Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tilen Nedanovski

Procesor grafov

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Patricio Bulić

Ljubljana, 2017

COPYRIGHT. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Procesor grafov izkorišča paralelizem operacij v grafih. Opišite arhitekturo in organizacijo takega procesorja ter ga implementirajte v programabilnem vezju FPGA.

Zahvaljujem se mentorju izr. prof. dr. Patriciu Buliću za usmerjanje pri izdelavi diplomskega dela in kolegom v podjetju Beyond Semiconductor, posebno Juretu Cigliču in Matjažu Breskvarju za strokovno pomoč, vse pridobljeno znanje, nasvete pri pisanju ter pobudo za nastanek dela. Nazadnje se zahvaljujem tudi družini in prijateljem za pomoč in podporo v času študija.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Graf **3**

2.1 Graf in matrika 5

2.2 Algoritmi 10

3 Podatkovni model **15**

3.1 Graf 15

4 Arhitektura **19**

4.1 Paralelni sistemi 19

4.2 Transakcijski pomnilnik 26

4.3 Procesor grafov 28

4.4 Pomnilniška hierarhija 29

4.5 Enota za koordinacijo dela 33

4.6 Prioritetni kodirnik 34

4.7 Procesorsko polje 36

4.8 Enota za transformacije grafa 36

4.9 Aritmetična procesna enota 37

4.10 Lokacija operandov in načini naslavljanja 43

4.11 Strojni jezik 46

5 Implementacija **51**
5.1 FPGA 51
5.2 Zgradba procesorja 52
5.3 Dodeljevalnik 52
5.4 AXI 53
5.5 Pomnilnik 56
5.6 Predpomnilnik L1 57
5.7 Enota za transformacije grafa 59
5.8 Polje procesnih enot 59

6 Zaključek **65**

Literatura **68**

Seznam uporabljenih kratic

kratica	angleško	slovensko
AMBA	advanced microcontroller bus architecture	napredna arhitektura vodila v mikrokrmilnikih
AXI	advanced extensible interface	napredni razširljivi vmesnik
DSP	digital signal processing	digitalno procesiranje signalov
FIFO	first in first out	prvi noter in prvi ven, vrsta
FPGA	field-programmable gate array	polje programabilnih logičnih blokov
LUT	look-up table	tabela preslikave, iskalna tabela
NUMA	non-uniform memory access	neenotni pomnilniški dostop
UMA	uniform memory access	enotni pomnilniški dostop

Povzetek

Naslov: Procesor grafov

Avtor: Tilen Nedanovski

Grafe običajno uporabljamo za opisovanje podatkov z visoko stopnjo medsebojne povezanosti oz. odvisnosti ali v primerih, ko so informacije o povezavah med podatki, kar imenujemo tudi topologija podatkov, pomembnejše kot podatki sami. Pogosto so implementacije grafa in operacij nad grafom izključno programske. Programske implementacije so konvencionalnim procesorjem v veliko breme, saj tipično ne izkoriščajo pomnilniške lokalnosti. Pri delu z grafi se tako poraja potreba po učinkoviti strojni implementaciji podatkovne strukture. Delo obravnava računalniško arhitekturo, ki je rezultat izkoriščanja grafu inherentnega paralelizma in referenčne lokalnosti, ki jo povzroča njegova matrična reprezentacija. Začetna poglavja vsebujejo nekaj nauka o grafih in algebri za delo z grafi. Naslednje poglavje podaja teoretično zasnovo organizacije in arhitekture procesorja ter govori o premislekih in spoznanjih med njegovim snovanjem. Delo se zaključi z nekaj podrobnostmi o implementaciji procesorja v vezju FPGA in predlogom o integraciji vseh komponent sistema v smiselno celoto oz. konfiguracijo.

Ključne besede: računalniška arhitektura, računalniška organizacija, procesor, graf.

Abstract

Title: A Graph Processor

Author: Tilen Nedanovski

Graphs are frequently used in cases where data to be described is densely interconnected or the information about said connections, also referred to as topology of the data, is more important than the data itself. Common solutions to graph processing and computation often rely on software, which in itself is a burden to the conventional widespread computer architecture. Henceforth, the need for an efficient hardware implementation of graph structures and their manipulation arises. This work is a treatise on hardware accelerated graph computation. It provides some knowledge about graphs and graph algebra, for use in what endeavours in the matter follow. It conveys some information on graph data structure and graph database. The latter is followed by the conception of the graph processor architecture and the reasoning behind it. Lastly, some details of a suitable implementation using an FPGA circuit are given and some common protocols are described to achieve a good overall integration as well as the integration of the processor itself.

Keywords: computer architecture, computer organisation, processor, graph.

Poglavje 1

Uvod

V obilju podatkov, ki jih z digitalizacijo pridobimo iz različnih področij človekovega življenja in dela, se s problematiko učinkovite taksonomije, analize ter obdelave velike količine informacij v digitalnih sistemih srečujemo zelo pogosto.

Ob načrtovanju podatkovnega modela za zajem informacij v bazo, pogosto odkrijemo, da nekateri podatki ne ustrezajo vnaprej določeni podatkovni strukturi [24, 8]. Ti podatki nimajo formalne strukture, kljub temu pa v njih lahko ločimo semantične elemente od sintaksnih. Podatke s šibko strukturo ali brez strukture enostavno predstavimo z grafom. Uporaba grafa kot podatkovnega modela pri opisovanju nestrukturiranih podatkov je smiselna zaradi prednosti podatkovne strukture graf.

Graf omogoča intuitivno modeliranje dinamičnih podatkov in podatkov brez formalne strukture. Poizvedbe v grafu so hierarhične in smiselneje posnemajo, kako razmišljamo o podatkih ter relacijah med njimi. Vozlišča lahko predstavljajo vrsto konceptov, na primer entiteto, shemo ali podgraf, kar omogoča predstavitev kompleksnih in zelo povezanih podatkov. V splošnem se prednosti graf baze[2] kažejo predvsem v primerih, ko je stopnja medsebojne odvisnosti v podatkih velika in so informacije o povezanosti oz. topologiji podatkov pomembnejše kot podatki sami.

Količina informacij in zahtevnost obdelave podatkov pogosto presegata

računske zmogljivosti dominantne računalniške arhitekture. Te omejitve premagujemo s povezovanjem več računalnikov v gruče ali s posebno strojno opremo, ki je namenjena le obdelavi podatkov in delu z njimi. Zaradi dinamične in iregularne strukture grafa ter splošne namembnosti ukazno pretokovne računalniške arhitekture je za ravnanje s podatki v grafu zadnja rešitev smiselnejša od prve.

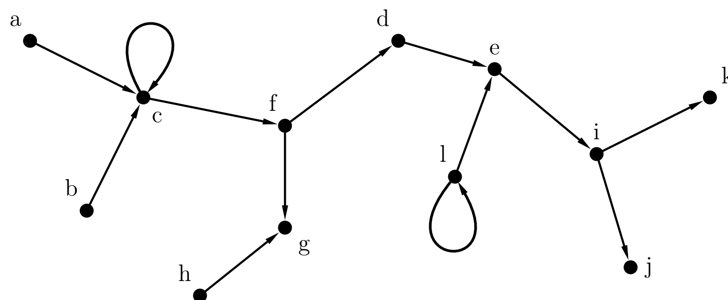
Struktura grafa dovoljuje veliko fleksibilnosti pri shranjevanju podatkov. V oziru uporabnika je ta lastnost predvsem dobrodošla, hkrati pa je razlog za številne težave, s katerimi se spopadamo pri načrtovanju učinkovite strojne opreme. Operacije nad podatkovno strukturo ki ponazarja graf, namreč vodijo do naključnih dostopov do pomnilnika. Nezaporedni dostopi so v klasičnih procesorskih arhitekturah z večnivojsko pomnilniško hierarhijo iregularni – netipični, z visoko ceno na dostop. Vsak iregularen dostop privede do zgrešitve v predpomnilnikih. Slednje se nanaša na lokalnosti dostopov do pomnilnika. O lokalnosti pomnilniških dostopov govorimo, kadar program zaporedoma oz. v krajšem časovnem obdobju dostopa do operandov, ki so v pomnilniku na zaporednih lokacijah. Tak vzorec dostopanja do pomnilnika je pogost v ukazno pretokovnih računalnikih. Operacije nad grafi pa ravno nasprotno izkazujejo slabo lokalnost pomnilniških dostopov. To pomeni, da prednosti, ki jih z izkoriščanjem lokalnosti pridobimo s pomnilniško hierarhijo niso več prednosti, temveč dodatno breme pri izvajanju programa.

Poglavje 2

Graf

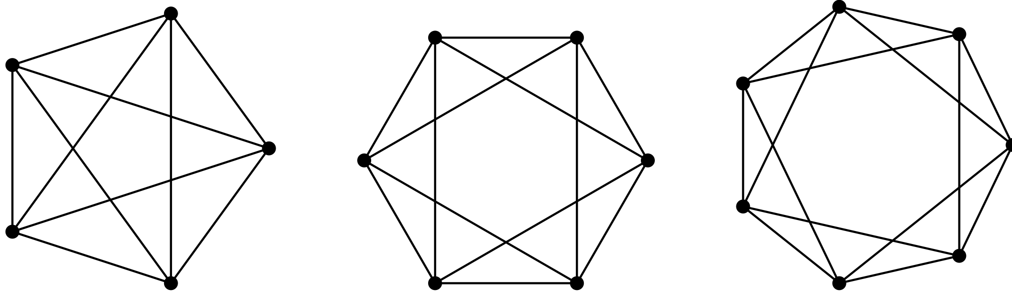
Graf je diskretna struktura, ki ponazarja skupino objektov in razmerja med njimi. Objekte v grafu zastopajo vozlišča, razmerja med objekti pa povezave, ki tvorijo pare vozlišč.

Definicija 2.1 *Graf je urejen par $G = (V, E)$, sestavljen iz množice vozlišč V in množice povezav E. Množico povezav E sestavljajo pari vozlišč množice V [13].*



Povezava je v usmerjenem grafu urejen in v neusmerjenem grafu neurejen par vozlišč iz množice V. Krajišči povezave sta torej vozlišči v paru. Število povezav, ki jim vozlišče pripada, imenujemo stopnja ali valenca vozlišča. Če sta vozlišči para enaki, je povezava refleksivna – tvori zanko. Graf brez zank in večkratnih povezav je enostaven. Dve vozlišči v neusmerjenem grafu veže enkratna povezava, kadar obstaja med vozliščema kvečjemu ena neusmerjena

povezava in v usmerjenem grafu, kadar sta vozlišči v vsako smer povezani z največ eno usmerjeno povezavo. Enostavni graf, v katerem ima vsako vozlišče enako število povezav kot vsa druga vozlišča v grafu – so valence vozlišč v grafu enake – imenujemo regularni graf [13][26].



Slika 2.1: 5-, 6- in 7-regularni grafi.

Definicija 2.2 *Končno zaporedje vozlišč $v_i \in V$*

$$S = v_0 v_1 v_2 ... v_{k-1} v_k$$

imenujemo sprehod v grafu $G = (V, E)$. Pri tem obstaja med poljubnima zaporednima vozliščema v_i in v_{i+1}, $i \in \{0, ..., k-1\}$, sprehoda S, povezava v grafu G, torej $(v_i, v_{i+1}) \in E$. [13]

Število k imenujemo dolžina sprehoda in zapišemo $k = |S|$. Vozlišči v_0 in v_k sta začetno in končno vozlišče sprehoda. Če se sprehod začne in konča v istem vozlišču, ga imenujemo sklenjeni sprehod ali obhod. Sprehod je enostaven, če nobena povezava v njem ne nastopi dvakrat. Enostavnemu sprehodu pravimo pot, če so poleg povezav različna tudi vozlišča. Izjemoma lahko začetno in končno vozlišče sovpadata. Takrat je pot sklenjena. Vozlišče u je iz vozlišča v dosegljivo takrat, ko v grafu obstaja sprehod z začetkom v v in koncem v u. Za graf pravimo, da je povezan, če za vsak par vozlišč v grafu obstaja vsaj ena pot med vozliščema. Torej, če je vsako vozlišče dosegljivo iz vseh drugih vozlišč v grafu [26].

Naj bo $w : E \to \mathbb{R}$ preslikava, ki vsaki povezavi v grafu priredi realno vrednost. Grafu $G(V, E, w)$ pravimo uteženi graf. Preslikava w v uteženem

grafu lahko namesto $\mathbb{R}$ slika v poljubno množico elementov. Vrednost oz. utež lahko na podlagi w priredimo tudi sprehodu ali končni množici sprehodov. Preslikavo razširimo z naslednjimi predpisi [5]:

- naj bo Z_v ničelni sprehod v točki v, potem je $w(Z_v) = 1$
- naj bo $S = (v_0, v_1), (v_1, v_2)...(v_{k-1}, v_k)$ sprehod po grafu G, potem je vrednost sprehoda

$$w(S) = \prod_{i=1}^{k} w((v_{i-1}, v_i))$$

- za prazno množico sprehodov $\emptyset$ velja $w(\emptyset) = 0$
- naj bo $\mathcal{S}$ končna množica sprehodov. Njena vrednost je

$$w(\mathcal{S}) = \sum_{S \in \mathcal{S}} w(S)$$

2.1 Graf in matrika

K predstavitvi grafov in reševanju problemov v povezavi z njimi lahko pristopimo z linearno algebro. Dovolj je spoznanje o dvojnosti kanonične definicije grafa in njegove matrične reprezentacije ter nekaj nauka o linearnih algebrskih strukturah in njihovih transformacijah.

Obravnava grafa v matrični obliki ima določene prednosti. Algoritmi z linearnimi operacijami jasno opredeljujejo dostope do matričnih struktur v pomnilniku in ustvarjajo lokalnost, ki jo v dostopih do drugih podatkovnih struktur, s katerimi lahko predstavimo graf, le s težavo odkrijemo. Algoritmi so tako bolj predvidljivi za pomnilniške dostope in tudi za operacije, ki jih izvajajo. Determinizem v algoritmih pa omogoča optimizacijske izboljšave tako na programskem kot tudi strojnem nivoju. Področje linearne algebre in linearnih algoritmov je rodovitno z raziskovalnimi deli, specifikacijami, protokoli in standardi za pospešeno ter porazdeljeno delo z linearnimi algebrskimi strukturami. Obstoječe znanje lahko tako prenesemo na transformacije nad grafi.

2.1.1 Polkolobar

S širšo definicijo matričnega in vektorskega množenja lahko veliko problemov v povezavi z grafi prevedemo v jezik linearne algebre. To storimo z algebrsko strukturo. Množico matrik navadno obravnavamo z algebro nad polkolobarjem realnih števil $(\mathbb{R}, +, \star, 0, 1)$. Koeficienti matrike so lahko elementi poljubnega polkolobarja. Z izbiro neklasičnega polkolobarja poleg elementov v matriki na novo opredelimo operaciji množenja in seštevanja matrik.

Definicija 2.3 *Algebrska struktura* $(P, \oplus, \otimes, 0, 1)$*, z binarnima operacijama seštevanje* $\oplus$ *in množenje* $\otimes$ *in nevtralnima elementoma za seštevanje* 0 *in množenje* 1*, je polkolobar, če velja:*

- *množica* $(P, \oplus, 0)$ *je Abelova grupa z nevtralnim elementom* 0 *(operacija seštevanje* $\oplus$ *je komutativna in asociativna, velja* $a \oplus 0 = 0 \oplus a = a$ *za vsak* $a \in P$*),*
- *množica* $(P, \otimes, 1)$ *je polgrupa z nevtralnim elementom* 1 *(operacija množenje* $\otimes$ *je asociativna, velja* $a \otimes 1 = 1 \otimes a = a$ *za vsak* $a \in P$*),*
- *distributivnost za poljubne elemente* $a, b, c \in P$*, ki povezuje operaciji seštevanje* $\oplus$ *in množenje* $\otimes$*:*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

- *element* 0 *je absorpcijski element za množenje* $\otimes$*, tj. velja*

$$a \otimes 0 = 0 \otimes a = 0 \text{ za vsak } a \in P. \text{ [4][21]}$$

Zgledi nekaterih polkolobarjev, s katerimi se srečujemo pri matričnih obravnavah grafa, so [5]:

Kombinatorični polkolobar $(\mathbb{N}, +, \cdot, 0, 1)$ z operacijo seštevanja $+$ in množenja $\cdot$. Vrednosti povezav so lahko vsa naravna števila $w((u, v)) \in \mathbb{N}$. Preslikava $w : E \to \mathbb{N}$ povezavi (u, v) priredi število načinov, na katere lahko iz vozlišča u po povezavi pridemo v točko v.

Polkolobar regularnih izrazov $(P(\Sigma^\star), \cup, \cdot, \emptyset, \varepsilon)$, pri čemer je $\Sigma^\star$ množica končnih nizov nad abecedo Σ, $\cdot$ operacija stikanja nizov in $\varepsilon = \{\lambda\}$. Vrednosti povezav so znaki abecede Σ, tj. $w((u, v)) \in \Sigma$. Preslikava $w(S)$ je torej beseda oz. zaporedje oznak povezav na sprehodu S.

Polkolobar najkrajših poti $(\mathbb{R}_0^+, \min, +, \infty, 0)$ z operacijama min, ki izmed dveh števil vrne najmanjše, in seštevanje $+$. Preslikava $w((u, v)) \in \mathbb{R}_0^+$ je cena povezave med vozliščema u in v, $w(S)$ pa cena sprehoda S.

Povezanostni polkolobar $(\{0, 1\}, \vee, \wedge, 0, 1)$ z operacijama disjunkcija $\vee$ in konjunkcija $\wedge$. Preslikava $w((u, v))$ priredi povezavi (u, v) vrednost 1, če sta vozlišči u in v povezani, in 0 v nasprotnem primeru. Vrednost sprehoda $w(S)$ je 1, če vse povezave v zaporedju povezav S obstajajo in 0 v nasprotnem primeru.

Verjetnostni polkolobar $([0, 1], +, \cdot, 0, 1)$ z operacijama seštevanje $+$ in množenje $\cdot$. Vrednosti povezav so lahko vsa števila na intervalu od 0 do 1 – $w((u, v)) \in [0, 1]$. Preslikava $w : E \to [0, 1]$ povezavi priredi verjetnost prehoda iz vozlišča u po povezavi v vozlišče v.

V klasičnem polkolobarju $(\mathbb{R}, +, \cdot, 0, 1)$ množimo matriki A in B na naslednji način

$$(A \star B)_{ij} = \sum_{k=1}^{n} (a_{ik} \cdot b_{kj}) \tag{2.1}$$

V kontekstu polkolobarjev zapišemo matrični produkt na naslednji način:

$$A\ op1\ .\ op2\ B. \tag{2.2}$$

Klasični matrični produkt torej zapišemo kot:

$$A \cdot B = A\ +.\star\ B. \tag{2.3}$$

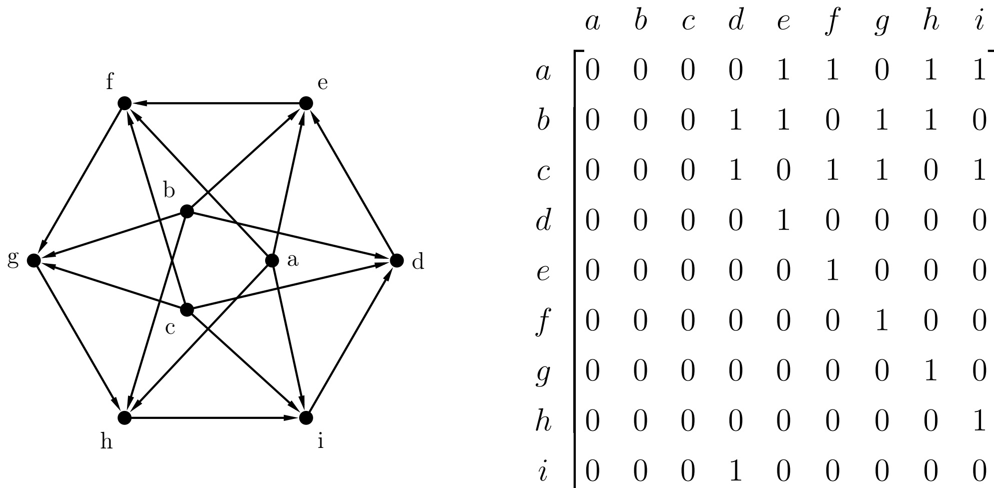
Graf lahko z matriko predstavimo na več načinov.

2.1.2 Matrika sosednosti

Matrika sosednosti je kvadratna matrika, ki nosi informacije o sosednosti vozlišč v grafu. Vsak element sosednostne matrike predstavlja povezavo med vozliščema, ki ju označuje položaj elementa v matriki. Vrstica običajno predstavlja izvorno, stolpec pa ponorno vozlišče v usmerjenem grafu. Sosednostna matrika neusmerjenega grafa je simetrična.

Za matriko sosednosti v splošnem velja:

$$A_{ij} = \begin{cases} 1; & (i,j) \in E, \\ 0; & sicer \end{cases} \tag{2.4}$$



	a	b	c	d	e	f	g	h	i
a	0	0	0	0	1	1	0	1	1
b	0	0	0	1	1	0	1	1	0
c	0	0	0	1	0	1	1	0	1
d	0	0	0	0	1	0	0	0	0
e	0	0	0	0	0	1	0	0	0
f	0	0	0	0	0	0	1	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	0	0	1
i	0	0	0	1	0	0	0	0	0

Slika 2.2: Graf z 9 vozlišči in pripadajočo matriko sosednosti.

Matriko sosednosti, poleg reprezentacije grafa, uporabljamo tudi za ugotavljanje izomorfizma med dvema grafoma in povezanost vozlišč v pripadajočem grafu.

Definicija 2.4 *Grafa G_1 in G_2 z matrikama sosednosti A_1 in A_2 sta izomorfna, če obstaja permutacijska matrika P, tako da velja:*

$$PA_1P^{-1} = A_2 \tag{2.5}$$

Pri tem imata matriki A_1 in A_2 enaka karakteristična polinoma, enaka minimalna polinoma, enake lastne vrednosti ter enaki determinanti in sledi. Obratno pa ne velja vedno. Dva grafa z enakimi lastnimi vrednostmi pripadajočih sosednostnih matrik nista nujno izomorfna [18].

Matriko sosednosti uporabljamo tudi za ugotavljanje povezanosti grafa. Potence matrike namreč za vsako vozlišče v grafu povedo število sprehodov med poljubnim parom vozlišč dolžine, ki ustreza potenci. Specifično je element (i, j) matrike A^n enak številu sprehodov dolžine n iz vozlišča i v vozlišče j.

Strukturo grafa lahko z matriko sosednosti predstavimo zelo strnjeno. Vsak element matrike hrani le bit informacije – povezava med vozliščema obstaja ali ne. Prostor, ki ga zavzema takšna matrika je torej kvečjemu $|V|^2$ bitov za usmerjene in kvečjemu $\frac{|V|^2}{2}$ za neusmerjene grafe. Matrika sosednosti je idealna podatkovna struktura za določene preproste operacije. Ustvarjanje novih povezav in branje, posodabljanje ter brisanje obstoječih so trivialne operacije – njihova časovna zahtevnost je $O(1)$.

2.1.3 Matrika dosegljivosti

Naj bo A matrika sosednosti, matrika I pa identiteta. Matriko

$$R = I + A \tag{2.6}$$

imenujemo matrika dosegljivosti.

Matrika dosegljivosti sporoča dostopnost vsakega vozlišča v ostalih vozliščih grafa. Element v i-ti vrstici in j-tem stolpcu je različen od 0, če lahko iz vozlišča v v vozlišče j pridemo v največ enem koraku. V nasprotnem primeru je vrednost matrike v j-tem stolpcu in i-ti vrstici enaka 0.

Označimo z $R^{\diamond r}$ r-to potenco matrike R v polkolobarju $(\{0,1\}, \vee, \wedge, 0, 1)$:

$$R^{\diamond r}_{i,j} = \bigvee_{k=1}^{n} (R^{\diamond r-1}_{i,k} \wedge R_{k,j}) \tag{2.7}$$

Matrika $R^{\diamond r}$ predstavlja dosegljivost vozlišč v r korakih. Torej $R^{\diamond r}_{i,j} = 1$, če od vozlišča i do vozlišča j obstaja pot z največ r koraki.

2.1.4 Matrika cen

Uteženi graf $G = (V, E, w)$ predstavimo z matriko cen C. Za elemente matrike cen velja:

$$c_{ij} = \begin{cases} 0 & i == j, \\ w_{ij} & \text{če med } i \text{ in } j \text{ obstaja povezava,} \\ \infty & \text{sicer} \end{cases} \tag{2.8}$$

Matriko cen pogosto uporabljamo pri iskanju poti ali pretokov v uteženem grafu.

2.1.5 Redke matrike

Običajno je število povezav v grafu bistveno manjše od števila elementov, ki jih zasedajo matrične reprezentacije grafa. Matrika je prostorsko potratna podatkovna struktura predvsem, kadar je graf, ki ga opisuje, redek. Graf $G(V, E)$ je gost, če velja $|E| \approx |V|^2$. Nasprotno je graf G redek, če velja $|E| \approx |V|$. V zadnjem primeru uporabljamo redke matrike in operacije nad njimi.

Redka matrika je matrika z veliko elementi, ki so enaki 0 in z malo elementi, ki so od 0 različni. Redko matriko lahko v pomnilniku hranimo učinkoviteje, če vanj zapišemo le elemente matrike, ki so od 0 različni. Poleg teh elementov vpišemo v pomnilnik indeksno strukturo, ki omogoča rekonstrukcijo originalne matrike brez izgub, napak ali dvoumnosti. Pri tem, v zameno za izboljšano prostorsko kompleksnost, izgubimo učinkovitost dostopa do posameznih elementov matrike in matrike same. Na podlagi lastnosti grafa in računskega problema se zato odločimo za redko matriko ali ne.

2.2 Algoritmi

Z grafi se pogosto srečujemo v analizi omrežij. To so lahko cestno omrežje, kjer križišča predstavljajo vozlišča, ceste pa povezave med njimi, omrežje

letalskih povezav, v katerem leti predstavljajo povezave med vozlišči letališč, ali internetno omrežje, kjer so vozlišča računalnikov povezana enako, kot so fizično povezani računalniki v omrežju.

Omenjeni grafi imajo med sabo različne strukture. Graf letalskih povezav je verjetno precej povezan. Vsako vozlišče, ki predstavlja letališče, ima visoko valenco. Vozlišča cestnega omrežje pa gotovo ne, saj se v križišču običajno srečajo štirje cestni odseki. Graf cestnega omrežja je v veliki meri planaren, saj nadvozov in podvozov ni veliko. Graf internetnega omrežja je podoben grafu letalskih povezav, vendar je v njem veliko več močno povezanih komponent, ki predstavljajo lokalna omrežja računalnikov. Problemov, ki se porajajo v takšnih in podobnih grafih, je mnogo.

2.2.1 Iskanje najcenejših poti

V grafu lahko med vozlišči iščemo tiste poti, preko katerih do vozlišč pridemo v najmanj korakih ali pa je cena poti najmanjša. Najkrajša pot med dvema vozliščema je pot, preko katere pridemo iz izvornega vozlišča v ponorno v najmanjšem številu korakov. Podobno je najcenejša pot med vozliščema tista, katere vsota cen ali uteži povezav, iz katerih sestoji, je izmed vsot uteži na vseh poteh z istim izvorom in ponorom najmanjša.

V usmerjenih grafih lahko na podlagi matrike cen ugotovimo najkrajše poti s poljubnim začetnim vozliščem v vsa druga vozlišča v grafu, če pot med vozliščema sploh obstaja. Problem rešimo z matričnim množenjem v polkolobarju $(\min, +, \infty, 0)$.

V splošnem je cena najcenejše poti od vozlišča i do vozlišča j z največ h koraki rekurzivno podana z

$$D_{i,j}^{h} = \min_{k=1}^{|V|}(D_{i,k}^{h-1} + C_{k,j}) \tag{2.9}$$

Natančneje je v usmerjenem grafu $G = (V, E, w)$ z utežmi $w : E \rightarrow \mathbb{R}_\infty$ najkrajša oz. najcenejša pot med vozliščema u in v podana na naslednji

način:

$$\Delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}; & \text{če pot med u in v obstaja} \\ \infty; & \text{sicer} \end{cases} \tag{2.10}$$

Najkrajša oz. najcenejša pot $u \overset{p}{\rightsquigarrow} v$ iz vozlišča u v vozlišče v je torej tista z vrednostjo ali utežjo $w(p) = \Delta(u,v)$ [7][18].

2.2.2 Bellman-Fordov algoritem

Bellman-Fordov algoritem[25] je postopek za iskanje najkrajše poti iz izvornega vozlišča v vsa druga vozlišča v usmerjenem in uteženem grafu. Algoritem deluje na načelu postopne rešitve. To pomeni, da se približek prave razdalje postopoma nadomesti z natančnejšimi vrednostmi v vsaki iteraciji, dokler sčasoma ne doseže pravilne ali optimalne rešitve. Razdalja med izvornim in ponornimi vozlišči je v vsaki iteraciji presežna ocena prave razdalje. V postopku izvajanja algoritma se pesimistične ocene nadomestijo z novimi, če te izboljšajo rezultat – so nove ocene boljše, torej poti krajše. Algoritem dosega optimalne rešitve v $|V| - 1$ ali manj iteracijah.

Na poti k algebraični formaciji algoritma se najprej ozremo na splošno definicijo problema, ki spominja na tisto v 2.10. Najkrajša razdalja med vozliščema u in v v največ k korakih je enaka:

$$\Delta_k(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v, |p| \leq k\}; & \text{če pot z največ } k \text{ koraki med } u \\ & \text{in } v \text{ obstaja} \\ \infty; & \text{sicer} \end{cases} \tag{2.11}$$

Algoritem z izjemo vedno privede do optimalne rešitve. Izjema so grafi z negativnimi cikli. To so cikli z negativnimi vsotami uteži povezav, ki jih sestavljajo. Če velja $\Delta_n(s,v) < \Delta_{|V|-1}(s,v)$ za katerokoli vozlišče v, potem v grafu obstaja negativno uteženi cikel.

Bellman-Fordov algoritem formaliziramo v linearni algebri. Potrebujemo matriko cen C in vektor d_k velikosti $|V|$, ki hrani razdalje najkrajših poti

v največ k korakih do vseh ostalih vozlišč. Računanje $\Delta_k(s,v)$ na podlagi prejšnjega koraka $\Delta_{k-1}(s,v)$ je intuitivno.

$$\Delta_k(s,v) = \min\{\Delta_{k-1}(s,u) + W(u,v)\} \tag{2.12}$$

Razdaljo najkrajše poti z največ k koraki iz izvornega vozlišča s v ponorno vozlišče v v grafu dobimo z:

$$d^k_{s,v} = \min_{\forall u \in V}(d^{k-1}_{s,u} + a_{u,v}) \tag{2.13}$$

Zgornja definicija spominja na formulacijo matričnega produkta $C = A \cdot B$:

$$C_{i,j} = \sum_k (A_{i,k} * B_{k,j}) \tag{2.14}$$

Iz tega sledi

$$D^h = D^{h-1} \min . + C. \tag{2.15}$$

Vektor d^k_s predstavlja najcenejše poti z največ k koraki iz vozlišča s do vseh ostalih vozlišč v grafu:

$$d^k_{s,*} = D^{k-1} \min . + C_{*,s} \tag{2.16}$$

Razdalje najkrajših poti iz vozlišča s do vseh ostalih vozlišč lahko torej predstavimo z $d_{s,*} = d^0_{s,*} \min . + C^{|V|-1}$, pri čemer je $d^0_{s,v}$:

$$d^0_{s,v} = \begin{cases} 0; & v = s \\ \infty; & \text{sicer} \end{cases} \tag{2.17}$$

Poglavje 3

Podatkovni model

Podatkovni model je, v računalništvu in matematiki, skupina konceptov za predstavitev znanja. V kontekstu podatkovnih baz sestoji vsaj iz množice podatkovnih struktur, množice operatorjev ali pravil sklepanja o podatkovnih strukturah in množice zakonov – pravil o integriteti – za delo s podatki.

Podatkovno strukturo dobimo z združevanjem osnovnih podatkovnih tipov. Podatkovni tip določa vrednosti, ki jih lahko zavzame podatek. Instance podatkovnih struktur predstavljajo konkretne podatke in tvorijo zbirko podatkov v podatkovni bazi. Za operatorje imenujemo vse operacije nad podatkovnimi strukturami za pridobivanje podatkov iz njih ali njihovih delov. Pravila o integriteti določajo veljavne posege v podatkovni bazi.

3.1 Graf

Z grafom enostavno predstavimo različne podatke in z njimi povezane podatkovne modele v bazi. Ustvarjanje vozlišč in povezav ter s tem spreminjanje strukture grafa je trivialno in ne zahteva posodabljanja obstoječih podatkov v bazi. Ob dodajanju ali brisanju vozlišč ali povezav se spreminja le struktura grafa in ne podatki sami. Z ločevanjem strukture od dejanskih podatkov se izognemo pogostim dostopom do pomnilnika.

Relacije med entitetami ustrezajo povezavam in potem v grafu. To po-

meni, da so poizvedbe v bazi kar sprehodi po grafu in nadalje, da so poizvedbe lahko hierarhične – se nanašajo na entitete ali relacije, ki niso neposredno povezane. Hierarhične poizvedbe so velika prednost graf podatkovnega modela. V splošnem velja, da prednost graf baze pri hierarhičnih poizvedbah raste z globino poizvedbene strukture oz. njeno kompleksnostjo. Če je drevo, s katerim ponazorimo hierarhično poizvedbo plosko ali izrojeno, je poizvedba primerljiva tistim v relacijskih bazah.

V vsakem vozlišču grafa enostavno najdemo njegove sosede, če se sprehodimo po povezavah, ki izvirajo ali ponikajo v vozlišču. Vsako vozlišče namreč neposredno naslavlja oz. se z nekakšno referenco nanaša nanj. Iz tega razloga ne potrebujemo imeniške oz. indeksne strukture, ki bi naslavljala vsako vozlišče ločeno od njegovih sosedov. Posledica brezindeksnega sosedstva je krajši čas poizvedbe, ki je deloma ali v celoti neodvisen od velikosti grafa v bazi.

Graf nas, skupaj s podatkovno strukturo in transformacijami, privede do podatkovnega modela, za katerega velja

- podatki in/ali podatkovna shema so predstavljeni z grafom. Najpreprostejši pristop k temu je predstavitev celotne baze z grafom – tj. shema in podatki skupaj tvorijo graf. Drugi pristopi vključujejo z grafom predstavljene podatke in ločeno implementacijo sheme, graf sheme in ločeno implementacijo predstavitve podatkov ter različne kombinacije naštetih in drugih strategij;

- obstaja dualnost med transformacijami podatkov – branje, ustvarjanje, spreminjanje in brisanje – ter transformacijami grafa oz. operacijami nad primitivi grafa – vozlišče, povezava, pot, podgraf;

- integritetne omejitve v podatkovni bazi ustrezajo tistim nad grafom.

V podatkovnem modelu z grafom sheme so entitetni tipi zastopani z vozlišči s prirejenim podatkovnim tipom. Povezave med podatkovnimi tipi, torej relacije med entitetnimi tipi, so predstavljene s povezavami v grafu. Povezave so označene z imeni relacij.

Podatkovni model z grafom instanc oz. grafom podatkov pa namesto sheme v grafu upodablja podatke same. Vozlišča torej predstavljajo instance entitet in so označena z imenom entitete, enoličnim identifikatorjem ali množico oz. kombinacijo le-teh. Vozlišča lahko poleg entitet predstavljajo primitivne podatkovne tipe. Oznake takšnih vozlišč so kar vrednosti iz domene primitivnega tipa. Povezave v grafu predstavljajo relacije med instancami entitet in primitivnimi podatkovnimi tipi.

Relacije so v podatkovnem modelu lahko preproste ali kompleksne oz. sestavljene. Preproste relacije so relacije, ki povezujejo dve entiteti z neko semantično vrednostjo – atribut. Kompleksne relacije pa so relacije, ki sestojijo iz več relacij – torej hierarhične – in bolj zahtevnimi semantičnimi elementi (na primer entiteta). V grafu predstavimo preproste relacije s povezavo, ki jo označimo z atributom. Kompleksne relacije se odvisno od podatkovnega modela v grafu odražajo različno.

Razširimo znanje o pogostejših tipih podatkovnih modelov v graf bazah.

3.1.1 Označeni graf lastnosti

Najbolj priljubljena oblika graf podatkovnega modela je označeni graf lastnosti. Vozlišča v lastnostnem grafu predstavljajo entitete, povezave pa relacije med njimi. Vozlišča so označena z eno ali več oznakami, ki določajo njihove vloge v podatkovni domeni. Poleg vlog lahko oznake določajo tudi skupine vozlišč, ki jim vozlišče pripada. Nasprotno so povezave lahko označene le z eno oznako. Edinstveno oznako povezave imenujemo tudi ime povezave ali tip povezave. Povezave so usmerjene in se vedno nanašajo na dve vozlišči – vodijo iz izvornega vozlišča v ponorno. Ker vsaka povezava vedno sestoji iz natanko dveh vozlišč, se pri odstranjevanju vozlišča, poleg vozlišča, ustrezno odstranijo tudi povezave, ki imajo vozlišče za izvorno ali ponorno. Tako predpostavimo, da vsaka obstoječa povezava v grafu nikoli ne naslavlja neobstoječih vozlišč.

Tako vozlišča kot povezave lahko nosijo več lastnosti ali atributov. Lastnost imenujemo urejen par, sestavljen iz ključa in vrednosti. Ključ je na-

vadno niz znakov in predstavlja ime lastnosti. Vrednost je lahko poljubnega podatkovnega tipa. Povezave imajo navadno kvantitativne lastnosti, kot so utež, cena, razdalja, moč ...

3.1.2 Hipergraf

Določeni podatki izkazujejo razmerja, ki se nanašajo na več kot dve entiteti. Takšne podatke preprosto predstavimo z označenim grafom lastnosti, vendar so nekatere povezave redundantne. Hipergraf je generalizacija označenega grafa lastnosti, ki omogoča poljubno kardinalnost povezav. To pomeni, da ima lahko povezava v hipergrafu poljubno število izvornih in ponornih vozlišč. Povezave v hipergrafu so lahko poljubnih dimenzij in jih imenujemo hiperpovezave.

Podatkovni model hipergraf je izomorfen podatkovnemu modelu označenega grafa lastnosti. Hipergraf lahko vedno predstavimo z označenim grafom lastnosti in obratno. Razlika je le v številu povezav, ki jih je v hipergrafu največ toliko kot v označenem grafu lastnosti.

Poglavje 4

Arhitektura

Procesorjev, ki avtonomno upravljajo z nestrukturiranimi podatki, je malo. Obstoječe implementacije strojne opreme pogosto le pospešujejo določene dele procesa transformacije in upravljanja s podatki. V času nastanka tega dela govori o specializaciji strojne opreme za procesiranje podatkov v grafu le delo [22]. Delo [20] obravnava ogrodje GraphGen, ki s pretvarjanjem obstoječega grafa v sintezo procesorja na FPGA izrablja prednosti strojno pospešenega računanja. Grafi, ki jih ogrodje GraphGen izvaja na FPGA, so statični. Spremembe v strukturi grafa pomenijo ponovno prevajanje in sintezo. O podobnem ogrodju govori tudi delo [6].

Poglavje je usmerjeno k teoretični obravnavi in evalvaciji procesorja grafov. Procesor grafov je ime za paralelno procesorsko arhitekturo, ki omogoča pospešeno računanje nad nestrukturiranimi podatki in podatki z visoko stopnjo medsebojne odvisnosti oz. povezanosti.

4.1 Paralelni sistemi

Zaporedno izvajanje ukazov je, izvzemajoče v primeru, ko je algoritem popolnoma sekvenčen, suboptimalno. Veliko računskih problemov namreč dovoljuje sočasno ali paralelno izvajanje opravil, ki sodelujejo pri reševanju posameznega problema.

S pridevnikom *sočásen* označujemo sisteme z zmožnostjo izvajanja več opravil s prekrivajočimi časovnimi obdobji. Soroden pojmu sočasni sistem, vendar konceptualno samostojen, je pojem paralelni sistem. Paralelni oz. vzporedni sistem je sočasni sistem, ki v vsakem trenutku hkrati izvaja več opravil. Paralelne sisteme osnujemo z izkoriščanjem paralelizma na več nivojih. Vsebina naslednjih podpoglavij, ki govorijo o različnih principih in tehnikah paralelizacije računalniškega sistema, je povzeta po delih [19], [16], [11], [12] in [15].

4.1.1 Paralelizem na nivoju ukazov

V zaporedju ukazov, ki jih izvršuje računalnik, pogosto odkrijemo množico medsebojno neodvisnih ukazov, ki se lahko izvršijo sočasno. S to motivacijo prilagodimo obstoječo arhitekturo tako, da kar najbolje izkoristimo ukazni paralelizem. Kot izboljšave v tehnologiji, so tehnike za izkoriščanje paralelizma na nivoju ukazov, s perspektive uporabnika, transparentne. Nasprotno od tehnik za izkoriščanje paralelizma na drugih nivojih, se sekvenčni uporabniški program, kljub dodatni strojni podpori za izkoriščanje ukaznega paralelizma, ne spremeni.

Stopnja paralelizma na nivoju ukazov se nanaša na število ukazov, ki jih sistem lahko izvede hkrati. Hkrati sme izvajati le ukaze, ki medsebojno ne vplivajo drug na drugega, ne naslavljajo istega registra ali iste pomnilniške besede. Predpogoj za to je dobra interferenčna analiza ukazov v programu. V sekvenčnem programu vrstni red ukazov, poleg pomena ukazov samih, tudi semantično določa pomen programa. Včasih je možno nekatere ukaze ali podzaporedje ukazov izvajati v drugačnem vrstnem redu, kot so zapisani v programu, brez da bi to vplivalo na rezultat ob koncu programa. Treba je le določiti njihov vpliv na množico ukazov, ki jim sledijo in poskrbeti, da se dva interferenčna ukaza nikoli ne izvajata sočasno. Torej ne naslavljata istih operandov ali ne dostopata do istih strojnih virov hkrati.

Medsebojne odvisnosti med ukazi mora tako prepoznati in ustrezno izvesti strojna oprema. Ob vsakem ukazu, ki ga superskalarni računalnik prevzame,

naj pred izstavljanjem preveri interferenco operandov – registrov ali pomnilniških lokacij – prevzetega ukaza s predhodno prevzetimi in izstavljenimi ukazi oz. ukazi, ki jih računalnik v tistem trenutku izvršuje. V primeru, ko se operandi prevzetega ukaza prekrivajo s tistimi predhodno prevzetih in izstavljenih, naj se izvajanje ukaza zamudi do trenutka, ko računalnik preneha z izvrševanjem konfliktnih ukazov. V tem času lahko računalnik prevzame in v izvajanje izstavi več naslednjih ukazov, pod enakimi interferenčnimi pogoji, kot veljajo za ukaze v čakalni vrsti. Na tem mestu gre poudariti, da za doseganje boljše performančne kapacitete prevzem več ukazov v enem procesorskem ciklu ni obvezen. Ukazno prepustnost lahko povečamo tudi na druge načine.

Ukazni cevovod je razmeroma preprosta in zelo pogosta razširitev procesorske arhitekture. Cevovod imenujemo metodo sočasnega izvrševanja več ukazov, ki omogoča večjo ukazno prepustnost sistema. V vsakem ciklu se torej izvaja več ukazov, dokončno pa se izvede le eden. Število izvedenih ukazov na cikel se ne poveča, poveča se le prepustnost. Izvrševanje ukazov je razdeljeno na manjše operacije oz. podoperacije, ki se izvajajo sočasno. Vsako od operacij izvrši del sistema, ki ga od ostalih delov sistema ločijo pomnilne celice in ki mu rečemo tudi stopnja cevovoda. Stopnje cevovoda so med sabo povezane s pomnilnimi celicami tako, da tvorijo vrsto, v katero ukazi vstopajo na eni strani in izstopajo na drugi. Vsak ukaz potuje skozi vse stopnje, a je v vsakem trenutku le v eni izmed njih.

Čas, ki ga cevovod potrebuje za izvajanje ukazov, je navzdol omejen s časom izvedbe najpočasnejše izmed stopenj. Zato mora biti delo, ki ga opravi posamezna stopnja, med vsemi stopnjami enakomerno porazdeljeno. V n-stopenjskem cevovodu se hkrati izvaja n ukazov. To pomeni, da se v istem časovnem obdobju na procesorju s cevovodom izvede n-krat več ukazov kot na procesorju brez. Število urinih period na ukaz je zato n-krat manjše. O n-krat izboljšani pretočnosti govorimo v idealnem primeru. V resnici je, zaradi zakasnitev v cevovodu, faktor prepustnosti v povprečju manjši. Zakasnitve v cevovodu preprečujejo cevovodne nevarnosti. Nastanek cevovodne nevarnosti

povzročijo odvisnosti operandov v zaporednih ukazih.

Z dobro delitvijo cevovoda na podoperacije oz. stopnje se skrajša urina perioda najpočasnejše izmed stopenj in posledično urina perioda sistema. To pomeni, da je čas, ki ga procesor potrebuje za izvrševanje katerekoli izmed stopenj, krajši kot sicer. Kako dobro lahko cevovod razdelimo na stopnje, je odvisno od kompleksnosti ukazov, ki jih določa arhitektura računalnika. V splošnem velja, da s povečanjem števila cevovodnih stopenj lahko vedno skrajšamo urino periodo, a ne brez posledic.

Kljub dobri prepustnosti cevovoda spoznamo, da se, zaradi interferenc v ukazih in posledičnih zakasnitev, ukazi ne izstavijo v vsaki urini periodi. Procesor bi ob nedejavnih periodah lahko izvajal druge ukaze, ki niso odvisni od izstavljenih ukazov. Z dinamičnim razvrščanjem ukazov glede na razpoložljivost operandov oz. podatkov lahko dosežemo prav to. Procesor z dinamičnim razvrščanjem ukazov spremeni vrstni red izvajanja strojnih ukazov tako, da zapolni urine periode, v katerih bi sicer čakal na zaključek izvajanja drugih ukazov z ukazi, ki se nanašajo na trenutno proste operande. Vrstni red strojnih ukazov torej določa razpoložljivost operandov in ni enak vrstnemu redu, kot je zapisan v programu.

4.1.2 Paralelizem na nivoju podatkov

Možnosti za paralelizacijo lahko izhajajo tudi iz podatkov, s katerimi računamo. Ključna karakteristika podatkovnega paralelizma je sposobnost izvajanja ene operacije nad več elementi neke regularne podatkovne strukture hkrati; najpogosteje nad elementi polja oz. vektorja, matrike ali druge zaporedne podatkovne strukture. Vsaka vektorska operacija se torej izvede nad več operandi hkrati.

Vektorski računalnik, tipično ime za tovrstno arhitekturo za računanje nad več operandi hkrati, namenja več vektorskih, pogosto tudi cevovodnih funkcijskih enot. Poleg funkcijskih enot vsebuje vektorske registre in ustrezne vektorske ukaze za delo s tovrstnimi podatki. Poleg procesorja mora vektorske operacije podpirati tudi pomnilnik oz. pomnilniški krmilnik. Pomnilnik

mora v enakem času, kot ga porabi za izstavitev enega operanda, procesorju dostaviti več operandov v vektorju. Operandi v vektorju so običajno na zaporednih naslovih, kar zmanjša kompleksnost pomnilniškega krmilnika, še vedno pa se v krajšem času med pomnilnikom in procesorjem izmenja več podatkov kot sicer. Performančna prednost (vrednost) podatkovno paralelne arhitekture je torej močno odvisna od hitrosti in lokalnosti pomnilniških dostopov. Velik zakasnitveni čas pomnilniškega dostopa se kaže v degradaciji zmogljivosti oz. učinkovitosti. Temu je moč nasprotovati s tehnikami za izkoriščanje paralelizma na nivoju ukazov iz prejšnjega razdelka. Dinamično razvrščanje ukazov na primer zmanjša vpliv ukazov za delo s pomnilnikom na računske ukaze. Prav tako izboljša učinkovitost rabe pomnilnika in amortizira zakasnitve, ki jih povzroča.

4.1.3 Paralelizem na nivoju procesorjev

Smiselna razširitev enoprocesorskega sistema je sistem z več procesorji. Večprocesorska arhitektura omogoča neodvisno izvajanje več opravil, procesov ali programov hkrati. Procesorjem je skupen le pomnilnik in vodilo, ki povezuje procesorje s pomnilnikom ter ostalimi komponentami v sistemu. Naslovni prostor je med procesorji deljen, fizični pomnilnik pa je lahko centraliziran ali porazdeljen.

V centralizirani pomnilniški konfiguraciji – imenujemo jo tudi arhitektura s poenotenim pomnilniškim dostopom – se vsak procesor s skupnim pomnilnikom povezuje prek vodila neposredno ali, bolj običajno, prek procesorju lokalnega predpomnilnika. Procesorji se medsebojno sporazumevajo prek skupnega pomnilnika, ki je običajno razdeljen na več modulov. Z medsebojnim izključevanjem lahko do vsakega modula dostopa največ en procesor hkrati. Fragmentacija pomnilniškega prostora na več modulov torej pomeni več hkratnih dostopov.

Nasprotno je v decentralizirani pomnilniški konfiguraciji – arhitektura z deljenim pomnilniškim dostopom – pomnilnik porazdeljen. Vsak procesor dostopa do svojega dela pomnilnika prek lokalnega vodila neposredno ali prek

predpomnilnika in do preostalih delov pomnilnika preko skupnega vodila. Skupno vodilo je tako bolj razbremenjeno kot v arhitekturi s centraliziranim pomnilnikom, kar vpliva na boljšo horizontalno raztegljivost sistema.

V večprocesorskih sistemih, specifično v primerih, kjer znotraj enega opravila procesor dostopa do virov skupnim drugim opravilom, se srečamo s sinhronizacijskimi in organizacijskimi problemi. Nenadzorovano oz. neomejeno izvajanje opravil lahko namreč vodi do napačnih rezultatov, anomalij v podatkih, nepredvidenega delovanja, paralelne upočasnitve[1] in podobno.

Sovisnost

Procesorji v večprocesorskem sistemu si delijo pomnilniški naslovni prostor. To pomeni, da lahko v nekem časovnem obdobju do iste pomnilniške lokacije dostopa več procesorjev, ki podatke zapisujejo ali zgolj berejo. Med predpomnilniki, ki hranijo lokalne kopije podatkov za vsak procesor, lahko nastanejo odstopanja v podatkih iste pomnilniške lokacije, če vsaj eden izmed procesorjev na to mesto v pomnilniku zapiše nove podatke. Predpomnjene vrednosti ostalih procesorjev so potemtakem napačne oz. nesovisne s podatki v glavnem pomnilniku. V boju proti nesovisnosti uporabljamo različne protokole, ki sovisnost zagotavljajo.

Vohunjenje[19] je eden takšnih protokolov. V procesu vohljanja vsak predpomnilnik opazuje vodilo, ki je skupno vsem predpomnilnikom in glavnemu pomnilniku ter spremlja zahteve po branju in pisanju za pomnilniške lokacije, ki jih predpomni. Ob odkriti pisalni operaciji za predpomnjeno pomnilniško lokacijo, predpomnilnik podatke za to lokacijo razveljavi. Vohljanje je lahko zelo hitro, če je pasovna širina vodila ustrezna. V učinkoviti konfiguraciji, kjer naj bodo vse zahteve na vodilu vidne vsem predpomnilnikom hkrati, je pasovna širina vodila sorazmerna s številom procesorjev. Sistem s protokolom vohljanja je posledično slabo raztegljiv. Alternativa vohljanju so direktorijski protokoli.

[1] fenomen sočasnih sistemov, pri katerem paralelizirano izvajanje programa teče počasneje kot sekvenčno.

Sinhronizacija

O sinhronizaciji govorimo, kadar želimo preprečiti sočasni dostop procesorjev do skupnih virov ali sočasno izvrševanje ukazov določenega programa ali dela programa. Posledice, ki nastanejo zaradi nepravilne sinhronizacije, se kažejo v degradaciji hitrosti, neodzivnosti ali zastoju sistema.

Osnova vsem sinhronizacijskim konceptom so atomične operacije. V paralelnih sistemih proglasimo operacijo ali množico operacij nad skupnim virom za atomično, če je njeno izvrševanje v celoti neprekinjeno in asinhrono z ostalimi operacijami nad istim virom. Atomične operacije omogočajo medsebojno izključevanje procesov oz. procesorjev v paralelnem sistemu in s tem sekvenčno ter neprekinjeno izvajanje. Na podlagi atomičnih operacij lahko načrtujemo obsežnejše sinhronizacijske protokole, kot so ključavnica, prepreka in semafor.

Zaklepanje oz. medsebojno izključevanje je pogosta metoda medprocesorske sinhronizacije. Ključavnico sestavljata pomnilniška lokacija – pomnilniška beseda, rezervirana v ta namen – in atomična operacija, ki v pomnilniško lokacijo vpisuje vrednost glede na stanje ključavnice oz. njeno razpoložljivost. Ob dostopu do izključujočega vira ali vstopu v kritični odsek programa, procesor atomično vpiše vnaprej dogovorjeno vrednost, ki označuje, da je ključavnica zaklenjena, na mesto ključavnice v pomnilniku. Preden procesor zapiše novo vrednost, preveri obstoječe stanje ključavnice tako, da prebere vsebino pomnilniške besede na naslovu ključavnice. Ob zaklenjeni ključavnici se, odvisno od implementacije, procesor ustavi ali počaka na odklep ključavnice. V prvem primeru prekine z izvrševanjem trenutnega ukaza in nadaljuje z njegovim izvajanjem ob odklepu ključavnice. V nasprotnem primeru procesor prav tako preneha z izvrševanjem ukaza, vendar se ne ustavi, temveč v zanki poizveduje po trenutnem stanju ključavnice. Ob odklepu ključavnice nadaljuje s ponovnim poskusom zaklepa.

Nepreviden raba ključavnic lahko povzroči smrtni objem ali živo zanko. Smrtni objem je stanje sistema, v katerem dva ali več procesorjev s prisvojenimi viri čaka na sprostitev virov v lasti drugega procesorja. Do stanja pride,

ko več procesorjev zaklepa več kritičnih sekcij zaporedoma, vendar v različnem vrstnem redu. Procesorji čakajo na sprostitev naslednje ključavnice, vendar nikoli ne sprostijo svoje. V izogib smrtnemu objemu vzpostavimo hierarhijo ključavnic in dovolimo le zaklepanje ključavnic, ki so po hierarhiji višje od že zaklenjenih ključavnic.

Za sinhronizacijo vseh procesorjev, tudi tistih, ki pri zaklepanju iste ključavnice ne sodelujejo, uporabimo pregrade. Pregrado imenujemo sinhronizacijski protokol, v katerem sodelujoči procesorji prenehajo z izvajanjem, ko preidejo v določeno stanje ali dosežejo določeni ukaz v programu. Izvajanje lahko nadaljujejo šele takrat, ko so v istem stanju oz. ko izvajajo isti ukaz tudi vsi drugi udeleženi procesorji. S pregrado torej ločimo stanja oz. dele programov, v katerih so oz. ki jih lahko izvajajo procesorji hkrati.

4.2 Transakcijski pomnilnik

Transakcija je zaporedje operacij z zagotovilom atomičnosti. Celotno zaporedje operacij se izvede uspešno ali neuspešno, preden se lahko izvede naslednja operacija ali transakcija. V primeru, da se vsaj ena izmed operacij v zaporedju ne izvede uspešno, se transakcija označi za neuspešno in se spremembe, ki so rezultat operacij, ovržejo.

Transakcijsko izvajanje operacij je veliko bolj optimistično kot ključavnice in drugi sinhronizacijski protokoli na nivoju procesorjev ali niti. Operacije v transakciji se namreč, nasprotno od operacij v kritični sekciji programa, lahko izvajajo sočasno. Spremembe, ki jih povzročijo, postanejo nepreklicne – se v skupnem viru uveljavijo – le v primeru, da niso konfliktne s spremembami, ki so jih med izvajanjem transakcije povzročile operacije drugih transakcij. Nasprotno se po analizi konfliktov spremembe transakcijskih operacij ovržejo, če sovpadajo. Preden postane transakcija veljavna znotraj skupnega vira, so nastali rezultati znotraj transakcije popolnoma spekulativni. V kontrastu s ključavnično sinhronizacijo, kjer se, v izogib korupciji vira kritične operacije izvajajo strogo zaporedno, se več transakcij lahko izvaja sočasno.

Pomembno je le, koliko operacij znotraj transakcij spreminja isti skupni vir in na kakšen način. Če ob koncu transakcij spremembe ne sovpadajo, se štejejo vse transakcije s skupnim ciljem za uspešne.

Dve transakciji sta konfliktni druga z drugo le v primeru, ko sta množici pomnilniških lokacij, ki jih naslavljata, v preseku. Kratke transakcije oz. transakcije, ki pomnilnika ne spreminjajo znatno, bodo v velikem pomnilniškem prostoru redko v konfliktni situaciji. Konflikte zaznamo na različne načine in v različnih fazah izvajanja transakcij. Pogosto se odkrivanje konfliktov zgodi ob končani transakciji, preden spremembe, ki jih je povzročila transakcija, postanejo obstojne. Konflikti se v tem primeru iščejo v množici pomnilniških lokacij, ki so jih pisalne operacije znotraj transakcije naslavljale. Nekateri sistemi uporabljajo sprotno preverjanje konfliktov. Ob vsaki pisalni operaciji znotraj transakcije se konflikti zaznavajo na pomnilniški lokaciji, na katero operacija piše. V primeru konflikta z drugo transakcijo, ena izmed transakcij preneha z izvajanjem, druga pa z izvajanjem nadaljuje na naslednji operaciji. Ker je sprotno preverjanje konfliktov lahko časovno potratno, omejimo preverjanje tako, da se izvede le po določenem številu pisalnih operacij, torej le nekajkrat na transakcijo, v vsakem primeru pa vsaj enkrat.

Pri programiranju s transakcijami uporabniku odvzamemo moč eksplicitnega navajanja ekskluzivnosti dostopa do skupnega vira ali vrstnega reda izvajanja kritičnih sekcij programa. S tem se izognemo smrtnim objemom in drugim posledicam nepravilne rabe nizkonivojskih sinhronizacijskih protokolov.

Poleg omenjenih prednosti se v specifičnih rabah transakcijskih protokolov kažejo njihove slabosti. Neidempotentne operacije – operacije, ki ob vsaki izvedbi proizvedejo drugačne rezultate, npr. operacije nad vhodno-izhodnimi napravami – so za transakcijski sistem problematične, saj se, v primeru konfliktov, znotraj transakcije izvedejo večkrat. Transakcije z neidempotenčnimi operacijami postanejo z uporabniškega vidika neprosojne, saj neposredno vplivajo na stanje in izid programa oz. dela programa.

Ob konfliktu dveh transakcij smo primorani eno izmed njiju ali celo obe

ponoviti, da se izognemo neskladju podatkov. Večkratno ponavljanje transakcij je v določenih primerih problematično za učinkovitost sistema. Več majhnih transakcij lahko negativno vpliva na izvajanje transakcij z velikim številom operacij. To se kaže v stradanju velikih transakcij. Večje transakcije lahko zaradi konfliktov, ki jih med izvajanjem povzročijo manjše transakcije, nenehno ponavljajo svoje operacije. Stradanje pričakujemo predvsem v primerih, ko veliko procesov sočasno dostopa do majhne množice skupnih virov, velikosti transakcij, ki jih izvajajo, pa se med posameznimi procesi močno razlikujejo.

Transakcije v procesih z majhno prioriteto lahko v primeru velike interference podaljšajo izvajanje transakcij v procesih z velikimi prioritetami. Čas, ki ga procesi z visoko prioriteto potrebujejo za izvajanje v celoti, je torej daljši. Slednje lahko ogroža delovanje sistema, od katerega uporabnik pričakuje odziv v realnem času – je časovno kritičen.

Pogosto rabo transakcij najdemo v paralelnih sistemih z deljenim pomnilnikom, kjer dobro raztegljivost sistema težko dosežemo, če uporabljamo ključavniške mehanizme pomnilniške sinhronizacije in sovisnosti podatkov.

4.3 Procesor grafov

Procesor grafov je paralelni računalnik z izredno podporo za delo s podatki v grafu oz. s podatki, ki izkazujejo visoko stopnjo medsebojne povezanosti. Od tipične računalniške arhitekture se oddaljuje z več avtonomnimi procesnimi jedri, ki opravljajo grafu specifične operacije, hierarhičnim transakcijskim pomnilnikom in s programskim vmesnikom, ki se močno razlikuje od strojnih jezikov sodobnih procesorskih arhitektur. Zaradi večprocesorske zasnove ga uvrščamo v razred večprocesorskih oz. večračunalniških sistemov – po Flynnovi klasifikaciji tudi razred MIMD. Zasnova procesorja omogoča dobro vertikalno raztegljivost z dodajanjem strojnih virov, kot tudi horizontalno raztegljivost s povezovanjem večih instanc procesorja grafov v širokopasovno omrežje.

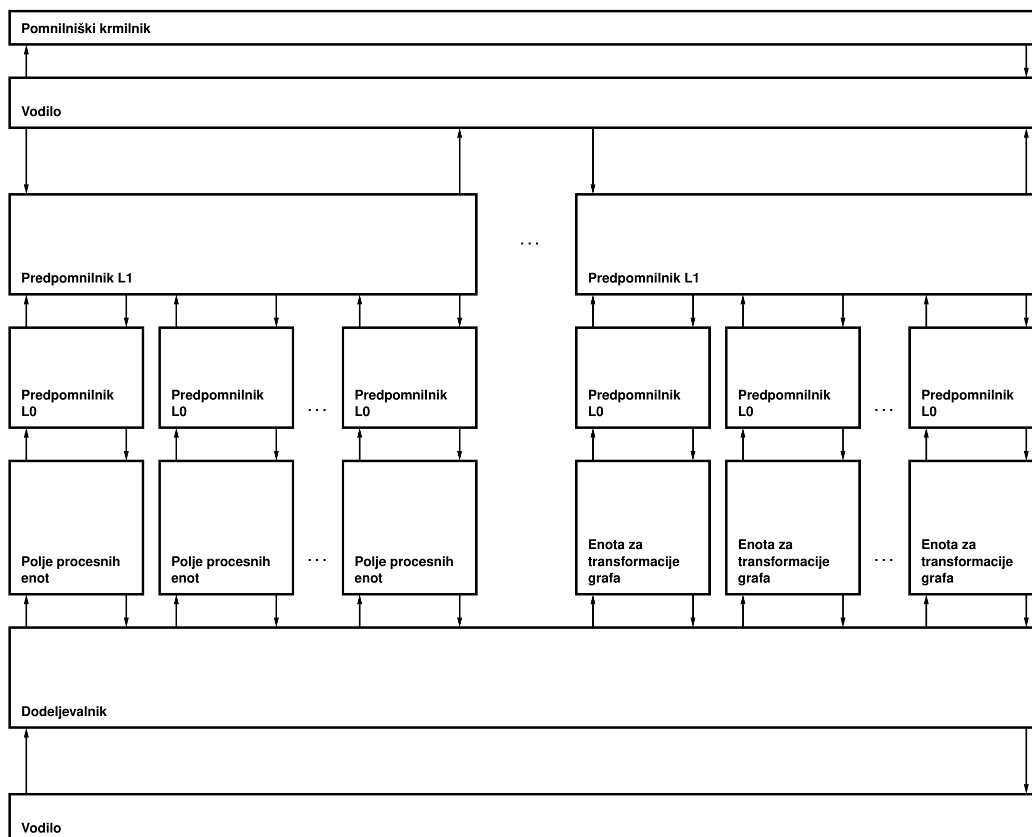
Neobičajni arhitekturi primeren je prav tako programski vmesnik. Programski vmesnik procesorja grafov omogoča programiranje kot ga pričakujemo pri sodobnih procesorjih, a se zaradi arhitekturnih razlik procesnih enot, ki izvajajo ukaze programa, oddaljuje od tovrstnih procesorjev. Strojni jezik je posledično manj kompleksen in bolj usmerjen k specifičnim operacijam, ki jih izvajamo nad podatki grafa.

Shemo procesorja prikazuje slika 4.1. Procesor sestavlja hierarhični predpomnilnik, ki je prek vodila povezan s pomnilniškim krmilnikom in nadalje z glavnim pomnilnikom ter izpolnjuje zahteve procesnih enot o branju in pisanju v pomnilniški prostor. Enota za koordinacijo dela, krajše dodeljevalnik, deli delo med procesne enote, skrbi za enakomerno porazdelitev dela in prek komunikacijskega vmesnika izmenjuje podatke z zunanjimi napravami. Transformacije grafa izvajajo procesne enote, ki so prek prioritetne vrste povezane v predpomnilniško hierarhijo in prek vodila z enoto za koordinacijo dela. Več procesnih enot, ki sodelujejo pri reševanju nalog, skupaj tvori procesorsko polje.

4.4 Pomnilniška hierarhija

Delovanje procesorja grafov v veliki meri poverjajo matrične operacije, ki zahtevajo dobro prepustnost med pomnilnikom in procesnimi enotami. Hitrost, predvsem pa učinkovitost sistema, je tako najbolj odvisna od hitrosti pomnilniških operacij, tj. branja in pisanja podatkov v pomnilnik.

V tipični von Neumannovi računalniški arhitekturi si centralna procesna enota lasti peščico registrov, iz katerih bere in vanj piše podatke. Ti registri so sestavljeni iz ene ali več pomnilnih celic. Dostop do registrov je zelo hiter, a je njihovo število omejeno. Podatke, ki trenutno niso v rabi, centralna procesna enota hrani v glavnem pomnilniku. Glavni pomnilnik je velik, a je zaradi cene pomnilniške tehnologije veliko počasnejši od registrov v centralni procesni enoti. Hitrost glavnega pomnilnika, še posebej pri pogostih dostopih, omejuje učinkovitost centralno procesne enote. Ob



Slika 4.1: Shema procesorja grafov. Jedro procesorja predstavljajo dodeljevalnik, polja procesnih enot, enote za transformacije grafa in pomnilniška hierarhija. Prek vodil se procesor povezuje s pomnilniškim krmilnikom in ostalimi komponentami v vgrajenem sistemu.

vsakem dostopu do glavnega pomnilnika mora CPE namreč čakati na zaključek pomnilniške operacije, preden lahko izvaja ukaze nad operandi, ki jih od pomnilnika zahteva ali prepiše register operanda, ki ga v pomnilnik shranjuje. Negativnim posledicam velikosti glavnega pomnilnika lahko nasprotujemo z manjšim pomnilnikom, ki hrani podatke, ki so pogosto v rabi. Dostopi do tega pomnilnika so tako hitrejši in bolj pogosti kot dostopi do glavnega pomnilnika. Branje ali pisanje operandov v glavni pomnilnik se izvede le v primeru, ko operandov v hitrem pomnilniku ni. Hitri pomnilnik, ki hrani podmnožico množice operandov glavnega pomnilnika, imenujemo predpomnilnik. Uvedba predpomnilnikov v računalniškem sistemu je smiselna, če so si operandi v pomnilniku, do katerih dostopa procesor, blizu, tj. na zaporednih lokacijah v pomnilniku in/ali če procesor v časovnih intervalih pogosto dostopa le do množice operandov v pomnilniku.

Lastnost takšnih pomnilniških dostopov imenujemo lokalnost. Pomnilniške dostope, ki znotraj krajšega časovnega obdobja pogosto naslavljajo iste operande, torej majhno podmnožico množice operandov v pomnilniku, označimo za časovno lokalne. Podobno označimo pomnilniške dostope, ki znotraj krajšega časovnega obdobja naslavljajo operande na zaporednih ali bližnjih naslovih, za prostorsko lokalne. Lokalnost pomnilniških dostopov vedno ugotavljamo na več kot enem dostopu. Dostopi v množici dostopov so lahko prostorsko in časovno lokalni.

Pomnilniška hierarhija v procesorju grafov sestoji iz treh nivojev. Glavni pomnilnik je skupen vsem enotam v procesorskem polju. Z njim se povezujejo posredno prek predpomnilnikov na nivoju L0 in L1. Predpomnilnik L1 pripada skupini enot, sestavljeni iz procesnih enot ali enot za transformacijo grafa, ki se z njim povezujejo prek predpomnilnika L0. Predpomnilnik L0 je za posamezno enoto lokalni pomnilnik in predstavlja njene registre. Ti registri niso programsko naslovljivi. Predpomnilnik L1 nad vodilom, ki si ga deli s predpomnilniki na nivoju nižje, izvaja arbitražo. Prednost daje pomnilniškim zahtevam tistim enotam v skupini, ki izvajajo prioritetno kritične operacije. Če takšnih procesnih enot v čakalni vrsti ni, izpolnjuje zahteve

v vrstnem redu, kot ga določa razvrščanje s krožnim dodeljevanjem (ang. round robin scheduling). Del predpomnilniške logike na nivoju L1 zato predstavlja tudi prioritetni kodirnik, ki razvršča zahteve v vrsti glede na njihove prioritete.

Zaščita pomnilnika in omejitve dostopa so pomemben del abstrakcije rokovanja s pomnilnikom. Preprečujejo namreč naključno oz. nepričakovano pisanje v pomnilnik in s tem tudi prepisovanje obstoječih podatkov ter pisanje v del pomnilnika, ki ni bil predhodno dodeljen v ta namen.

Ker do predpomnilnika L1 hkrati dostopa več enot v skupini, mora predpomnilnik skrbeti za sovisnost podatkov z beleženjem dostopov in ustreznim označevanjem delov pomnilnika, ki so trenutno v lokalnem pomnilniku (tudi predpomnilniku) posamezne enote. Predpomnilnik L1 v procesorju grafov je zato transakcijski.

4.4.1 Segmentacija pomnilnika

Pomnilniški prostor skorajda v celoti zasedajo matrične reprezentacije grafa. Poleg matrik pomnilnik hrani podatke o zasedenosti pomnilnika, razne statistične informacije o grafih in druge podatke o sistemu. Velikost ene pomnilniške besede ustreza velikosti matrike, ki jo lahko v okviru ene transakcije enota v procesorskem polju prenese iz pomnilnika ali ob pisanju nazaj v pomnilnik. Beseda v pomnilniku predstavlja podmatriko ene izmed bločnih matrik grafov v sistemu. Več pomnilniških besed na zaporednih naslovih tvori segment, ki predstavlja še večji blok matrike grafa kot posamezna pomnilniška beseda. Segment je, z vidika upravljanje pomnilniških virov, atomarna enota v pomnilniškem prostoru. To pomeni, da se ob dodelitvi prostora in sproščanju pomnilnika vedno dodeli ali sprosti vsaj en ali več segmentov v celoti. Ob alokaciji pomnilniškega prostora za podatke v specifični matriki se tako dodeli celoten segment pomnilniških besed, četudi matrika oz. podmatrika matrike ne zaseda celotnega segmenta. Na ta način se izognemo časovno zahtevnemu alokacijskemu postopku, hkrati pa izboljšamo prostorsko lokalnost pomnilniških dostopov. Če je matrika redka in je segment velik, izkorišče-

nost pomnilnika upade. Med načrtovanjem sistema moramo zato premišljeno določiti velikost segmenta v pomnilniku.

S segmenti neposredno upravljajo kontrolne enote polj procesnih enot in enote za transformacije grafa s poseganjem v pomnilniški prostor, posredno pa tudi dodeljevalnik, ki z razvrščevanjem ukazov nadzoruje in omejuje pomnilniške operacije.

4.5 Enota za koordinacijo dela

Enota za koordinacijo dela oz. dodeljevalnik povezuje zunanje naprave s procesorjem grafa in opravlja funkcijo programskega vmesnika v sistemu. Od zunanjih naprav prejema ukaze in podatke, na podlagi katerih se v procesorju grafov izvajajo transformacije ter poizvedbe nad grafom. Na prejete ukaze odgovarja z njihovimi rezultati, ki jih pošlje izdajatelju ukaza.

Ob prejemu ukazov ustvari enota za koordinacijo dela ustrezne operacijske pare, sestavljene iz množice naslovov segmentov v pomnilniku in operacije, ter jih odpremi v ustrezno procesorsko polje. Pri tem semantično in sintaksno ovrednoti ukaz ter prevaja operande ukazov v ustrezne pomnilniške lokacije. Ciljno polje za vsak ukaz izbere dodeljevalnik na podlagi trenutnega stanja in operacijske zgodovine posameznega polja. Operandi v predpomnilniku polja, zasedenost polja, zahtevnost in prioriteta ukaza so pomembni faktorji pri izbiri cilja. S tem skuša maksimirati podatkovno in ukazno prepustnost sistema ter minimizirati odzivni čas za ukaze z visoko prioriteto. Hkrati skuša delo pošteno porazdeliti med procesorska polja. Običajno te lastnosti sovpadajo. V procesorju grafov se dodeljevalnik zato privzeto posveča prepustnosti sistema. Pri tem spodbuja ponovno uporabo obstoječih podatkov z razvrščanjem ukazov v polja, ki v predpomnilniku že hranijo operande, ki jih ukaz naslavlja.

4.6 Prioritetni kodirnik

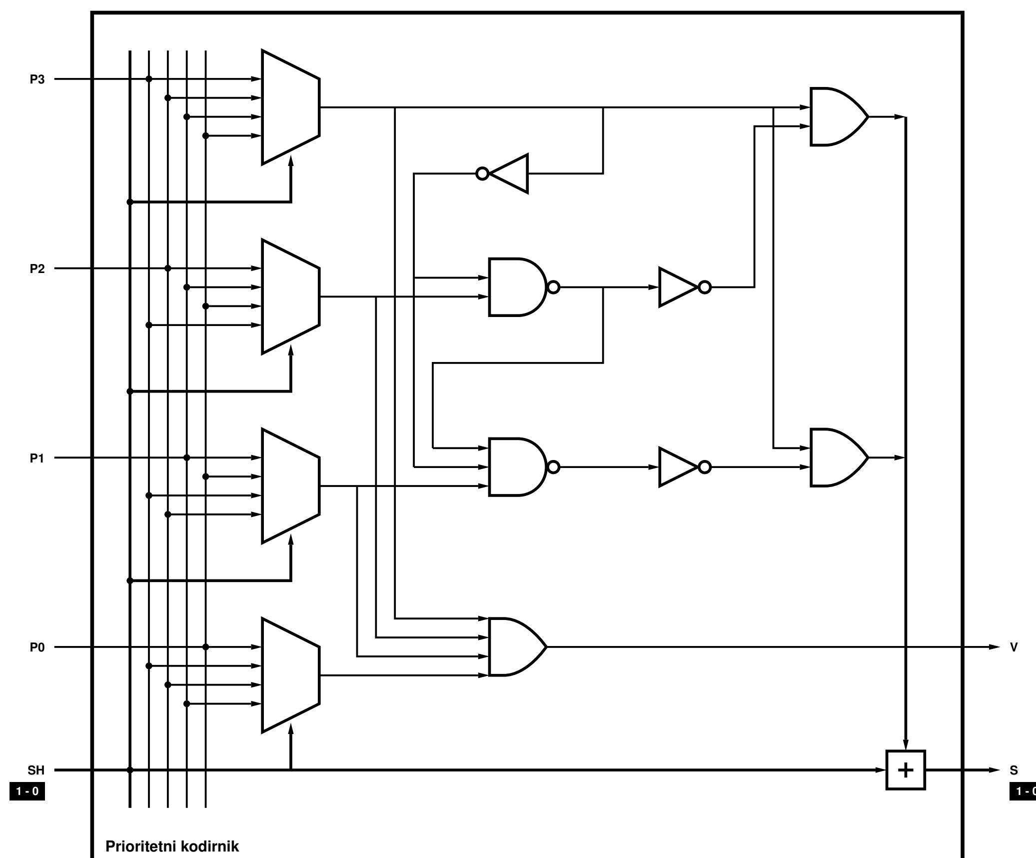
O omejitvi oz. prioriteti dostopa do skupnega vira skupine enot se sistem odloča na več mestih. Konfliktne situacije dinamično razsodi enota, ki ji pravimo prioritetni kodirnik.

Prioritetni kodirnik opravlja podobno logično funkcijo kot običajni kodirnik, le da definicijsko območje logične funkcije obsega tudi primere, ko je v visokem stanju več kot en vhod. V stanju z več aktivnimi vhodi, izhod funkcije odraža položaj vhoda z največjo težo. Logična funkcija prioritetnega kodirnika z n vhodi, priredi izhodu $log_2(n)$-bitno vrednost, ki v binarnem zapisu določa položaj najvišjega vhoda – vhoda z največjo težo – v visokem stanju. Če je najvišji aktivni vhod $k \leq n$ in so izhodi $k < j \leq n$ neaktivni, je na izhodu vrednost $(k-1)_2$. Vhodi $1 \leq l < k$, ne glede na stanje, v katerem se nahajajo, ne vplivajo na stanje izhoda.

Logično vezje prioritetnega kodirnika je preprosto. Vsak bit izhoda določajo pripadajoča vrata OR, ki so prek negatorja (vrata NOT) povezana z izhodi vrat NAND ustreznih vhodov. Vrata NAND vsakega vhoda preslikajo stanje pripadajočega vhoda in stanje izhodov vrat NAND vhodov z večjo težo v komplement konjunkcije. V takšni strukturi vezja je očitno, da je s prehodom vhoda z višjo prioriteto v visoko stanje, logična funkcija do stanja vhodov z nižjo prioriteto izključujoča.

Osnovni prioritetni kodirnik za rabo v procesorju grafov prilagodimo tako, da omogočimo dinamično izbiro prioritete posameznih vhodov ne glede na njihovo začetno utežitev. Dinamični prioritetni kodirnik z n vhodi v procesorju grafov torej sestoji iz osnovnega prioritetnega kodirnika z n vhodi, n multiplekserjev in seštevalnika. Vsak izmed vhodov je povezan z n multiplekserji, ki predstavljajo različne permutacije prioritete izhodov.

Izhodno stanje multiplekserjev določa ločen vhod z $log_2 n$ biti. Izhodi multiplekserjev so povezani pripadajočim vhodom prioritetnega kodirnika. Izhod prioritetnega kodirnika predstavlja vhod z najvišjo prioriteto, če vhodni vektor aritmetično pomaknemo za število mest, kot jih določa vhod za pomik. Izhodni vrednosti prioritetnega kodirnika se v seštevalniku prišteje



Slika 4.2: Shema prioritetnega kodirnika. Vhod P0 ima najmanjšo in vhod P3 največjo težo oz. prioriteto. Signali na vhodu SH določajo prioriteto s krožnim pomikanjem signalov P0-P3 za število mest, kot jih kolektivno določajo njihove vrednosti oz. stanja. Izhod S določa izbran vhod prioritetnega kodirnika, izhod V pa veljavnost izhoda S. Izhod S predstavlja veljavno vrednost, če je vsaj eden izmed vhodov P0-P3 v visokem stanju.

vrednost vhoda za pomik. Na ta način izhodna vrednost določa pravi položaj vhoda z najvišjo prioriteto brez zamika. Slika 4.2 prikazuje shemo dinamičnega prioritetnega kodirnika s 4 vhodi.

4.7 Procesorsko polje

V procesorju grafov se vse delo odvija na procesnih enotah. Glede na delo, ki ga opravljajo, delimo procesne enote na dve vrsti – enote za transformacije grafa in aritmetične procesne enote. Več procesnih enot iste vrste skupaj tvori procesorsko polje. Vsako procesorsko polje poleg procesnih enot sestavlja še predpomnilnik L1 in lokalni dodeljevalnik.

Vsaka enota za transformacije grafa deluje neodvisno od ostalih enot iste vrste – izvajanje ukazov poteka sočasno. Komunikacije med posameznimi procesnimi enotami ni. Enote v skupini si delijo le predpomnilnik L1, vsaka pa si lasti svoj predpomnilnik L0. Nasprotno je delovanje posamezne procesne enote v polju aritmetičnih procesnih enot sinhrono delovanju ostalim procesnim enotam v polju. Enote so s predpomnilnikom L0 povezane posredno prek kontrolne enote polja. Kontrolna enota skrbi za dostavo operandov iz predpomnilnika in rezultatov vanj ter usklajuje proces izvajanja ukazov v aritmetičnih procesnih enotah. Aritmetične procesne enote v procesorskem polju so med sabo povezane s komunikacijskimi kanali tako, da tvorijo mrežo. Prek kanalov si izmenjujejo operande, ki jih potrebujejo in s tem razbremenijo kontrolno enoto ter posledično predpomnilnik.

4.8 Enota za transformacije grafa

Enote za transformacije grafa so odgovorne za spremembe in poizvedbe v podatkovni strukturi oz. delih podatkovne strukture ter za dodeljevanje in sproščanje pomnilniškega prostora, ki ga podatkovna struktura zaseda. Podatkovno strukturo v tem kontekstu sestavljajo različne matrike grafa. Posegi v strukturo, ki jih opravi enota za transformacije grafa, so na primer doda-

janje in odstranjevanje vozlišč ter povezav, spreminjanje smeri obstoječih povezav, označevanje elementov grafa in podobno.

Več enot lahko sočasno opravlja transformacije nad istim delom grafa. Za sovisnost podatkovne strukture skrbi transakcijski predpomnilnik v polju. Če spremembe sovpadajo – dve enoti sočasno spreminjata vrednost uteži iste povezave – je ena izmed enot primorana transformacijo ponoviti. Posamezna enota za transformacije grafa sestoji iz kontrolne enote in aritmetično-logične enote. Vsak prejet ukaz se najprej pretvori v ustrezne signale kontrolne enote. Absolutni naslovi operandov v ukazu, tj. indeks vozlišča v grafu, se prevedejo v naslove segmentov ali besed v pomnilniku. Naslovljene besede se nato preberejo iz predpomnilnika L0 in se po končani transformaciji v aritmetično-logični enoti zapišejo nazaj v predpomnilnik ali posredujejo dodeljevalniku.

4.9 Aritmetična procesna enota

Vse analitične informacije, ki jih lahko izvemo o grafu – na primer najkrajša pod med vozlišči ali povezanost – so rezultat aritmetičnih in logičnih operacij procesnih enot nad različnimi matričnimi reprezentacijami grafa. Procesne enote v polju sodelujejo pri izvajanju aritmetičnih operacij. Med seboj so povezane z registri, ki so namenjeni prenosu operandov med procesnimi enotami in komunikaciji. Tako je na primer matrično množenje za matrike večje od tistih, ki jih lahko zmnoži ena procesna enota, razporejeno med več enot v polju. Porazdeljeno računanje pozitivno vpliva na hitrost in učinkovitost sistema, komunikacija med enotami pa na količino prometa med procesorskim poljem ter predpomnilnikom. Vsako procesno enoto v polju sestavljajo kontrolna enota, cevovodna aritmetično-logična enota in manjši pomnilnik.

4.9.1 Aritmetično-logična enota

Aritmetično-logična enota procesne enote je v celoti paralelizirana in vsebuje več manjših aritmetičnih ter logičnih enot, ki izvršujejo specifične operacije. Na ta način omogoča, poleg operacij nad skalarji, množenje matrike z ma-

triko, matrike z vektorjem, vektorja z vektorjem in skalarne operacije nad vektorjem ter matriko. Največja velikost matrike, nad katero je mogoče izvajati aritmetične in logične operacije, je parametrično določena v opisu strojne opreme arhitekture.

Ker želimo graf predstaviti z matriko in transformacije nad grafov z matričnimi operacijami, naj bodo matrične operacij nadvse učinkovite. Naivno množenje dveh kvadratnih matrik razsežnosti $n \times n$, pri čemer so vse operacije nad posameznimi skalarji zaporedne, zahteva n^3 operacij množenja in najmanj $(n-1)n^2$ operacij seštevanja. Očitno je, da bodo zaporedne aritmetične in logične operacije ozko grlo sistema. Aritmetično-logično enoto paraleliziramo na način, ki omogoča pospešeno delo z matrikami in matričnimi operacijami.

Matrike, katerih dimenzije presegajo zmožnosti vektorske aritmetično-logične enote, je smiselno porazdeliti med več procesnih enot. Za porazdelitev matričnega množenja med več procesnih enot obstaja več strategij oz. pristopov.

4.9.2 Deli in vladaj

Metoda deli in vladaj je ena izmed strategij za uspešno paralelno matrično množenje [10]. V splošnem z metodo deli in vladaj načrtujemo algoritme tako, da reševanje začetnega problema razdelimo na manjše naloge, ki rešujejo le del problema. Naloge rekurzivno delimo še na manjše naloge, dokler ni nalog, ki jih dobimo, mogoče trivialno rešiti.

Metodo uporabimo pri reševanju matričnega množenja. Pri tem začetno matriko oz. matriki razdelimo v manjše matrike oz. podmatrike. Matriko, sestavljeno iz manjših podmatrik, imenujemo bločna matrika. Bloke v matriki rekurzivno delimo v manjše bloke, dokler blok ne predstavlja ene same skalarne vrednosti ali matrike, ki jo trivialno zmnožimo.

Časovna kompleksnost matričnega množenja z metodo deli in vladaj je

podana rekurzivno

$$T(1) = \Theta(1) \tag{4.1}$$

$$T(n) = 8\Theta\left(\frac{n}{2}\right) + \Theta(n^2) \tag{4.2}$$

Po Masterjevem izreku dobimo iz zgornje enačbe za $T(n)$

$$a = 8, b = 2, f(n) = \Theta(n^2), c = 2 \tag{4.3}$$

$$log_b a = log_2 8 = 3 > c \tag{4.4}$$

$$T(n) = \Theta(n^{log_b a}) = \Theta(n^3). \tag{4.5}$$

Iz rezultata je razvidno, da z metodo deli in vladaj dosežemo enako časovno zahtevnost kot z naivnim oz. iterativnim načinom množenja.

4.9.3 Cannonov algoritem

Matrično množenje zahteva veliko podatkovnega prometa med procesorjem in pomnilnikom oz. med procesorji in pomnilnikom v večprocesorskem sistemu. Lokalnost pomnilniških dostopov je ob matričnih operacijah vseeno boljša kot ob operacijah nad drugimi podatkovnimi strukturami, s katerimi lahko predstavimo graf, vendar se kljub pomnilniški hierarhiji oz. dobri pomnilniški organizaciji hitro znajdemo v situaciji, ko čas branja iz pomnilnika in pisanja podatkov vanj presega čas aritmetičnih operacij, ki jih je zmožen opraviti procesor med temi pomnilniškimi operacijami. To pomeni, da bo procesor, medtem ko bo čakal na operande, nedejaven, kar zmanjšuje učinkovitost sistema. Na večprocesorskem sistemu je ozko grlo komunikacije s pomnilnikom še bolj izrazito, saj s skupnim pomnilnikom podatke izmenjuje več procesorjev. Pogostih pomnilniških operacij se deloma lahko znebimo, če med procesorje v večprocesorskem sistemu umestimo komunikacijske vmesnike – registre, vrsto registrov, vodila in podobno. Procesorji tako s pošiljanjem operandov, ki jih potrebujejo, zmanjšajo pogostost pomnilniških dostopov. Medprocesna komunikacija prav tako zahteva svoj davek. Komunikacija mora potekati nadzorovano in biti v skladu z nekim komunikacijskim protokolom, da v sistemu ne bo prihajalo do napak ali korupcije v podatkih.

Cannonov algoritem[14] omogoča učinkovito množenje matrik na večprocesorskih sistemih z zmanjšanim posegom v pomnilnik in minimalnim gibanjem podatkov v sistemu. Matriki sta podobno kot pri metodi deli in vladaj razdeljeni v bloke. Vsak procesor v polju zmnoži dva bloka originalnih matrik na naivni način in posreduje svoje operande bližnjim procesorjem. Procesorji so v polju povezani tako, da tvorijo torus v dveh dimenzijah.

V primeru naivno porazdeljenega množenja vsak izmed $p \times p$ procesorjev dostopa do pomnilnika neodvisno od ostalih, med njimi pa ni komunikacije. Nekateri elementi originalnih matrik, kot tudi elementi rezultatne matrike, bodo iz pomnilnika prebrani in vanj zapisani večkrat. Vseh branj bo največ $n^3 + 3n^2$ – torej $O(n^3)$. V idealnem primeru dimenzije bločnih matrik, ki ju množimo, ustrezajo dimenzijam polja procesorjev. Ob tej predpostavki vzemimo za zgled množenje matrik A in B dimenzij $n \times n$ v polju procesorjev z enakimi dimenzijami. Bloki matrik so na začetku porazdeljeni med procesorje v polju, tako da si vsak procesor lasti blok matrike A in blok matrike B. Procesor v i-ti vrstici in j-tem stolpcu, pri čemer velja $0 < i, j \leq n$, prejme blok v presečišču vrstice i in stolpca $(j - 1) \bmod n$ matrike A in blok v vrstici $(i - 1) \bmod n$ in stolpcu j matrike B. Celotno množenje se izvede v n korakih. V vsakem koraku se s permutacijo obstoječih operandov, tj. blokov matrik A in B, v procesorju zmnoži nov par operandov, rezultat operacije pa se prišteje k drugim delnim rezultatom. Procesor v vrstici i in stolpcu j si v koraku k lasti bloka $A(i, (i + j + k) \bmod n)$ in $B((i + j + k) \bmod n, j)$. Slika 4.3 prikazuje operande, ki si jih, v vsakem izmed 4 korakov matričnega množenja dveh matrik, lastijo procesne enote v polju razsežnosti 4×4.

V polju s $p \times p$ procesorji se med matričnim množenjem s Cannonovim algoritmom med procesorji prenese $\frac{2n^2}{p}$ elementov – skalarjev ali podmatrik. Z upoštevanjem začetnih prenosov operandov iz pomnilnika in pisanja rezultatov vanj, skupno $\frac{2n^2}{p} + 3n^2$. Pod predpostavko, da je zahtevnost komunikacije $O(1)$ in so zahtevnosti prenosa podatka iz pomnilnika, prenosa podatka med procesorji ter operacije v plavajoči vejici α, β in γ, je cena Cannonovega

A_{00} B_{00}	A_{01} B_{11}	A_{02} B_{22}	A_{03} B_{33}
A_{11} B_{10}	A_{12} B_{21}	A_{13} B_{32}	A_{10} B_{03}
A_{22} B_{20}	A_{23} B_{31}	A_{20} B_{02}	A_{21} B_{13}
A_{33} B_{30}	A_{30} B_{01}	A_{31} B_{12}	A_{32} B_{23}

A_{01} B_{10}	A_{02} B_{21}	A_{03} B_{32}	A_{00} B_{03}
A_{12} B_{20}	A_{13} B_{31}	A_{10} B_{02}	A_{11} B_{13}
A_{23} B_{30}	A_{20} B_{01}	A_{21} B_{12}	A_{22} B_{23}
A_{30} B_{00}	A_{31} B_{11}	A_{32} B_{22}	A_{33} B_{33}

A_{02} B_{20}	A_{03} B_{31}	A_{00} B_{02}	A_{01} B_{13}
A_{13} B_{30}	A_{10} B_{01}	A_{11} B_{12}	A_{12} B_{23}
A_{20} B_{00}	A_{21} B_{11}	A_{22} B_{22}	A_{23} B_{33}
A_{31} B_{10}	A_{32} B_{21}	A_{33} B_{32}	A_{30} B_{03}

A_{03} B_{30}	A_{00} B_{01}	A_{01} B_{12}	A_{02} B_{23}
A_{10} B_{00}	A_{11} B_{11}	A_{12} B_{22}	A_{13} B_{33}
A_{21} B_{10}	A_{22} B_{21}	A_{23} B_{32}	A_{20} B_{03}
A_{32} B_{20}	A_{33} B_{31}	A_{30} B_{02}	A_{31} B_{13}

Slika 4.3: Razporeditev operandov med procesorji v štirih korakih Cannonovega algoritma.

algoritma za matrično množenje

$$n\left[\frac{2n^2}{p}\gamma + 2\left(\alpha + \frac{n}{p}\beta\right)\right] \tag{4.6}$$

Časovna zahtevnost je

$$T(n,p) = \frac{2n^3}{p}\gamma + 2n\alpha + \frac{2n^2}{p}\beta, \tag{4.7}$$

pohitritev pa

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p^2}\gamma + n\alpha + \frac{n^3}{p}} = \frac{p}{\frac{1}{p} + \frac{\alpha}{2n^2\gamma} + \frac{1}{2\gamma}} \tag{4.8}$$

Učinkovitost algoritma je torej

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{\frac{1}{p} + \frac{\alpha}{2n^2\gamma} + \frac{1}{2\gamma}} \quad [9][23] \tag{4.9}$$

Cannonov algoritem predpostavlja sinhrono računanje nad podatki v korakih po končani začetni komunikaciji. Podatki oz. operandi iz pomnilnika prek robnih procesorjev polja potujejo do ostalih procesorjev v polju, dokler vsak izmed procesorjev ne prejme ustreznih operandov. Temu sledi računanje.

V idealnem primeru vsak procesor prejme svoj podatek sočasno z drugimi procesorji v enem koraku. Ker je teh podatkov veliko, natančneje p^2, je takšna komunikacija pogosto težko izvedljiva. Podatki se zato širijo od robnih procesorjev do vseh ostalih postopoma. To pomeni, da so procesorji polja v času komunikacije nedejavni, saj čakajo na stanje polja, v katerem si vsak procesor lasti ustrezne operande.

Cannonov algoritem za rabo v procesorju grafov zato prilagodimo tako, da izkoristimo čas nedejavnosti oz. prikrijemo čas postopne komunikacije. Procesne enote v polju, namesto v 2D torus, s komunikacijskimi kanali povežemo v mrežo. To prikazuje slika 4.4. Podatki še vedno potujejo od robnih procesnih enot do vseh drugih enot v polju, vendar vsaka procesna enota začne z računanjem takoj, ko sta na voljo operanda obeh procesnih enot, ki enoti pošiljata podatke. Robne procesne enote prejemajo operande prek vrst

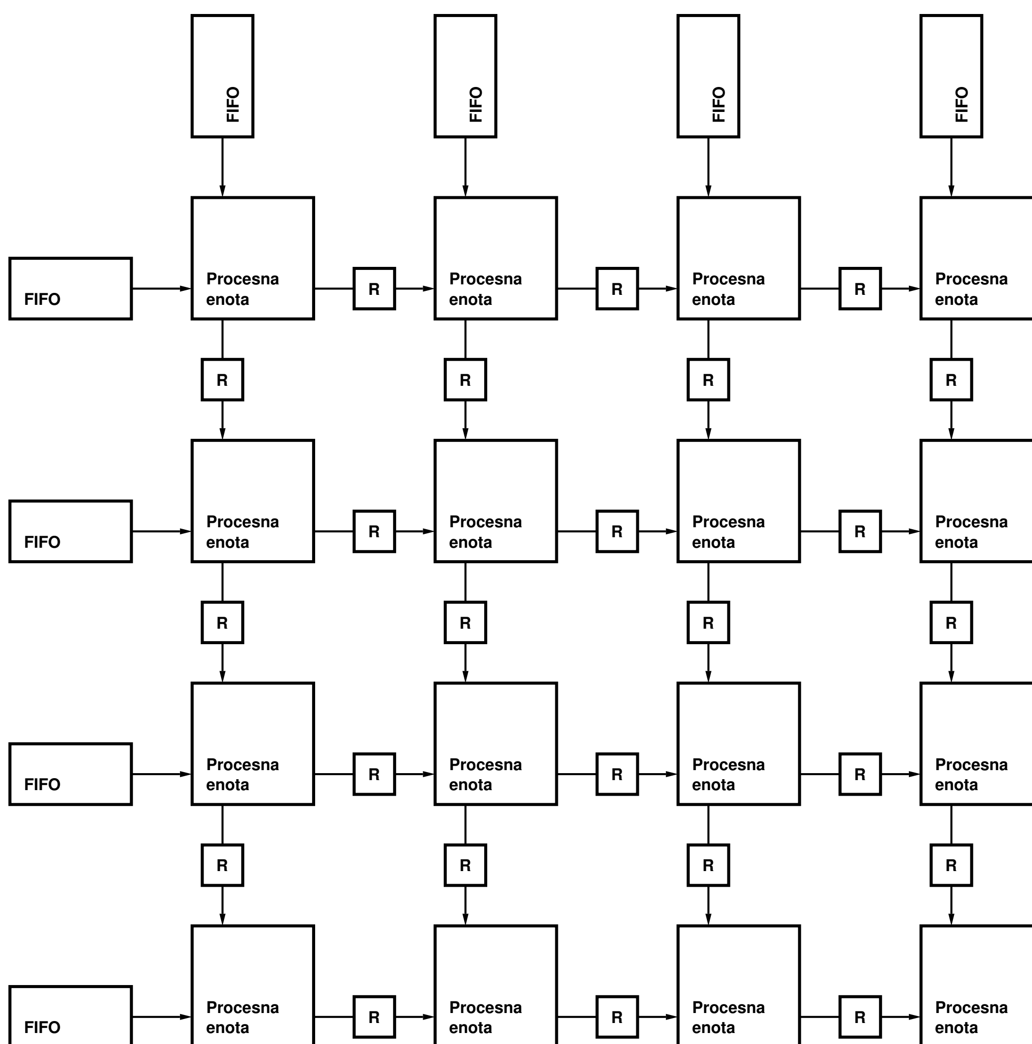
FIFO, kamor jih vstavlja kontrolna enota polja. Komunikacija med enotami poteka prek registrov velikosti enega operanda. Vrste FIFO, registri in procesne enote skupaj tvorijo cevovod. V polju procesnih enot dimenzij $p \times p$ se matrika enake razsežnosti, ob idealni razpoložljivosti operandov, torej vse vrste FIFO vsebujejo operande, zmnoži v $2p - 1$ korakih.

4.10 Lokacija operandov in načini naslavljanja

Operandi so količine, na katerih se naredijo računske operacije. V procesorju grafov so operandi 32-bitna cela in realna števila ter matrike, ki sestojijo iz več 32-bitnih operandov. Enota za transformacije grafa izvaja operacije le nad enim ali dvema 32-bitnima operandoma. Torej enim ali dvema celima ali realnima številoma. Polje procesnih enot, ki je izključno namenjeno aritmetičnim in logičnim operacijam nad matrikami, izvaja eno operacijo nad enim ali več 32-bitnimi operandi hkrati. Tem razlikam so primerni tudi ukazi posameznih komponent procesorja grafov.

V splošnem se operandi v računalniku nahajajo na enem izmed treh mest. Majhen in hiter pomnilnik, ki je običajno del procesne enote in ki je sestavljen iz množice programsko dostopnih registrov, je eno izmed takšnih mest. Poleg množice registrov, so operandi lahko tudi v eni ali več sosednjih pomnilniških besedah. V računalnikih z večnivojsko pomnilniško hierarhijo so lahko v različnih nivojih hkrati, saj podatki v celotni pomnilniški hierarhiji spadajo v isti naslovni prostor. Nazadnje lahko računalnik operande hrani v enem od registrov krmilnika vhodno-izhodne naprave ali drugih komponent računalniškega sistema. Za delo z operandi v teh registrih arhitektura računalnika pogosto namenja posebne ukaze in načine naslavljanja. Izjemoma se za delo s temi operandi uporabljajo pomnilniški ukazi, če so registri naprave preslikani v isti pomnilniški prostor, kar imenujemo pomnilniško preslikan vhod/izhod. Operandi v teh napravah se z uporabniškega vidika ne razlikujejo s pomnilniškimi [19].

Operandi se v procesorju grafov nahajajo izključno v pomnilniku oz. po-



Slika 4.4: Shema polja procesnih enot. Enote v polju s pošiljanjem operandov sodelujejo pri matričnem množenju. Podatke prejemajo in pošiljajo prek registrov in vrst FIFO.

mnilniški hierarhiji. Polje procesnih enot in enota za transformacije grafa si namreč ne lastita množice registrov, pač pa operande neposredno bereta in pišeta v predpomnilnik na najnižjem nivoju – lokalni predpomnilnik. Zaradi hitrosti lokalnega predpomnilnika, kar gre v večji meri pripisati cevovodu in ekskluzivni pripadnosti pomnilnika le enemu gospodarju, je interakcija z njim podobna tisti z registrskim blokom. Ta v tipični računalniški arhitekturi izstavlja in zapisuje operande enako hitro, kot jih centralna procesna enota bere in piše, vendar obsega veliko manj prostora, kot ga običajno zaseda predpomnilnik na najnižjem nivoju. Strojni jezik procesorja grafov zato ne predpisuje posebnih ukazov za delo s pomnilniškimi operandi. Do operandov v pomnilniku lahko dostopa vsak ukaz.

Ker so operandi v procesorju grafov izključno pomnilniški in ker procesor grafov ne vsebuje registrov, ki bi jih potrebovali za posredno ali katerokoli drugo od registrov odvisno naslavljanje, je mogoče operande v ukazih nasloviti le na dva načina.

Najpreprosteje operand v ukazu podamo kar z njegovo vrednostjo. Tako podanemu operandu pravimo takojšnji operand. Operand je del ukaza in se v polje procesnih enot ali enoto za transformacije grafa prenese skupaj z ukazom. Branje pomnilnika ob izvajanju ukaza s takšnim operandom torej ni potrebno. Tak način naslavljanja imenujemo takojšnje naslavljanje.

Operande lahko v ukazu podamo tudi z njihovimi naslovi v pomnilniku, kar imenujemo neposredno ali tudi absolutno naslavljanje. Pri prevzemu ukaza se z dostopom do lokalnega predpomnilnika, in ob zgrešitvi tudi do višjih nivojev pomnilniške hierarhije, iz pomnilnika prebere vsak neposredno naslovljen operand. Neposredno naslavljanje omogoča dostop do poljubnega pomnilniškega operanda. Prostost oz. splošnost dostopa pa se odraža v težavah, ki jih povzroča neposredno naslavljanje. Dolžina naslova v pomnilniškem prostoru je sorazmerna s kapaciteto glavnega pomnilnika. Ker so operandi ukaza v celoti podani z absolutnimi pomnilniškimi naslovi, bodo ukazi na računalnikih z velikim naslovnim prostorom dolgi. Težave nastanejo tudi, če želimo v obstoječi računalniški arhitekturi povečati naslovljiv

pomnilniški prostor.

4.11 Strojni jezik

Strojni jezik polja procesnih enot in enote za transformacije grafa je sovisen z njunima arhitekturama – tj. arhitektura močno vpliva na pomen, število in formate ukazov strojnega jezika, ki vzvratno vpliva na velik del arhitekturne zasnove.

Vselej dobra in nadvse pomembna lastnost arhitekture vsakega računalnika je ortogonalnost njenih sestavin, tj. množice operacij, množice podatkovnih tipov oz. vrst operandov ter načinov naslavljanja. Dve sestavini sta ortogonalni, kadar sta neodvisni oz. se njune prvine, ki določajo ukaze, medsebojno ne izključujejo. Na primer množica operacij in množica vrst operandov sta ortogonalni, če lahko operacijo izvedemo nad vsemi vrstami operandov v množici. Informacija o operaciji je torej neodvisna od informacije o operandih ukaza. Dobra ortogonalnost računalniške arhitekture pomeni lažje programiranje, saj je izjem v pravilih za podajanje strojnih ukazov manj kot sicer [19, 16].

Arhitektura procesorja grafov je le deloma ortogonalna. Način podajanja operandov je za ukaze polja procesnih enot in enote za transformacije grafa različen. Prav tako so za operacije, ki jih izvaja enota za transformacije grafa, veljavni le osnovni tipi operandov, tj. 32-bitno celo ali realno število. Popolnoma ortogonalni so z vrstami operacij in podatkovnimi tipi le načini naslavljanja. Neodvisno od ostalih informacij lahko v ukazu podamo takojšnje operande ali naslovimo pomnilniške operande neposredno.

Dolžina neposredno naslovljenih in takojšnjih operandov je privzeto 32 bitov. Format ukaza je različno določen glede na to, ali ukaz zadeva polje procesnih enot ali enoto za transformacije grafa in glede na vrsto operacije, ki naj se z ukazom izvede. V vsakem primeru sestoji posamezni ukaz procesorja grafov iz 2-bitnega določila ciljne enote za izvedbo, vsaj enega 4-bitnega polja za izbiro funkcije v aritmetično-logični enoti, enega polja 32 bitov, ki

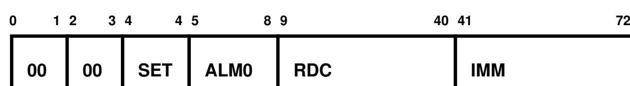
naslavlja ponorno ali izvorno pomnilniško besedo – operanda, in enega ali več sekundarnih operandov. Sekundarni operandi so, odvisno od formata ukaza, pomnilniški naslovi ali številske konstante.

Prvi štirje biti imajo, ne glede na vrsto ali format ukaza, isti pomen, zato jih imenujemo tudi glava ukaza. Vrsto enote, v katero dodeljevalnik odpremi ukaz, določata prvi in drugi bit. Bita, ki tem bitom sledita, določata format ukaza, ki je specifičen ciljni enoti. Glavi ukaza sledi več bitov, ki, odvisno od informacij o cilju in formatu ukaza, pomenijo različna določila o stanju enote ter izvajanju ukaza.

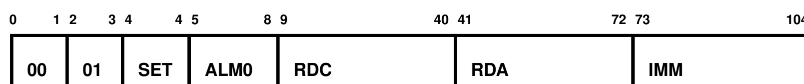
4.11.1 Ukazi enote za transformacije grafa

Ukaz enote za transformacije grafa poleg glave vsebuje zaporedoma še enobitno polje, ki predpisuje vrsto posega v pomnilniški prostor, torej ali je končni rezultat operacije branje pomnilnika ali pisanje vanj. Bitu sledijo polje, ki določa funkcijo v aritmetično-logični enoti, 32-bitni pomnilniški naslov in en ali dva operanda.

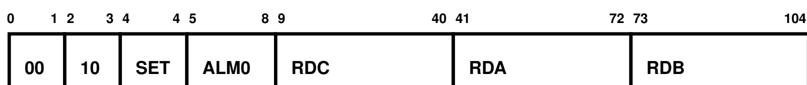
Format ETG-A, odvisno od polja `SET`, pomeni enočleno aritmetično-logično operacijo nad takojšnjim operandom `IMM` ali branje pomnilniške besede na naslovu `RDC`.



Format ETG-B pomeni dvočleno aritmetično-logično operacijo nad pomnilniškim in takojšnjim opernadom `RDA` in `IMM`.



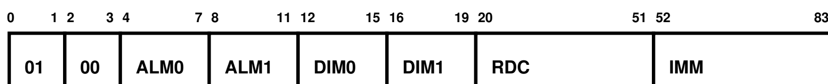
Format ETG-C pomeni dvočleno aritmetično-logično operacijo nad pomnilniškima operandoma `RDA` in `RDB`.



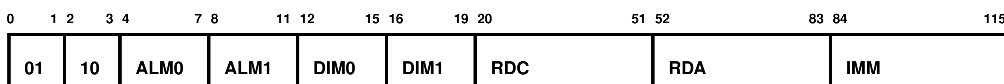
4.11.2 Ukazi polja procesnih enot

Za glavo ukaza polja procesnih enot stojita po vrsti polji, ki določata operaciji $op1$ in $op2$ matričnega množenja $A\ op1\ .\ op2\ B$ v aritmetično-logični enoti, 4-bitni polji, ki določata dimenziji matrik pri matričnem množenju, 32-bitni pomnilniški naslov in en ali dva operanda.

Format PPE-A pomeni operacijo `ALM0` nad matriko dimenzij `DIM0` × `DIM1`, na naslovu `RDC`, in skalarjem `IMM`.



Format PPE-B pomeni operacijo `ALM0` nad matriko dimenzij `DIM0` × `DIM1`, na naslovu `RDA`, in skalarjem `IMM`.



Format PPE-C pomeni matrično množenje, z operacijama `ALM0` in `ALM1`, matrik, na naslovih `RDA` in `RDB`, dimenzij `DIM0` × `DIM1` in `DIM1` × `DIM0`.

0–1	2–3	4–7	8–11	12–15	16–19	20–51	52–83	84–115
01	10	ALM0	ALM1	DIM0	DIM1	RDC	RDA	RDB

Poglavje 5

Implementacija

Poglavje obsega nekaj informacij o fizični arhitekturi procesorja grafov, navaja sredstva, s katerimi izvedemo učinkovito implementacijo in vsebuje implementacijsko specifične informacije, ki so v teoretično obarvanem poglavju o arhitekturi odveč.

5.1 FPGA

Arhitekturo, ki je predmet obravnave prejšnjega poglavja, zavoljo simulacije, verifikacije in testiranja implementiramo v programabilnem vezju FPGA.

FPGA imenujemo polje konfigurabilnih logičnih blokov, nastavljivih povezav med bloki in drugih komponent, ki v skupnem substratu iz polprevodniškega materiala tvorijo programabilno integrirano vezje. Poleg logičnih blokov vsebuje FPGA navadno tudi pomnilniške celice oz. pomnilniške bloke, vezje za upravljanje z uro in celice za digitalno procesiranje signalov.

Motivacij za implementacijo FPGA je več. Programabilnost, kot primarna odlika FPGA, omogoča fleksibilno načrtovanje strojne opreme. Iteracije v načrtovanju lahko izvedemo hitreje, kot če načrtujemo namensko specifično integrirano vezje. Hitrost načrtovanja gre delno pripisati programski opremi proizvajalca FPGA, ki namesto načrtovalca umešča načrt strojne opreme v integrirano vezje, izdela časovno analizo in opravlja druge naloge, ki

jih pri načrtovanju namensko specifičnih integriranih vezij običajno izvajamo ročno.

Konfigurabilni logični bloki vsebujejo osnovne gradnike za implementacijo kombinatorične in sekvenčne logike. Med gradniki pogosto najdemo vsaj eno tabelo LUT, eno pomnilno celico, elemente aritmetične in prenosne logike ter multiplekserje za usmerjanje signalov skozi blok.

Z nastavljanjem tabel LUT in povezovanjem logičnih blokov v skupine blokov, ki predstavljajo bolj kompleksne logične funkcije, ustvarimo vezje, ki ustreza arhitekturnemu načrtu strojne opreme. V ta namen uporabljamo orodje za sintezo proizvajalca vezja FPGA. Arhitekturo opišemo v jeziku za opis strojne opreme, ki ga orodje prevede v konfiguracijo logičnih blokov in drugih programabilnih elementov v FPGA.

5.2 Zgradba procesorja

Procesor grafov sestoji iz več procesnih enot in enot za transformacije grafa, dodeljevalnika in trinivojske pomnilniške hierarhije.

5.3 Dodeljevalnik

Dodeljevalnik v sistemu je povezan z vsemi polji procesnih enot in vsemi enotami za transformacije grafa. Zavedajoč se vsebine v predpomnilnikih, torej razpoložljivosti operandov, skuša najbolje izbrati primerno polje oz. enoto za izvedbo prevzetega ukaza. Njegovo zgradbo prikazuje slika 5.1.

Ustrezno enoto za transformacije grafa, ki naj izvede delo, izbere dodeljevalnik na podlagi operandov ukaza in zgodovine ukazov, ki so se predhodno izvajali na enotah tako, da zmanjša verjetnost zgrešitev v predpomnilnikih in s tem izboljša lokalnost sistema. Dodeljevalnik v ta namen vodi evidenco o prevzetih ukazih in enotah, v katere jih odpremlja. Idealna velikost evidence je enaka velikosti predpomnilnika L0. Predpomnilnik na nivoju L0 hrani 2^8 pomnilniških besed – skupno 4 KiB podatkov, kar je za kapaciteto evidence

vsekakor sprejemljivo, vendar je v smislu porabe strojnih virov in posledično cene procesorja grafov veliko preveč. Velikost evidence je torej manjša od predpomnilnika L0, zato se dodeljevalnik pri delitvi dela omeji na nedavno zgodovino največ c ukazov, pri čemer je c kapaciteta evidence.

Evidence o odpremljenih ukazih so v dodeljevalniku enotam pripadajoči pomnilniki, ki vsebujejo informacije o enoti izstavljenih ukazih in ki jih v sklopu tega dela imenujemo tudi zgodovinski pomnilniki. Specifično hrani vsak zgodovinski pomnilnik naslove o operandih zadnjih c ukazov, ki jih je enota izvedla ali čakajo na izvedbo – so v ukazni vrsti.

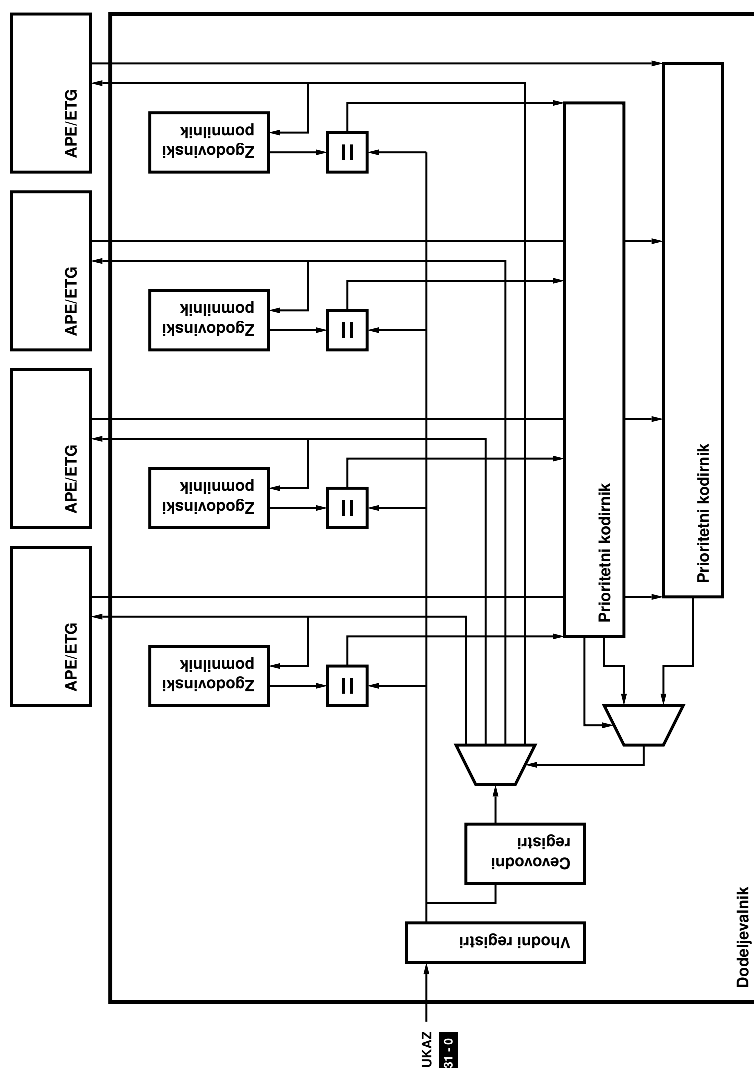
Ob prevzemu vsakega ukaza se naslovi njegovih operandov primerjajo z zapisi v zgodovinskih pomnilnikih vseh enot, ki so zmožne sprejeti oz. opravljati delo. Ukaz se dodeli enoti, za katero v zgodovinskem pomnilniku dodeljevalnik odkrije enakovrednost oz. ujemanje naslova z naslovom operanda prevzetega ukaza. Če se naslov operanda ukaza za dodelitev ne ujema z nobenim zapisom v pomnilnikih, dodeljevalnik izstavi ukaz v prosto enoto z najvišjo prioriteto. V zgodovinskem pomnilniku te enote se na mesto najstarejšega zapisa vpiše naslov operanda prevzetega ukaza. Če je enot z ujemanji več, se ukaz dodeli tisti z najvišjo prioriteto, pripadajoči zgodovinski pomnilnik pa ostane nespremenjen.

Z večanjem kapacitete zgodovinskih pomnilnikov enot se izboljša natančnost napovedi za razpoložljivost operandov v predpomnilnikih L0 posameznih enot in s tem lokalnost, poleg tega se zmanjša število konfliktov, ki nastopijo zaradi dostopa vsaj dveh enot do istih operandov v krajšem časovnem obdobju.

5.4 AXI

AXI[3] je protokol iz skupine protokolov za medsebojno povezovanje logičnih enot v vgrajenih sistemih AMBA. Protokol je v javni domeni in je standard v mnogih procesorskih arhitekturah ter drugih vgrajenih sistemih.

AXI je protokol tretje generacije protokolov AMBA in je namenjen vi-



Slika 5.1: Shema dodeljevalnika. Število sužnjev, nad katerimi gospodari, je določeno s parametrom v opisu strojne opreme procesorja. V vsakem primeru vsebuje sužnjem pripadajoče zgodovinske pomnilnike in primerjalnike, dva prioritetna kodirnika in cevovodne ter vhodne registre.

soko zmogljivim sistemom ter sistemom z visoko frekvenco ure. Predvsem je primeren za širokopasovno komunikacijo, kjer je latenca kritičnega pomena.

Protokol AXI odlikujejo številne lastnosti. Kanali za kontrolne in naslovne signale, podatkovne signale pri branju ter podatkovne signale pri pisanju so ločeni, kar omogoča fleksibilni vrstni red transakcij, paralelno obdelavo transakcij in dovoljuje, da se naslovni podatki pošljejo pred prenosom podatkov, ki tem signalom ustrezajo. Protokol podpira več zaporednih transakcij, ki berejo ali pišejo podatke na zaporednih ali fiksnih naslovih. S tem razkriva možnost združevanja transakcij, še posebej v pomnilniški hierarhiji ali v primeru arbitraže. Pri tem se na naslovno vodilo objavi le naslov prve transakcije in s signali na kontrolnih vodilih sporoči vrsta in število zaporednih transakcij.

V komunikacijo sta vedno vključeni dve entiteti – gospodar in suženj. Gospodar je odgovoren za začetek komunikacije in pošilja ukaze sužnju, s katerim izmenjuje podatke. Suženj nikoli ne začne komunikacije in tudi ne pošilja ukazov. Zgolj odziva se na zahteve gospodarja s kontrolnimi signali in/ali podatki.

Gospodar in suženj sta med seboj povezana s petimi kanali. Prek naslovnega kanala gospodar sužnju pošilja naslovne in kontrolne signale ob zahtevi po branju ali pisanju podatkov. Naslovni kanal za branje podatkov in naslovni kanal za pisanje sta ločena.

Podatki se, skupaj z drugimi informacijami vezanimi na podatke, pošiljajo po bralnem podatkovnem kanalu, če jih gospodar od sužnja zahteva in po pisalnem podatkovnem kanalu, če jih gospodar sužnju želi posredovati.

Suženj na zahtevo o pisanju podatkov prejeme podatke, ki jih gospodar želi posredovati. Ob prejemu zadnjega podatka, ki ga gospodar tako označi, suženj gospodarja obvesti o uspešnosti zahteve z ustreznimi signali v odzivnem kanalu. Odzivni kanal je namenjen izključno odgovorom na pisalne zahteve gospodarja. Po končanem branju podatkov ali tudi med branjem podatkov suženj gospodarja obvesti o uspešnosti branja preko bralnega podatkovnega kanala. Suženj lahko gospodarja obvesti o statusu posameznega

prenosa med pošiljanjem podatkov ali o statusu vseh prenosov ob koncu zadnjega prenosa.

Preden se prenos podatkov ali kontrolnih signalov lahko začne, morata gospodar in suženj potrditi razpoložljivost ter udeležbo v komunikaciji. V vseh petih kanalih protokola AXI, preko katerih teče komunikacija med gospodarjem in sužnjem, se udeleženca o pošiljanju informacij dogovorita v procesu rokovanja. Pošiljatelj oz. izvor s signalom `valid` v visokem stanju označi razpoložljivost podatkov v kanalu. Prejemnik oz. ponor se na signal `valid` odzove s signalom `ready`, prav tako v visokem stanju. S tem pošiljatelju sporoča, da lahko sprejme podatke. Prenos podatkov se izvede le v primeru, ko sta oba signala `valid` in `ready` v visokem stanju.

5.5 Pomnilnik

Pomnilniško hierarhijo v procesorju grafov tvorita dva nivoja predpomnilnikov, L0 in L1, ter glavni pomnilnik. Predpomnilnik L0 je tesno povezan s procesno enoto, ki ji pripada in predstavlja polje programsko naslovljivih registrov enote. Procesna enota se s predpomnilnikom L0 povezuje s kontrolnimi in podatkovnimi signali. Kontrolni signali označujejo pripravljenost enote za sprejem oz. pošiljanje podatkov in sporočajo informacije o podatkih ter njihovih naslovih v pomnilniku. Podatkovni signali tvorijo vodilo za komunikacijo podatkov v obe smeri. Protokol med procesno enoto in pripadajočim predpomnilnikom L0 je zaradi manjše kompleksnosti ter večje hitrosti veliko bolj preprost kot protokol med višjimi nivoji v pomnilniški hierarhiji. Procesna enota ob branju ali pisanju v pomnilnik postavi v visoko stanje enega izmed dveh kontrolnih signalov, ki označujeta pripravljenost za sprejem ali pošiljanje podatkov. Pri pisanju v pomnilnik pošlje še ustrezne podatke, v obeh primerih pošlje tudi pomnilniški naslov za branje oz. pisanje.

Predpomnilniki na nivoju višje, tj. L1, ki posredujejo med predpomnilniki L0 in glavnim pomnilnikom, so v strukturi ter delovanju podobni predpomnilnikom L0, le da poleg predpomnilniških funkcij opravljajo tudi nalogo

arbitraže in sovisnosti podatkov. S pomnilnikom L1 se namreč povezuje več predpomnilnikov L0, ki lahko izdajajo zahteve sočasno. Za dinamično določanje vrstnega reda izpolnjevanja zahtev vsebuje predpomnilnik L1 prioritetni kodirnik, ki določa prevzem naslednje zahteve v vsaki urini periodi oz. v periodah, ko je to mogoče. Več predpomnilnikov L1 se povezuje prek enotnega vodila z glavnim pomnilnikom. Arbitražo nad tem vodilom opravlja pomnilniški krmilnik. Komunikacija med predpomnilnikom L1 in glavnim pomnilnikom poteka v skladu s protokolom AXI.

5.6 Predpomnilnik L1

Predpomnilnik na nivoju L1 je transakcijski. V komunikaciji z glavnim pomnilnikom zavzame vlogo gospodarja, v komunikaciji s predpomnilniki na nivoju nižje, tj. L0, pa vlogo sužnja. Nad vsakim predpomnilnikom L1 v sistemu lahko gospodari več predpomnilnikov L0. Predpomnilnik L1 med njihovimi zahtevami izbira s prioritetnim kodirnikom. Pri tem upošteva prioriteto vrstnega reda, kot ga narekujejo gospodarji s povezavami na ustreznih mestih in ki se krožno ponavlja, če hkrati transakcije zahteva več gospodarjev.

Konflikti med transakcijami se odkrivajo v pomnilniških besedah, ki jih transakcije pišejo. Vsaka beseda v pomnilniku, ki predstavlja kvadratno podmatriko razsežnosti $n \times n$, pri čemer je $n \geq 1$, je sestavljena iz n^2 elementov, ki jih je znotraj transakcije mogoče neodvisno spreminjati. Konflikt nastane v primeru, ko se dve ali več pisalnih operacij znotraj različnih transakcij nanaša na isti element neke podmatrike.

Ob vsakem branju besede iz predpomnilnika oz. bralni transakciji se poleg besede prenese informacija o trenutni različici besede. Ob pisanju besede nazaj v pomnilnik se njena različica primerja z obstoječo v predpomnilniku L1. Če se različici obstoječe in nove besede razlikujeta, se elementi podmatrike, ki jo predstavlja nova beseda in ki se od elementov iste, a prebrane različice podmatrike razlikujejo, primerjajo z istoležnimi elementi podmatrike obsto-

ječe besede. Če vsaj en element nove podmatrike sovpada z novejšo različico istoležnega elementa podmatrike, se šteje pisalna transakcija za konfliktno in posledično neuspešno. Predpomnilnik o uspešnosti transakcije obvesti gospodarja z ustreznimi signali.

Predpomnilnik L1 oz. njegovo delovanje je razdeljeno v tri cevovodne stopnje. Razlog za to je boljša prepustnost predpomnilnika in večja frekvenca ure, predvsem pa cevovod omogoča možnost dinamičnega razvrščanja pomnilniških transakcij oz. dinamično določanje njihovega reda.

Z dinamičnim razvrščanjem transakcij zmanjšamo čas nedejavnosti procesnih enot, ki zaradi zgrešitve v predpomnilniku čakajo na podatke oz. operande. Pri delu z grafi pogosto ni pomembno, v kakšnem vrstnem redu izvedemo transformacije nad grafom. Pomembno je le, da se transakcije med sabo ne prekrivajo oz. med njimi ni interferenc. Transakcije lahko izvajamo v drugačnem vrstnem redu od tistega, ki ga določa uporabniški program. Tiste, za katere so operandi na voljo v predpomnilniku na najnižjem nivoju, lahko tako izvedemo pred tistimi, ki zaradi nerazpoložljivosti operandov upočasnjujejo delovanje sistema.

V prvi cevovodni stopnji se na podlagi naslova opredeli naslov v predpomnilniku, značka in odmik znotraj besede. Prav tako se iz pomnilnika značk v predpomnilniku prebere značka na predhodno opredeljenem naslovu. V drugi cevovodni stopnji se primerjata prebrana značka in značka naslova transakcije. Če se ujemata, se v tretji cevovodni stopnji podatki preberejo iz podatkovnega pomnilnika, ki jih predpomnilnik posreduje predpomnilniku na nivoju nižje (gospodarju). V nasprotnem primeru pošlje predpomnilnik zahtevo po branju manjkajočih podatkov v glavni pomnilnik. Izvajanje cevovodnih stopenj se zaustavlja do časa, ko podatki prispejo iz glavnega pomnilnika do predpomnilnika L1. Ker je predpomnilnikov L1, gospodarjev nad glavnim pomnilnikom, torej sužnjem, v sistemu lahko več, lahko cevovod stoji dlje časa. Zato prav z namenom, da izkoristimo prosti čas, transakcije izvajamo v dinamično določenem vrstnem redu, ki upošteva predvsem razpoložljivost operandov v predpomnilnikih.

5.7 Enota za transformacije grafa

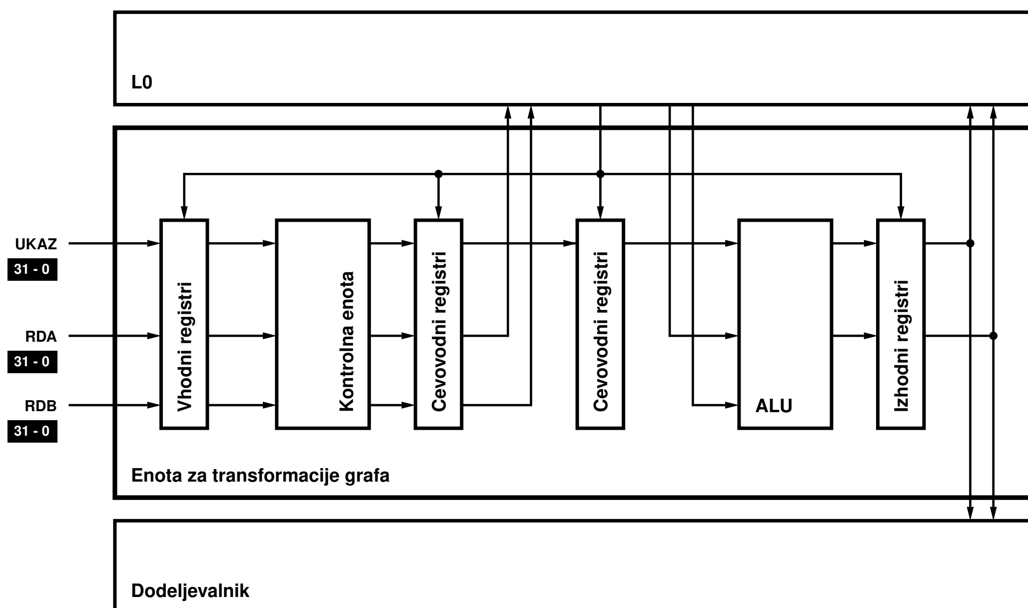
Cevovod je, kot v drugih komponentah sistema, prisoten tudi v enoti za transformacije grafa. Poleg registrov 5-stopenjskega cevovoda vsebuje enota za transformacije grafa kontrolno enoto, ki na podlagi ukaza in operandov nastavlja ustrezne signale ter aritmetično-logično enoto za delo z operandi. Shemo enote prikazuje slika 5.2.

V prvi stopnji cevovoda se v kontrolno enoto prek vhodnih registrov prevzamejo ukaz in operandi. Na podlagi slednjih podatkov v vhodnih registrih kontrolna enota določi ustrezne izhodne signale, ki prek cevovodnih registrov potujejo v vse naslednje cevovodne stopnje. V stopnji, ki sledi, se v predpomnilnik L0 skupaj z naslovi pošlje zahteva po branju podatkov. Podatki so na voljo v naslednji cevovodni stopnji, če predpomnilnik L0 že pomni njihovo vrednost. V primeru zgrešitve se napredovanje cevovoda enote za transformacijo grafa zakasni za število urinih period, ki jih potrebuje predpomnilnik L0 za pridobitev podatkov z višjih nivojev pomnilniške hierarhije. Signal, s katerim predpomnilnik L0 sporoča uspešnost zahteve, določa dejavnost oz. nedejavnost cevovoda. Naslednja cevovodna stopnja vsebuje aritmetično-logično enoto za izvajanje tovrstnih operacij nad operandi, pridobljenimi iz predpomnilnika L0. Rezultat ALE se v zadnji stopnji, glede na kontrolne signale, ki so skozi cevovod prispeli iz kontrolne enote v drugi stopnji, zapiše v predpomnilnik L0 ali posreduje dodeljevalniku.

5.8 Polje procesnih enot

Vsaka procesna enota je sinhrona z drugimi enotami v istem polju. Računanje in komunikacija potekata sočasno v sinhronih korakih. Robne procesne enote v mreži prejemajo podatke oz. operande prek vrste FIFO, kamor jih pošilja kontrolna enota procesorskega polja. Ukaze prejmejo vse procesne enote v polju sočasno, operandi pa v korakih potujejo od robnih procesnih enot do ostalih enot v mreži. Slika 5.3 prikazuje primer zgradbe polja velikosti 3×3.

n^2 procesnih enot v polju je z vrstami FIFO povezanih v mrežo dimenzije

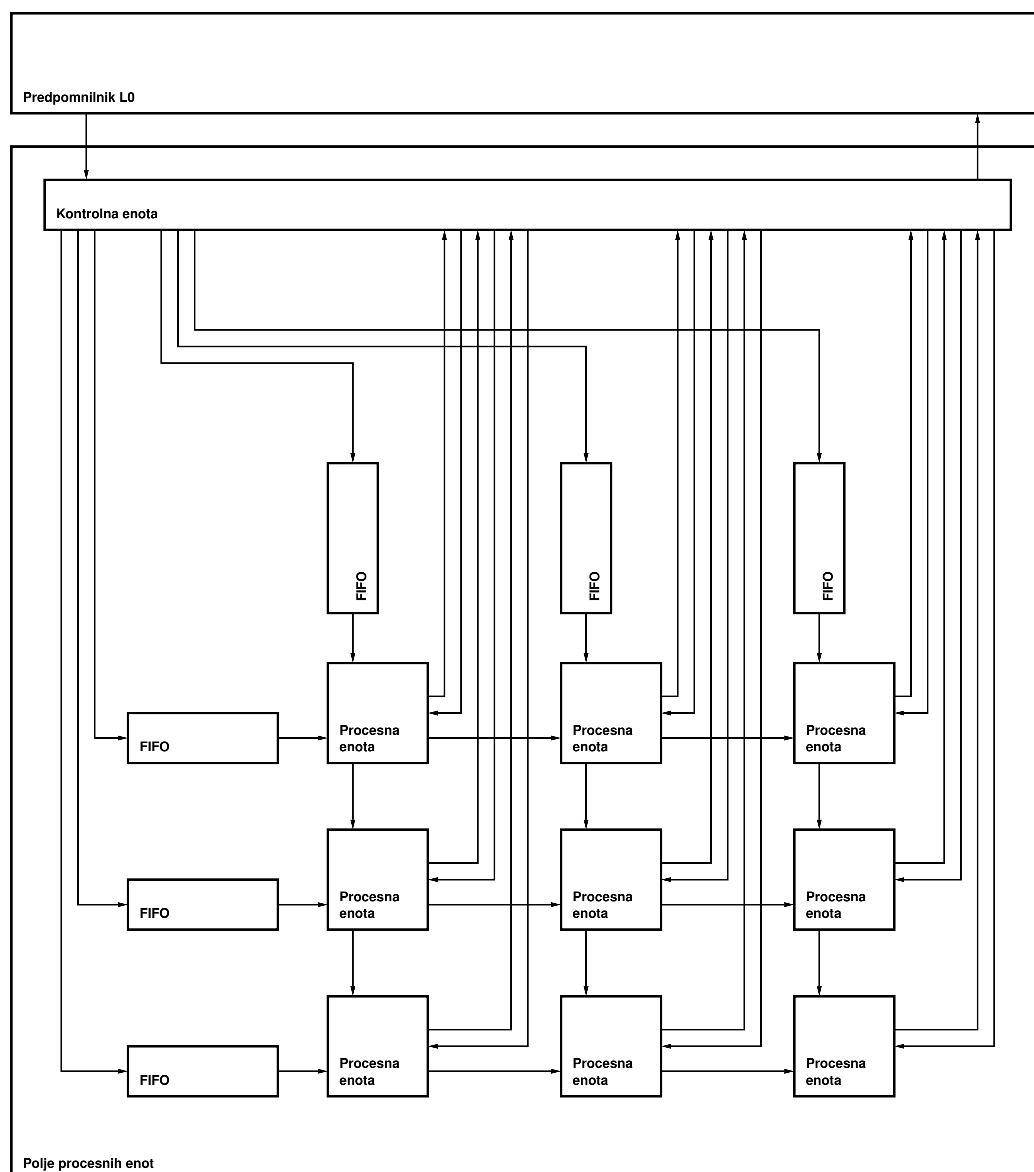


Slika 5.2: Shema enote za transformacije grafa.

$n \times n$. Vsako izmed procesnih enot v polju sestavlja aritmetično-logična enota, kontrolna enota in par registrov. Za doseganje večje prepustnosti je vsaka procesna enota razdeljena na 5 cevovodnih stopenj, od tega se v 4 cevovodnih stopnjah izvajajo aritmetične in logične operacije.

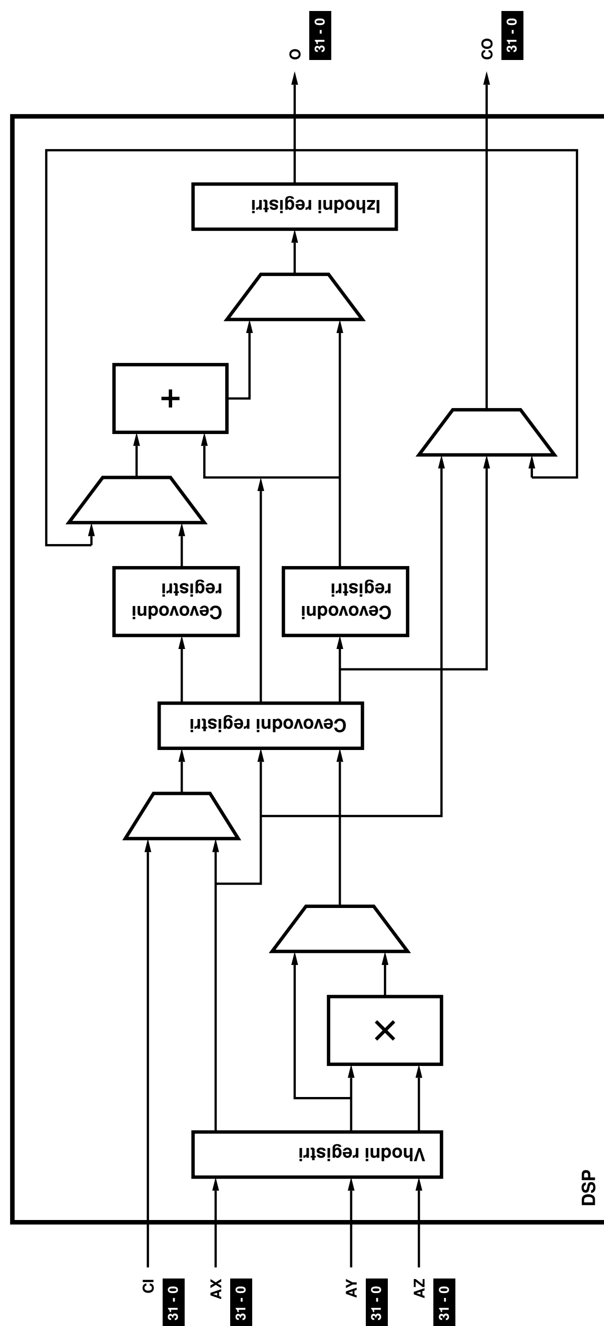
5.8.1 Aritmetično-logična enota

Aritmetično-logična enota znotraj procesne enote vsebuje aritmetične funkcijske enote za seštevanje, odštevanje in množenje ter logične funkcijske enote za negacijo, konjunkcijo, disjunkcijo in primerjavo. Množenje, ki je najzahtevnejša izmed operacij, se zaradi boljše podatkovne in ukazne prepustnosti izvede v več urinih periodah cevovodno. V vezju FPGA se za implementacijo množilnika, namesto običajnih konfigurabilnih logičnih blokov, uporabljajo bloki za digitalno procesiranje signalov. Bloki DSP, različno od konfigurabilnih logičnih blokov, opravljajo le aritmetične funkcije. Zaradi slednje ekskluzivnosti so aritmetične funkcije v blokih DSP veliko bolj energijsko



Slika 5.3: Shema polja procesnih enot razsežnosti 3×3. Procesne enote prejemajo podatke od kontrolne enote prek vrst FIFO. Druga z drugo so povezane z registri.

učinkovite, hkrati omogočajo krajšo urino periodo oz. višjo frekvenco ure. Enote, ki tipično tvorijo en blok DSP, so seštevalnik, odštevalnik, akumulator, množilnik in registri za hrambo koeficientov. Z ustreznim povezovanjem enot in več blokov DSP je mogoče realizirati poljubno aritmetično funkcijo. Vsak blok DSP se z drugimi bloki povezuje prek kaskadnega vodila, ki je ločen od drugih vodil in poti v vezju. Namensko vodilo omogoča večjo hitrost in natančnost aritmetičnih operacij s povezovanjem blokov DSP v večje funkcijske enote [1]. Slika 5.4 prikazuje shemo bloka DSP v vezju FPGA Altera Arria 10 [17].



Slika 5.4: Shema bloka DSP v vezju FPGA Altera Arria 10.

Poglavje 6

Zaključek

Rezultat dela je računalniška arhitektura, ki učinkovito procesira podatke v grafu oz. šibko strukturirane in nestrukturirane podatke v splošnem. Izraba načel in spoznanj algebraične formalizacije grafa ter njegovih transformacij je privedla do paralelnega računalnika z dobro lokalnostjo pomnilniških dostopov in posplošenim pristopom k reševanju problemov v povezavi z grafi s pomočjo linearne algebre.

V delu predstavljena arhitektura zadostuje reševanju tipičnih, vendar osnovnih problemov v grafih, kot so iskanje najkrajše poti ali odkrivanje močno povezanih komponent grafa. Zaradi splošnosti arhitekture se sposobnost procesorja grafov za reševanje bolj zahtevnih oz. domensko specifičnih problemov predpostavlja, a delo njihove implementacije v celoti izpušča. Nadaljevanje dela, poleg reševanja večjih problemov, potencialno obsega tudi obravnavo vertikalnega raztezanja sistema z dodajanjem strojnih virov – na primer povečanje oz. dodajanje elementov v procesorska polja ali povečanje pomnilnika – kot tudi horizontalnega raztezanja s povezovanjem več instanc procesorja grafov v širokopasovno omrežje. Poleg naštetega delo v veliki meri dopušča prosto interpretacijo tudi o uporabniškem vmesniku oz. interakciji med uporabnikom in sistemom. Za arhitekturo delo sicer opredeljuje strojni jezik, vendar podrobne specifikacije o interakciji, poleg vodila AXI, preko katerega se procesor povezuje z ostalimi napravami, ne obsega. Način inte-

rakcije s procesorjem grafa je tako prepuščen implementaciji in je odvisen od namena procesorja oz. njegove umestitve v večji sistem.

Literatura

[1] Altera Corporation. *Implementing Multipliers in FPGA Devices*, 3.0 edition, July 2004.

[2] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[3] ARM. *AMBA® AXI™ and ACE™ Protocol Specification*, 3.0 edition, October 2011.

[4] Vladimir Batagelj. Semirings for social networks analysis. *Journal of Mathematical Sociology*, 19(1):53–68, 1994.

[5] Vladimir Batagelj and Anuška Ferligoj. Analiza omrežij. Prosojnice s predavanj, 2006.

[6] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj. A framework for fpga acceleration of large graph problems: Graphlet counting case study. In *2011 International Conference on Field-Programmable Technology*, pages 1–8, Dec 2011.

[7] John Adrian Bondy. *Graph Theory With Applications*. Elsevier Science Ltd., Oxford, UK, UK, 1976.

[8] Peter Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, pages 117–121, New York, NY, USA, 1997. ACM.

[9] Jaeyoung Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia '97*, pages 224–229, Apr 1997.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[11] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[12] Roger Espasa and Mateo Valero. Exploiting instruction-and data-level parallelism. *IEEE micro*, 17(5):20–27, 1997.

[13] Gašper Fijavž. *Diskretne strukture.* Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2015. Elektronski vir.

[14] H. Gupta and P. Sadayappan. Communication efficient matrix-multiplication on hypercubes. Technical Report 1994-25, Stanford Infolab, 1994.

[15] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[17] Intel. *Intel® Arria® 10 Native Fixed Point DSP IP Core User Guide*, March 2017.

[18] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[19] D. Kodek. *Arhitektura in organizacija računalniških sistemov.* Bi-tim, 2008.

[20] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, May 2014.

[21] Selena Praprotnik and Vladimir Batagelj. Semirings for temporal network analysis. *CoRR*, abs/1603.08261, 2016.

[22] William S. Song, Vitaliy Gleyzer, Alexei Lomakin, and Jeremy Kepner. Novel graph processor architecture, prototype system, and results. *CoRR*, abs/1607.06541, 2016.

[23] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.

[24] Peter C. Verhoef, Edwin Kooge, and Natasha Walk. *Creating Value with Big Data Analytics: Making Smarter Marketing Decisions.* Routledge, New York, NY, 10001, 2016.

[25] Boštjan Vilfan. *Osnovni algoritmi.* Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2002.

[26] Riste Škrekovski. *Diskretne strukture II.* 2010. Elektronski vir.