ELEKTROTEHNIŠKI VESTNIK 78(X):
EXISTING SEPARATE ENGLISH EDITION

Cevovod in stiskanje podatkov

Jaka Vrhovec, Adrijan Rogan

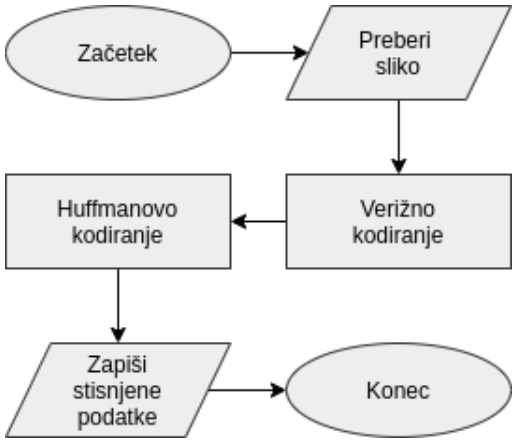
Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Večna pot 113, 1000 Ljubljana, Slovenija

Povzetek. Stiskanje podatkov je pomembno za učinkovito prenašanje sporočil. Dobro poznan in relativno enostaven način stiskanja je verižno kodiranje, ki mu sledi Huffmanovo kodiranje z vnaprej določenimi ali dinamičnimi kodnimi zamenjavami. Za stiskanje ene datoteke moramo opraviti štiri stopnje: (i) branje datoteke, (ii) verižno kodiranje, (iii) Huffmanovo kodiranje in (iv) zapis stisnjenih podatkov. Posamezne korake algoritma težko pohitrimo, zato raje konstruiramo cevovod po principu proizvajalec-porabnik, da se lahko koraki izvajajo vzporedno.

Ključne besede: cevovod, nitenje, proizvajalec–porabnik, stiskanje podatkov, verižno kodiranje, Huffmanovo kodiranje

1 SEKVENČNI ALGORITEM

Pri sekvenčnem algoritmu posamezne stopnje kličemo zaporedno. Stopnje implementiramo posebej, da jih lahko kasneje uporabimo še za cevovod brez večjih sprememb. Če imamo slik več, lahko algoritem na sliki 1 poganjamo v zanki.



Slika 1: Koraki sekvenčnega algoritma

Na hitro si oglejmo vse štiri dele algoritma. Dejanska koda je napisana v jeziku *C++*, izseki pa bodo predstavljeni v psevdokodi. Najprej omenimo, da za lažji zapis definiramo dva tipa, ki se pogosto pojavljata, dvodimenzionalni vektor (hranjenje podatkov po vrsticah in stolpcih) in par celega števila z logično vrednostjo (število ponovitev črnih ali belih točk).

```
template <typename T>
using Vec = vector<vector<T>>;

typedef pair<int, bool> int_bool;
```

V prvem koraku moramo sliko prebrati. Odločili smo se, da bodo vse slike shranjene v formatu PNG; za branje uporabljamo knjižnico *stb*[*]. Pri branju preverimo, ali je slika pričakovane velikosti: stran A4 je sestavljena iz 1145 vrstic s po 1728 slikovnimi točkami. Sliko preberemo kot sivinsko; knjižnica sama poskrbi za ustrezno pretvorbo, če je potrebna, in nam vrne **unsigned char** $*$, kjer vsaka komponenta predstavlja eno slikovno točko. Kot rezultat prve stopnje vpeljemo **struct** image, ki hrani vrnjene slikovne točke, širino in višino.

Na rezultatu iz prve stopnje enostavno opravimo verižno kodiranje. Sliko si predstavljamo kot dvodimenzionalno strukturo in štejemo število enakih zaporednih slikovnih točk.

V tretjem koraku moramo opraviti Huffmanovo kodiranje z optimalnim Huffmanovim kodom, zato moramo zgraditi Huffmanovo drevo. Za gradnjo drevesa najprej pripravimo frekvence za bele in črne znake raznih dolžin. *Node* je podatkovna struktura za binarno drevo.

```
BLACK = true
WHITE = false

class Node {
  int value
  bool bit
  Node left
  Node right

  constructor(int value, bool bit)
}
```

Pripravimo funkcijo prepare, ki nam vrne multimap (preslikava, kjer ima lahko več elementov enak ključ) za bele in za črne točke, ki slika iz frekvenc v liste, ki vsebujejo osnovne znake (tj. število zaporednih točk enake barve). Elementi so urejeni naraščajoče po ključih.

Prejet
Odobren

[*] GitHub nothings/stb – single-file public domain libraries

```
multimap<int, Node>[2] prepare(
  Vec<int_bool> rle
) {
  // {dolzina zaporedja} -> {frekvenca}
  map<int, int> white
  map<int, int> black

  for line in rle
    for {count, color} in line
      if color is WHITE -> white[count]++
      else              -> black[count]++

  // {frekvenca} -> {list oz. vozlisce}
  multimap<int, Node> white_nodes
  multimap<int, Node> black_nodes

  for {length, frequency} in white
    node = Node(length, WHITE)
    white_nodes.add({frequency, node})

  for {length, frequency} in black
    node = Node(length, BLACK)
    black_nodes.add({frequency, node})

  return {white_nodes, black_nodes}
}
```

Sedaj lahko za črne in bele točke zgradimo posebej drevesi, tako da združujemo vozlišča z najmanjšimi frekvencami. Ker je multimap urejen, preprosto združujemo prvi in drugi element, dokler ne dobimo korena drevesa.

```
Node huffman_tree(
  multimap<int, Node> freq
) {
  while freq.size > 1
    {int f1, Node c1} = freq.first
    {int f2, Node c2} = freq.second
    // Levi sin, desni sin, frekvenca
    Node parent = Node(c1, c2, f1 + f2)

    freq.add({parent.value, parent})
    freq.remove(c1)
    freq.remove(c2)

  // Vrnemo koren
  return freq.first.second
}
```

Zgrajeno drevo sedaj uporabimo na verižno kodiranih podatkih. Za lažje kodiranje iz drevesa zgradimo *map*, ki slika iz dolžine zaporedja enakih točk v kodno zamenjavo. Kodne zamenjave gradimo iz korena binarnega drevesa, tako da ob vsaki vejitvi dodamo obstoječi kodni zamenjavi 0 ali 1, odvisno ali gremo v levega ali desnega sina. V korenu začnemo s prazno kodno zamenjavo.

```
// Preslikavi iz dolzin zaporedij
// v kodne zamenjave
map<int, string> white
map<int, string> black


// int_bool: dolzina zaporedja in barva
Vec<string> encode(Vec<int_bool> rle) {
  // 2D sprehod po verizno
  // kodiranih podatkih
}
```

S sprehodom po verižno kodiranih podatkih in z zgrajenimi preslikavami opravimo Huffmanovo kodiranje. Za vsak par dolžine zaporedja in barve v končni dvodimenzionalni vektor dodamo ustrezno kodno zamenjavo.

Na zadnjem, četrtem koraku, Huffmanovo kodirane podatke zapišemo v datoteko. Pomembno je, da na začetku datoteke zapišemo zaglavje z Huffmanovima drevesoma za bele in črne slikovne točke, da lahko podatke kasneje dekodiramo. Podatke lahko zapišemo v tekstovno datoteko, še boljše pa je, da podatke zapišemo dvojiško preko posameznih bajtov.

2 Preprost cevovod

Ob ideji, da je slik lahko več, opazimo pomanjkljivost sekvenčnega algoritma. Preden začnemo brati naslednjo sliko, se mora v celoti zaključiti kodiranje in zapisovanje trenutne slike. Ker posamezne stopnje težko pohitrimo, uporabimo cevovod. Z uporabo cevovoda lahko naslednjo sliko začnemo brati, takoj ko začnemo verižno kodiranje trenutne slike. Podobno velja še za druge stopnje algoritma. Vendar, namesto da bi se ukvarjali s tem, da prebrano sliko oddamo naslednji stopnji, verižnemu kodiranju, in šele nato začnemo brati novo sliko, lahko prebrano sliko oddamo v medpomnilnik in se tako izognemo neposredni komunikaciji. Posamezne stopnje algoritma izvajajo niti, pri čemer zaporedni stopnji (recimo branje slike in verižno kodiranje) povezuje medpomnilnik.

Slika 2: Visokonivojski pregled cevovoda

Konstrukcija nam da preprost cevovod, kot je prikazan na sliki 2, kjer ena nit skrbi za izvajanje ene stopnje algoritma. Velikost medpomnilnika je smiselno omejiti, saj je lahko naslednja stopnja počasnejša od trenutne, zaradi česar bi zasedenost pomnilnika rasla v nedogled.

Medpomnilnik 1 na sliki 2 vsebuje imena datotek, ki jih bo prebrala prva stopnja cevovoda. Namesto njega lahko uporabimo globalen seznam, iz katerega nit prevzema naloge. Smiselno je, da medpomnilnik 1

ne omejimo preveč (elementi so po velikosti majhni – nizi, ki predstavljajo poti do slik), saj bi s tem lahko blokirali glavno nit, ali pa uporabimo drugačen pristop za dodajanje nalog v cevovod.

Za medpomnilnik smo pripravili abstrakcijo Stream, ki ponuja dodajanje in jemanje nalog s pripadajočimi ključi po principu proizvajalec-porabnik. Parameter K je tip ključa (predstavlja nalogo in ga lahko uporabimo za ime izhodne datoteke), parameter T pa tip podatkov v medpomnilniku.

```
class Stream<K, T> {
  void produce(K key, T data)
  pair<K, T> consume()
}
```

Abstrakcija je implementirana z vrsto in pthread knjižnico v razredu FifoStream. Za dodajanje nalog zaklenemo ključavnico, preverimo zasedenost medpomnilnika (po potrebi čakamo, dokler je vrsta polna s pogojem za bujenje), dodamo nalogo, odklenemo ključavnico in sprožimo pogoj za buditev. Za jemanje nalog zaklenemo ključavnico, po potrebi čakamo, dokler je vrsta prazna (s pogojem za bujenje), prevzamemo nalogo, odklenemo ključavnico in sprožimo pogoj za buditev.

```
class FifoStream<K, T> : Stream<K, T> {

  queue<pair<K, T>> queue;
  mutex mtx;
  condition produced;
  condition consumed;
  int max_size;

  void produce(K key, T data) {
    mutex_lock(mtx)
    while (queue.size >= max_size)
      condition_wait(consumed, mtx)
    queue.push(pair(key, data))
    mutex_unlock(mtx)
    condition_signal(produced)
  }

  pair<K, T> consume() {
    mutex_lock(mtx)
    while (queue.empty) {
      condition_wait(produced, mtx)
    }
    pair<K, T> keydata = queue.front
    queue.pop()
    mutex_unlock(mtx)
    condition_signal(consumed)
    return keydata
  }
};
```

Posamezen medpomnilnik še ni dovolj za implementacijo stopnje cevovoda. Vsaka stopnja potrebuje dostop do dveh medpomnilnikov. Iz prvega bo jemala naloge, ki jo bo obdelala, in obdelane podatke predala v drug medpomnilnik kot nalogo za naslednjo stopnjo. Izjema je zadnja stopnja cevovoda, ki ima samo podporo za jemanje nalog. Za enostavnejši dostop do medpomnilnikov to funkcionalnost enkapsuliramo v PipelineStage, ki ga lahko kot argument posredujemo nitim.

```
class PipelineStage<K, T, U> {
  Stream<K, T> consumer
  Stream<K, U> producer

  pair<K, T> consume()
    return consumer.consume()

  void produce(K key, U data)
    return producer.produce(key, data)
};
```

Stopnjo cevovoda dodamo tako, da implementiramo funkcijo, ki jo bo izvajala nit, in ji kot argument podamo kazalec na PipelineStage, preko katerega potem sprejema in oddaja naloge v neskončni zanki. V glavni niti ustvarimo potrebne medpomnilnike, PipelineStage objekte in zaženemo niti. Za stopnjo branja izgleda v C++ to takole:

```
void* read(void* arg) {
  // Argument castamo v PipelineStage
  PipelineStage<...>* stage = ...
  while (true) {
    auto p = stage->consume();
    auto key = p.first;
    auto filename = p.second;
    auto im_data = Input::read(filename);
    stage->produce(key, im_data);
  }
  return nullptr;
}

// Imena datotek za branje
// Omejitev velikosti 0 = neomejeno
FifoStream<int, string> instream(0);
// Slike za verizno kodiranje
FifoStream<int, struct image> imstream(0);

// Stopnja cevovoda za branje
PipelineStage<int, string, struct image>
read_stage(&instream, &imstream);

// Ustvarimo nit za branje
pthread_t read_thread;
pthread_create(&read_thread, NULL,
  read, &read_stage);
}
```

3 DEKODIRNIK

Pripravimo tudi cevovodno rešitev, ki opravlja ravno nasprotno nalogo - iz naših zapisanih kodiranih datotek sestavi nazaj A4 sliko. Najprej moramo prebrati kodirane podatke, iz zaglavja ponovno zgradimo Huffmanovi drevesi in ju uporabimo v postopku dekodiranja. Dekodirane podatke lahko zapišemo verižno kodirano ali pa že neposredno kot zaporedje slikovnih točk. Za zapis slike podoobno kot za branje slike uporabimo knjižnico *stb*.

Branje kodiranih podatkov je implementirano v Input :: read_encoded, dekodiranje v Huffman.decode in zapisovanje slike v Output :: write_image.

4 VEČ VZPOREDNIH NITI

Štiristopenjski cevovod iz prejšnjega poglavja lahko nadgradimo, tako da za posamezno stopnjo uporabimo več kot eno nit. S pripravljeno zasnovo je to preprosto, saj je za varnost sočasnega dostopanja do podatkov že zagotovljeno. Število niti za posamezno stopnjo lahko enostavno nastavimo preko spremenljivk ali vhodnih argumentov brez potrebe po spreminjanju ostale kode. Za zgornji primer to recimo izgleda takole:

```
pthread_t read_thread[NUM_READ];
for (int i = 0; i < NUM_READ; i++)
    pthread_create(&read_threads[i],
      NULL, read, &read_stage);
```

Optimalno porazdelitev števila niti lahko ugotovimo empirično s testiranjem različnih konfiguracij. V okviru naloge smo uporabili program perf za vzorčenje procesorskega časa in klicev in hotspot za vizualicijo dobljenih podatkov. TODO

5 DINAMIČNO PRILAGAJANJE

Namesto, da nastavimo fiksno porazdelitev števila niti, lahko število niti na posamezni stopnji dinamično prilagajamo potrebam. Enostaven pristop je, da glavna nit periodično preveri zasedenost medpomnilnika posamezne stopnje in uporabi nek klasifikator, ki na podlagi tega stanja in števila niti odloča, kaj storiti.

Z uporabo dinamičnega prilagajanja lahko za dano število niti na dolgi rok naloge procesiramo kar se da učinkovito. Pri fiksnem pristopu to težko dosežemo, saj je optimalno število niti ravno med dvema celima številoma.

6 REZULTATI

Algoritem smo pognali za različne konfiguracije in za $N \in \{10, 100, 1000\}$, kjer je N število obdelanih slik. Čase smo povprečili preko deset zaporednih pogonov. V tabelah je konfiguracija označena kot $a - b - c - d$; a označuje število niti prve stopnje (branje), b število niti druge stopnje (verižno kodiranje) in tako naprej. Za prevajanje smo uporabili g++ 10.2.0 z −std=c++11 −O2 −lpthread opcijami. Rezultate smo pridobili na dveh platformah:

1) CPU: Intel i5-8250U (4/8 jedrni), prenosnik, med testiranjem priključen na napajanje
2) CPU: AMD Ryzen 7 2700X (8/16 jedrni)

V idealnem svetu bi s štirimi nitmi dosegli štirikratno pohitritev, z osmimi nitmi (dve za vsako stopnjo) pa osemkratno pohitritev (kot da bi vzporedno izvajali dva idealna cevovoda). V našem cevovodu ne moremo pričakovati štirikratne pohitritve. Namreč, delo med nitmi ni enakomerno porazdeljeno, za tretjo stopnjo pa lahko še dodatno povemo, da je potreben čas odvisen od samih slik, saj je gradnja drevesa in kodiranje odvisno od dolžine in variabilnosti verižno kodiranih podatkov.

N		10	100	1000
sekvenčni				
čas	[s]	0,19	1,9	18,7
max RAM	[MB]	21	21	21
preprost cevovod				
čas	[s]	0,12	1,0	9,3
max RAM	[MB]	32	33	38
$2-1-1-1$				
čas	[s]	0,09	0,65	5,8
max RAM	[MB]	52	62	104

Tabela 1: Rezultati prve platforme

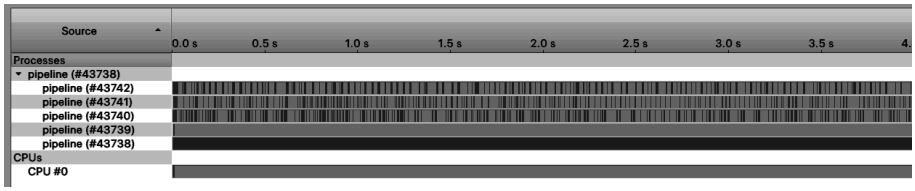
N		10	100	1000
sekvenčni				
čas	[s]	0,15	1,4	14,3
max RAM	[MB]	21	21	21
preprost cevovod				
čas	[s]	0,11	0,80	7,5
max RAM	[MB]	32	34	40
$2-1-1-1$				
čas	[s]	0,09	0,45	3,8
max RAM	[MB]	45	79	86
$3-2-2-2$				
čas	[s]	0,07	0,33	2,9
max RAM	[MB]	54	87	98
$4-3-4-4$				
čas	[s]	0,07	0,30	2,4
max RAM	[MB]	58	105	132

Tabela 2: Rezultati druge platforme

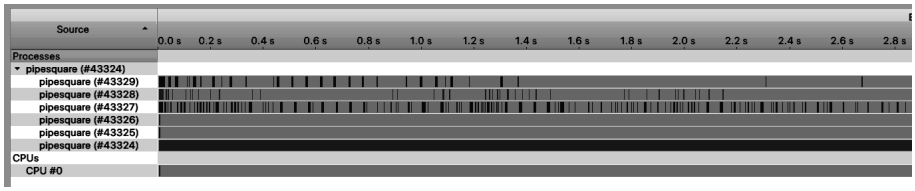
Dodatno, uporaba več niti pripelje še stroške morebitne sinhronizacije in varovanja pred neželenim hkratnim dostopom. Tudi če imamo za vsako stopnjo le eno nit, moramo že zaradi dostopa do podatkovnih struktur (npr. std :: queue) poskrbeti za pravilno zaklepanje.

Rezultati ob smiselni omejitvi medpomnilnika (vsaj nekaj mest za naloge) so praktično enaki. Z omejitvimo medpomnilnika umetno upočasnimo stopnje, ki so ”prehitre”; recimo, če stopnja za verižno kodiranje ne dohaja količine prebranih slik, z omejitvimo medpomnilnika prisilimo niti za branje, da nekaj časa čakajo, nima pa to vpliva na končen čas.

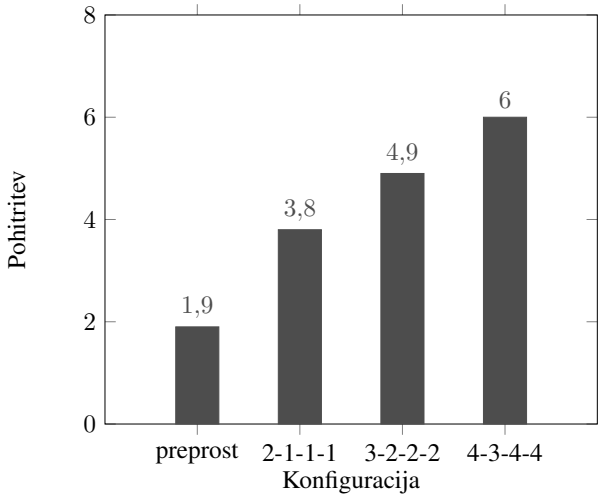
Preprost cevovod je konfiguracija $1 - 1 - 1 - 1$.



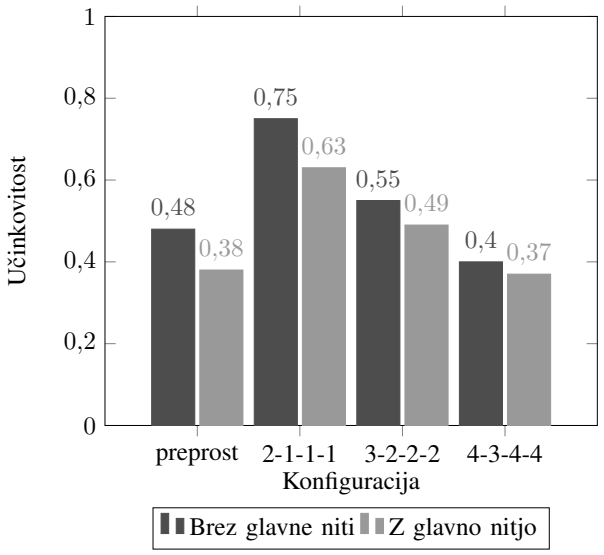
Slika 3: Analiza stanja niti pri preprostem cevovodu



Slika 4: Analiza stanja niti pri konfiguraciji $2-1-1-1$



Slika 5: Pohitritve na drugi platformi



Slika 6: Učinkovitosti na drugi platformi

S programskima orodjema perf in hotspot dobimo analizo po nitih na testni množici s 1000 slikami, ki jo prikazujeta sliki 3 in 4 (sliki sta odrezani približno na polovici za boljšo berljivost). Črna barva predstavlja časovno obdobje, ko nit spi. Nit z najmanjšim indeksom – najbolj spodaj – je glavna nit, nato pa si proti vrhu sledijo niti po posameznih stopnjah. Opazimo, da je pri preprostem cevovodu nit za branje cel čas zaposlena, medtem ko so ostale tri slabo izkoriščene. S konfiguracijo $2-1-1-1$ stanje bistveno izboljšamo. Še vedno sta najbolj zaposleni niti za branje, a ostale niti veliko manj spijo.

Pohitritev s številom niti raste, kot je prikazano na sliki 5. Če imamo za branje dve niti, dosežemo kar dvakratno pohitritev v primerjavi s preprostim cevovodom (ena nit za eno stopnjo). Z večanjem števila niti lahko dosežemo še večje pohitritve, vendar je težje poiskati optimalno razdelitev.

Z večanjem števila niti opazimo še eno stvar – učinkovitost začne hitro upadati na nivo, primerljiv z enostavnim cevovodom. Sklepamo lahko, da obstajajo konfiguracije, kjer dosežemo lokalne maksimume učinkovitosti. Namreč, ob določenih konfiguracijah je delo niti precej bolj enakomerno razdeljeno kot ob ostalih sosednjih konfiguracijah (kot je to recimo pri konfiguraciji $2-1-1-1$). Učinkovitost smo izračunali na dva načina. Pri prvem glavne niti nismo upoštevali v izračunu, pri drugem pa smo jo. Namreč, glavna nit bi lahko prav tako opravljala delo ene stopnje. V naši implementaciji temu ni tako, zato na sliki 6 prikazujemo oba izračuna.

7 Zaključek

Cevovod je preprost koncept, s katerim lahko pospešimo delovanje večstopenjskih algoritmov, ki jih z običajnimi pristopi težko pohitrimo. Za dosego čim večje pohitritve in učinkovitosti je pomembno, da so niti čim bolj enakomerno zaposlene.