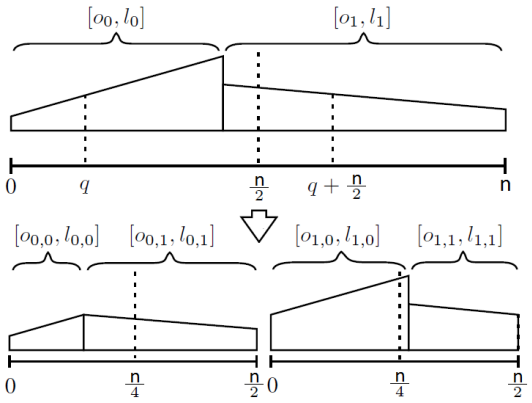
$$L(E) = (e_0, e_1, ..., e_{q-1}, e_{\frac{n}{2}+q+1}, e_{\frac{n}{2}+q+2}, ..., e_{n-1}),$$
$$U(E) = (e_{\frac{n}{2}}, e_{\frac{n}{2}+1}, ..., e_{\frac{n}{2}+q-1}, e_q, e_{q+1}, ..., e_{\frac{n}{2}-1}).$$

Lahko ju zapišemo tudi kot intervala:

$$L(E) \simeq [o_0,\ q],\ [o_1 + (l_1 - \frac{n}{2}) + q,\ \frac{n}{2} - q], \tag{5.12}$$

$$U(E) \simeq [o_1 + (l_1 - \frac{n}{2}),\ q],\ [o_0 + q,\ l_0 - q],\ [o_1,\ l_1 - \frac{n}{2}]. \tag{5.13}$$

Kot vidimo, je zaporedje *L(E)* (5.12) predstavljeno z naraščajočim in nato s padajočim intervalom. Zaporedje *U(E)* (5.13) je predstavljeno s tremi



Slika 5.4: Bitono zaporedje sestavljeno iz monotono naraščajočega in monotono padajočega podzaporedja, pri čemer velja, da je prvo podzaporedje krajše ali enako dolgo kot drugo podzaporedje (prirejeno po [29]).

intervali. Težavo lahko rešimo s cikličnim zamikom $U(E)$ za $l_1 - \frac{n}{2}$:

$$
\begin{aligned}
&U_E = S_{l_1 - \frac{n}{2}}(U(E)) \\
&= (e_{\frac{n}{2}-(l_1-\frac{n}{2})}, e_{\frac{n}{2}-(l_1-\frac{n}{2})+1}, ..., e_{\frac{n}{2}-1}, e_{\frac{n}{2}}, e_{\frac{n}{2}+1}, ..., e_{\frac{n}{2}+q-1}, e_q, e_{q+1}, ..., e_{\frac{n}{2}-(l_1-\frac{n}{2})-1}) \\
&= (e_{l_0}, e_{l_0+1}, ..., e_{\frac{n}{2}+q-1}, e_q, e_{q+1}, ..., e_{l_0-1}).
\end{aligned}
$$

Na ta način lahko zaporedje U_E zapišemo samo z dvema intervaloma:

$$U_E \simeq [o_1,\ q + l_1 - \frac{n}{2}],\ [o_0 + q,\ l_0 - q]. \tag{5.14}$$

Prikazana predstavitev (5.14) je veljavna prestavitev, saj je prvi interval monotono padajoč in drugi monotono naraščajoč. Tako lahko s pomočjo enačb (5.12) in (5.14) določimo podzaporedji L_E in U_E [29].

5.2.4 Implementacija na gostitelju

Psevdokoda 5.6 prikazuje implementacijo na gostitelju. Algoritem deluje podobno kot algoritma 3.2 in 4.1. Kadar dolžina tabele ni enaka potenci števila 2, jo moramo glede na smer urejanja najprej zapolniti z minimalnimi oziroma maksimalnimi vrednostmi (vrstica 17). Potem izvedemo klic ščepca za

```
1  // tab:      vhodna tabela,
2  // pomTab:   pomožna tabela potrebna za bitono zlivanje,
3  // n:        dolžina vhodne in pomožne tabele brez zapolnjenih elementov,
4  // fazeBU:   št. faz izvedenih v ščepcu za bitono urejanje,
5  // korakiLZ: št. korakov izvedenih v ščepcu za lokalno zlivanje,
6  // korakiII: maks. št. korakov tvorjenja inter. v ščepcu za inicializacijo,
7  // korakiTI: maks. št. korakov tvorjenja inter. v ščepcu za obstoječe inter.,
8  // s:        smer urejanja (0: naraščajoče, 1: padajoče).
9  vzporUrejanjePI(tab, pomTab, n, fazeBU, korakiLZ, korakiII, korakiTI, s):
10   // Izračuna število faz potrebnih za bitono urejanje
11   stFaz = log2(nasledjaPotencaStevila2(n))
12   // Pomožni tabeli za hranjenje intervalov
13   inter[], interPom[]
14
15   // Če dolžina tabel ni enaka potenci števila 2, ščepec zapolni preostanek
16   // tabel z min. oziroma maks. vrednostmi (glede na smer urejanja)
17   scepecZapolniTabelo(tab, pomTab, n, s)
18   // Izvede "fazeBU" faz bitonega urejanja v deljenem pomnilniku
19   scepecBitonoUrejanje(tab, n, fazeBU, s)
20
21   // Izvedba preostalih faz bitonega urejanja
22   for (faza = fazeBU + 1; faza < stFaz; faza++):
23     // Določi začetni in končni korak tvorjenja intervalov
24     korakZ, korakK = faza, max(korakiLZ, faza - korakiII)
25     scepecInicializirajIntervale(tab, inter, n, stFaz, korakZ, korakK, s)
26
27     // Tvori preostanek intervalov z več klici ščepca, saj je število
28     // tvorjenih intervalov na en klic ščepca omejeno
29     while (korakK > korakiLZ):
30       zamenjajKazalca(inter, interPom)
31       korakZ, korakK = korakK, max(korakiLZ, korakZ - korakiTI);
32       // Prebere obstoječe intervale in jih razvije
33       scepecTvoriIntervale(
34         tab, inter, interPom, n, stFaz, faza, korakZ, kokrakK, s
35       )
36
37     // Izvede "korakiLZ" korakov zlivanja v deljenem pomnilniku
38     scepecLokalnoBitonoZlivanje(tab, tabPom, inter, n, faza, korakiLZ, s)
39     zamenjajKazalca(tab, tabPom)
```

Psevdokoda 5.6: Bitono urejanje s preureditvijo intervalov na gostitelju (prirejeno po [29]).

bitono urejanje (vrstica 19), ki uredi podzaporedja velikosti $2^{\texttt{fazeBU}}$. Omenjen ščepec uredi samo originalno tabelo brez zapolnjenih vrednosti, saj se vrsti red zapolnjenih elementov ne bi spremenil.

Globalno zlivanje izvajamo nad celotno tabelo (tudi nad zapolnjenim delom). Najprej pokličemo ščepec, ki inicializira intervale (vrstica 25). Omenjen ščepec nato razvija inicializirane intervale za nadaljnjih `korakiII` korakov oziroma do koraka `korakiLZ`, ko se začne izvajati lokalno bitono zlivanje (vrstica 38). Število tvorjenih intervalov ob enem klicu ščepca je ome-

jeno z velikostjo deljenega pomnilnika oziroma s parametroma `korakiII` in `korakiTI`. Pri daljših zaporedjih je zato potrebnih več klicev ščepca, ki razvije obstoječe intervale za `korakiTI` nadaljnjih korakov (vrstice 29 - 35). Na koncu vsake faze se izvede lokalno bitono zlivanje (vrstica 38). Zlivanje v deljenem pomnilniku deluje zelo podobno, kot smo opisali v poglavju 3.2.2. Razlika je le v tem, da niti ob branju elementov iz globalnega pomnilnika uporabijo funkcijo `vrniElement` 5.4. S tem dosežemo preureditev oziroma zlivanje podzaporedij, ki so predstavljena z intervali. Lokalnega zlivanja ne izvajamo nad celotno zapolnjeno tabelo, temveč zgolj samo nad naslednjim večkratnikom $2^{\texttt{faza}}$ dolžine tabele `n`. To pomeni, da izvedemo zlivanje nad celotno zapolnjeno tabelo samo v zadnji fazi.

Pri urejanju parov ključ-vrednost nam v ščepcih za inicializacijo in tvorjenje intervalov ni potrebno dostopati do vrednosti, ker za določanje intervalov zadostujejo ključi. Posledično pride do zelo majhnega padca v hitrosti delovanja, saj v številnih fazah urejanja ne dostopamo do vrednosti. Dostopanje do vrednosti je potrebno samo v ščepcu za začetno bitono urejanje (vrstica 19) in v ščepcu za lokalno bitono zlivanje (vrstica 38). Ščepec za dodajanje polnila kličemo samo nad primarno in pomožno tabelo ključev.

5.2.5 Implementacija na napravi

Psevdokoda 5.7 prikazuje ščepec za *inicializacijo intervalov*, ki ga pokličemo na začetku vsake faze urejanja f (vrstica 25 v psevdokodi 5.6). Kot vidimo najprej za vsa bitona podzaporedja p dolžine $m = 2^f$ tvorimo začetna intervala $[p \cdot m,\ \frac{m}{2}]$ in $[p \cdot m + \frac{m}{2},\ \frac{m}{2}]$ (vrstice 14 - 21). Pri tem intervale v lihih zaporedjih zamenjamo. Kljub temu, [29] ne navajajo potrebe po zamenjavi lihih intervalov, je bila za našo implementacijo nujno potrebna Ustvarjene intervale tako razvijamo nadaljnjih (`korakZ` - 1) - `korakK` korakov (vrstice 29 - 48). Pri tem v vsakem koraku omejimo število aktivnih niti (vrstici 14 in 30), ki je proporcionalno številu novo ustvarjenih intervalov. Za obstoječe intervale najprej poiščemo vrednost q (vrstice 36 - 38), na podlagi katere nato tvorimo dva nova intervala (vrstice 42 - 44). Za intervala iz enačbe

```
1  // tab:       vhodna tabela,
2  // intervali: tabela za shranjevanje intervalov,
3  // n:         dolžina vhodne tabele,
4  // korakZ:    začetni korak zlivanja,
5  // korakK:    končni korak zlivanja,
6  // s:         smer urejanja (0: naraščajoče, 1: padajoče).
7  scepecInicializirajIntervale(tab, intervali, n, korakZ, korakK, s):
8    // Primarna in pomožna tablea za hranjenje intervalov
9    __shared__ lokInter, lokInterPom
10   tx, aktivneNiti = threadIdx.x, n / (1 << korakZ) / gridDim.x
11
12   // Inicializira intervale. Ščepec za tvorjenje intervalov prebere obstoječe
13   // intervale iz globalnega pomnilnika.
14   if (threadIdx.x < aktivneNiti):
15     // Izračuna velikost enega bitonega podzaporedja in indeks intervala
16     velP, indeksInt = 1 << korakZ, blockIdx.x * blockDim.x + tx
17     // Izračuna odmik za nova intervala
18     odmik0, odmik1 = indeksInt * velP, indeksInt * velP + velP / 2
19     // Intervale lihih podzaporedij je potrebno zamenjati
20     odmik0, odmik1 = (indeksInt % 2) == 0 ? odmik0, odmik1 : odmik1, odmik0
21     lokInter[tx] = [[odmik0, velP], [odmik1, velP]]
22
23   __syncthreads()
24   // Število intervalov na bitono podzaporedje dolžine "1 << faza". V ščepcu
25   // za tvorjenje intervalov je ta vrednost enaka "1 << (faza - korakZ)".
26   stInterNaBitPodzap = 1
27
28   // Tvorjenje intervalov od začetnega do končnega koraka
29   for (korak = korakZ - 1; korak >= korakK; korak--):
30     if (tx < aktivneNiti):
31       // Določi smer urejanja podzaporedja
32       indeksInter = blockIdx.x * aktivneNiti + tx
33       sNarascajoce = s XOR ((indeksInter / stInterNaBitPodzap) & 1)
34
35       // Uporabi funkcijo iz psevdokode 5.1
36       q = najdiQzIntervali(
37         tabela, lokInter[tx][0], lokInter[tx][1], 1 << korak, sNarascajoce
38       )
39
40       // Tvori nova intervala po enačbah (5.12) in (5.14). Intervala (5.14)
41       // tvori v OBRATNEM vrstnem redu kot sta navedena v enačbi.
42       lokInterPom[2 * tx, 2 * tx + 1] = tvoriIntervala(
43         lokInter[tx], q, 1 << korak
44       )
45
46     stInterNaBitPodzap, aktivneNiti *= 2
47     zamenjajKazalca(lokInter, lokInterPom)
48     __syncthreads()
49
50   shraniVGlobalniPomnilnik(intervali, lokInter, lokInterPom)
```

Psevdokoda 5.7: Ščepec za inicializacijo in nadaljnje tvorjenje intervalov.

(5.14) je potrebno poudariti, da ju tvorimo v obratnem vrstnem redu kot sta zapisana v omenjeni enačbi. Članek [29] zopet ne omenja potrebe po menjavi intervalov, vendar je bila potrebna za našo implementacijo. Na koncu ščepca shranimo intervale v globalni pomnilnik. Ščepec za *tvorjenje intervalov* deluje zelo podobno (vrstica 33 v psevdokodi 5.6). Razlika je le, da v prvem

koraku ne inicializira novih intervalov, ampak prebere obstoječe intervale iz globalnega pomnilnika.

5.2.6 Časovna zahtevnost

Pred analizo časovne zahtevnosti je potrebno poudariti, da sta algoritma *prilagodljivega bitonega urejanja* in *bitonega urejanja s preureditvijo intervalov* različna algoritma, ki temeljita na zelo podobnih idejah.

Pri vzporedni implementaciji smo iskanje vrednosti q (oziroma določanje podzaporedij *L(E)* in *U(E)*) v istih korakih zlivanja izvajali vzporedno, pri čemer smo korake zlivanja izvajali zaporedno. Število bitonih podzaporedij, in posledično število iskanj vrednosti q, je odvisno samo od trenutnega koraka bitonega zlivanja. To pomeni, da se bo tekom izvajanja algoritma večina iskanj izvajala z veliko mero vzporednosti [29].

Za določitev časovne zahtevnosti upoštevamo, da je število faz f in korakov k enako kot pri vzporednem bitonem urejanju. To pomeni, da sta kritični poti obeh algoritmov enaki, saj moramo korake bitonega zlivanja izvajati zaporedno. Kot smo omenili, lahko izvajamo vzporedno le posamezna iskanja vrednosti q znotraj istega koraka urejanja. Časovna zahtevnost za p procesorjev je enaka (na podlagi enačbe (3.12)):

$$T_p = O\left(\frac{n}{p} \cdot \log^2 n\right).$$

Pri dovolj velikem številu procesorjev (tj. $O(n)$) lahko dosežemo optimalno časovno zahtevnost:

$$T_n = O(\log^2 n). \tag{5.15}$$

Pohitritev vzporedne implementacije za p procesorjev je enaka:

$$S_P = \frac{T_1}{T_p} = O\left(\frac{n \cdot \log n}{\frac{n}{p} \cdot \log^2 n}\right) = O\left(\frac{p}{\log n}\right). \tag{5.16}$$

Kot vidimo, pohitritev ni idealna. Pohitritev pri $O(n)$ procesorjih je enaka:

$$S_n = \frac{T_1}{T_n} = O\left(\frac{n \cdot \log n}{\log^2 n}\right) = O\left(\frac{n}{\log n}\right). \tag{5.17}$$

Poglavje 6

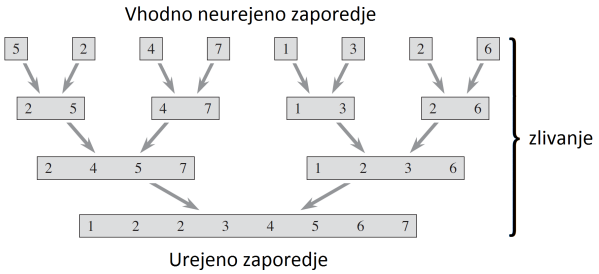
Urejanje z zlivanjem

Urejanje z zlivanjem (ang. *merge sort*) temelji na pristopu deli in vladaj (ang. *divide-and-conquer*). Algoritmi, implementirani na podlagi omenjenega pristopa delujejo tako, da razdelijo probleme na manjše podprobleme, jih rešijo in nato združijo dobljene rešitve. S tem rešijo tudi prvoten problem. Urejanje sledi opisanemu pristopu tako, da vhodno zaporedje najprej razdeli na manjše bloke, jih uredi in na koncu zlije v urejeno zaporedje. Ob uporabi ustreznega algoritma zlivanja je lahko urejanje tudi stabilno [12, 30].

6.1 Zaporedni algoritem

6.1.1 Algoritem zlivanja

Algoritem zlivanja predstavlja temelj za delovanje urejanja z zlivanjem. Opisali bomo osnovno in v praksi največkrat uporabljeno različico zlivanja. Ta na vhodu prejme dve *urejeni* zaporedji A in B dolžine p in q, kateri *zlije* v *urejeno* zaporedje C dolžine $p+q$. Algoritem v prvem koraku nastavi kazalca a in b na začetek vhodnih zaporedij A in B. Potem v vsakem koraku zlivanja i primerja elementa na mestih a in b. V tabelo C na mesto i shrani manjšega izmed primerjanih elementov. V primeru enakih elementov shrani element iz zaporedja A (ob predpostavki, da se zaporedje A nahaja pred oziroma levo od B v celotnem zaporedju), kar je tudi ključno za ohranjanje stabilnosti.



Slika 6.1: Urejanje z zlivanjem zap. (5, 2, 4, 7, 1, 3, 2, 6) (prirejeno po [12]).

Po vsakem koraku premakne ustrezen kazalec a ali b za eno mesto naprej. V primeru, da pride do konca zaporedja A oziroma B, kopira preostanek zaporedja B oziroma A v C. Po $p + q$ korakih zlivanja se algoritem konča. Pri tem je potrebno poudariti, da je opisan algoritem zlivanja *stabilen* [12].

6.1.2 Algoritem urejanja

Algoritem najprej razdeli vhodno zaporedje dolžine n na bloka dolžine $\frac{n}{2}$. To rekurzivno ponavlja, dokler ne dobimo blokov dolžine 1, ki so po definiciji urejena zaporedja. Nato omenjene bloke zlije, s čimer dobimo *urejene* bloke dolžine 2. Postopek ponavlja za urejene bloke dolžine 4, 8, itd., dokler v zadnji fazi zlivanja ne dobimo urejenega zaporedja. Slika 6.1 prikazuje primer urejanja z zlivanjem. Vhodno zaporedje na začetku razbijemo na bloke dolžine 1, ki jih nato v 3 fazah zlivanja uredimo [12].

6.1.3 Implementacija

Kot smo opisali v prejšnjem poglavju, je urejanje z zlivanjem rekurziven algoritem, zaradi česar smo preizkusili rekurzivno in iterativno implementacijo. Odločili smo se za slednjo, saj se je izkazala za hitrejšo.

Psevdokoda 6.1 prikazuje implementacijo algoritma. Kot vidimo v vrstici 7, iteriramo čez vse velikosti blokov oziroma čez vseh $\log_2 n$ faz urejanja. V vsaki iteraciji zlijemo vse podbloke (vrstice 9 - 23). Med zlivanjem shranjujemo elemente v pomožno tabelo `tabPom` (vrstici 17 in 18). Po opravljenem zlivanju preverimo, če smo prišli do konca levega oziroma desnega bloka. V

```
1  // tab:    vhodna tabela,
2  // tabPom: pomožna tabela,
3  // n:      dolžina vhodne in pomožne tabele,
4  // s:      smer urejanja (0: naraščajoče, 1: padajoče).
5  zaporednoUrejanjeZZlivanjem(tab, tabPom, n, s):
6    // Izvede log2 n faz zlivanja
7    for (velB = 1; velB < n; velB *= 2):
8      // Zlivanje vseh podzaporedij
9      for (p = 0; p < (n - 1) / (2 * velB) + 1; p++):
10       // Začetek in konec levega podzaporedja
11       indeksLZac = p * (2 * velB)
12       indeksLKon = indeksLZac + velB <= n ? indeksLZac + velB : n
13       // Začetek in konec desnega podzaporedja
14       indeksDZac = indeksLZac + velB
15       indeksDKon = indeksDZac + velB <= n ? indeksDZac + velB : n
16
17       while (indeksLZac < indeksLKon && indeksDZac < indeksDKon):
18         // Izvajaj zlivanje, kot je opisano v poglavju 6.1.1
19
20       if (indeksLZac >= indeksLKon && indeksDZac <= IndeksDKon):
21         // Kopiraj iz "tab" v "tabPom" od "indeksDZac" do "IndeksDKon"
22       else:
23         // Kopiraj iz "tab" v "tabPom" od "indeksLZac" do "IndeksLKon"
24
25     zamenjajKazalca(tab, tabPom)
```

Psevdokoda 6.1: Zaporedno urejanje z zlivanjem.

tem primeru izvedemo kopiranje preostanka desnega oziroma levega bloka iz `tab` v `tabPom` (vrstice 20 - 23). V ta namen smo uporabili funkcijo `std::copy`, ki je hitrejša od kopiranja s pomočjo zanke. Prikazan algoritem urejanja deluje za poljubno dolžino tabele in je stabilen.

6.1.4 Časovna zahtevnost

Iz opisa v poglavju 6.1.1 vidimo, da ima zlivanje dveh zaporedij dolžine p in q časovno zahtevnost $O(p+q)$. V vsaki fazi urejanja f dobimo $\frac{n}{2^f}$ urejenih blokov dolžine 2^f. Število primerjav in kopiranj v eni fazi urejanja je enako:

$$\frac{n}{2^f} \cdot 2^f = n. \tag{6.1}$$

Urejanje z zlivanjem zaporedja dolžine $n = 2^r$ je sestavljeno iz $\log_2 n$ oziroma r faz zlivanja, od katerih vsaka zahteva $O(n)$ (6.1) operacij primerjav in kopiranj. Iz omenjenega sledi, da je časovna zahtevnost enaka:

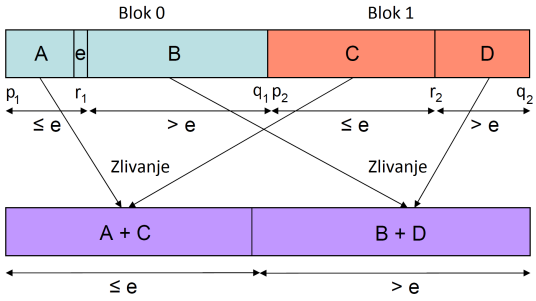
$$T_1 = O(n \cdot \log n). \tag{6.2}$$

6.2 Vzporedni algoritem

Urejanje z zlivanjem lahko izvedemo tudi vzporedno. V vsaki fazi urejanja lahko vzporedno zlijemo pare urejenih blokov, saj so ti medsebojno neodvisni. Opisan način zlivanja predstavlja težavo, kajti število parov blokov se razpolovi z vsako fazo urejanja. Kot vidimo na sliki 6.1, imamo v prvi fazi 8 blokov, v drugi 4 in v zadnji samo 2. To predstavlja neizogibno ozko grlo za vzporedni algoritem. Pohitrimo lahko le algoritem zlivanja, ki ga lahko izvedemo vzporedno [30].

6.2.1 Osnoven algoritem zlivanja

Algoritem zlivanja želimo optimizirati tako, da bi bilo potrebno v vsaki fazi urejanja zliti enako število blokov. Na ta način bi bil algoritem enako učinkovit v vseh fazah urejanja, ne samo v zgodnjih fazah. Iz tega razloga potrebujemo algoritem *zlivanja z delitvijo na več podblokov* [16]. Pred podrobnejšo analizo algoritma definirajmo *rang* r (ang. *rank*) elementa e urejene tabele. To je število elementov v urejeni tabeli, ki so manjši ali enaki vrednosti elementa e. Slika 6.2 prikazuje delovanje algoritma na dveh *urejenih* blokih dolžine $n_1 = q_1 - p_1 + 1$ in $n_2 = q_2 - p_2 + 1$. Najprej določimo vzorec $e = T[r_1]$ urejenega bloka 0, pri čemer je $r_1 = \lfloor \frac{(p_1+q_1)}{2} \rfloor$. Število r_1 je pravzaprav rang elementa e. Potem s pomočjo binarnega iskanja poiščemo rang r_2 elementa e v urejenem bloku 1. Blok 0 razdelimo na podbloka A in B, kjer A vsebuje prvih r_1 elementov in B preostanek bloka. Na enak način razdelimo tudi blok 1 na podbloka C in D glede na rank r_2. Ugotovimo, da so vsi elementi podblokov A in C manjši ali enaki e, medtem ko so vsi elementi podblokov B in D večji od e. To pomeni, da lahko podbloka A in C ter B in D zlijemo neodvisno oziroma vzporedno, pri čemer element e vstavimo na indeks $r_3 = r_1 + (r_2 - p_2)$. Opisan algoritem izboljša vzporednost samo za faktor 2. V splošnem moramo pare blokov razbiti na veliko več podblokov, zaradi česar moramo uporabiti več vzorcev. Število podblokov je vedno za 1 večje kot število vzorcev [12, 16, 30].



Slika 6.2: Zlivanje z delitvijo na dva podbloka (prirejeno po [30]).

6.2.2 Napreden algoritem zlivanja

Če želimo doseči maksimalno učinkovitost urejanja z zlivanjem, moramo podbloke zliti v deljenem pomnilniku. Algoritem zlivanja mora zagotoviti hitro iskanje vzorcev v blokih. Poleg tega mora zagotoviti, da so vsi podbloki manjši od velikosti deljenega pomnilnika GPE. Osnovni algoritem, opisan v prejšnjem poglavju, nam ne zagotavlja naštetih lastnosti. Da bi dosegli želeno, potrebujemo napredni algoritem zlivanja, ki je prikazan na sliki 6.3.

6.2.2.1 Določanje vzorcev

Podobno kot v osnovnem algoritmu zlivanja najprej izberemo vzorce. Te določimo kot vsak M-ti element obeh blokov, pri čemer je M manjši od velikosti deljenega pomnilnika. V naši implementaciji smo za vrednost M izbrali 256. S tem smo omejili velikost podblokov na 256, zaradi česar jih lahko zlijemo v deljenem pomnilniku. Na sliki vidimo, da smo iz bloka 0 izbrali vzorca 0 in 20 ter iz bloka 1 vzorca 10 in 50 [30].

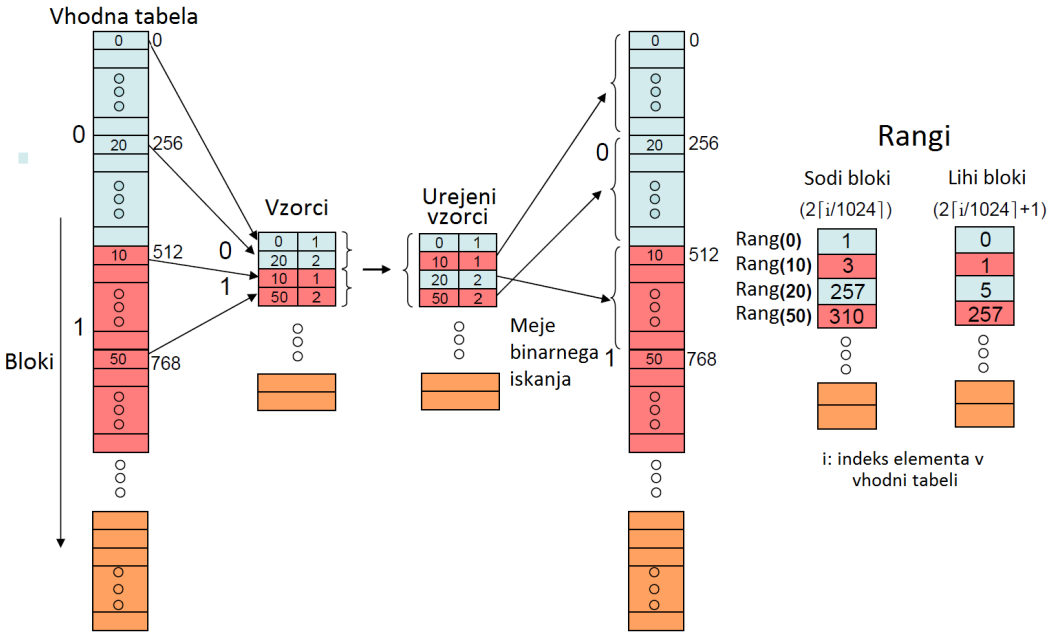
6.2.2.2 Določanje rangov vzorcev

Po določitvi vzorcev v obeh blokih moramo poiskati njihove range. Rang vzorcev v njihovem pripadajočem bloku je določen trivialno, ker je enak indeksu elementa znotraj bloka (ranga vzorcev bloka 0: 1 in 257, ranga vzorcev bloka 1: 513 in 767). To pomeni, da moramo za oba podbloka poiskati range vzorcev v nasprotnem bloku [30].

Z vsako fazo urejanja narašča velikost blokov in je zato potrebnih več korakov binarnega iskanja za določitev ranga v nasprotnem bloku. Za pohitritev zlivanja je potrebno učinkovitejše iskanje ranga. Na začetku tako najprej shranimo vse vzorce v tabelo vzorcev. Poleg vzorcev shranimo tudi njihove range v omenjeni tabeli glede na posamezen blok. Na sliki 6.3 so v levem stolpcu prikazani vzorci (0, 20, 10, 50) in v desnem njihovi rangi glede na blok (1, 2, 1, 2). Tabelo *zlijemo* glede na vzorce, s čimer dobimo urejeno zaporedje (0, 10, 20, 50). Pri zlivanju vzorcev izvajamo tudi zamenjave njihovih pripadajočih rangov. S pomočjo urejene tabele lahko za vsak vzorec določimo indeks podbloka nasprotnega bloka, v katerem moramo poiskati rang vzorca. Določimo ga kot razliko med rangom vzorca v urejeni tabeli in rangom vzorca pred urejanjem. Vzemimo za primer vzorec 20, katerega rang v neurejeni tabeli vzorcev bloka 0 je enak 2. Po zlivanju se njegov rang v urejeni tabeli vzorcev poveča na 3. Iz prej opisane definicije ugotovimo, da je njegov rang v tabeli vzorcev bloka 1 enak $3 - 2 = 1$. To pomeni, da je vzorec 20 manjši od vseh vzorcev bloka 1, z izjemo vzorca 10. Iz opisanega sklepamo, da moramo rang vzorca 20 iskati v prvih 256 elementih bloka 1. Podrobneje analizirajmo tudi vzorec 0 bloka 0, katerega rang je enak 1 tako v tabeli vzorcev bloka 0 kot tudi v urejeni tabeli vzorcev. Njegov rang v tabeli vzorcev bloka 1 je zato enak $1 - 1 = 0$. Iz navedenega lahko sklepamo, da so vsi elementi v bloku 1 večji od omenjenega vzorca. Zaradi tega nam ni potrebno iskati njegovega ranga, saj je ta že trivialno določen z 0. Na enak način določimo tudi podbloka v bloku 0 za vzorca 10 in 50. Po določitvi podblokov v nasprotnem bloku poiščemo range vzorcev. To storimo s pomočjo binarnega iskanja. S tem v veliki meri pohitrimo iskanje rangov, ker morajo niti izvesti iskanje v podblokih velikosti M oziroma 256 in ne v celotnem nasprotnem bloku. Slika 6.3 na desni prikazuje dobljene range vzorcev [30].

6.2.2.3 Določanje podblokov za zlivanje

S pomočjo rangov prikazanih na sliki 6.3 določimo podbloke bloka 0 in 1, ki jih moramo zliti. Prikazani so v tabeli 6.1. Kot vidimo, je število podblokov v



Slika 6.3: Napredno vzporedno zlivanje (prirejeno po [30]).

vsakem bloku enako 5, kar je za 1 več kot število vzorcev. Obseg posameznega podbloka določimo s pomočjo dveh zaporednih rangov. Vzemimo za primer podblok 3 bloka 0. Kot vidimo na sliki 6.3, ima drugi rang bloka 0 vrednost 3 in tretji rang vrednost 257. To pomeni, da podblok 3 vsebuje elemente 3 - 256 bloka 0. Na podoben način določimo tudi ostale podbloke. Podbloki prikazani v tabeli 6.1 tudi potrjujejo prej omenjeno dejstvo, da je njihova velikosti manjša ali enaka M oziroma 256 [30].

6.2.2.4 Zlivanje podblokov

Algoritem, opisan v poglavju 6.2.1, zliva podbloke zaporedno, kar predstavlja veliko slabost za vzporednost. Zato potrebujemo učinkovitejši algoritem, prikazan v [10], v katerem lahko več niti zliva iste pare podblokov. Omenjen algoritem deluje tako, da vsaka nit izračuna rang svojega pripadajočega elementa v končni zliti tabeli in ga posledično lahko shrani na pravo mesto zlite tabele. Rang elementa v zliti tabeli je enak vsoti rangov elementa v

Podblok	Elementi bloka 0	Elementi bloka 1
0	0	-
1	1 - 2	0
2	3 - 256	1 - 4
3	257 - 309	5 - 256
4	310 - 511	257 - 511
...	...	...

Tabela 6.1: Podbloki blokov 0 in 1, ki sta prikazana na sliki 6.3 (prirejeno po [30]).

obeh podblokih, katera zlivamo. Rang elementa v njegovem pripadajočem podbloku je trivialno določen, ker je enak njegovem indeksu. To pomeni, da moramo ponovno poiskati rang elementa v nasprotnem podbloku, kar storimo z binarnim iskanjem. Potem niti izračunajo range v zliti tabeli in elemente shranijo na ustrezno mesto v globalnem pomnilniku. Velika prednost opisanega postopka je dejstvo, da lahko niti opravljajo binarna iskanja vzporedno. Pri tem je potrebno omeniti, da so dostopi do pomnilnika med binarnim iskanjem neporavnani. Zaradi tega moramo zlivanje izvajati v deljenem pomnilniku, ki je veliko hitrejši od globalnega pomnilnika [10, 30].

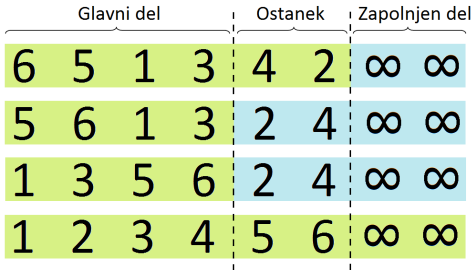
6.2.3 Implementacija

Psevdokoda 6.2 prikazuje implementacijo vzporednega urejanja z zlivanjem na gostitelju. V vrstici 13 najprej pokličemo ščepec, ki glede na smer urejanja zapolni primarno in pomožno tabelo z minimalnimi oziroma maksimalnimi vrednostmi. To je potrebno, kadar dolžina tabele ni enaka potenci števila 2. Nato pokličemo ščepec, ki uredi podbloke velikosti $2^{\texttt{fazeUZ}}$ (vrstica 15). Za ohranjanje stabilnosti urejanja mora biti stabilno tudi urejanje v omenjenem ščepcu, zato smo v našem primeru uporabili urejanje z zlivanjem. Ta izvaja urejanje v deljenem pomnilniku, pri čemer bloke zliva s pomočjo binarnega iskanja. Tukaj je potrebno izpostaviti, da ščepec ne uredi celotne zapolnjene tabele, temveč uredi le tabelo do naslednjega večkratnika $2^{\texttt{fazeUZ}}$ dolžine originalne tabele n, saj se vrstni red zapolnjenih elementov ne bi spremenil.

```
1  // tab:    vhodna tabela,
2  // tabPom: pomožna tabela,
3  // rangi:  tabela za hranjenje rangov vzorcev,
4  // fazeUZ: število faz izvedenih v ščepcu za urejanje z zlivanjem,
5  // n:      dolžina vhodne in pomožne tabele brez zapolnjenih elementov,
6  // s:      smer urejanja (0: naraščajoče, 1: padajoče).
7  vzporednoUrejanjeZZlivanjem(tab, tabPom, fazeUZ, n, s):
8    // Izračuna število faz potrebnih za urejanje z zlivanjem
9    steviloVsehFaz = log2(nasledjaPotencaStevila2(n))
10
11   // Če dolžina tabel ni enaka potenci števila 2, ščepec zapolni preostanek
12   // tabel z minimalno oziroma maksimalno vrednostjo (glede na smer urejanja)
13   scepecZapolniTabelo(tab, pomTab, n, s)
14   // Izvede "fazeUZ" faz urejanja z zlivanjem v deljenem pomnilniku
15   scepecUrejanjeZZlivanjem(tab, n, fazeUZ, s)
16
17   for (faza = fazeUZ + 1; faza < steviloVsehFaz; faza++):
18     zamenjajKazalca(tab, tabPom)
19     kopirajOstanekZaporedja(tab, tabPom, n, faza)
20     scepecDolociRange(tabPom, rangi, n, faza, s)
21     scepecZlijPodbloke(tabPom, tab, rangi, n, faza, s)
```

Psevdokoda 6.2: Vzporedno urejanje z zlivanjem na gostitelju.

Urejanje v preostalih fazah poteka tako, kot smo opisali v prejšnjem poglavju 6.2.2 (vrstice 17 - 21). Najprej se izvede ščepec, ki določi range podblokov (vrstica 20). Pri tem vsaka nit izvaja binarno iskanje za svoj pripadajoči vzorec. Za iskanje mora vsaka nit izvesti samo $\log_2 M = \log_2 256 = 8$ korakov, pri čemer je število vzorcev manjše kot število elementov tabele. Zato lahko iskanje izvajamo v globalnem pomnilniku, ker bi v nasprotnem primeru kopiranje podblokov v deljeni pomnilnik zahtevalo več časa. V naslednjem koraku se izvede še ščepec, ki na podlagi rangov določi meje podblokov, ki jih zlije s pomočjo binarnega iskanja (vrstica 21). Vsaka nit zopet izvaja binarno iskanje za svoj pripadajoči element. Število elementov je veliko večje kot število vzorcev, kar posledično pomeni veliko več neporavnanih dostopov do pomnilnika med binarnim iskanjem. Iskanje rangov in zlivanje podblokov moramo zato izvajati v deljenem pomnilniku. Ponavljajoči elementi pomenijo težavo pri binarnih iskanjih, kar moramo posebej izpostaviti. Za ohranjanje stabilnosti moramo v sodih (oziroma lihih) blokih uporabiti *vključujoče* (ang. *inclusive*) binarno iskanje in v lihih (oziroma sodih) blokih *izključujoče* (ang. *exclusive*) binarno iskanje. Kakšno vrsto binarnega iskanja uporabljamo v posamezni vrsti blokov ni pomembno, pomembno je le, da uporabljamo isto iskanje (vključujoče ali izključujoče) v vseh podblokih



Slika 6.4: Zlivanje zaporedja, katerega dolžina ni potenca števila 2.

iste vrste (sodi ali lihi), kjer se mora vrsta binarnega iskanja med sodimi in lihimi bloki razlikovati [30].

Potrebno je tudi zagotoviti, da urejanje deluje za poljubno dolžino zaporedja. V ta namen razdelimo vhodno zaporedje na sledeča podzaporedja:

- **glavni del**, kateri ima dolžino prejšnjega večkratnika potence števila 2 dolžine tabele n oziroma n, kadar je dolžina tabele enaka potenci števila 2,
- **ostanek**, katerega dolžina je enaka razliki med dolžino tabele n in dolžino prej opisanega *glavnega dela*,
- **zapolnjen del**, kateri vsebuje minimalne oziroma maksimalne možne vrednostmi ($\pm\,\infty$, odvisno od smeri urejanja) do naslednjega večkratnika potence števila 2 dolžine zaporedja.

Slika 6.4 prikazuje urejanje tabele dolžine 6. Na sliki vidimo, da ima glavni del dolžino 4, ostanek 2 in zapolnjen del prav tako 2. Zelen okvir prikazuje del zaporedja, v katerem se izvaja zlivanje. Modri okvir prikazuje del zaporedja, katerega ne spreminjamo. Kot vidimo, glavni del zlivamo v vseh fazah urejanja. Nasprotno ostanek dolžine d zlivamo samo $f = \lceil \log_2 d \rceil$ faz, saj je po f fazah že urejen. Zapolnjen del zlivamo samo v zadnji fazi urejanja in ob zlivanju ostanka, katerega dolžina ni enaka potenci števila 2. Zlivanje celotnega zaporedja (vključno z zapolnjenim delom) izvedemo samo v zadnji fazi urejanja.

V vsaki fazi zlivanja se elementi zaporedja prenašajo med primarnim in

pomožnim pomnilnikom oziroma obratno. Kot smo omenili, se ostanek zaporedja dolžine d ureja samo $\lceil \log_2 d \rceil$ faz. Zaradi tega se lahko pred izvedbo zadnje faze zlivanja zgodi, da se glavni del in ostanek nahajata v različnih tabelah (primarna in pomožna tabela). V tem primeru je potrebna izvedba funkcije, ki kopira ostanek v tabelo, v kateri se nahaja glavni del (vrstica 19 psevdokode 6.2). Zapolnjenega dela nam ni potrebno kopirati, saj ščepec v vrstici 13 doda polnilo obema tabelama.

6.2.4 Časovna zahtevnost

V poglavjih 6.2.2.1 in 6.2.2.2 smo omenili, da moramo najprej poiskati vzorce in njihove pripadajoče range. Vzorce določimo kot vsak M-ti element zaporedja dolžine n. Vseh vzorcev v zaporedju je zato $O\left(\frac{n}{M}\right)$. Njihove range poiščemo s pomočjo binarnega iskanje v blokih dolžine M. Za vsak rang je zato potrebnih $\log M$ korakov binarnega iskanja. Časovna zahtevnost iskanja rangov vzorcev je zato enaka:

$$O\left(\frac{n}{M} \cdot \log M\right). \tag{6.3}$$

Potem je potrebno izvesti zlivanje blokov (poglavje 6.2.2.4). Določiti moramo še rang vseh elementov v zliti tabeli. To storimo s pomočjo binarnega iskanja v blokih dolžine $O(M)$, kar za vsak element zahteva $O(\log M)$ korakov. Časovna zahtevnost iskanja rangov vseh elementov tabele je zato enaka:

$$O\left(n \cdot \log M\right). \tag{6.4}$$

Vidimo, da časovna zahtevnost zlivanja (6.4) prevlada nad časovno zahtevnostjo iskanja rangov vzorcev (6.3). Kot smo omenili, lahko binarna iskanja izvajamo vzporedno, ker so ta medsebojno neodvisna. Vzporedno zlivanje lahko s p procesorji pohitrimo na:

$$O\left(\frac{n}{p} \cdot \log M\right). \tag{6.5}$$

Za urejanje zaporedja dolžine $n = 2^f$ je potrebnih $f = \log n$ faz. Faze zlivanja moramo izvajati zaporedno. Časovna zahtevnost vzporednega urejanja s p procesorji je enaka:

$$T_p = O\left(\frac{n}{p} \cdot \log n \cdot \log M\right). \tag{6.6}$$

Vrednost M je konstanta. Poleg tega je veliko manjša od $O(n)$. Zato jo lahko odstranimo iz časovne zahtevnosti:

$$T_p = O\left(\frac{n}{p} \cdot \log n\right). \tag{6.7}$$

Z $O(n)$ procesorji lahko časovno zahtevnost zmanjšamo na:

$$T_n = O(\log n). \tag{6.8}$$

Pohitritev vzporedne implementacije za p procesorjev je enaka:

$$S_p = \frac{T_1}{T_p} = O\left(\frac{n \cdot \log n}{\frac{n}{p} \cdot \log n}\right) = O(p). \tag{6.9}$$

Vidimo, da je pohitritev (6.9) optimalna, saj je enaka p. Pohitritev pri $O(n)$ procesorjih je enaka:

$$S_n = \frac{T_1}{T_n} = O\left(\frac{n \cdot \log n}{\log n}\right) = O(n). \tag{6.10}$$

Poglavje 7

Hitro urejanje

Hitro urejanje (ang. *quicksort*) velja za enega najučinkovitejših algoritmov za urejanje podatkov. Temelji na paradigmi *deli in vladaj* (ang. *divide-and-conquer*) [12].

- **Deli** - določimo *pivot*, na podlagi katerega razdelimo (ang. *partition*) vhodno zaporedje $T[p...r]$ na dve podzaporedji. Po opravljeni delitvi se mora pivot nahajati na mestu $T[q]$, vsi elementi prvega podzaporedja $T[p...q-1]$ morajo biti manjši ali enaki $T[q]$ in vsi elementi drugega podzaporedja $A[q+1...r]$ morajo biti večji od $T[q]$.
- **Vladaj** - z rekurzivnimi klici hitrega urejanja (prejšnji korak *deli*) uredimo podzaporedji $T[p...q-1]$ in $T[q+1...r]$.
- **Združi rezultate** - v prejšnjem koraku urejamo podzaporedja toliko časa, dokler ne dobimo podzaporedij dolžine 1. S tem uredimo tudi celotno zaporedje $T[p...r]$, zaradi česar korak združevanja ni potreben.

7.1 Zaporedni algoritem

7.1.1 Implementacija

Zaporedni algoritem hitrega urejanja je prostorsko učinkovit, ker za delovanje ne potrebuje pomožnih tabel (ang. *in-place*). Prikazan je v psevdokodi 7.1.

```
1  // tabela: vhodna tabela,
2  // n:      dolžina vhodne tabele,
3  // s:      smer urejanja (0: naraščajoče, 1: padajoče).
4  zaporednoHitroUrejanje(tabela, n, s):
5    if (n <= 1):
6      return
7
8    // Razdeli tabelo na elemente manjše in večje od pivota
9    delitev = razdeliTabelo(tabela, n, s)
10   // Rekurzivno uredi dobljeni tabeli
11   zaporednoHitroUrejanje(tabela, delitev, s)
12   zaporednoHitroUrejanje(tabela + delitev + 1, n - delitev - 1, s)
13
14 // Enaki vhodni parametri kot v funkciji zaporednoHitroUrejanje.
15 razdeliTabelo(tabela, n, s):
16   // Določi indeks pivota (npr. srednji element zaporedja)
17   indeksPivot = dolociIndeksPivota(tabela, n)
18   zamenjajElementa(tabela[indeksPivot], tabela[n - 1])
19
20   // Razdeli tabelo na elemente manjše in večje od pivota
21   for (j = 0, i = 0; j < n - 1; j++):
22     if ((s == NARASCAJOCE) XOR (tabela[j] > tabela[n - 1])):
23       zamenjajElementa(tabela[j], tabela[i])
24       i++
25
26   // Vstavi pivot med levo in desno podzaporedje
27   zamenjajElementa(tabela[i], tabela[n - 1])
28   return i
```

Psevdokoda 7.1: Zaporedno hitro urejanje (prirejeno po [12]).

Najprej razdelimo tabelo na elemente manjše ali večje od pivota (vrstica 9). Potem dobljeni tabeli rekurzivno uredimo. To ponavljamo, dokler ne dobimo tabel dolžine 1, ki so po definiciji urejene. Veliko vlogo pri učinkovitosti urejanja igra algoritem razdelitve oziroma izbira pivota (vrstica 17). Obstaja veliko algoritmov izbire pivota, kot so naključni element tabele, mediana k naključnih elementov tabele, itd. V naši implementaciji smo za vrednost pivota izbrali mediano prvega, srednjega in zadnjega elementa tabele. Izbrani pivot vstavimo na zadnjo mesto tabele (vrstica 18). Nato razdelimo tabelo na elemente manjše (levi del tabele) in elemente večje (desni del tabele) od pivota (vrstice 21 - 24). Po opravljeni delitvi vstavimo pivot med prej omenjeni podzaporedji (vrstica 27) in vrnemo njegov indeks (vrstica 28) [12].

7.1.2 Časovna zahtevnost

Učinkovitost hitrega urejanja je v veliki meri odvisna od izbire pivot (vrstica 17 v psevdokodi 7.1). Časovna zahtevnost delitve tabele dolžine n je

$\theta(n)$ (funkcija `razdeliTabelo`). Najprej preučimo najslabši možni primer delitve. Ta se pripeti, ko za vrednost pivota izberemo minimalno oziroma maksimalno vrednost. Takrat eno povzaporedje vsebuje $n-1$ elementov, medtem ko je drugo prazno. Predpostavimo, da dobimo najslabšo možno delitev ob vsakem rekurzivnem klicu (na primer tabela z enakimi vrednostmi). Izvajanje algoritma lahko zapišemo z rekurzivno enačbo [12]:

$$T(n) = T(n-1) + T(0) + \theta(n). \tag{7.1}$$

V vrsticah 5 in 6 psevdokode 7.1 vidimo, da se klici $T(0)$ zaključijo v konstantnem času $O(1)$. Enačbo lahko poenostavimo:

$$T(n) = T(n-1) + \theta(n). \tag{7.2}$$

S pomočjo metode zamenjave (ang. *substitution method*) lahko določimo časovno zahtevnost [12].

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= 1 + 2 + ... + n - 1 + n = \sum_{i=1}^{n} i \\ &= O(n^2) \end{aligned} \tag{7.3}$$

Preučimo še najboljši primer delitve, v katerem dobimo podzaporedji dolžine $\lfloor \frac{n}{2} \rfloor$ in $\lceil \frac{n}{2} \rceil - 1$. Predpostavimo, da tekom urejanja vedno dobimo najboljšo delitev. Izvajanje algoritma lahko zapišemo z rekurzivno enačbo:

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n). \tag{7.4}$$

Po *izreku o asimptotičnem obnašanju rešitve rekurenčne enačbe* (ang. *master theorem*) ima rekurzivna enačba (7.4) rešitev [12]:

$$T(n) = O(n \cdot \log n). \tag{7.5}$$

Povprečna časovna zahtevnost hitrega urejanja je veliko bližje najboljšemu kot najslabšemu primeru delitve. Vzemimo za primer delitev v razmerju 9 proti 1. V tem primeru bo imelo rekurzivno drevo globino $\log_{10/9} n = \theta(\log n)$. Ugotovimo, da katerakoli konstantna delitev privede do rekurzivnega drevesa globine $\theta(\log n)$, pri čemer je zahtevnost v vsakem nivoju drevesa $O(n)$. Sklepamo, da je povprečna časovna zahtevnost algoritma enaka (formalni dokaz se nahaja v [12]):

$$T_1 = O(n \cdot \log n). \tag{7.6}$$

7.2 Vzporedni algoritem

Vzporedno hitro urejanje deluje zelo podobno kot zaporedno. Izberemo pivot in nato izvedemo *delitev* - vse elemente manjše od pivota premaknemo na levo stran pivota in vse elemente večje od pivota na desno stran. Vzporedni algoritem je sestavljen iz *dveh faz*. V *prvi fazi* algoritma so podzaporedja zelo dolga, zato jih mora obdelati več blokov niti. Po določenem času (v *drugi fazi*) dobimo dovolj veliko število podzaporedij oziroma so podzaporedja dovolj kratka, da lahko vsak blok niti izvaja delitev nad enim podzaporedjem. Ko dolžina podzaporedja pade pod določeno mejo, postane operacija delitve preveč potratna. Iz tega razloga kratka podzaporedja uredimo z bitonim urejanjem [9].

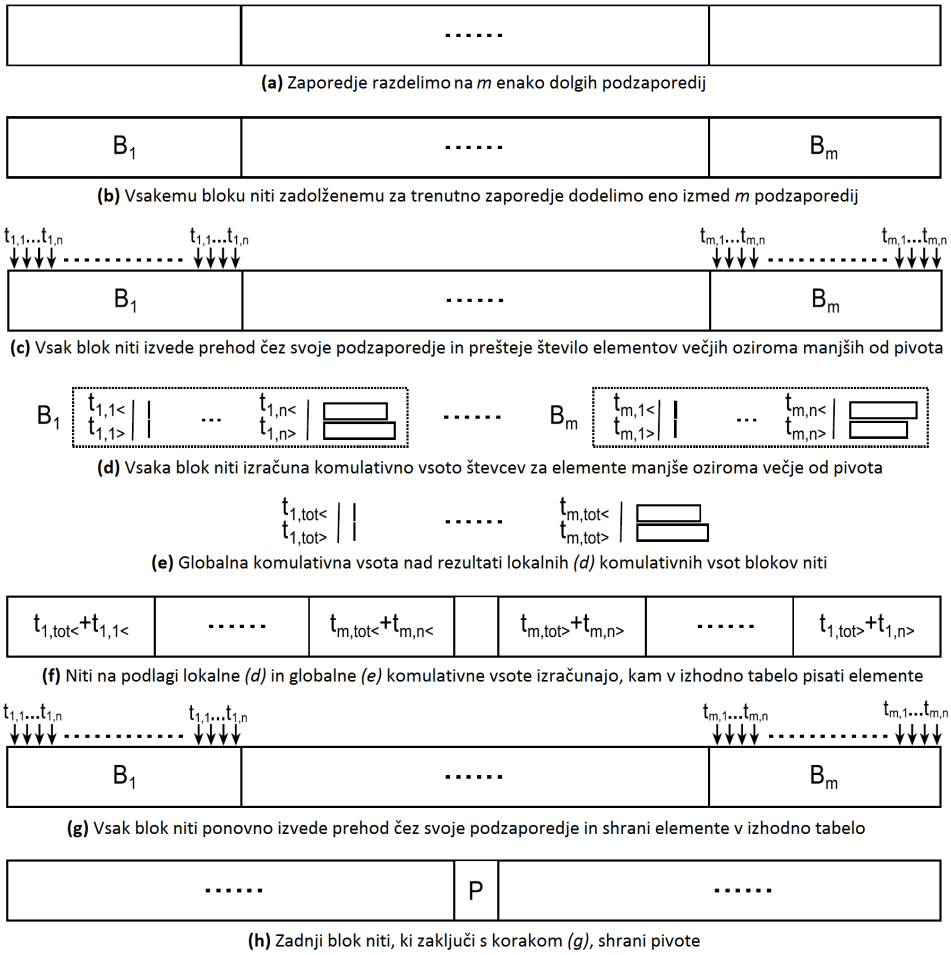
7.2.1 Delitev v dveh prehodih

V prvi fazi hitrega urejanja mora več blokov niti razdeliti isto zaporedje. Težava je v tem, da morajo niti ob opravljanju delitve vedeti, kam v izhodno

tabelo pisati svoje pripadajoče elemente. Možna rešitev bi bila, če bi ob *vsakem* branju iz globalnega pomnilnika s pomočjo komulativne vsote izračunali, koliko niti želi pisati na levo oziroma desno stran pivota. S tem bi vsaka nit vedela, da bo x niti z nižjim identifikatorjem niti pisalo levo in y niti z nižjim identifikatorjem niti pisalo na desno stran pivota. Glede na vrednost svojega pripadajočega elementa, bi vsaka nit lahko zapisala svoj element na mesto $x + 1$ oziroma $n - (y + 1)$. Opisan pristop je slab, saj operacija komulativne vsote zahteva veliko časa [9].

Da se izognemo zgoraj opisanim težavam potrebujemo algoritem *delitve v dveh prehodih* (ang. *two pass partition*), prikazan na sliki 7.1. Zaporedje najprej razdelimo na m enako dolgih podzaporedij (korak a). Dolžina podzaporedij mora biti večkratnik števila niti v blokih. Nato blokom niti dodelimo podzaporedja (korak b), nad katerimi izvedejo prvi prehod (korak c). V prehodu vsaka nit prebere svoje pripadajoče elemente in prešteje, koliko elementov je manjših oziroma večjih od pivota. Za tem vsak blok niti izračuna lokalno komulativno vsoto nad dobljenimi števci (korak d). Kot smo omenili, v prvi fazi urejanja več blokov niti obdeluje isto zaporedje. Zato je potrebno opraviti še globalno komulativno vsoto nad rezultati lokalnih vsot (korak e). Namesto tega lahko uporabimo tudi atomarne operacije (podrobneje v poglavju 7.2.3). V drugi fazi algoritma ta korak ni potreben, saj vsak blok niti obdeluje le eno podzaporedje. Niti na podlagi lokalne in globalne komulativne vsote izračunajo, kam morajo zapisovati svoje pripadajoče elemente (korak f). Nato izvedejo drugi prehod čez svoje pripadajoče podzaporedje, pri čemer elemente shranijo v izhodno tabelo na levo oziroma desno stran pivota (korak g). Zadnji blok, ki zaključi s korakom g, na koncu shrani še pivote (korak h) [9].

Prednost delitve v dveh prehodih je v tem, da komulativno vsoto izračunamo samo enkrat. Poleg tega ni potrebe po sinhronizaciji niti ob vsakem branju iz globalnega pomnilnika. Kot slabost lahko navedemo, da niti berejo iste podatke dvakrat, vendar je omenjena slabost veliko manjša od prej naštetih prednosti.



Slika 7.1: Delitev v dveh prehodih (prirejeno po [9]).

7.2.2 Implementacija na gostitelju

Kot smo omenili, je algoritem sestavljen iz *dveh faz*. V *prvi fazi* so zaporedja zelo dolga, zaradi česar jih mora obdelati več blokov niti. To dosežemo s pomočjo algoritma *globalnega hitrega urejanja*. Ko postanejo zaporedja dovolj kratka (*druga faza* algoritma) in lahko posamezno zaporedje obdela en blok niti, uporabimo algoritem *lokalnega hitrega urejanje*.

Psevdokoda 7.2 prikazuje hitro urejanje na gostitelju. Najprej s pomočjo ščepca za vzporedno redukcijo poiščemo minimalno in maksimalno vrednost zaporedja (vrstica 9). Pri tem smo uporabili implementacijo optimizirane

```
1  // tab:           vhodna tabela,
2  // tabPom:        pomožna tabela,
3  // n:             dolžina vhodne tabele,
4  // minDolzinaZap: minimalna dolžina zaporedja za globalno hitro urejanje,
5  // velikostBloka: dolžina podzaporedja, katerega obdela en blok niti v ščepcu
6  //                za globalno hitro urejanje.
7  vzporednoHitroUrejanje(tab, tabPom, n, minDolzinaZap, velikostBloka):
8    // Z redukcijo poišče minimalno in maksimalno vrednost v zaporedju
9    min, maks = scepecMinMaksRedukcija(tab, tabPom, n)
10   if min == maks: return
11
12   // Števec števila zaporedij in konstanta maksimalnega števila zaporedij
13   stZap, maksStZap = 1, (n - 1) / minDolzinaZap + 1
14   // Zaporedja, ki jih mora razdeliti GLOBALNO hitro urejanje
15   globZap = [{zacetek=0, dolzina=n, min=min, maks=maks, smer=PRIM_POM}]
16   // Zaporedja, ki jih mora razdeliti in urediti LOKALNO hitro urejanje
17   lokZap = []
18
19   while globZap != []:
20     // Naprava potrebuje drugačne metapodatke o zaporedjih kot gostitelj
21     zapNaprava, indeksiZapNaprava, indeksBloka = [], [], 0
22
23     for (indeksZap = 0; indeksZap < ||globZap||; indeksZap++):
24       // Število blokov niti, ki obdela zaporedje
25       stBlokovNaZap = (globZap[indeksZap].dolzina - 1) / velikostBloka + 1
26       zapNaprava += {
27         globZap[indeksZap], stElManjsihOdPivota=0, stElVecjihOdPivota=0,
28         pivot=(globZap[indeksZap].min + globZap[indeksZap].maks) / 2,
29         maksElManjsiOdPivota=MINIMUM, minElVecjiOdPivota=MAKSIMUM,
30         zacetniIndeksBloka=indeksBloka, stevecBlokovNaZap=stBlokovNaZap
31       }
32
33       // Za vsak blok niti označi, katero zaporedje mora obdelati
34       for (blok = 0; blok < stBlokovNaZap; blok++):
35         indeksiZapNaprava[indeksBloka++] = indeksZap
36
37     scepecGlobalnoHitroUrejanje(tab, tabPom, zapNaprava, indeksiZapNaprava)
38     globZap = []
39
40     // Na podlagi delitve določi nova globalna in lokalna zaporedja
41     for z in zapNaprava:
42       // Podzaporedje manjše od pivota - levi del delitve
43       if z.stElManjsihOdPivota > minDolzinaZap && stZap++ < maksStZap:
44         globZap += {
45           zacetek=z.zacetek, dolzina=z.stElManjsihOdPivota, min=z.min,
46           maks=z.maksElManjsiOdPivota, smer=(!z.smer)
47         }
48       else if z.stElManjsihOdPivota > 0:
49         lokZap += // Enako kot globalno zaporedje, le brez "min" in "maks"
50
51       // Podzaporedje večje od pivota - desni del delitve
52       if z.stElVecjihOdPivota > minDolzinaZap && stZap++ < maksStZap:
53         globZap.dodaj({
54           zacetek=z.zacetek + z.dolzina - z.stElVecjihOdPivota,
55           dolzina=z.stElVecjihOdPivota, min=z.minElVecjiOdPivota,
56           maks=z.maks, smer=(!z.smer)
57         })
58       else if zap.stElVecjihOdPivota > 0:
59         lokZap.dodaj(/* Enako kot "globZap", le brez "min" in "maks" */mas)
60
61   if n < minDolzinaZap: lokZap = [{zacetek=0, dozina=n, smer=PRIM_POM}]
62   scepecLokalnoHitroUrejanje(tab, tabPom, lokZap)
```

Psevdokoda 7.2: Vzporedno hitro urejanje na gostitelju (prirejeno po [9]).

redukcije, ki je opisana v poglavju 2.2.3. V primeru enakih vrednosti gre za ničelno porazdelitev (vrstica 10). Vrednosti sta potrebni za izračun začetnega pivota celotnega zaporedja. Zelo pomemben parameter algoritma je `minDolzinaZap`. Ta predstavlja minimalno dolžino zaporedja, ki ga še lahko obdelamo s ščepcem za globalno hitro urejanje. Krajša zaporedja obdelamo z lokalnim hitrim urejanjem. Na podlagi omenjenega parametra določimo maksimalno število zaporedij, ki jih lahko tvorimo z globalnim urejanjem (vrstica 13). Nato definiramo seznama z metapodatki zaporedij za globalno in lokalno urejanje. Seznam zaporedij za globalno urejanje vsebuje celotno vhodno zaporedje (vrstica 15), ki se na začetku nahaja v primarni tabeli. V ta namen nastavimo smer prenosa podatkov ob izvajanju delitve na `PRIM_POM` (iz primarne v pomožno tabelo). Seznam zaporedij za lokalno urejanje je na začetku prazen (vrstica 17) [9].

Za tem sledi globalno hitro urejanje (vrstice 19 - 59). Naprava potrebuje drugačne metapodatke o zaporedjih kot gostitelj, zato definiramo nov seznam metapodatkov `zapNaprava` (vrstica 21). Pomemben parameter algoritma je tudi `velikostBloka`. Ta predstavlja število elementov, ki jih lahko obdela en blok niti v ščepcu za globalno urejanje. Smiselno je, da je večkratnik števila niti v bloku. S tem dosežemo, da vsaka nit obdela več in obenem tudi enako število elementov. S pomočjo omenjenega parametra izračunamo število blokov niti potrebnih za obdelavo enega zaporedja (vrstica 25). Za vsako zaporedje na napravi shranimo poleg metapodatkov potrebnih na gostitelju `globZap[indeksZap]` še dodatne metapodatke:

- `stElManjsihOdPivota` oziroma `stElVecjihOdPivota` - števca števila elementov, ki so manjši oziroma večji od pivota,
- `pivot` - pivot trenutnega zaporedja, ki ga izračunamo kot povprečje minimuma in maksimuma zaporedja [34],
- `maksElManjsiOdPivota` oziroma `minElVecjiOdPivota` - največji element manjši oziroma najmanjši elementov večji od pivota,
- `zacetniIndeksBloka` - indeks prvega bloka, ki je zadolžen za trenutno zaporedje,

- `stevecBlokovNaZap` - število blokov niti, ki obdeluje zaporedje.

Našteti metapodatki (vrstice 26 - 31) so potrebni za izvedbo ščepca globalnega urejanja (podrobneje v poglavju 7.2.3.1), in kasneje za tvorjenje novih zaporedij (vrstice 41 - 59). Definiramo tudi seznam `indeksiZapNaprava`, v katerega shranimo indekse zaporedij, ki jih morajo obdelati posamezni bloki niti (vrstica 21). Omenjen seznam napolnimo v vrsticah 34 in 35.

V naslednjem koraku pokličemo ščepec za globalno urejanje, ki izvede delitev zaporedij (vrstica 37). Ščepec poleg tega za vsako zaporedje poišče maksimum elementov manjših oziroma minimum elementov večjih od pivota in število elementov manjših oziroma večjih od pivota. Slednji par vrednosti pravzaprav določa dolžino podzaporedij dobljenih s pomočjo delitve. Predolga podzaporedja je potrebno ponovno dodati v seznam za globalno urejanje, medtem ko kratka zaporedja dodamo v seznam za lokalno urejanje (vrstice 41 - 59). Ko dosežemo maksimalno število zaporedij za globalno urejanje, dodamo vsa preostala zaporedja v seznam za lokalno urejanje. Na tej točki je potrebno omeniti, da v metapodatkih zaporedij za lokalno urejanje ne potrebujemo minimuma in maksimuma zaporedja, saj določamo pivot na drugačen način (podrobneje v poglavju 7.2.3.2). Po prvi iteraciji globalnega urejanja, se bodo vsa novo ustvarjena zaporedja nahajala v pomožni tabeli. Po drugi iteraciji se bodo nova zaporedja nahajala v primarni tabeli, itd. Iz tega razloga je potrebno vsem novim zaporedjem obrniti smer prenosa podatkov (vrstici 46 in 56).

Po določenem številu iteracij je seznam zaporedij za globalno hitro urejanje prazen. Takrat nad zaporedji izvedemo lokalno hitro urejanje, ki se v celoti izvede na GPE (vrstica 62) [9].

7.2.3 Implementacija na napravi

7.2.3.1 Globalno hitro urejanje

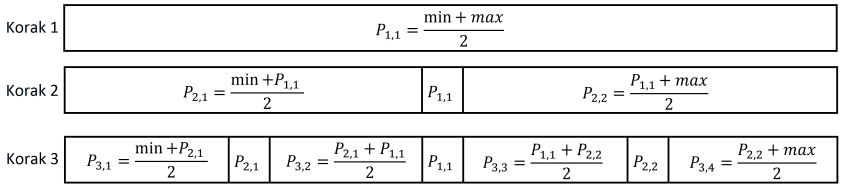
Psevdokoda 7.3 prikazuje ščepec za globalno hitro urejanje. Najprej preberemo metapodatke pripadajočega zaporedja (vrstica 6). Na podlagi metapo-

```
1  // tab:        vhodna tabela,
2  // tabPom:     pomožna tabela,
3  // zaporedja:  tabela z metapodatki zaporedij,
4  // indeksiZap: za vsak blok niti vsebuje indeks zap., katerega mora obdelati.
5  scepecGlobalnoHitroUrejanje(tab, tabPom, zaporedja, indeksiZap):
6    zap = zaporedja[indeksiZap[blockIdx.x]]
7    // Izračuna meje podzaporedja, katerega obdela trenutni blok niti
8    zacetekPodz, konecPodz = izracunajMejePodzaporedja(zap.zacetniIndeksBloka)
9    lTab, lTabPom = dolociSmerPrenosa(tab, tabPom, zap.smer)
10
11   stManjsih, stVecjih, __shared__ globStManjsih, __shared__ globStVecjih = 0
12   lMin, lMaks = MAKSIMUM, MINIMUM
13
14   for (i = zacetekPodz; i < konecPodz; i += blockDim.x):
15     // Vsaka nit prešteje število elementov manjših oziroma večjih od pivota
16     sStManjsih += lTab[i] < zap.pivot, sStVecjih += lTab[i] > zap.pivot
17     // Iščemo maks. el. manjši od pivota in min. el. večji od pivota
18     lMaks = max(lMaks, lTab[i] < zap.pivot ? lTab[i] : MINIMUM)
19     lMin = min(lMin, lTab[i] > zap.pivot ? lTab[i] : MAKSIMUM)
20   __syncthreads()
21
22   // Izvedemo redukcijo nad minimumi in maksimumi vseh niti bloka
23   rMin, rMaks = minRedukcija(lMin), maksRedukcija(lMaks)
24   // S pomočjo atomarnih operacij poiščemo min. in maks. trenutnega zaporedja
25   if threadIdx.x == 0:
26     atomicMin(zap.minElVecjiOdPivota, rMin)
27     atomicMax(zap.maksElManjsiOdPivota, rMaks)
28   __syncthreads()
29
30   // Komulativna vsota nad števci št. elementov manjših oz. večjih od pivota
31   vsotaManjsi, vsotaVecji = komulVsota(stManjsih), komulVsota(stVecjih)
32   __syncthreads()
33
34   // S pomočjo atomarnih operacij izračuna globalni odmik za trenutno zap.
35   if threadIdx.x == blockDim.x - 1:
36     globStManjsih = atomicAdd(zap.stElManjsihOdPivota, vsotaManjsi)
37     globStVecjih = atomicAdd(zap.stElVecjihOdPivota, vsotaVecji)
38   __syncthreads()
39
40   // Izvede delitev - shrani elemente na levo in desno stran pivota
41   indeksManjsi = zap.zacetek + globStManjsih + vsotaManjsi - stManjsih
42   indeksVecji = zap.zacetek + zap.dolzina - globStVecjih - vsotaVecji
43
44   for (i = zacetekPodz; i < konecPodz; i += blockDim.x):
45     if lTab[i] < zap.pivot: lTabPom[indeksManjsi++] = lTab[i]
46     if lTab[i] > zap.pivot: lTabPom[indeksVecji++] = lTab[i]
47
48   if threadIdx.x == blockDim.x - 1:
49     stevecBlokovNiti = atomicSub(zap.stevecBlokovNaZap, 1) - 1
50   __syncthreads()
51
52   // Zadnji blok niti zadolžen za trenutno zaporedje shrani še pivote
53   if stevecBlokovNiti == 0:
54     indeksPivot = zap.zacetek + zap.stElManjsihOdPivota + threadIdx.x
55     indeksKonec = zap.zacetek + zap.dolzina - zap.stElVecjihOdPivota
56
57     for (; indeksPivot < indeksKonec; indeksPivot += blockDim.x):
58       tabPom[indeksPivot] = zap.pivot
```

Psevdokoda 7.3: Ščepec za globalno hitro urejanje (prirejeno po [9]).

datkov izračunamo odsek zaporedja, ki ga mora obdelati trenutni blok niti (vrstica 8). Glede na smer prenosa podatkov (iz globalnega pomnilnika v pomožni ali obratno) ustrezno nastavimo kazalca na primarno in pomožno tabelo (vrstica 9). Nato niti preberejo vse svoje pripadajoče elemente, pri čemer prištejejo število elementov manjših oziroma večjih od pivota (vrstice 14 - 19). Obenem vsaka nit tudi shrani maksimum elementov manjših in minimum elementov večjih od pivota. Za tem s pomočjo vzporedne redukcije v deljenem pomnilniku poiščemo minimum in maksimum celotnega bloka niti (vrstica 23). To izvedemo s pomočjo osnovne vzporedne redukcije opisane v poglavju 2.2.1. Za omenjeno redukcijo smo se odločili, ker za izvedbo potrebuje samo $\frac{n}{2}$ niti. To pomeni, da smo lahko minimum in maksimum poiskali vzporedno. S pomočjo atomarnih operacij shranimo dobljeni vrednosti za trenutno zaporedje (vrstice 25 - 27). Nato v deljenem pomnilniku izračunamo komulativno vsoto nad prej omenjenimi števci elementov manjših oziroma večjih od pivota (vrstica 31). Za izračun uporabimo algoritem optimizirane komulativne vsote (poglavje 2.3.2). S pomočjo vsote vsaka nit ugotovi, koliko elementov manjših oziroma večjih od pivota so našle niti z nižjim identifikatorjem. Posledično niti vejo, kam v izhodno tabelo morajo zapisati svoje elemente ob opravljanju delitve. Opozoriti velja, da več blokov niti obdeluje isto zaporedje. Zato morajo niti vedeti, koliko elementov je bilo manjših oziroma večjih od pivota v blokih, ki so že opravili delitev nad istim zaporedjem. To izračunamo s pomočjo atomarnega seštevanja (vrstice 35 - 37). Na podlagi dobljenih vrednosti lahko algoritem za vsako nit določi indeks pomožne tabele, kamor mora shranjevati elemente manjše ali večje od pivota (vrstici 41 in 42). Nadalje niti opravijo drug prehod čez svoje podatke, v katerem izvedejo delitev (vrstice 44 - 46). Zadnji blok niti, ki obdela zaporedje, shrani tudi pivote (vrstice 54 - 58). Pri tem je potrebno poudariti, da moramo pivote shraniti v končno oziroma pomožno tabelo (ne glede na smer urejanja), saj so ti že urejeni in jih ni več potrebno prestavljati [9].

Ključ za delovanje opisanega ščepca so atomarne operacije. V primeru starejših GPE, ki ne podpirajo atomarnih operacij, je potrebno v vrstici 32



Slika 7.2: Postopek za iskanje pivota v prvih treh korakih hitrega urejanja.

zaključiti izvajanje ščepca. Poleg tega ni potrebno izvesti atomarnega minimuma in maksimuma v vrsticah 25 - 27. Pred zaključkom ščepca moramo v globalni pomnilnik shraniti minimum in maksimum (vrstica 23) ter rezultata komulativnih vsot števcev elementov manjših oziroma večjih od pivota (vrstica 31). Potem pokličemo ščepec, ki nad naštetimi vrednostmi izračuna komulativno vsoto oziroma izvede redukcijo za iskanje minimuma in maksimuma. Na koncu pokličemo ščepec za globalno hitro urejanje od vrstice 41 dalje. Pri tem vsak blok niti prebere svoj pripadajoč rezultat komulativne vsote. S tem ugotovi število elementov manjših oziroma večjih od pivota v blokih niti z nižjim identifikatorjem, ki obdelujejo isto zaporedje (v implementaciji z atomarnimi operacijami to določimo v vrsticah 35 - 37). Preostanek ščepca izvedemo enako, kot prikazuje psevdokoda 7.3. Kot vidimo moramo v primeru starejših GPE brez podpore atomarnih operacij izvesti tri klice ščepcev namesto enega [9].

Kot smo omenili, vrednost pivota vsakega zaporedja določimo kot povprečje njegovega minimuma in maksimuma (psevdokoda 7.2, vrstica 28). Zaradi tega je v ščepcu za globalno hitro urejanje potrebno poiskati minimum in maksimum, kar zahteva izvedbo redukcije. Slabost opisanega pristopa je v tem, da izvedba redukcije zahteva določen čas. V ta namen smo izdelali učinkovitejši postopek za iskanje pivota, kot to prikazuje slika 7.2. Kot vidimo, začetni pivot $P_{1,1}$ še vedno izračunamo kot povprečje minimuma in maksimuma. Po prvi opravljeni delitvi se vsi elementi manjši od $P_{1,1}$ nahajajo na levi strani $P_{1,1}$, večji pa na desni. To dejstvo izkoristimo pri izračunu pivotov $P_{2,1}$ in $P_{2,2}$ ter pri izračunu vseh nadaljnjih pivotov, kot to prikazuje slika 7.2. Opisan pristop se pri večini porazdelitvah izkaže za približno

```
1  // tab:         vhodna tabela,
2  // tabPom:      pomožna tabela,
3  // zaporedja:   tabela z metapodatki zaporedij.
4  scepecLokalnoHitroUrejanje(tab, tabPom, zaporedja):
5    sklad.dodajNaVrhSklada(zaporedja[blockIdx.x])
6
7    while ||sklad|| > 0:
8      __syncthreads()
9      zap = sklad.vrniZgornjiElementSklada()
10     if zap.dolzina <= MEJA_BITONO_UREJANJE:
11       normBitonoUrejanje(zap.smer = PRIM_POM ? tab : tabPom, tabPom, zap)
12       continue
13
14     lTab, lTabPom = dolociSmerPrenosa(tab, tabPom, zap.smer)
15     // Določi vrednost pivota kot mediano prvega, srednjega in zadnjega el.
16     pivot = dolociPivotMediana(lTab, zap)
17     stManjsih, stVecjih = 0
18
19     // Vsaka nit prešteje število elementov manjših oziroma večjih od pivota
20     for (i = zap.zacetek; i < zap.zacetek + zap.dolzina; i += blockDim.x):
21       sStManjsih += lTab[i] < zap.pivot, sStVecjih += lTab[i] > zap.pivot
22
23     // Komulativna vsota nad števci št. el. manjših oz. večjih od pivota
24     vsotaManjsi, vsotaVecji = komulVsota(stManjsih), komulVsota(stVecjih)
25     __syncthreads()
26
27     // Izvede delitev - shrani elemente na levo in desno stran pivota
28     indeksManjsi = zap.zacetek + vsotaManjsi + stManjsih
29     indeksVecji = zap.zacetek + zap.dolzina - vsotaVecji
30     for (i = zap.zacetek; i < zap.zacetek + zap.dolzina; i += blockDim.x):
31       if lTab[i] < zap.pivot: lTabPom[indeksManjsi++] = lTab[i]
32       if lTab[i] > zap.pivot: lTabPom[indeksVecji++] = lTab[i]
33
34     if threadIdx.x == blockDim.x - 1:
35       // Doda na sklad levo in desno zap., pri čemer je krajše zap. na vrhu
36       sklad.dodajLevoInDesnoZap(sklad, zap, pivot, vsotaManjsi, vsotaVecji)
37       // Vse niti dobijo število elementov manjših in večjih od pivota
38       __shared__ pivotManjsi, pivotVecji = vsotaManjsi, vsotaVecji
39     __syncthreads()
40
41     // Shrani pivote v izhodno tabelo
42     indeksPivot = zap.zacetek + pivotManjsi + threadIdx.x
43     indeksKonec = zap.zacetek + zap.dolzina - pivotVecji
44     for (; indeksPivot < indeksKonec; indeksPivot += blockDim.x):
45       tabPom[indeksPivot] = zap.pivot
```

Psevdokoda 7.4: Ščepec za lokalno hitro urejanje (prirejeno po [9]).

10 % hitrejšega kot iskanje minimuma in maksimuma ob vsakem klicu ščepca za globalno hitro urejanje.

7.2.3.2 Lokalno hitro urejanje

Psevdokoda 7.4 prikazuje ščepec za lokalno hitro urejanje. Kot vidimo, deluje zelo podobno kot globalno urejanje. Ponovno preštejemo število elementov

manjših oziroma večjih od pivota, izračunamo komulativno vsoto nad števci ter na koncu elemente in pivote tudi shranimo. Opazimo razliko, da ni več potrebe po atomarnem seštevanju za izračun globalnega odmika, ker vsako blok niti obdela samo eno zaporedje. Poleg tega ni več potrebe po vzporedni redukciji za iskanje minimuma in maksimuma, saj vrednost pivota določimo kot mediano prvega, srednjega in zadnjega elementa (vrstica 16) [9, 34].

Izpostaviti velja, da je hitro urejanje rekurzivni algoritem, v katerem zaporedja razbijamo na krajša podzaporedja. To pomeni, da bi morali shranjevati metapodatke o številnih zaporedjih. Zato moramo poskrbeti, da vedno obdelamo najkrajše zaporedje. S tem zagotovimo, da nam ne bo nikoli potrebno hraniti več kot $O(\log_2 n)$ zaporedij. To dosežemo s pomočjo sklada. Ob dodajanju zaporedij na sklad moramo zagotoviti, da je krajše zaporedje na vrhu sklada (vrstica 36) [9, 21, 32].

Hitro urejanje je neučinkovito za urejanje kratkih zaporedij. Ko zaporedja postanejo dovolj kratka, da jih shranimo v deljen pomnilnik, jih lahko uredimo z drugim algoritmom. Odločili smo se za normalizirano bitono urejanje, ker je zelo učinkovito za urejanje kratkih zaporedij (vrstica 11). Izvedemo ga v deljenem pomnilniku. Pri tem moramo paziti, da urejeno zaporedje shranimo v izhodno tabelo (ne glede na smer urejanje), saj je zaporedje že urejeno. Enako velja tudi za shranjevanje pivotov (vrstice 42 - 45) [9, 21].

7.2.4 Posebnosti implementacije

Zaradi enostavnejšega prikaza algoritma smo se v psevdokodah 7.2, 7.3 in 7.4 osredotočili samo na prikaz urejanja v naraščajočem vrstnem redu. Za urejanje v padajočem vrstnem redu bi morale niti v globalnem in lokalnem hitrem urejanju shranjevati elemente manjše od pivota na desno stran pivota in elemente večje ne levo stran. Tudi bitono urejanje bi moralo urejati v obratnem vrstnem redu. Poleg tega bi morali v psevdokodi 7.2 zamenjati vrednosti minimuma in maksimuma med levim in desnim zaporedjem (vrstice 45 in 46 ter 55 in 56).

S shranjevanjem pivotov v ščepcih za lokalno in globalno hitro urejanje za-

polnimo praznino med levim in desnim podzaporedjem. Pri tem morajo niti poznati samo pivot. Nasprotno od tega moramo pri urejanju parov ključ-vrednost poleg ključev shraniti tudi vrednosti pivotov. Da to dosežemo, potrebujemo pomožno tabelo dolžine celotnega zaporedja, v katero niti shranjujejo vrednosti elementov, katerih ključ je enak pivotu. Ko niti izvajajo delitev zaporedja, morajo ob tem shranjevati vrednosti pivotov v pomožno tabelo. Za shranjevanje morajo vnaprej poznati število pivotov v njihovih pripadajočih elementih. Kot vemo vsaka nit prešteje število elementov manjših oziroma večjih od pivota. Iz omenjenih števcev in njihovih komulativnih vsot lahko za vsako nit ugotovimo, koliko pivotov so naštele niti z nižjim identifikatorjem. S tem posledično lahko izračunamo, kam v pomožno tabelo mora vsaka nit shranjevati vrednosti svojih pivotov. Po opravljeni delitvi lahko niti shranijo pivote v izhodno tabelo. To storijo tako, da kopirajo vrednosti pivotov iz pomožne v izhodno tabelo. V nasprotju z vrednostmi, ključe samo shranijo oziroma kopirajo, saj so ti enaki pri vseh pivotih. Opisan algoritem potrebuje majhno dopolnitev za globalno hitro urejanje, ker več blokov niti obdeluje isto zaporedje. Vsak blok niti mora poznati število pivotov v blokih, ki so že obdelali zaporedje. Zato je v metapodatkih vsakega zaporedja potreben še števec pivotov. S pomočjo atomarnega seštevanja lahko vsak blok niti poveča omenjen števec (podobno kot povečevanje števca elementov manjših oziroma večjih od pivota v vrsticah 36 in 37 psevdokode 7.3).

7.2.5 Časovna zahtevnost

Časovna zahtevnost začetne redukcije je enaka $O(\frac{n}{p} \cdot \log n)$ (2.1). Iz stališča časovne zahtevnosti je delo opravljeno v prvi (globalno hitro urejanje) in drugi (lokalno hitro urejanje) fazi urejanja enako. Vsaka delitev v obeh fazah zahteva dva prehoda čez celotno zaporedje dolžine n. V prvem prehodu niti preštejejo število elementov manjših oziroma večjih od pivota in nato v drugem prehodu shranijo podatke na levo oziroma desno stran pivota. Ob uporabi p procesorjev imata prehoda časovno zahtevnost $O(\frac{n}{p})$. Operacije komulativne vsote ter redukcija za iskanje minimuma in maksimuma niso

odvisne od n ali p, ampak le od velikosti bloka niti t. Časovna zahtevnost omenjenih operacij je zato konstantna in je enaka $O(\log t)$ oziroma $O(1)$. Bitono urejanje mora urediti samo zaporedja, ki so krajša od meje m. Omenjena meja je odvisna le od velikosti deljenega pomnilnika GPE. V splošnem velja $m << n$. To pomeni, da časovna zahtevnost bitonega urejanja ni odvisna od n ali p in je konstantna. Ugotovimo, da je časovna zahtevnost ene iteracije hitrega urejanja enaka $O(\frac{n}{p})$.

V poglavju 7.1.2 smo ugotovili, da je povprečna globina rekurzivnega drevesa oziroma povprečno število iteracij hitrega urejanja enako $\theta(\log n)$. Iz opisanega lahko sklepamo, da je časovna zahtevnost vzporednega hitrega urejanja za p procesorjev enaka [9]:

$$T_p = O\left(\frac{n}{p} \cdot \log n\right). \tag{7.7}$$

Z $O(n)$ procesorji lahko časovno zahtevnost zmanjšamo na:

$$T_n = O\left(\log n\right). \tag{7.8}$$

Pohitritev vzporedne implementacije za p procesorjev je enaka:

$$S_P = \frac{T_1}{T_p} = O\left(\frac{n \cdot \log n}{\frac{n}{p} \cdot \log n}\right) = O(p). \tag{7.9}$$

Kot vidimo, dosežemo pohitritev za faktor p, kar je idealna pohitritev. Pohitritev pri $O(n)$ procesorjih je enaka:

$$S_n = \frac{T_1}{T_n} = O\left(\frac{n \cdot \log n}{\log n}\right) = O(n). \tag{7.10}$$

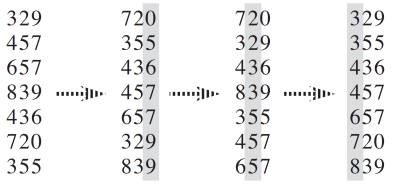
Poglavje 8

Urejanje po delih

Urejanje po delih (ang. *radix sort*) velja za enega najhitrejših algoritmov za urejanje. Je tudi edini algoritem v magistrskem delu, ki ne temelji na primerjavah. To posledično pomeni, da ni generičen kot ostali predstavljeni algoritmi. Ob spremembi podatkovnega tipa (npr. števila, datumi, besede, itd.) je potrebno pri ostalih algoritmih le spremeniti funkcijo primerjave, medtem ko je potrebno pri omenjenem urejanju prilagoditi algoritem [12].

8.1 Zaporedni algoritem

Algoritem urejanja po delih deluje tako, da podatke (števila, datume, besede, itd.) razdeli na d delov. V primeru števil, te razdeli na d r-bitnih števk. Števila nato uredi od najmanj do najbolj pomembne števke. Algoritem urejanja števk mora biti stabilen, ker moramo obdržati vrstni red števil s ponavljajočimi števkami. To posledično pomeni, da je tudi algoritem urejanja po delih stabilen. Po d opravljenih prehodih je zaporedje števil urejeno. Primer opisanega algoritma je prikazan na sliki 8.1. Števila najprej uredimo po zadnji oziroma najmanj pomembni števki, nato po srednji in na koncu po prvi oziroma najpomembnejši števki. Za hitro delovanje algoritma je ključno, da uporabimo učinkovit algoritem za urejanje števk. Zato smo se odločili za *urejanje s štetjem* (ang. *counting sort*) [12].



Slika 8.1: Primer urejanja po delih sedmih trimestnih števil [12].

8.1.1 Urejanje s štetjem

Urejanje s štetjem deluje na podlagi predpostavke, da je vsak izmed n vhodnih elementov celo število (ang. *integer*) na intervalu med 0 in k. Delovanje algoritma je prikazano na sliki 8.2.

a) Tabela A dolžine $n = 8$ vsebuje vhodno neurejeno zaporedje nepredznačenih celih števil. Vsi elementi zaporedja so manjši ali enaki $k = 5$. Tabela C ima dolžino $k + 1$ in vsebuje števce pojavitev elementov v tabeli A. Na primer element 0 se v tabeli A pojavi 2-krat, zato je vrednost števca enaka $C[0] = 2$. Element 1 se ne pojavi v zaporedju, zato je $C[1] = 0$, itd. do elementa k.

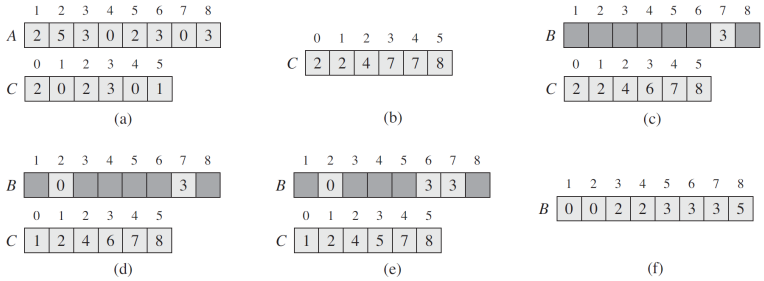
b) Nad tabelo števcev C izvedemo algoritem komulativne vsote (opisano v poglavju 2.3).

c) V tem koraku pričnemo polniti izhodno tabelo B. Izberemo zadnji element tabele A, to je 3. Nato preverimo vrednost števca v tabeli C na mestu 3, ki ima vrednost $C[3] = 7$. To pomeni, da moramo število 3 vstaviti na sedmo mesto tabele B. Nato zmanjšamo števec $C[3]$ za 1.

d) Izberemo predzadnji element tabele A, to je 0. Vrednost števca v tabeli C je $C[0] = 2$, zaradi česar ga vstavimo na drugo mesto tabele B. Na koncu zopet zmanjšamo števec $C[0]$ za 1.

e) Po enakem postopku vstavimo predpredzadnje število 3 tabele A.

f) Enak postopek ponovimo za ostale elemente.



Slika 8.2: Primer urejanja s štetjem. A je vhodna tabela, B je izhodna tabela, C je pomožna tabela števcev pojavitev elementov tabele A [12].

8.1.2 Časovna zahtevnost

Algoritem ureja števke števil s pomočjo urejanja s štetjem. Podrobneje analizirajmo urejanje s štetjem za vhodno tabelo dolžine n, ki vsebuje elemente na intervalu $[0,\ k]$. Na začetku moramo inicializirati pomožno tabelo števcev na 0, kar ima časovno zahtevnost $\theta(k)$. Preštevanje elementov (korak *(a)* na sliki 8.2) ima zahtevnost $\theta(n)$. Komulativna vsota (korak *(b)* na sliki 8.2) ima zahtevnost $\theta(k)$. Shranjevanje elementov v izhodno tabelo ima časovno zahtevnost $\theta(n)$. Časovna zahtevnost urejanja je zato enaka [12]:

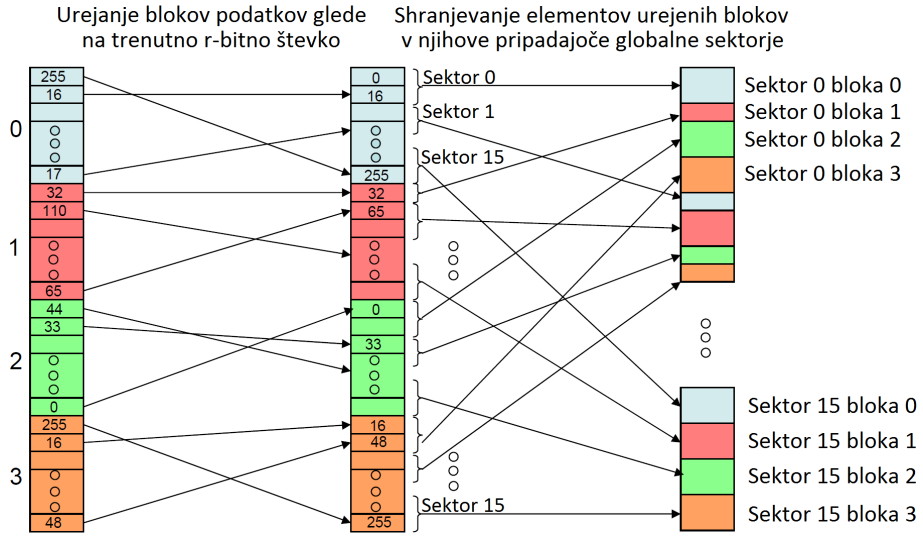
$$T_1 = O(n + k). \tag{8.1}$$

V praksi izvajamo urejanje s štetjem samo takrat, ko velja $k = O(n)$. V tem primeru je časovna zahtevnost urejanja enaka:

$$T_1 = O(n). \tag{8.2}$$

Vzemimo za primer tabelo b-bitnih števil dolžine n, ki jo uredimo s pomočjo urejanja po delih. V vsakem prehodu uredimo r-bitne števke, kar zahteva $d = \frac{b}{r}$ prehodov. Vsaka števka lahko zavzame vrednost na intervalu $k = 2^r$. Ob predpostavki, da števke urejamo s pomočjo urejanja s štetjem (8.1), je časovna zahtevnost urejanja po delih enaka:

$$T_1 = O\left(\left(\frac{b}{r}\right)(n + 2^r)\right) = O(d(n + k)). \tag{8.3}$$



Slika 8.3: Algoritem vzporednega urejanja po delih (prirejeno po [30]).

Kadar velja $k = O(n)$, lahko urejanje po delih izvedemo v linearnem času. V splošnem želimo izbrati $r \leq b$, ki minimizira izraz $(\frac{b}{r})(n + 2^r)$. Urejanje po delih je najprimernejše, kadar velja $b = O(\log n)$. V tem primeru velja za poljuben $r \leq b$, da je časovna zahtevnost urejanje enaka $\theta(n)$, pri čemer optimalno vrednost r določimo s pomočjo testiranja [12].

8.2 Vzporedni algoritem

8.2.1 Implementacija

Vzporedno urejanje po delih deluje na podlagi urejanja s štetjem in sektorskega urejanja (ang. *bucket sort*). Sestavljeno je iz treh korakov:

- urejanje blokov podatkov glede na trenutno r-bitno števko,
- določanje velikosti in odmikov sektorjev urejenih blokov,
- urejanje celotne tabele glede na trenutno r-bitno števko.

Za urejanje b-bitnih števil po r-bitnih števkah, je potrebnih $d = \frac{b}{r}$ iteracij prej naštetih korakov. V vsaki iteraciji se izvede algoritem, ki je prikazan na sliki 8.3.

8.2.1.1 Urejanje blokov podatkov glede na trenutno r-bitno števko

Algoritem najprej izvede klic ščepca za urejanje po delih, v katerem vsak blok niti uredi svoj pripadajoč blok podatkov glede na trenutno r-bitno števko. Implementacija ščepca je prikazana v psevdokodi 8.1. Kot vidimo, urejanje poteka v r iteracijah, pri čemer v vsaki iteraciji uredimo elemente glede na trenutni bit oziroma predikat r-bitne števke (vrstica 20). To storimo s pomočjo urejanja s štetjem. Omenjeno urejanje deluje tako, da za vsak element tabele poišče njegov *rang* (ang. *rank*) oziroma njegov indeks v urejeni tabeli. Za izračun ranga uporabimo funkcijo `najdiRangElementa` (vrstica 2). Ta najprej izračuna komulativno vsoto vseh trenutnih predikatov (vrstica 4), kar stori s pomočjo komulativne vsote za predikate (poglavje 2.3.3). Na podlagi vsote vsaka nit ugotovi število vseh niti z nižjim identifikatorjem, ki vsebujejo predikat 1. V naslednjem koraku zadnja nit v bloku izračuna število vseh predikatov z vrednostjo 0 (vrstica 6). Iz omenjenih vrednosti lahko enostavno določimo rang vsakega elementa (vrstici 9 in 10). Ko poznamo range vseh elementov tabele, jo lahko uredimo. To storimo tako, da vsak element shranimo na indeks njegovega pripadajočega ranga. Enak postopek ponovimo še na ostalih bitih trenutne r-bitne števke [30].

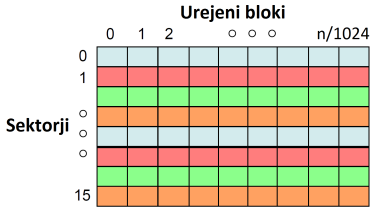
Izkaže se, da je ščepec bolj učinkovit, če ena nit obdela več elementov. V tem primeru moramo podati funkciji `najdiRangElementa` več predikatov naenkrat. Pri tem so potrebne spremembe tudi v komulativni vsoti. Klic funkcije `komVsotaSnopPredikat` (psevdokoda 2.8) je potrebno opraviti za vsak predikat, pri čemer preostanek funkcije `optKomulativnaVsota` ostane enak [30].

8.2.1.2 Določanje velikosti in odmikov sektorjev urejenih blokov

Število vseh možnih r-bitnih števk je enako $k = 2^r$. Z urejanjem blokov podatkov glede na trenutno r-bitno števko (opisano v prejšnjem poglavju) pravzaprav dosežemo, da elemente razporedimo v k sektorjev. Vsi elementi znotraj istega sektorja imajo isto vrednost trenutne r-bitne števke [30].

```
1  // pred: binarni predikat 0 ali 1.
2  najdiRangElementa(pred):
3    // Število niti z nižjim identifikatorjem, ki imajo predikat 1
4    stResnicnih = optKomulativnaVsota(pred)
5    if (threadIdx.x == blockDim.x - 1):
6      __shared__ steviloVsehNeresnicnih = blockDim.x - (stResnicnih + pred)
7    __syncthreads()
8
9    if pred: return stResnicnih - 1 + steviloVsehNeresnicnih
10   else: return threadIdx.x - stResnicnih
11
12 // tab:    vhodna tabela b-bitnih števil,
13 // bOdmik: odmik (št. bitov), na katerem se nahaja trenutna r-bitna števka.
14 scepecUrejanjePoDelih(tab, bOdmik):
15   extern __shared__ sTab[]
16   indeks = blockIdx.x * blockDim.x + threadIdx.x
17   sTab[threadIdx.x] = tab[indeks]
18   __syncthreads()
19
20   for (bit = bOdmik; bit < bOdmik + r; bit++):
21     element = sTab[threadIdx.x]
22     rang = najdiRangElementa((element >> bit) & 1)
23     __syncthreads()
24     sTab[rang] = element
25     __syncthreads()
26
27   tab[indeks] = sTab[threadIdx.x]
```

Psevdokoda 8.1: Ščepec za urejanje po delih (prirejeno po [30]).



Slika 8.4: Histogram velikosti sektorjev urejenih blokov (prirejeno po [30]).

Za vsak prej urejen blok izračunamo velikost in odmik vsakega izmed k sektorjev. To storimo tako, da medsebojno primerjamo trenutno r-bitno števko sosednjih elementov v urejenih blokih. Na mestih, kjer se števki razlikujeta, se nahajajo meje oziroma odmiki sektorjev. Njihovo velikost izračunamo s pomočjo razlike med odmiki sosednjih sektorjev. Za vsak urejen blok shranimo dobljene odmike in velikosti [30].

Na podlagi tabele velikosti sektorjev lahko s pomočjo izključujoče komulativne vsote enostavno izračunamo globalne odmike sektorjev. Za izračun vsote moramo velikosti sektorjev shraniti na ustrezen način. V prvem delu tabele morajo biti velikosti sektorja 0 vseh urejenih blokov, potem velikosti

sektorji 1, itd. vse do zadnjega sektorja $k-1$. Slika 8.4 prikazuje opisan postopek shranjevanja za 16 sektorjev in urejene bloke velikosti 1024. Komulativno vsoto izračunamo s pomočjo knjižnice *CUDPP* [1, 30].

8.2.1.3 Urejanje celotne tabele glede na trenutno r-bitno števko

Na podlagi prej izračunanih lokalnih in globalnih odmikov sektorjev, lahko uredimo celotno tabelo glede na trenutno števko. Za vsak element tabele, preberemo ustrezen lokalni in globalni odmik, na podlagi katerega izračunamo globalni rang. To storimo tako, da izračunamo vsoto globalnega odmika in identifikatorja niti, od katere odštejemo lokalni odmik. Na koncu enostavno shranimo element na indeks, ki ga določa rang [30].

Kot smo omenili, se elementi z isto vrednostjo trenutne r-bitne števke nahajajo na zaporednih naslovih oziroma v istih sektorjih urejenih blokov. Zaradi stabilnosti urejanja s štetjem velja, da bodo zaporedni elementi znotraj istega sektorja v urejenem bloku shranjeni na zaporedne naslove v globalnem pomnilniku. Število sektorjev je le $k = 2^r,\ r \in \mathbb{N}_0$ (po navadi majhno število, npr. 16). Ob predpostavki, da je velikost urejenih blokov enaka 1024, je pričakovana velikost sektorjev enaka $\frac{1024}{16} = 64$. Iz tega razloga bo velika večina pisanj v globalni pomnilnik potekala zaporedno. [30].

Za urejanje tabel poljubne dolžine je potrebno zapolniti tabelo z minimalnimi oziroma maksimalnimi vrednostmi (odvisno od smeri urejanja) do naslednjega večkratnika velikosti urejenega bloka.

8.2.2 Časovna zahtevnost

Kot smo omenili, je za urejanje b-bitnih števil po r-bitnih števkah potrebnih $d = \frac{b}{r}$ prehodov. V vsakem prehodu vsak blok niti najprej uredi svoj odsek podatkov dolžine m glede na trenutno r-bitno števko. Tako mora vseh $O(\frac{n}{m})$ blokov niti r-krat izračunati komulativno vsoto, kar zahteva čas $O(\frac{nr}{mp} \cdot m \cdot \log m) = O(\frac{nr}{p} \cdot \log m)$ (2.5). V naslednjem koraku določimo lokalne odmike sektorjev urejenih blokov. Vseh lokalnih sektorjev je $O(\frac{nk}{m});\ k = 2^r$, zato omenjena operacija zahteva čas $O(\frac{nk}{mp})$. Komulativna vsota nad veliko-

stjo lokalnih sektorjev zahteva čas $O(\frac{nk}{mp} \cdot \log \frac{nk}{m})$ (2.5). V zadnjem koraku vsaka nit prebere globalni in lokalni odmik za sektor svojega pripadajočega elementa, na podlagi katerih izračuna rang elementa v celotni tabeli. Opisan postopek ima časovno zahtevnost $O(\frac{n}{p})$. Ugotovimo, da v vsakem izmed d prehodov prevlada časovna zahtevnost komulativne vsote. V splošnem velja, da sta m in k veliko manjša od n. Poleg tega sta tudi konstanti neodvisni od n in p, zato je časovna zahtevnost komulativne vsote enaka $O(\frac{n}{p} \cdot \log n)$. Časovna zahtevnost urejanja po delih za p procesorjev je zato enaka:

$$T_p = O\left(d\left(\frac{n}{p} \cdot \log n\right)\right). \tag{8.4}$$

V poglavju 8.1.2 smo omenili, da je urejanje po delih smiselno izvajati samo, kadar velja $b = O(\log n)$. V tem primeru velja:

$$T_p = O\left(\frac{n}{p} \cdot \log n\right). \tag{8.5}$$

Z $O(n)$ procesorji lahko časovno zahtevnost zmanjšamo na:

$$T_n = O(\log n). \tag{8.6}$$

Pri izračunu pohitritev bomo primerjali časovne zahtevnosti, kadar velja prej omenjen pogoj $b = O(\log n)$, saj je le takrat smiselno uporabljati urejanje po delih. Pohitritev za p procesorjev je enaka:

$$S_p = \frac{T_1}{T_p} = \frac{O(n)}{O\left(\frac{n}{p} \cdot \log n\right)} = O\left(\frac{p}{\log n}\right). \tag{8.7}$$

Kot vidimo, pohitritev ni optimalna, ker ni enaka p. Pohitritev pri O(n) procesorjih je enaka:

$$S_n = \frac{T_1}{T_n} = \frac{O(n)}{O\left(\log n\right)} = O\left(\frac{n}{\log n}\right). \tag{8.8}$$

Poglavje 9

Urejanje z vzorci

9.1 Zaporedni algoritem

9.1.1 Implementacija

Delovanje zaporednega *urejanja z vzorci* (ang. *sample sort*) najpreprosteje opišemo s psevdokodo 9.1. Na začetku preverimo, če je dolžina tabele manjša ali enaka konstanti `MEJA_UREJANJE` (vrstica 6). V tem primeru jo uredimo z alternativnim urejanjem (vrstica 7). Tukaj velja izpostaviti, da je urejanje z vzorci stabilno, če je stabilno tudi alternativno urejanje. Zato smo se v naši implementaciji odločili za urejanje z zlivanjem (poglavje 6), ker je zelo učinkovito za urejanje krajših zaporedij in je hkrati tudi stabilno. Kadar je tabela daljša od `MEJA_UREJANJE`, jo razdelimo v sektorje (ang. *bucket*). Za sektorje velja, da so vsi elementi v sektorju b manjši od elementov v sektorju $b + 1$. Delitev izvedemo tako, da najprej izberemo `ST_VZORCEV` naključnih vzorcev in jih uredimo (vrstice 10 - 13). Iz tabele urejenih vzorcev izberemo `ST_SEKTORJEV` vrednosti, ki bodo predstavljale meje sektorjev (vrstici 16 in 17). Na tem mestu je potrebno poudariti, da je število `ST_VZORCEV` navadno večkratnik števila `ST_SEKTORJEV`. Večje kot je razmerje med omenjenima konstantama $a = \frac{ST\_VZORCEV}{ST\_SEKTORJEV}$, bolj uravnotežene bodo velikosti sektorjev. Pri tem moramo biti pozorni, da razmerje a ni preveliko, ker

```
1  // tab:    vhodna tabela,
2  // tabPom: pomožna tabela,
3  // n:      dolžina vhodne in pomožne tabele,
4  // s:      smer urejanja (0: naraščajoče, 1: padajoče).
5  zaporednoUrejanjeZVzorci(tab, tabPom, n, s):
6    if n <= MEJA_UREJANJE:
7      urediTabelo(tab, n, s)
8      return
9
10   vzorci = []
11   for (i = 0; i < ST_VZORCEV; i++):
12     vzorci.dodaj(tabela[nakljucnoStevilo(0, n)])
13   urediTabelo(vzorci)
14
15   mejeSektorjev = [], velikostiSektorjev = []
16   for (i = 0; i < ST_VZORCEV; i += ST_VZORCEV / ST_SEKTORJEV):
17     mejeSektorjev.dodaj(vzorci[i])
18     velikostiSektorjev.dodaj(0)
19   velikostiSektorjev.dodaj(0)
20
21   sektorjiElementov = []
22   for (i = 0 ; i < n; i++):
23     sektor = vkljucujoceBinarnoIskanje(mejeSektorjev, tab[i])
24     velikostiSektorjev[sektor]++
25     sektorjiElementov.dodaj(sektor)
26
27   odmikiSektorjev = izkljucujocaKomulativnaVsota(velikostiSektorjev)
28   for (i = 0; i < n; i++):
29     tabPom[odmikiSektorjev[sektorjiElementov[i]]++] = tab[i]
30
31   for (i = 0; i <= ST_SEKTORJEV; i++):
32     odmikPrejsnjega = i > 0 ? odmikiSektorjev[i - 1] : 0
33     velikostSektorja = odmikiSektorjev[i] - odmikPrejsnjega
34     zaporednoUrejanjeZVzorci(
35       tabPom + odmikPrejsnjega, tab + odmikPrejsnjega, velikostSektorja
36     )
```

Psevdokoda 9.1: Zaporedno urejanje z vzorci (prirejeno po [23]).

bi v takšnem primeru urejanje vzorcev potekalo predolgo (vrstica 13). Ob določanju mej sektorjev hkrati inicializiramo tabelo njihovih velikosti (vrstici 18 in 19). Nato moramo za vsak element v tabeli določiti, v kateri sektor spada. To storimo s pomočjo vključujočega binarnega iskanja v prej omenjeni tabeli mej sektorjev (vrstica 23). Sočasno določimo tudi njihove velikosti (vrstica 24). Za vsak element si tudi zapomnimo, v kateri sektor spada (vrstica 25). Nad dobljenimi velikostmi sektorjev izračunamo komulativno vsoto (poglavje 2.3), s čimer določimo njihove odmike (vrstica 27). Na podlagi dobljenih odmikov lahko elemente tudi shranimo v njihove pripadajoče sektorje (vrstici 28 in 29). Nad dobljenimi sektorji rekurzivno pokličemo funkcijo za urejanje z vzorci (vrstice 31 - 36) [23].

Opisan algoritem je podoben algoritmu hitrega urejanja (poglavje 7). Razlikuje se v tem, da hitro urejanje razporeja elemente v 2 sektorja (elementa manjše ali večje od pivota), medtem ko urejanje z vzorci razporeja elemente v `ST_SEKTORJEV` sektorjev. V nasprotju s sektorskih urejanjem (ang. *bucket sort*) izvede vzorčenje v vsaki iteraciji, zaradi česar mu ni potrebno vnaprej poznati intervala števil vhodnega zaporedja. Posledično je tudi učinkovitejše pri urejanju neenakomernih porazdelitev.

9.1.2 Časovna zahtevnost

V vsaki fazi urejanja razdelimo zaporedje na k sektorjev. Alternativno urejanje izvedemo, ko je dolžina tabele manjša ali enaka m. Število faz, potrebnih za razdelitev tabele dolžine n v sektorje dolžine m, je enako $O(\log_k \frac{n}{m})$. V vsaki fazi je potrebno določiti sektorje in shraniti elemente vanje (psevdokoda 9.1), kar ima časovno zahtevnost $O(n)$. Časovna zahtevnost razdelitve zaporedja v sektorje je zato enaka [23]:

$$O\left(n \cdot \log_k \frac{n}{m}\right). \tag{9.1}$$

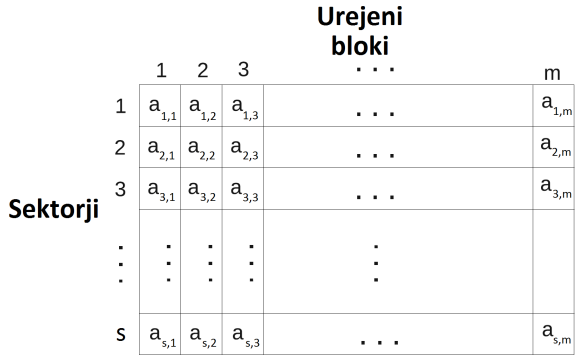
V naslednjem koraku je potrebno vse sektorje urediti. V naši implementaciji smo uporabili urejanje z zlivanjem s časovno zahtevnost $O(n \cdot \log n)$ (6.2). Časovna zahtevnost urejanja z vzorci je enaka:

$$\begin{aligned} T_1 &= O\left(n \cdot \log_k \frac{n}{m} + \frac{n}{m} \cdot m \cdot \log m\right) = O\left(n\left(\log_k \frac{n}{m} + \log m\right)\right) \\ &= O(n \cdot \log n). \end{aligned} \tag{9.2}$$

9.2 Vzporedni algoritem

9.2.1 Implementacija

V prvem ščepcu vsak izmed m blokov niti uredi svoj pripadajoč odsek podatkov dolžine $k = \frac{n}{m}$. Urejanje odsekov izvedejo s pomočjo bitonega urejanja



Slika 9.1: Histogram velikosti sektorjev urejenih blokov (prirejeno po [13]).

(poglavje 3), saj je ta zelo učinkovit za urejanje kratkih zaporedij. Po opravljenem urejanju izberejo s vzorcev iz svojega urejenega odseka podatkov. Vzorce izberejo kot vsak $\frac{k}{s}$ element. Urejene bloke in vzorce shranijo nazaj v globalni pomnilnik [13].

S pomočjo bitonega urejanja uredimo v prejšnjem koraku shranjeno tabelo lokalnih vzorcev. Iz urejene tabele lokalnih vzorcev dolžine $s \cdot m$ izberemo s globalnih vzorcev, ki bodo služili kot meje sektorjev. Te izberemo kot vsak m-ti vzorec tabele urejenih lokalnih vzorcev [13].

V naslednjem ščepcu vsak izmed m blokov niti prebere globalne vzorce. S pomočjo vključujočega binarnega iskanja poišče *rang* (ang. *rank*) vseh vzorcev v svojem pripadajočem odseku prej urejenih podatkov. Kot smo pojasnili v prejšnjih poglavjih, je rang elementa e enak številu elementov v urejeni tabeli, ki so manjši ali enaki vrednosti elementa e. Rangi vzorcev pravzaprav predstavljajo odmike oziroma meje sektorjev in posledično določajo tudi sektorje vsakega urejenega odseka podatkov. Na podlagi odmikov sektorjev lahko določimo njihove velikosti. Izračunamo jih kot razliko med odmiki sosednjih sektorjev. Vsak blok niti nato shrani velikosti svojih lokalnih sektorjev v globalni pomnilnik [13].

Nad velikostmi lokalnih sektorjev izračunamo komulativno vsoto, s čimer določimo globalne odmike sektorjev. V ta namen moramo velikosti lokalnih sektorjev shraniti tako, kot to prikazuje slika 9.1. V prvem delu tabele morajo biti velikosti sektorja 1 vseh urejenih blokov, potem velikosti sektorja

```
1  // tab:         vhodna tabela,
2  // tabPom:      pomožna tabela,
3  // m:           dolžina urejenega bloka podatkov vsakega bloka niti,
4  // odmikiS:     odmiki sektorjev trenutnega bloka niti,
5  // velikostiS: velikosti sektorjev trenutnega bloka niti.
6  shraniElemente(tab, tabPom, m, odmikiS, velikostiS):
7    aktivneNiti, aktivneNitiPrej, sektor = 0
8    tx, odmik = threadIdx.x, blockIdx.x * m
9
10   while (tx < m):
11     aktivneNiti += velikostiS[sektor]
12     for (;tx < aktivneNiti; tx += blockDim.x):
13       tabPom[odmikiS[sektor] + tx - aktivneNitiPrej] = tab[odmik + tx]
14
15     aktivneNitiPrej = aktivneNiti, sektor++
```

Psevdokoda 9.2: Funkcija za shranjevanje elementov v sektorje.

2, itd. vse do zadnjega sektorja s. Za izračun vsote smo uporabili knjižico *CUDPP* [1, 13].

V naslednjem koraku smo izdelali ščepec, v katerem vsak blok niti ponovno prebere svoj pripadajoč blok urejenih podatkov ter svoje pripadajoče velikosti in odmike sektorjev. Na podlagi omenjenih odmikov in velikosti lahko shrani elemente v globalne sektorje, kar prikazuje psevdokoda 9.2. Vse niti izvedejo prehod od sektorja 0 do s. V vsaki iteraciji preberejo velikost trenutnega sektorja (vrstica 11). Nato izvedejo shranjevanje elementov trenutnega sektorja (vrstici 12 in 13) in za tem izvedejo premik na naslednji sektor (vrstica 14) [13].

Na koncu s pomočjo bitonega urejanja uredimo vseh s sektorjev, pri čemer urejanja sektorjev izvajamo zaporedno [13].

9.2.2 Posebnosti implementacije

Najpomembnejši parameter algoritma je število globalnih vzorcev oziroma sektorjev s. S povečevanjem vrednosti s dobimo tudi večje število sektorjev, ki so posledično krajši. To pomeni veliko prednost, saj je bitono urejanje najučinkovitejše pri urejanju kratkih zaporedij. Slabost velike vrednosti s je veliko število lokalnih vzorcev in posledično njihovo dolgotrajno urejanje. S testiranjem moramo poiskati tako vrednost s, ki bo omogočala najboljše ravnotežje prej naštetih prednosti in slabosti [13].

Opisan algoritem razporejanja elementov v njihove pripadajoče sektorje je stabilen. Za ohranjanje stabilnosti bi bilo potrebno urediti sektorje s stabilnim urejanjem (npr. urejanje z zlivanjem). V naši implementaciji smo za urejanje sektorjev uporabili bitono urejanje, ki ni stabilno, in zato tudi naša implementacija urejanja z vzorci ne more biti stabilna.

Pri urejanju parov ključ-vrednost so za določitev sektorjev potrebni samo ključi, kar predstavlja veliko prednost, saj s tem zmanjšamo število dostopov do globalnega pomnilnika.

V algoritmu so potrebne majhne spremembe za urejanje tabel poljubnih dolžin. Tabelo je potrebno samo zapolniti z minimalnimi oziroma maksimalnimi vrednostmi (odvisno od smeri urejanja) do naslednjega večkratnika dolžine urejenega bloka $k = \frac{n}{m}$.

9.2.3 Časovna zahtevnost

Na začetku uredimo bloke podatkov velikosti $k = \frac{n}{m}$ in uporabimo vzporedno bitono urejanje. Časovna zahtevnost urejanja je $O(\frac{n}{p} \cdot \log^2 k)$ (3.12). Ob tem shranimo tudi $s \cdot m$ lokalnih vzorcev, ki jih zopet uredimo z bitonim urejanjem. Časovna zahtevnost urejanja vzorcev je $O(\frac{sm}{p} \cdot \log^2 sm)$. Določimo še globalne vzorce in poiščemo njihove range v vseh urejenih blokih. Število globalnih vzorcev je s, število urejenih blokov je m, časovna zahtevnost binarnega iskanja je $O(\log k)$. Časovna zahtevnost prej omenjene operacije je torej $O(\frac{sm}{p} \cdot \log k)$. Ob tem določimo tudi velikosti sektorjev, nad katerimi izvedemo komulativno vsoto. Časovna zahtevnost komulativne vsote je enaka $O(\frac{sm}{p} \cdot \log sm)$ (2.5). Nato premaknemo elemente v njihove pripadajoče sektorje. Kot vidimo iz psevdokode 9.2, mora vsaka nit opraviti prehod čez vseh s sektorjev. Časovna zahtevnost shranjevanje elementov je enaka $O(\frac{ns}{p})$. Na koncu s pomočjo bitonega urejanja uredimo vseh s sektorjev, kar ima časovno zahtevnost (3.12):

$$O\left(s\frac{n}{sp}\log^2\frac{n}{s}\right) = O\left(\frac{n}{p}\log^2\frac{n}{s}\right). \tag{9.3}$$

Ugotovimo, da izmed vseh časovnih zahtevnosti prevlada časovna zahtevnost urejanja s sektorjev. Število s je konstanta in ni povezana z dolžino vhodnega zaporedja n ali številom procesorjev p. Poleg tega velja, da je v splošnem zelo majhno število (npr. 32 ali 64). Zato je časovna zahtevnost urejanja z vzorci enaka:

$$T_p = O\left(\frac{n}{p} \cdot \log^2 n\right). \tag{9.4}$$

Z O(n) procesorji lahko časovno zahtevnost zmanjšamo na:

$$T_n = O(log^2 n), \tag{9.5}$$

Pohitritev za p procesorjev je enaka:

$$S_p = \frac{T_1}{T_p} = O\left(\frac{n \cdot \log n}{\frac{n}{p} \cdot \log^2 n}\right) = O\left(\frac{p}{\log n}\right). \tag{9.6}$$

Kot vidimo pohitritev ni optimalna. Pohitritev pri O(n) procesorjih je enaka:

$$S_n = \frac{T_1}{T_n} = O\left(\frac{n \cdot \log n}{\log^2 n}\right) = O\left(\frac{n}{\log n}\right). \tag{9.7}$$

Poglavje 10

Primerjava algoritmov

10.1 Testno okolje

Zaporedno računanje smo izvajali na štirijederni CPE *Intel Core i7-3770K* frekvence $3.5GHz$. Za vzporedno računanje smo uporabili GPE *GeForce GTX670* z $2GB$ pomnilnika. Algoritme smo preizkušali na zaporedjih celih števil, porazdeljenih po naslednjih porazdelitvah [20]:

a) **enakomerna** - naključna števila med 0 in $2^{32} - 1$ za 32-bitna števila oziroma $2^{64} - 1$ za 64-bitna števila,

b) **normalna** - vsak element je povprečje štirih naključnih števil,

c) **ničelna** - naključna konstantna vrednost,

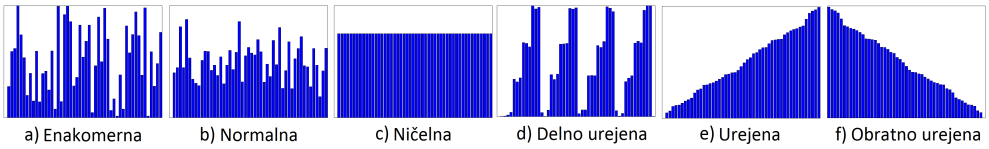
d) **delno urejena** (ang. *bucket*) - urejena podzaporedja dolžine 1024,

e) **urejena** - nepadajoče urejena naključna števila,

f) **obratno urejena** - nenaraščajoče urejena naključna števila.

Slika 10.1 prikazuje vizualizacijo prej naštetih porazdelitev. Za tvorjenje naključnih števil smo uporabili generator *Mersenne Twister* [24].

Algoritmi so urejali ključe in pare ključ vrednost nepredznačenih celih števil (ang. *unsigned integer*). Števila so bila bodisi 32-bitna ali 64-bitna. Algoritme smo preizkušali na različnih dolžinah vhodnega zaporedja, in sicer



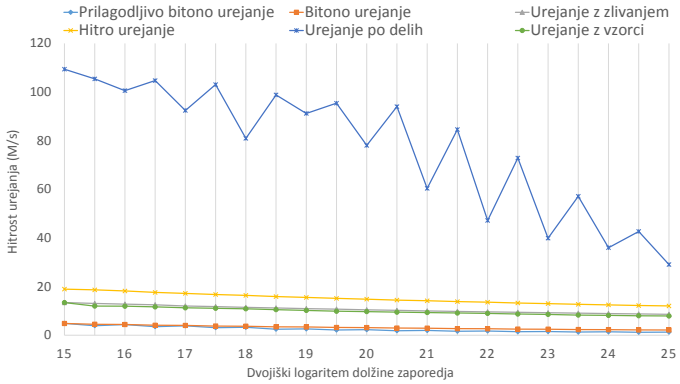
Slika 10.1: Vizualizacija porazdelitev vhodnih zaporedij (prirejeno po [9]).

od 2^{15} do 2^{25} za 32-bitna števila oziroma do 2^{24} za 64-bitna števila. Preizkušali smo tudi vmesne dolžine zaporedij $2^n + 2^{n-1}$(med 2^n in 2^{n+1}), saj nekateri algoritmi delujejo hitreje oziroma počasneje pri opisani dolžini zaporedij. Vsako urejanje smo ponovili 50-krat in dobljene čase povprečili. Hitrost urejanja smo merili v številu milijonov elementov, ki jih algoritem uredi v eni sekundi (M/s).

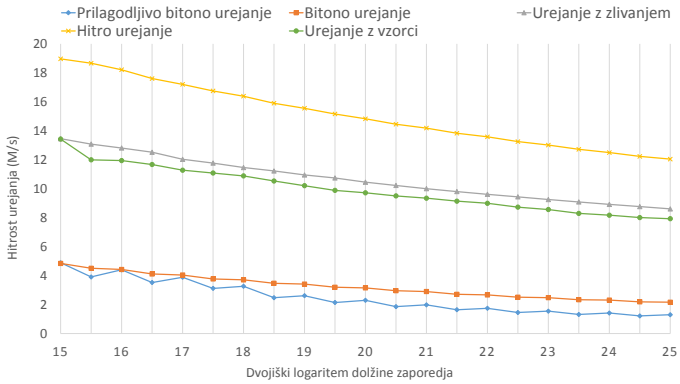
Pri vzporednih implementacijah smo upoštevali samo čas izvajanja urejanja na GPE, pri čemer nismo upoštevali časa prenosa podatkov iz gostitelja na napravo in obratno. Za opisan pristop smo se odločili, ker urejanje podatkov navadno predstavlja le en korak v zahtevnejših algoritmih. V tem primeru se podatki že nahajajo na GPE, zato je opisan način merjenja časa postal standarden v literaturi [9].

V poglavju 5.1 smo omenili, da zaporedno prilagodljivo bitono urejanje deluje na podlagi bitonega drevesa. Izgradnja drevesa zahteva veliko časa, zato bi ta algoritem v praksi uporabili le za urejanje binarnih dreves. To je bil tudi glavni razlog, da pri merjenju časa nismo upoštevali izgradnje drevesa in pretvarjanja drevesa v tabelo po končanem urejanju.

Na koncu poglavja 7.2.3.1 smo omenili, da smo implementirali dve različici ščepca za globalno hitro urejanje. Prva različica izvaja redukcijo za iskanje minimuma in maksimum ter na podlagi dobljenih vrednosti določi pivot. Druga različica določi pivot kot povprečje starega pivota in minimalne oziroma maksimalne vrednosti trenutnega podzaporedja. Naša GPE ima računsko zmožnost 3.0. To pomeni, da ne podpira atomarnega minimuma oziroma maksimuma za 64-bitna števila, ki je potreben za izvedbo prve različice. Zaradi pravičnejše primerjave smo zato preizkusili samo drugo različico hitrega urejanja. Poleg tega je druga različica v večini primerov za približno 10 % hitrejša od prve različice.



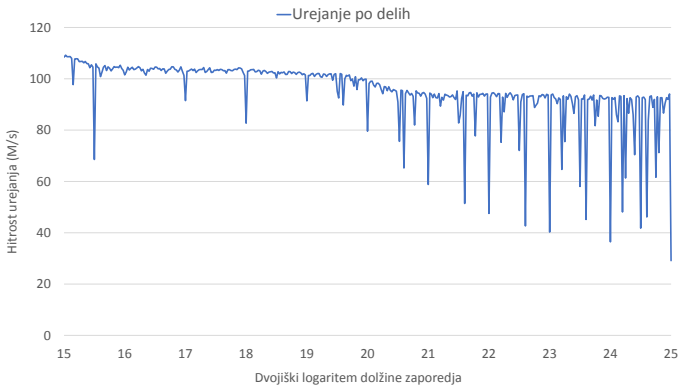
Slika 10.2: Zaporedno urejanje 32-bitnih ključev enakomerne porazdelitve.



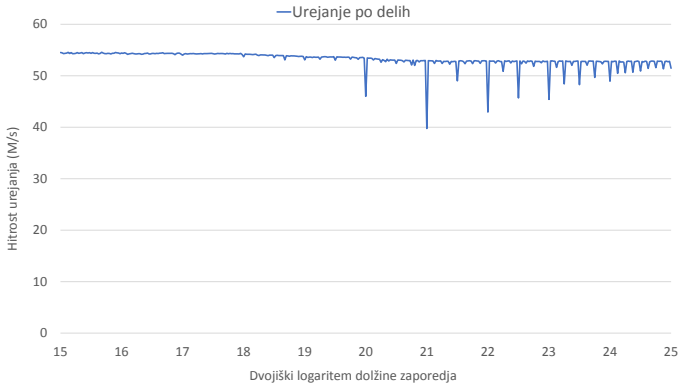
Slika 10.3: Zaporedno urejanje 32-bitnih ključev enakomerne porazdelitve brez urejanja po delih.

10.2 Rezultati zaporednih urejanj

Podrobneje bomo opisali samo rezultate urejanja enakomerne porazdelitve, saj v praksi najpogosteje urejamo naključna števila. Rezultati urejanj ostalih porazdelitev so prikazani v dodatku A. Na osi x so prikazani dvojiški logaritmi dolžine zaporedja. Os y prikazuje število milijonov elementov, ki jih algoritem uredi v eni sekundi (M/s). Opozoriti velja, da zaporednega hitrega urejanja nismo preizkušali na ničelni porazdelitvi, ker ta dosega hitrost le 0.3 M/s ali manj.



Slika 10.4: Zaporedno urejanje po delih 32-bitnih ključev enakomerne porazdelitve z uporabo 8-bitnih števk, vzorčeno 40-krat med zaporednima dvojiškima celoštevilskima logaritmoma dolžine zaporedja.



Slika 10.5: Zaporedno urejanje po delih 32-bitnih ključev enakomerne porazdelitve z uporabo 4-bitnih števk, vzorčeno 40-krat med zaporednima dvojiškima celoštevilskima logaritmoma dolžine zaporedja.

10.2.1 Urejanje 32-bitnih ključev

Slika 10.2 prikazuje graf urejanja enakomerne porazdelitve. Zaradi preglednosti smo vključili tudi sliko 10.3, ki ne vsebuje urejanja po delih. Kot vidimo, je urejanje po delih najhitrejše, ker dosega hitrost do 110 M/s. Med algoritmi, ki delujejo na podlagi primerjav, je najhitrejše hitro urejanje. Urejanje z zlivanjem in urejanje z vzorci imata podobno hitrost, ker urejanje z vzorci temelji na urejanju z zlivanjem. Najpočasnejša algoritma sta navadno in prilagodljivo bitono urejanje. Vzrok za nizko hitrost bitonega urejanja je časovna zahtevnost $O(n \cdot \log^2 n)$ (3.7). Vzrok za nizko hitrost prilagodljivega

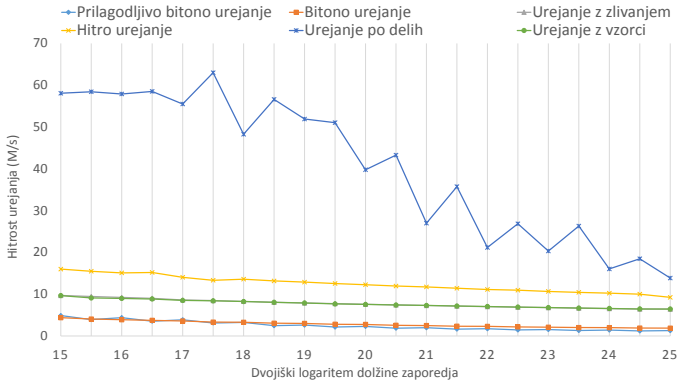
bitonega urejanja je najverjetneje v tem, da je algoritem rekurziven, zato je potrebno veliko dodatnega shranjevanja na sklad.

Grafi urejanj ostalih porazdelitev so prikazani na sliki A.1. Kot vidimo, je v splošnem vrsti red urejanj na vseh grafih skoraj enak. Opazimo tudi, da z daljšanjem vhodne tabele pada hitrost urejanj. Grafi enakomerne, normalne in delno urejene porazdelitve so skoraj popolnoma enaki. Pri ničelni, urejeni in obratno urejeni porazdelitvi algoritmi pridobijo na hitrosti. Izjema je urejanje po delih, katerega hitrost ostane kvečjemu enaka.

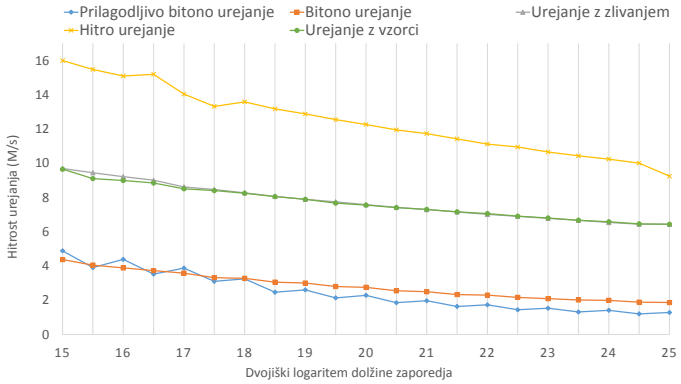
Pri urejanju po delih opazimo tudi zanimiv pojav, ki ga lahko vidimo tudi na večini ostalih grafov zaporednih urejanj. Algoritem je počasnejši pri urejanju zaporedij, katerih dolžina je potence števila 2. Pojav smo tudi podrobneje preučili tako, da smo interval dolžine zaporedja med 2^n in 2^{n+1} razdelili na 40 delov. Sliki 10.4 in 10.5 prikazujeta urejanje po delih z 8-bitnimi (uporabljeno tudi na ostalih grafih) in 4-bitnimi števkami. Kot vidimo, pride do številnih odstopanj v hitrosti urejanja, ne samo pri dolžini potence števila 2. Odstopanja so veliko bolj izrazita pri urejanju z 8-bitnimi števkami. To ni posledica implementacije algoritma, saj je ta povsem enaka za urejanje katerekoli dolžine zaporedja. Vzrok najverjetneje leži v strojni opremi. V vsaki iteraciji se s pomočjo urejanja s štetjem izvaja urejanje po trenutni b-bitni števki oziroma razporejanje števil v 2^b sektorjev. S 4-bitnimi števkami dobimo 16 sektorjev, medtem ko z 8-bitnimi 256 sektorjev. Zato je pri razporejanju v 4-bitne sektorje veliko večja verjetnost, da se bo odsek tabele s pravim sektorjem že nahajal v predpomnilniku in je posledično potrebnih manj predpomnilniških zamenjav, kot pri urejanju z 8-bitnimi števkami. Podoben trend opazimo tudi na nekaterih grafih hitrega urejanja, kar je najverjetneje posledica algoritma za izbiro pivota.

10.2.2 Urejanje 32-bitnih parov ključ vrednost

Na slikah 10.6 in 10.7 vidimo, da je trend hitrosti urejanj zelo podoben trendu urejanja 32-bitnih ključev. Podobno velja tudi za ostale porazdelitve, kar lahko vidimo na sliki A.2. Izpostaviti velja, da je hitrost urejanj manjša



Slika 10.6: Zaporedno urejanje 32-bitnih parov ključ vrednost enakomerne porazdelitve.



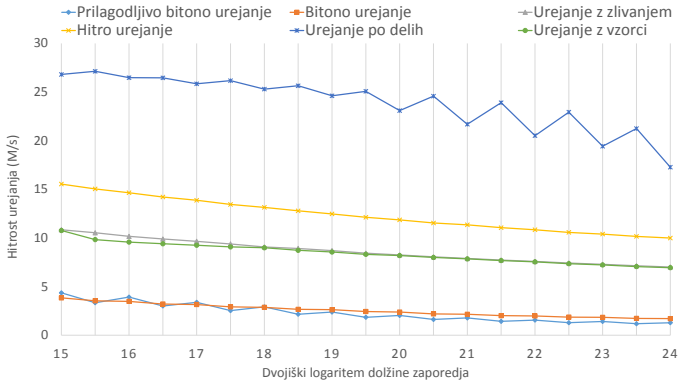
Slika 10.7: Zaporedno urejanje 32-bitnih parov ključ vrednost enakomerne porazdelitve brez urejanja po delih.

kot pri urejanju ključev, ker je potrebno ob vsaki menjavi ključa izvesti tudi menjavo vrednosti.

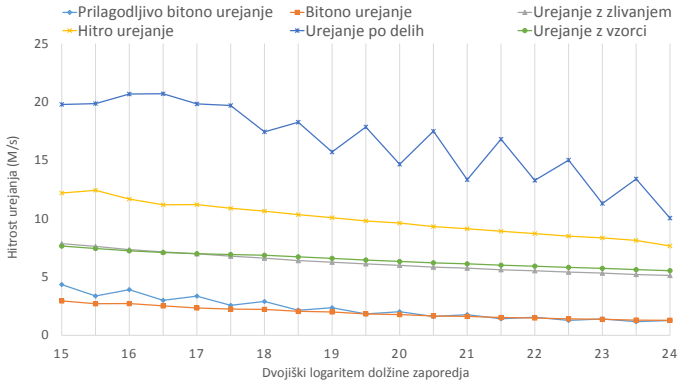
10.2.3 Urejanja 64-bitnih ključev

Na sliki 10.8 vidimo, da je razmerje med urejanji zelo podobno kot pri urejanju 32-bitnih ključev, pri čemer vsa urejanja izgubijo na hitrosti. Hitrost urejanja najbolj izrazito pade urejanju po delih. Vzrok za izgubo hitrosti omenjenega algoritma so dvakrat daljši ključi, zaradi česar je potrebnih dvakrat več faz urejanja.

Grafi enakomerne, normalne in delno urejene porazdelitve so si ponovno



Slika 10.8: Zaporedno urejanje 64-bitnih ključev enakomerne porazdelitve.



Slika 10.9: Zaporedno urejanje 64-bitnih parov ključ vrednost enakomerne porazdelitve.

podobni (slika A.3). Grafa urejene in obratno urejene porazdelitve sta zelo podobna grafu 32-bitnih ključev, pri čemer vsa urejanja izgubijo na hitrosti. Kot izjemo bi izpostavili delno urejeno porazdelitev, pri kateri hitro urejanje, urejanje z zlivanjem in urejanje z vzorci pridobijo na hitrosti ob urejanju tabel dolžine 2^{20} ali več. Opozoriti velja, da urejanje po delih občutno izgubi na hitrosti in postane približno enako hitro kot urejanje z zlivanjem. Najpočasnejši urejanji sta navadno in prilagodljivo bitono urejanje.

10.2.4 Urejanje 64-bitnih parov ključ vrednost

Graf urejanja 64-bitnih parov ključ vrednost (slika 10.9) je podoben grafu 64-bitnih ključev, pri čemer vsem urejanjem pade hitrosti za približno 25 %.

Podobno velja tudi za ostale porazdelitve, ki jih vidimo na sliki A.4. Izjema je urejanje po delih, ki pri ničelni, urejeni in obratno urejeni porazdelitvi zelo malo izgubi na hitrosti in je z izjemo urejene porazdelitve najhitrejše.

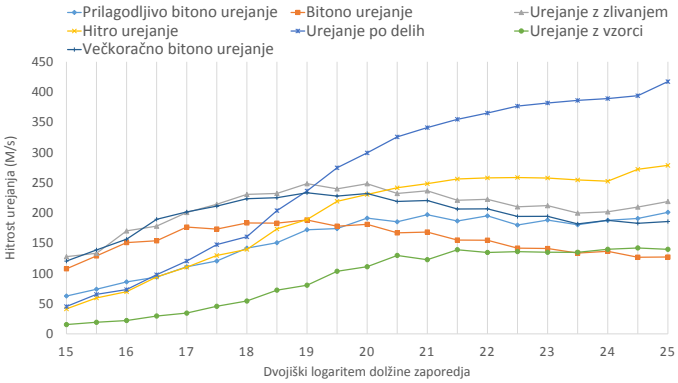
10.3 Rezultati vzporednih urejanj

Podobno kot v poglavju o rezultatih zaporednih urejanj, smo tudi v tem poglavju vključili le grafe enakomerne porazdelitve. Zaradi preglednosti grafi ničelne porazdelitve v prilogi A ne prikazujejo hitrega urejanja, ker to dosega hitrost do 35.000 M/s. Vzrok za to je dejstvo, da mora algoritem samo poiskati minimalno in maksimalno vrednost zaporedja.

10.3.1 Urejanje 32-bitnih ključev

Slika 10.10 prikazuje graf urejanja enakomerne porazdelitve. Pri urejanju zaporedij dolžine 2^{19} ali več, je najhitrejše urejanje po delih, ki dosega hitrost do 417 M/s. Najpočasnejše je urejanje z vzorci z maksimalno hitrostjo 142 M/s. Vzrok za njegovo počasno izvajanje je v tem, da se urejanje posameznih sektorjev izvaja zaporedno. Med urejanji s primerjavami sta za zaporedja dolžine 2^{20} ali manj najhitrejša urejanje z zlivanjem in večkoračno bitono urejanje, za daljša zaporedja pa hitro urejanje. Opazimo tudi, da je večkoračno bitono urejanje hitrejše od navadnega bitonega urejanja. Enako velja tudi za prilagodljivo bitono urejanje pri zaporedjih daljših od 2^{20}.

Opazimo tudi zanimivost, da z večanjem obsega vhodnih podatkov postanejo zaporedni algoritmi manj učinkoviti, medtem ko postanejo vzporedni bolj učinkoviti. Do opisanega pojava pride najverjetneje zato, ker je pri zaporednem urejanju zelo dolgih zaporedij potrebno izvajati več menjav pomnilniških blokov v predpomnilniku. Pri krajših zaporedjih je veliko večja verjetnost, da bomo izvajali menjave elementov zaporedja, ki se že nahajajo v predpomnilniku, kot pri daljših zaporedjih. Vzporedni algoritmi urejanja števil pa postajajo učinkovitejši pri daljših zaporedjih, ker je potrebnega več vzporednega procesiranja. V tem primeru pride vzporednost bolj do izraza,



Slika 10.10: Vzporedno urejanje 32-bitnih ključev enakomerne porazdelitve.

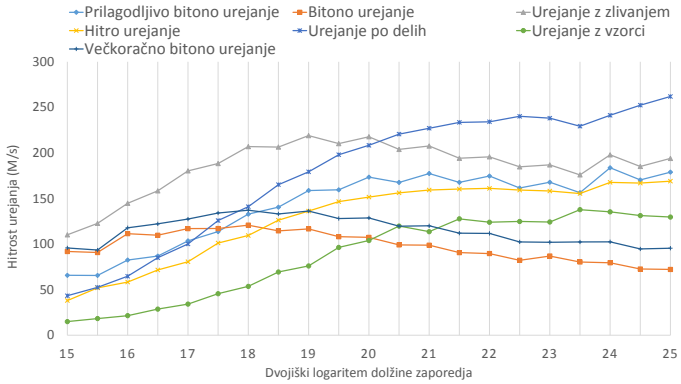
zaradi česar prenosi podatkov med globalnim in deljenim pomnilnikom pridejo manj do izraza.

Slika A.5 prikazuje grafe ostalih porazdelitev. Kot vidimo, so si grafi za enakomerno, normalno in delno urejeno porazdelitev zelo podobni. Podobna sta tudi grafa urejene in obratno urejene porazdelitve. Pri naštetih porazdelitvah vsa urejanja pridobijo na hitrosti. Med algoritmi s primerjavami je pri ničelni, urejeni in obratno urejeni porazdelitvi izrazito najhitrejše urejanje z zlivanjem. Opozoriti velja, da je hitrost prilagodljivega bitonega urejanja za zaporedja dolžine 2^{19} ali več razmeroma konstantna, medtem ko začne hitrost navadnega in večkoračnega bitonega urejanja pri tej meji padati.

Pri vzporednem urejanju z zlivanjem in vzporednem prilagodljivem bitonem urejanju opazimo, da sta hitrejša pri urejanju zaporedij, katerih dolžina je enaka potenci števila dve. Opisan pojav opazimo pri skoraj vseh porazdelitvah, kar lahko vidimo na slikah A.5, A.6, A.7 in A.8. To je posledica dejstva, da imata algoritma časovno zahtevnejšo implementacijo za urejanje zaporedij, katerih dolžina ni enaka potenci števila dve.

10.3.2 Urejanje 32-bitnih parov ključ vrednost

Trend urejanj je podoben kot pri urejanju 32-bitnih ključev (slika 10.11). Za tabele krajše od 2^{20} je najhitrejše urejanje z zlivanjem (maksimalna hitrost 219 M/s), za daljše tabele pa urejanje po delih (maksimalna hitrost



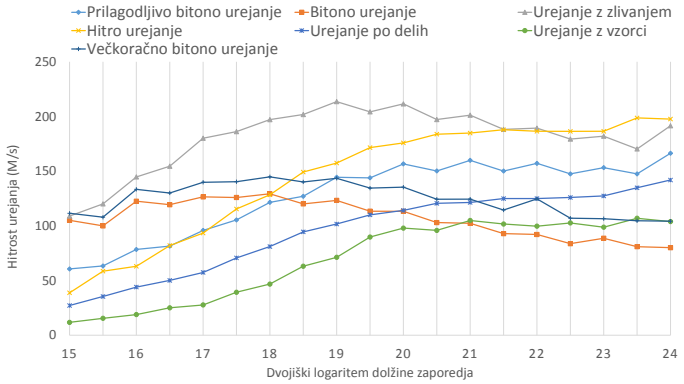
Slika 10.11: Vzporedno urejanje 32-bitnih parov ključ vrednost enakomerne porazdelitve.

261 M/s). Opazimo, da urejanje z vzorci ni več najpočasnejše za zaporedja dolžine 2^{20} ali več, saj postane hitrejše od navadnega in večkoračnega bitonega urejanja.

Grafi ostalih porazdelitev so prikazani na sliki A.6. Podobno kot pri zaporednih algoritmih tudi vzporedni algoritmi za urejanje parov ključ vrednost izgubijo na hitrosti v primerjavi z urejanjem ključev. Med algoritmi, ki ne temeljijo na primerjavah, je najhitrejše urejanje z zlivanjem. Pri ničelni, urejeni in obratno urejeni porazdelitvi je omenjeno urejanje celo hitrejše kot urejanje po delih. Poleg tega postane prilagodljivo bitono urejanje hitrejše od hitrega urejanja, ker pri tvorjenju intervalov dostopa samo do ključev, s čimer se zmanjša količina prenesenih podatkov iz globalnega pomnilnika med urejanjem.

10.3.3 Urejanje 64-bitnih ključev

V nasprotju z vzporednim urejanjem 32-bitnih ključev postane urejanje po delih skoraj najpočasnejši algoritem (slika 10.12). Algoritem deluje počasneje, ker števila vsebujejo dvakrat več bitov in potrebuje dvakrat več faz za urejanje. Za zaporedja krajša od 2^{22}, je najhitrejše urejanje z zlivanjem (maksimalna hitrost 213 M/s), medtem ko je za daljša zaporedja najhitrejše hitro urejanje (maksimalna hitrost 199 M/s).



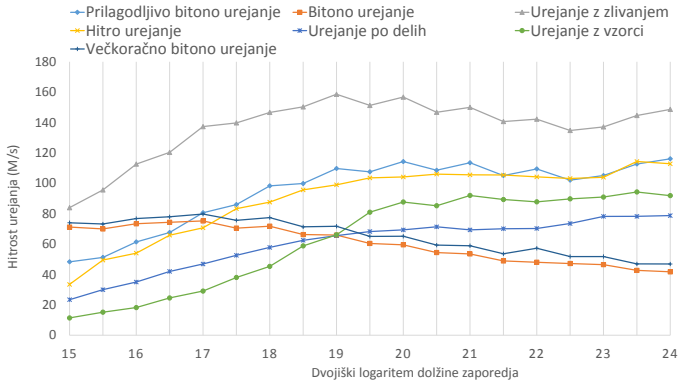
Slika 10.12: Vzporedno urejanje 64-bitnih ključev enakomerne porazdelitve.

Iz slike A.7 vidimo, da so si grafi enakomerne, normalne in delno urejene porazdelitve med seboj zelo podobni. V večini primerov je najhitrejše urejanje z zlivanjem. Pričakovano velja tudi, da je večkoračno bitono urejanje hitrejše od navadnega bitonega urejanja. Pri urejanju zaporedij dolžine 2^{21} ali manj je v večini primerov najpočasnejše urejanje z vzorci, pri daljših zaporedjih pa bitono urejanje.

Pri algoritmu hitrega urejanja delno urejene porazdelitve 64-bitnih ključev (slika A.7) in 64-bitnih parov ključ-vrednost (slika A.8) opazimo, da hitrost urejanja močno pade pri zaporedjih dolžine 2^{21} ali več. To je najverjetneje posledica algoritma za izbiro pivotov, ki izbere pivot kot povprečje minimalne oziroma maksimalne vrednosti zaporedja in starega pivota, zaradi česar lahko pride do slabe izbire pivota in slabe delitve zaporedja. Algoritem bi zelo verjetno deloval bolje, če bi pivota novih dveh zaporedij določili kot povprečje minimuma in maksimuma obeh zaporedij (opisano v poglavju 7.2.3.1). Opisane različice nismo mogli preizkusiti, ker naša *GPE* ne podpira atomarnega minimuma oziroma maksimuma za 64-bitna števila.

10.3.4 Urejanje 64-bitnih parov ključ vrednost

Na sliki 10.13 vidimo, da urejanja izgubijo na hitrosti v primerjavi z vzporednim urejanjem 64-bitnih ključev. Izrazito najhitrejše urejanje je urejanje z zlivanjem, ki dosega hitrosti do 159 M/s. V primerjavi z urejanjem



Slika 10.13: Vzporedno urejanje 64-bitnih parov ključ vrednost enakomerne porazdelitve.

64-bitnih ključev postane prilagodljivo bitono urejanje hitrejše od hitrega urejanja. Razlika med večkoračnim in navadnim bitonim urejanjem se zmanjša. Vzrok za to je najverjetneje posledica omejenega števila registrov na nit, zaradi česar mora algoritem izvajati večkoračne ščepce nižje stopnje. Precej zanimivo je dejstvo, da je urejanje z vzorci hitrejše od urejanja po delih za tabele daljše od 2^{19}.

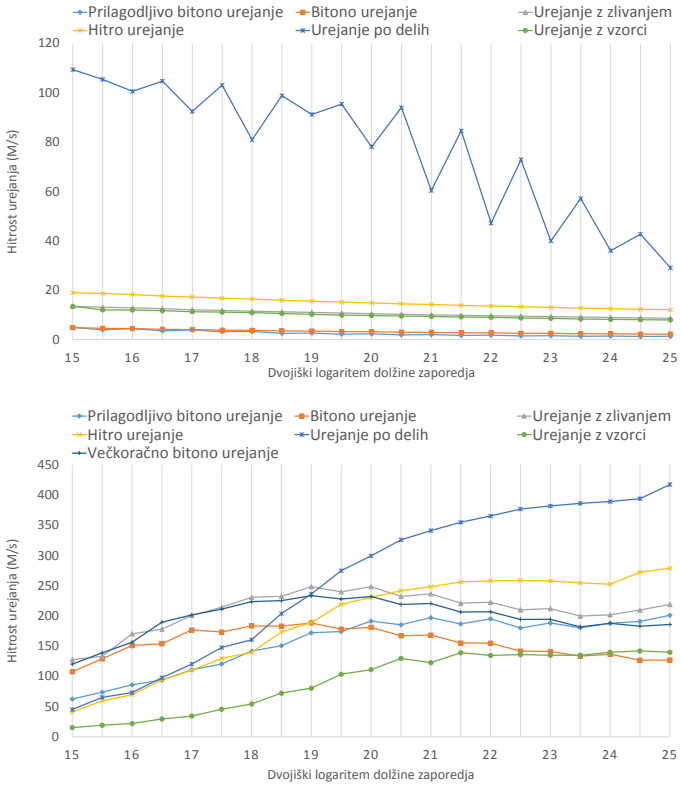
Podobno kot pri urejanju 64-bitnih ključev, tudi pri urejanju parov ključ vrednost hitro urejanje močno izgubi na hitrosti, kadar ureja delno urejeno porazdelitev (obrazloženo v prejšnjem poglavju 10.3.3). Prilagodljivo bitono urejanje postane hitrejše od hitrega urejanja pri vseh porazdelitvah (z izjemo ničelne porazdelitve). To je posledica tega, da prilagodljivo bitono urejanje potrebuje samo ključe za tvorjenje intervalov. Z izjemo ničelne porazdelitve je najhitrejše urejanje z zlivanjem. Zanimivo pri vsem tem je tudi podatek, da urejanje po delih postane v nekaterih primerih celo najpočasnejše. Pri ničelni in urejeni porazdelitvi velja tudi opozoriti, da postane večkoračno bitono urejanje počasnejše od navadnega bitonega urejanja.

10.4 Sklep

Slika 10.14 prikazuje primerjavo zaporednega in vzporednega urejanja 32-bitnih ključev enakomerne porazdelitve. Kot vidimo, je za zaporedno urejanje v splošnem najhitrejše urejanje po delih, ki dosega hitrosti do 110 M/s. Med urejanji s primerjavami je najhitrejše hitro urejanje (19 M/s pri enakomerni porazdelitvi). Pri urejeni porazdelitvi je hitro urejanje v nekaterih primerih hitrejše od urejanja po delih. Izpostaviti velja, da sta urejanje z zlivanjem in urejanje z vzorci podobno hitra, ker urejanje z vzorci temelji na urejanju z zlivanjem. Najpočasnejša algoritma sta navadno in prilagodljivo bitono urejanje. Navadno bitono urejanje je počasno zaradi časovne zahtevnosti $O(n \cdot \log^2 n)$. Vzrok za počasnost prilagodljivega bitonega urejanja je najverjetneje dejstvo, da je algoritem rekurziven.

Pri vzporednih algoritmih rezultati niso tako enotni. Za urejanje 32-bitnih ključev, je za zaporedja daljša od (približno) 2^{20}, najhitrejše urejanje po delih (do 417 M/s pri enakomerni porazdelitvi), medtem ko je za krajša zaporedja najhitrejše urejanje z zlivanjem (do 248 M/s pri enakomerni porazdelitvi). Pri urejanju 32-bitnih parov ključ vrednost velja podobno, pri čemer urejanje z zlivanjem prehiti urejanje po delih ob urejanju ničelne, urejene in obratno urejene porazdelitve. Pri urejanju 64-bitnih ključev in parov ključ vrednost je v večini primerov najhitrejše urejanje z zlivanjem. Izjema je le ničelna porazdelitev, pri kateri je v vseh primerih najhitrejše hitro urejanje, ki dosega hitrosti do 35.000 M/s. Opozoriti velja, da postane urejanje po delih veliko počasnejše, ker je zaradi dvakrat daljših ključev (64-bitnih namesto 32-bitnih) potrebnih dvakrat več faz urejanja. V večini primerov je najpočasnejše urejanje z vzorci, saj pri urejanju 32-bitnih ključev enakomerne porazdelitve dosega hitrost do 142 M/s. Pričakovano je v večini primerov večkoračno bitono urejanje hitrejše od navadnega bitonega urejanja.

Izpostaviti velja, da sta urejanje po delih in urejanje z zlivanjem edina stabilna algoritma, kar pomeni, da ohranjata vrstni red ponavljajočih se elementov. Pri urejanju z vzorcu je stabilna samo zaporedna implementacija. Vzporedna implementacija ni stabilna, ker je bitono urejanje, uporabljeno za

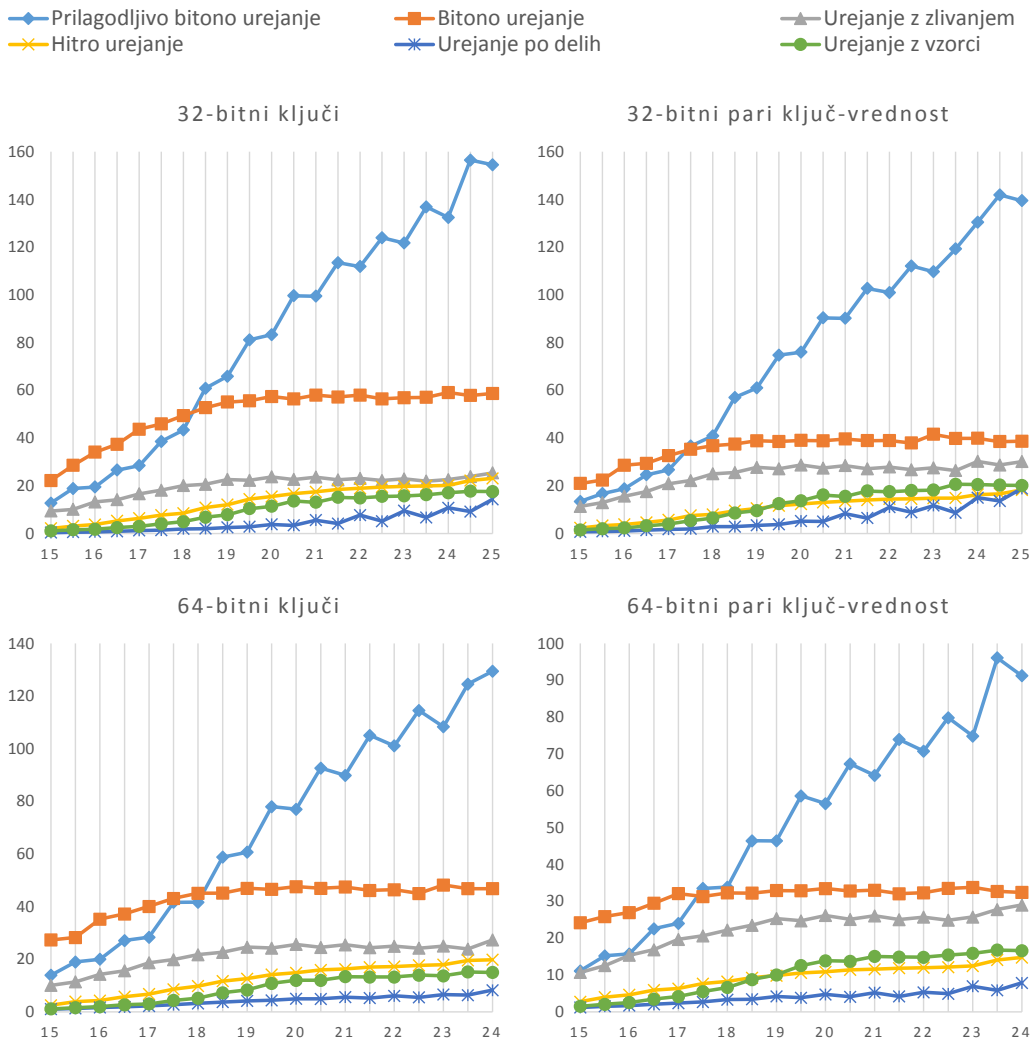


Slika 10.14: Primerjava zaporednega (zgoraj) in vzporednega (spodaj) urejanja 32-bitnih ključev enakomerne porazdelitve.

urejanje sektorjev, nestabilno.

Na sliki 10.14 opazimo tudi zanimivost, da pri zaporednih algoritmih z večanjem velikosti vhodnih podatkov učinkovitost algoritmov pada, pri vzporednih algoritmih pa velja ravno obratno. Do opisanega pojava pride najverjetneje zato, ker je pri zaporednem urejanju zelo dolgih zaporedij potrebno izvajati več menjav pomnilniških blokov v predpomnilniku kot pri krajših zaporedjih, kjer je veliko večja verjetnost, da se potrebni elementi že nahajajo v predpomnilniku. Vzporedni algoritmi urejanja števil pa postajajo učinkovitejši pri daljših zaporedjih, ker je potrebnega več vzporednega procesiranja, zaradi česar prenosi podatkov med globalnim in deljenim pomnilnikom pridejo manj do izraza.

Slika 10.15 prikazuje faktorje pohitritve vzporednih implementacij v primerjavi z zaporednimi implementacijami pri urejanju enakomerne porazde-



Slika 10.15: Pohitritev vzporednih implementacij v primerjavi z zaporednimi implementacijami pri urejanju enakomerne porazdelitve (os X: dvojiški logaritem dolžine zaporedja, os Y: faktor pohitritve).

litve. Kot vidimo, dosežemo največjo pohitritev pri prilagodljivem in navadnem bitonem urejanju. Kot smo omenili je urejanje po delih z naskokom najhitrejše zaporedno urejanje, zato z njim dosežemo najmanjšo pohitritev. Opazimo, da je trend pohitritev zelo podoben pri vseh vhodnih podatkih. V splošnem dosežemo največje pohitritve pri 32-bitnih ključih, najmanjše pa pri 64-bitnih parih ključ vrednost. Največjo pohitritev dosežemo pri prilagodljivem bitonem urejanju 32-bitnih ključev, in sicer do 157-kratno pohitritev.

Podobno velja tudi za navadno bitono urejanje. Do tako velikih pohitritev pride zaradi tega, ker sta zaporedni implementaciji omenjenih algoritmov zelo neučinkoviti. Če izvzamemo zgoraj navedeni bitoni urejanji, dosežemo največjo pohitritev pri urejanju z zlivanjem, in sicer do 30-kratno pohitritev. Opazimo tudi, da faktorji pohitritve naraščajo z daljšanjem vhodnega zaporedja. To je posledica zgoraj opisane ugotovitve, da pri zaporednih urejanjih hitrost pada in pri vzporednih narašča.

Pri vseh algoritmih se zaradi povečanja velikosti elementov zaporedja (bodisi zaradi urejanja parov ključ vrednost ali zaradi prehoda iz 32-bitnih na 64-bitna števila) zmanjša hitrost urejanj. Padec hitrosti je odvisen od porazdelitve, algoritma urejanja in dolžine vhodnega zaporedja. Zanimivo dejstvo je tudi, da je pri krajših tabelah (do dolžine približno 2^{18}) zaporedno urejanje po delih hitrejše od nekaterih vzporednih algoritmov (npr. od urejanja z vzorci, urejanja po delih in hitrega urejanja).

Poglavje 11

Zaključek

V magistrskem delu smo preučili, implementirali in primerjali zaporedne in vzporedne implementacije algoritmov za urejanje podatkov, in sicer hitro urejanje [9], bitono urejanje [28], prilagodljivo bitono urejanje [29], urejanje po delih [30], urejanje z zlivanjem [30] in urejanje z vzorci [13]. Vsi našteti algoritmi, z izjemo urejanja po delih, temeljijo na primerjavah. Med vzporednimi implementacijami se je za najbolj zahtevno izkazalo hitro urejanje, za najpreprostejše pa bitono urejanje. Velike težave smo imeli tudi s tvorjenjem intervalov pri vzporednem prilagodljivem bitonem urejanju, ker navodila v članku [29] niso dovolj podrobna. Ugotovili smo, da so vzporedne implementacije veliko bolj zahtevne in obsežne kot zaporedne.

Med zaporednimi urejanji je v splošnem najhitrejše urejanje po delih, ki dosega hitrosti do 110 M/s. Med urejanji s primerjavami je najhitrejše hitro urejanje, ki pri enakomerni porazdelitvi dosega hitrosti do približno 19 M/s. Presenetljivo je urejanje po delih najhitrejše tudi pri urejanju 64-bitnih števil.

Pri vzporednih urejanjih rezultati niso tako enotni. Za urejanje 32-bitnih ključev enakomerne porazdelitve, je za zaporedja daljša od 2^{20}, najhitrejše urejanje po delih (do 417 M/s), za krajša zaporedja pa urejanje z zlivanjem (do 248 M/s). Urejanju po delih pričakovano občutno pade hitrost pri urejanju 64-bitnih vrednosti. Pri urejanju 64-bitnih ključev in parov ključ-vrednost je v večini primerov najhitrejše urejanje z zlivanjem (do 213 M/s).

Tukaj velja izpostaviti, da sta urejanje po delih in urejanje z zlivanjem edina stabilna algoritma. To velja tako za zaporedno kot tudi za vzporedno implementacijo. Pri urejanju z vzorcu je stabilna samo zaporedna implementacija.

S tem smo potrdili našo domnevo, da bo za kratke ključe najhitrejši algoritem urejanja po delih. Pri zaporednih algoritmih to velja celo pri 64-bitnih številih. Potrdili smo tudi domnevo, da bodo nekateri algoritmi občutljivi na določene porazdelitve vhodnih podatkov. Tako je na primer zaporedno hitro urejanje zelo počasno pri ničelni porazdelitvi, urejanje z zlivanjem je hitrejše pri delno urejenih zaporedjih, in podobno.

Z vzporednimi implementacijami dosežemo pohitritve v primerjavi z zaporednimi implementacijami. Največjo pohitritev dosežemo pri prilagodljivem in navadnem bitonem urejanju, kjer dosežemo do 157-kratno pohitritev. Do tako velike pohitritve pride zaradi tega, ker sta zaporedni implementaciji omenjenih algoritmov zelo neučinkoviti. Prilagodljivo bitono urejanje je rekurziven algoritem, zaradi česar je potrebno veliko dodatnega pisanja na sklad. Navadno bitono urejanje ima časovno zahtevnost $O(n \cdot \log^2 n)$ in je zato neučinkovito. Če izvzamemo omenjeni bitoni urejanji, dosežemo največjo pohitritev pri urejanju z zlivanjem, in sicer do 30-kratno pohitritev. Najmanjšo pohitritev dosežemo pri urejanju po delih, in sicer do 19-kratno pohitritev.

Naši rezultati so delno skladni tudi z rezultati Mišića in Tomaševića [26]. Po njunih ugotovitvah je vzporedno urejanje po delih [1] za približno 45 % hitrejše od urejanja z zlivanjem [2] in hitrega urejanja [9]. Naši rezultati so podobni za zaporedja dolžine približno 2^{21} ali več. Pri enakomerni porazdelitvi dolžine 2^{25} je urejanje po delih za približno 50 % hitrejše od hitrega urejanja in celo za 90 % hitrejše od urejanja z zlivanjem. Naše ugotovitve se razlikujejo od [26] za zaporedja krajša od 2^{20}, kjer je urejanje z zlivanjem hitrejše od urejanja po delih. Pri enakomerni porazdelitvi 32-bitnih ključev dolžine 2^{15} je hitrejše celo za 165 %. Poleg tega naši implementaciji urejanja z zlivanjem in hitrega urejanja nista tako podobno hitri kot v [26]. Njuni hitrosti sta odvisni od dolžine vhodnega zaporedja, porazdelitve in obsega podatkov (32-bit ali 64-bit, ključ ali ključ-vrednost). V splošnem se urejanje

z zlivanjem izkaže za hitrejše od hitrega urejanja. Glavni razlog za razliko med rezultati leži v tem, da sta avtorja [26] uporabila optimizirane implementacije urejanj iz javno dostopnih knjižnic. Poleg tega smo preizkuse izvajali na novejši in zmogljivejši strojni opremi, pri čemer smo urejanja preizkušali na veliko več različnih vhodnih zaporedjih kot [26].

Naštete implementacije vzporednih algoritmov smo delno tudi izboljšali. Za večkoračno bitono urejanje smo izdelali postopek za izgradnjo d-koračnega dela. Z rekurzivnimi funkcijami za *branje*, *pisanje* in *primerjanje* elementov smo optimizirali d-koračne ščepce ter s tem zagotovili, da se elementi vedno nahajajo v registrih niti, kar omogoča hitrejše izvajanje. S pomočjo optimizirane redukcije [17] in optimizirane komulativne vsote [33] smo pohitrili hitro urejanje [9]. Izboljšali smo tudi postopek iskanja pivotov v hitrem urejanju, ki smo jih določili kot povprečje starega pivota in minimuma oziroma maksimuma novega podzaporedja. S pomočjo komulativne vsote za predikate [18] smo pohitrili urejanje po delih [30]. Pri urejanju z vzorci [13] smo izdelali tudi preprost in učinkovit postopek za shranjevanje elementov v sektorje. Za vse algoritme smo tudi zagotovili, da lahko urejajo zaporedja poljubne dolžine.

V nadaljnjem delu bi lahko algoritme preizkusili pri urejanju velikih količin podatkov, kot je to storil Amirul s sodelavci [3] z urejanjem nizov dolžine 20 znakov. V tem primeru vsak element zaporedja zahteva veliko več prostora kot pri urejanju števil, zaradi česar bi se učinkovitost vzporednih algoritmov zmanjšala. Zelo verjetno bi se tudi močno zmanjšala učinkovitost urejanja po delih, ker nizi zavzamejo veliko več bitov kot 32-bitna oziroma 64-bitna števila. Podrobneje bi lahko tudi preučili, zakaj pride pri zaporednem urejanju po delih do tako velikih odstopanj v hitrosti urejanja. Za nadaljnje delo so vse implementacije javno dostopne na [8].

Literatura

[1] Cudpp: CUDA data parallel primitives library. `https://github.com/cudpp/cudpp/`, 2015.

[2] Thrust 1.1: parallel algorithms library. `https://github.com/thrust/thrust`, 2015.

[3] M. Amirul, M. Omar, N. Aini, E. Karuppiah, Mohanavelu, S. S. Meng, and P. K. Chong. Sorting very large text data in multi GPUs. In *Computing and Engineering (ICCSCE), 2012 IEEE International Conference on Control System*, pages 160–165, Nov 2012.

[4] R. Baraglia, G. Capannini, F. M. Nardini, and F. Silvestri. Sorting using bitonic network with CUDA. In *7th Workshop on Large- Scale Distributed Systems for Information Retrieval (LSDS-IR)*, July 2009.

[5] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[6] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY, USA, 1986.

[7] D. Božidar and T. Dobravec. Comparison of parallel sorting algorithms. Concurrency and Computation: Practice and Experience (submitted for publication), June 2015.

[8] D. Božidar and T. Dobravec. A comparison study between sequential and parallel sorting algorithms. `https://github.com/darkobozidar/sequential-vs-parallel-sort`, 2015.

[9] D. Cederman and P. Tsigas. GPU-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[10] D. Z. Chen. Efficient parallel binary search on sorted arrays, with applications. *IEEE Trans. Parallel Distrib. Syst.*, 6(4):440–445, Apr. 1995.

[11] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] F. Dehne and H. Zaboli. Deterministic sample sort for GPUs. *CoRR*, abs/1002.4464, 2010.

[14] N. K. Govindaraju, N. Raghuvanshi, M. Henson, D. Tuft, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina, 2005.

[15] A. Greßand G. Zachmann. GPU-abisort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 45–45, Washington, DC, USA, 2006. IEEE Computer Society.

[16] T. Hagerup and C. Rüb. Optimal merging and sorting on the erew pram. *Inf. Process. Lett.*, 33(4):181–185, Dec. 1989.

[17] M. Harris. Optimizing Parallel Reduction in CUDA. Technical report, nVidia, 2008.

[18] M. Harris and M. Garland. Chapter 3 - Optimizing Parallel Prefix Operations for the Fermi Architecture. In W.-m. W. Hwu, editor, {*GPU*}

Computing Gems Jade Edition, Applications of GPU Computing Series, pages 29 – 38. Morgan Kaufmann, Boston, 2012.

[19] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3(39):851–876, 2007.

[20] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. *J. Parallel Distrib. Comput.*, 52(1):1–23, July 1998.

[21] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, Jan. 1962.

[22] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[23] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–10, April 2010.

[24] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998.

[25] D. Merrill and A. S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[26] M. Misic and M. Tomasevic. Data sorting using graphics processing units. In *Telecommunications Forum (TELFOR), 2011 19th*, pages 1446–1449, Nov 2011.

[27] Nvidia. *CUDA C Programming Guide*, 6.5 edition, August 2014.

[28] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. *Concurr. Comput. : Pract. Exper.*, 23(7):681–693, May 2011.

[29] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 227–237, 2012.

[30] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[31] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[32] R. Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, Oct. 1978.

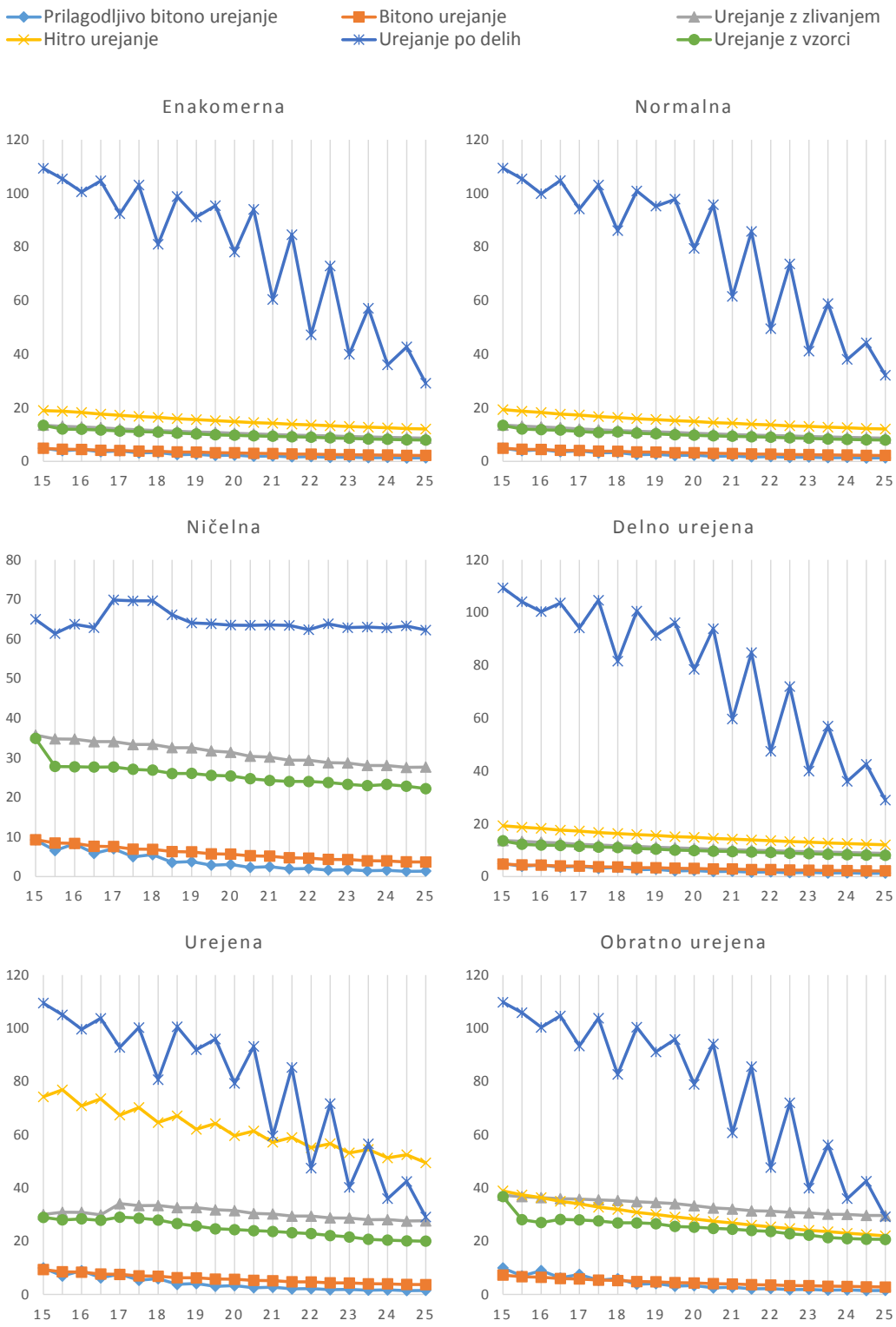
[33] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA Corporation, Dec. 2008.

[34] R. C. Singleton. Algorithm 347: An efficient algorithm for sorting with minimal storage [m1]. *Commun. ACM*, 12(3):185–186, Mar. 1969.

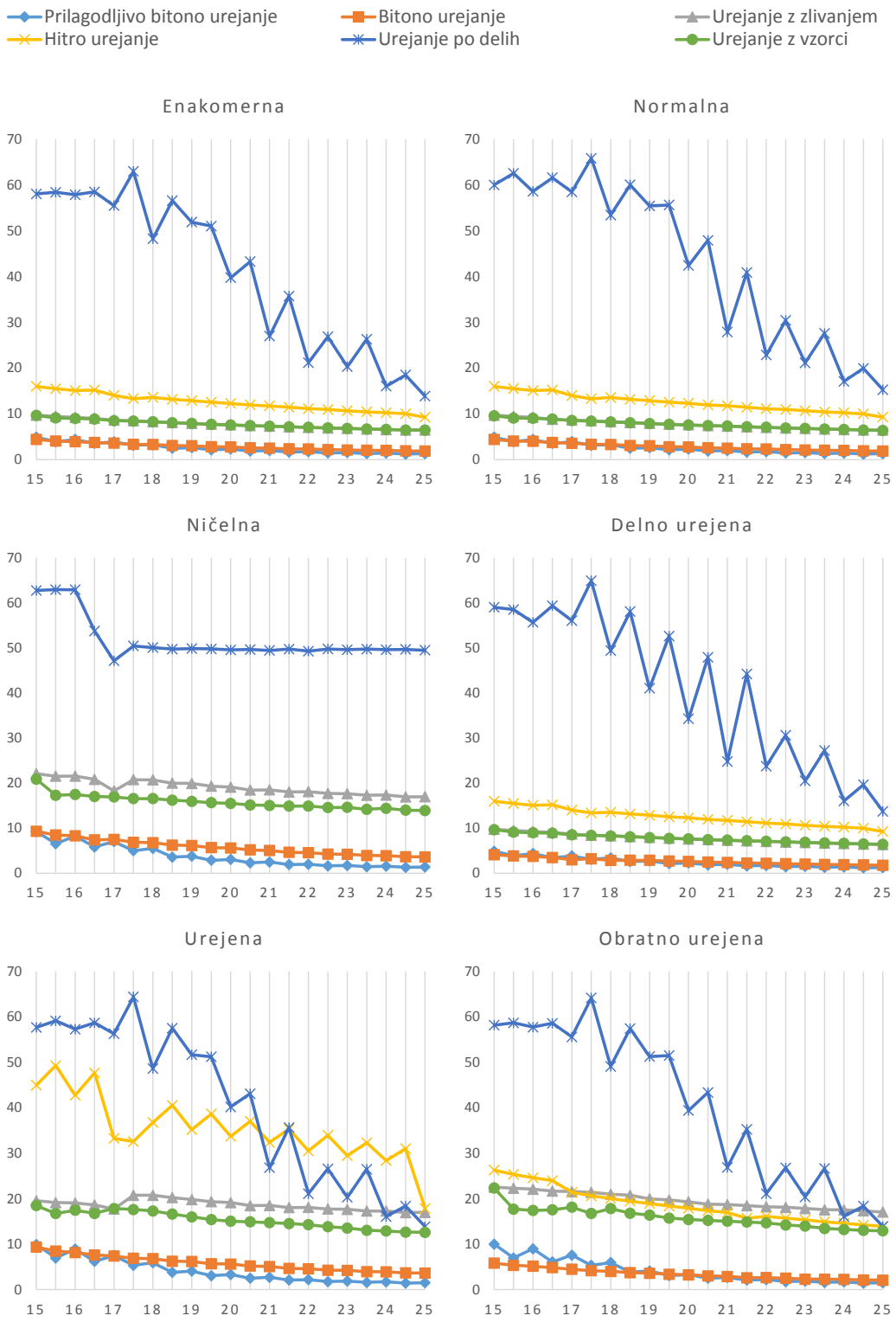
Dodatek A

Rezultati urejanj

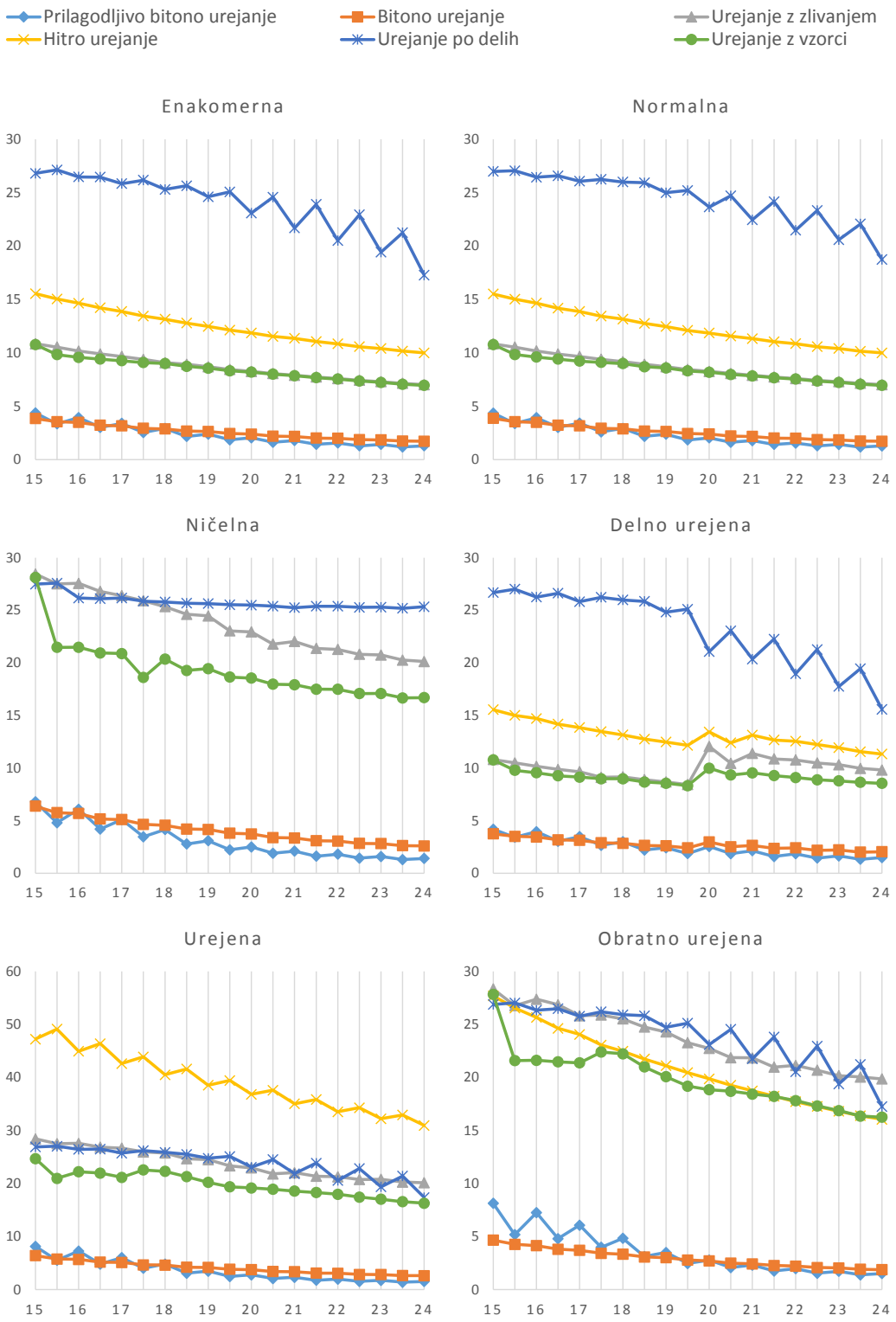
V dodatek smo vključili rezultate urejanj vseh šestih vhodnih porazdelitev, ki smo jih opisali v poglavju 10.1. V poglavjih 10.2 in 10.3 smo prikazali samo rezultate urejanja enakomerne porazdelitve. Na osi x se nahajajo dvojiški logaritmi dolžine zaporedja. Os y predstavlja hitrost urejanja oziroma število milijonov elementov, ki jih algoritem lahko uredi v sekundi. Izpostaviti velja, da grafi zaporednega urejanja ničelne porazdelitve ne vsebujejo hitrega urejanja, ker ta dosega hitrost le do 0.3 M/s. Enako velja tudi za vzporedno urejanje ničelne porazdelitve, pri katerem je hitro urejanje daleč najhitrejše in dosega hitrosti do 35.000 M/s.



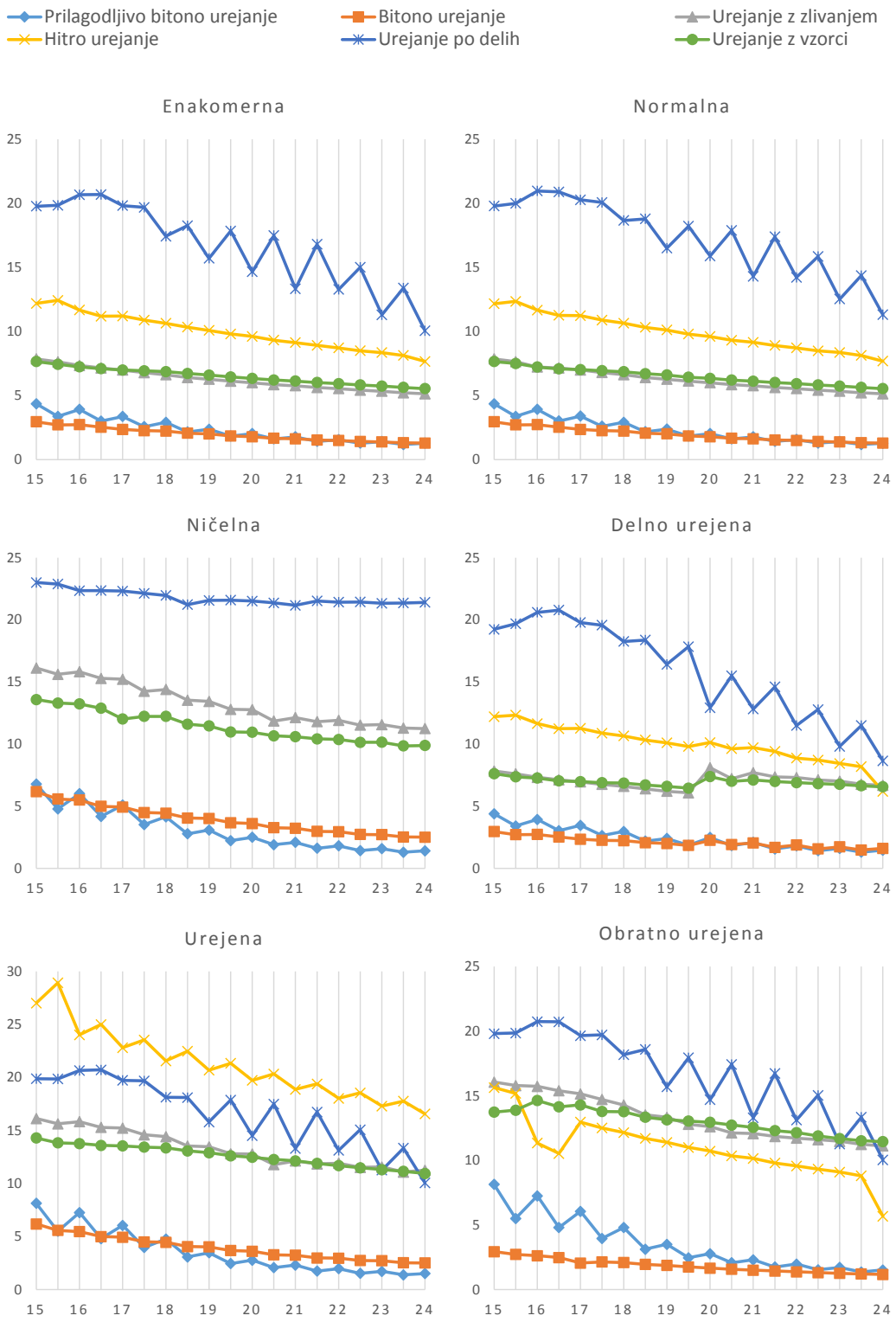
Slika A.1: Zaporedno urejanje 32-bitnih ključev (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



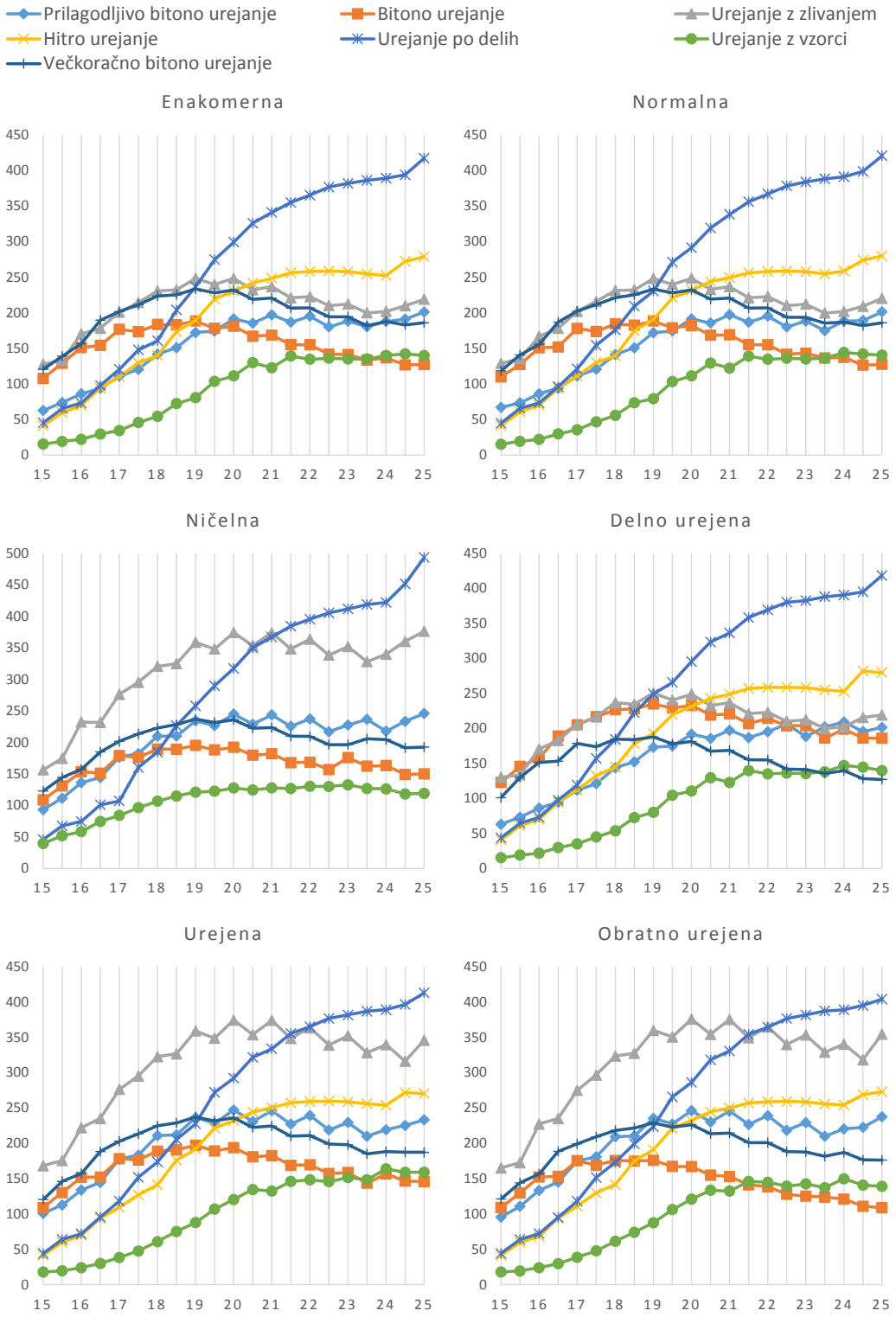
Slika A.2: Zaporedno urejanje 32-bitnih parov ključ vrednost (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



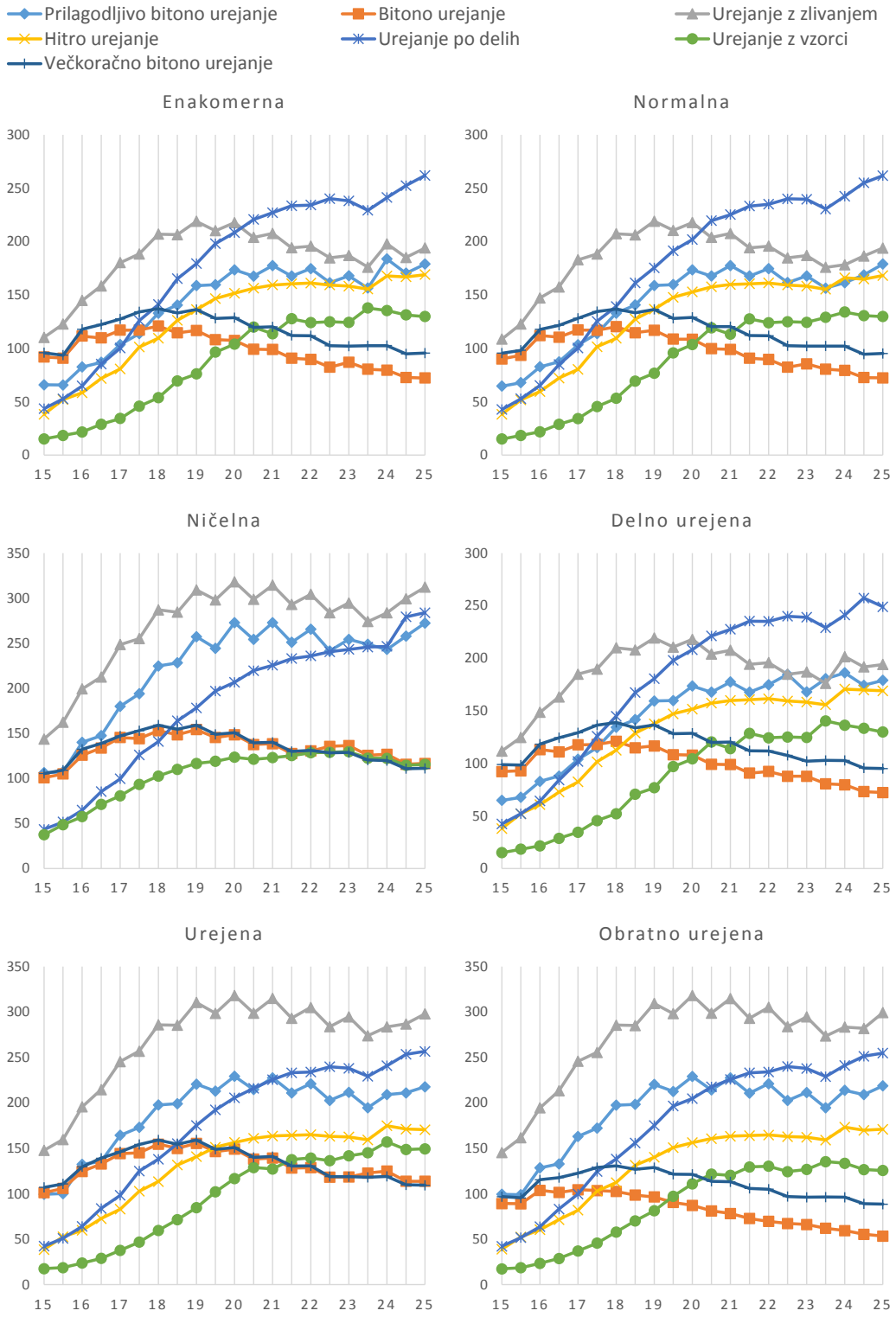
Slika A.3: Zaporedno urejanje 64-bitnih ključev (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



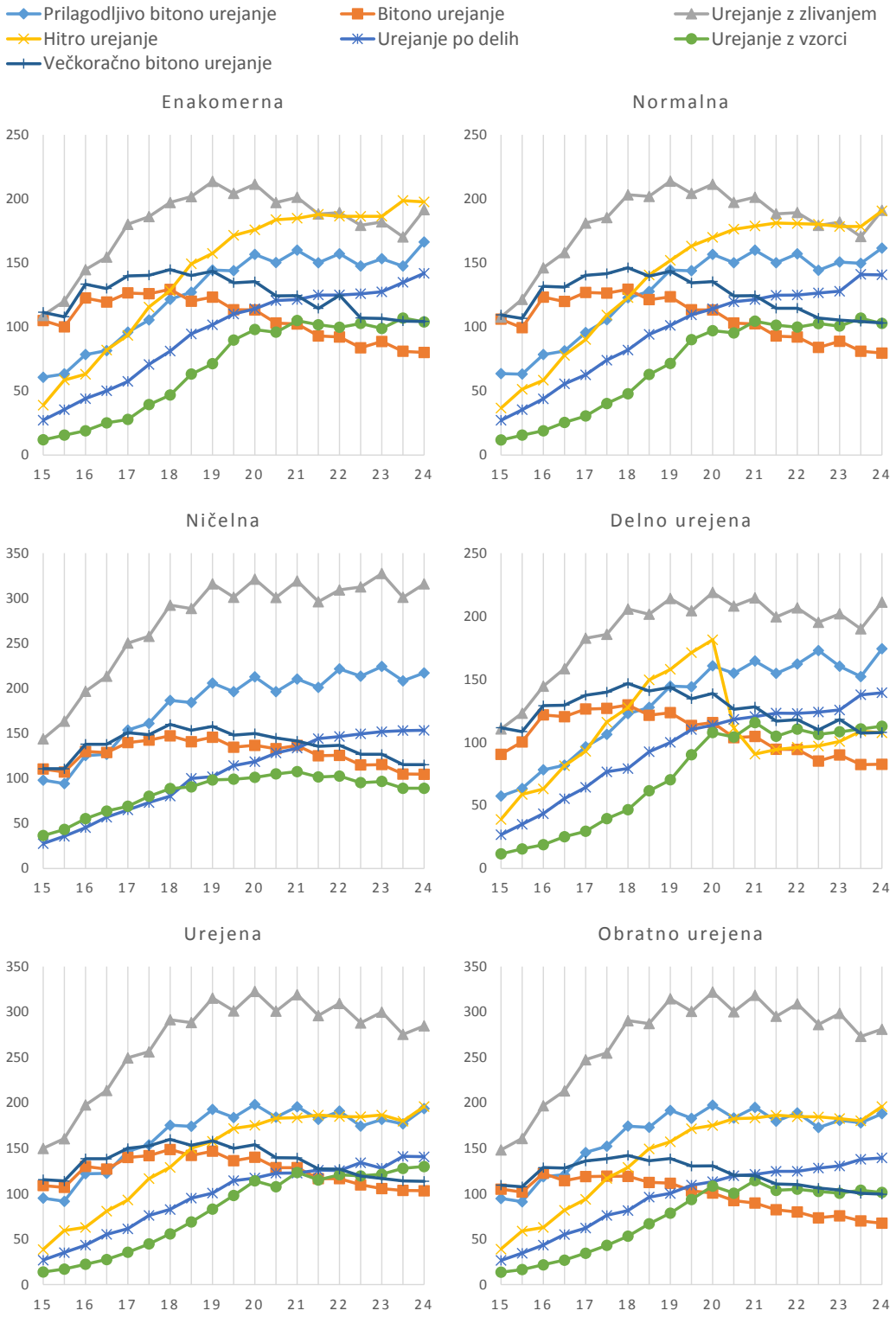
Slika A.4: Zaporedno urejanje 64-bitnih parov ključ vrednost (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



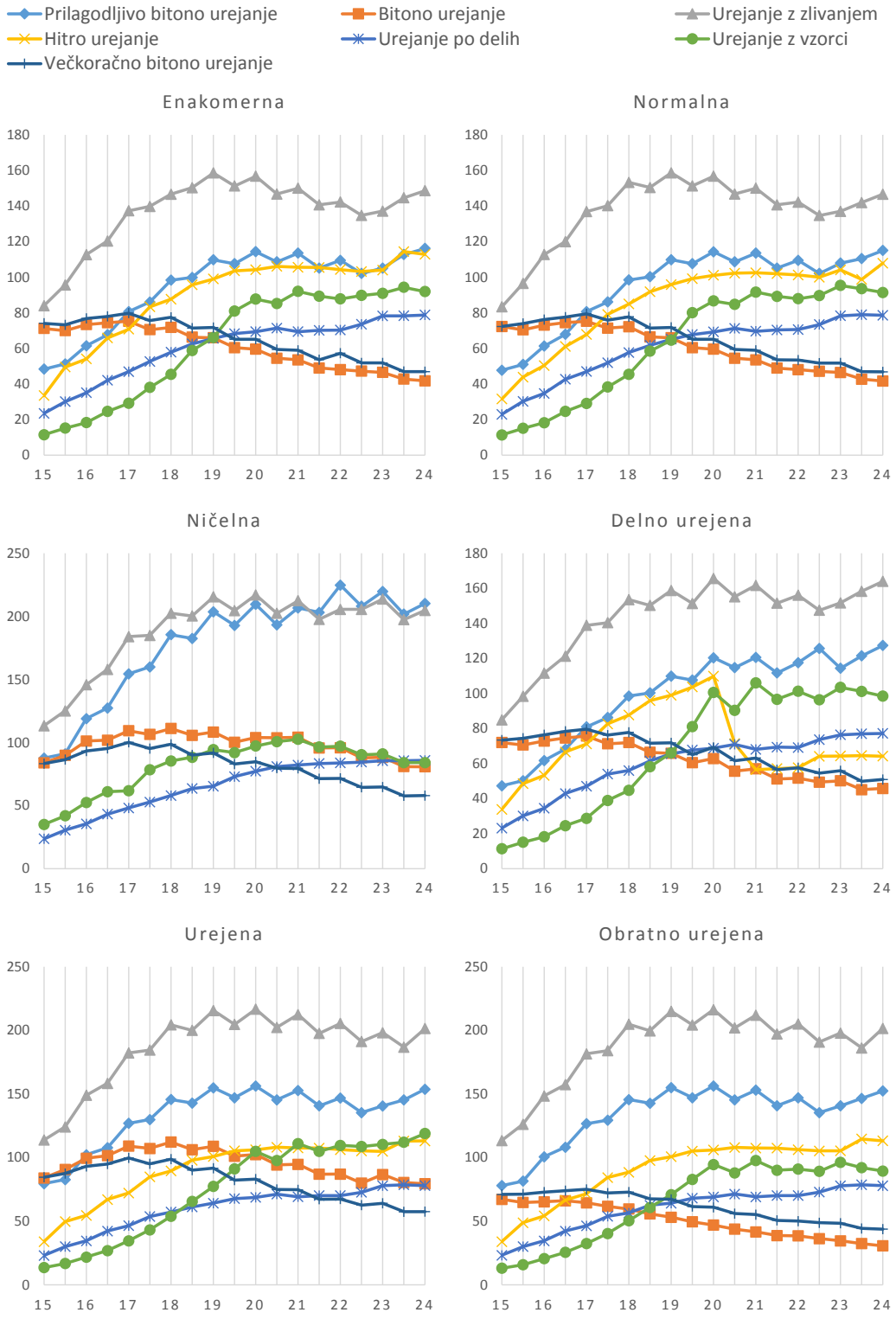
Slika A.5: Vzporedno urejanje 32-bitnih ključev (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



Slika A.6: Vzporedno urejanje 32-bitnih parov ključ vrednost (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



Slika A.7: Vzporedno urejanje 64-bitnih ključev (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).



Slika A.8: Vzporedno urejanje 64-bitnih parov ključ vrednost (os X: dvojiški logaritem dolžine zaporedja, os Y: hitrost urejanja v M/s).