SINHRONIZIRANA PODATKOVNA PRETOKOVNA RAČUNALNIŠKA ARHITEKTURA (I)

INFORMATICA 2/92

Keywords: parallel processing, dataflow computing, scheduling, allocator, performance evaluation parallel computer architecture.

Jurij Šilc
Laboratorij za računalniške arhitekture
Institut Jožef Stefan, Ljubljana

Delo obravnava problem časovne optimizacije asinhronega procesiranja na omejenem številu procesorjev. Predlagamo izvirno rešitev, ki temelji na uvedbi nekaterih mehanizmov sinhronizacije v asinhrono računanje. Graf pretoka podatkov, ki opisuje asinhrono procesiranje, opremimo s časovno optimalno sprožitveno funkcijo, ki služi tako pri vlaganju grafa v računalnik, kakor tudi pri njegovem časovno optimalnejšem izvrševanju. V ta namen smo razvili hevristična algoritma `pOptSinh` in `TOptSinh` za konstrukcijo optimalnih sprožitvenih funkcij, ki po pesimistični oceni vračata optimalno rešitev v 80% primerov. Nadalje predlagamo algoritma za dodeljevanje `MinGlDol` in `MinGlGor`, ki temeljita na optimizaciji medprocesorskih komunikacij. V primerjavi z znanimi razvrščevalnimi algoritmi dobimo s predlaganima algoritmoma boljše rezultate, kar potrjujejo analizirani primeri algoritmov za izračun hitre Fourierjeve transformacije, dinamične analize scene in LU razcepa matrike. Končno podajamo tudi zasnovo hibridne vzporedne arhitekture računalnika, ki podpira predlagano preoblikovanje asinhronega računanja.

SYNCHRONOUS DATAFLOW COMPUTER ARCHITECTURE - We discuss the problem of time optimization of asynchronous processing on a limited number of processors. We present an original solution to the problem based on introduction of synchronization mechanisms into asynchronous processing. The dataflow graph describing asynchronous processing is associated with the corresponding time-optimal firing function. This function is used both for loading a dataflow graph into the computer and for time-optimal graph execution. In order to do this, we have developed two heuristic algorithms, `pOptSinh` and `TOptSinh`, which are used for optimal firing function construction. According to conservative estimates, these algorithms return optimal functions with 80% probability. Furthermore, we propose two scheduling algorithms, `MinGlDol` and `MinGlGor`, which are based on interprocessor communication minimization. These two algorithms give better results compared to some other well known scheduling algorithms. This fact is illustrated in Fast Fourier Transformation, Dynamic Scene Analysis, and LU matrix decomposition algorithms. Finally, we present a design of hybrid parallel computer architecture capable of supporting modified asynchronous computing.

1. Uvod

Asinhrono (podatkovno vodeno) procesiranje [1, 2, 3, 4, 5, 6, 7] najlaže predstavimo z grafom pretoka podatkov [8], katerega točke ponazarjajo operacije oz. ukaze, usmerjene povezave pa so nosilke vrednosti operandov oz. podatkov. Ko v neko točko stečejo podatki po vseh njenih vhodnih povezavah, postane točka izvršljiva in prične takoj z izvrševanjem pridruženega ukaza. Pravimo, da se je točka sprožila. Ko točka konča izvrševanje ukaza, pošlje rezultat v vse svoje izhodne povezave. Graf pretoka podatkov je "strojni" jezik računalnikov, ki podpirajo podatkovno vodeno procesiranje in je rezultat prevajanja visokega programskega jezika z enkratno prireditvijo [9, 10]. Čeprav segajo osnovne ideje, na katerih temeljijo takšni računalniki, v pozna šestdeseta leta, so šele s pojavom VLSI tehnologije postale te ideje tudi uresničljive [11, 12]. V osemdesetih letih so se porodili v svetu številni projekti, katerih cilj je realizacija podatkovno pretokovnega računalnika [13, 14, 15, 16, 17, 18].

Intuitivno si lahko zamislimo implementacijo asinhronega procesiranja tako, da je vsaka točka podprta s svojim procesorjem in vsaka povezava s komunikacijskim kanalom. Izkaže se, da ni nujno vsaki točki prirediti lasten procesor že vnaprej, saj je v danem trenutku izvršljivih le nekaj točk. Takšen pristop se je uveljavil v krožni arhitekturi, kjer se dodeli točki procesor šele tedaj, ko

postane slednja izvršljiva. Ker se v veliki večini algoritmov vsebovana vzporednost dinamično spreminja, ni smiselno uporabiti toliko procesorjev, kot jih zahteva maksimalna vsebovana vzporednost, saj bi bila njihova izkoriščenost nizka. Po drugi strani pa lahko povzroči pomanjkanje procesorjev občutno upočasnitev procesiranja, ker prihaja do ti. nasičenj, ko izvršljive točke čakajo v vrsti na sprostitev procesorjev. Zaradi nedeterminističnega vstopanja so točke v vrsti neurejene glede na svojo pomembnost: npr. kritična točka (tj. točka, ki bi se morala takoj izvršiti) ne more prehiteti ostalih točk, ki se nahajajo v vrsti pred njo. Nedeterministično vstopanje v vrsto v splošnem podaljša čas izvrševanja grafa pretoka podatkov.

Predpostavimo, da je na voljo podatkovno pretokovni računalnik s potencialno neskončnim številom procesorjev ter izberimo poljuben graf pretoka podatkov. Denimo, da je za najhitrejšo asinhrono izvršitev izbranega grafa potrebnih vsaj m procesorjev. Tedaj označimo s T_m najkrajši čas, v katerem se ta graf izvrši na m procesorjih. Na računalniku z $n < m$ procesorjev pa bi se v splošnem isti graf asinhrono izvršil v času $T_n = T_m + \Delta T_n$, kjer $\Delta T_n \geq 0$. Za nalogo si zadajmo minimizirati podaljšek izvrševanja ΔT_n, tj. časovno optimizirati asihrono procesiranje pri n danih procesorjih.

2. Vzporedno procesiranje in arhitekture

Cilj vzporednega procesiranja in pripadajočih računalniških arhitektur [19, 20] je povečati "moč" računalnikov, tj. povečati njihovo zmogljivost in hitrost procesiranja ter odpraviti ozka grla v procesu računanja, ki so posledica *von Neumannove* arhitekture. Zaporedni računalniki so neučinkoviti pri reševanju nalog, ki se porajajo na področjih umetne inteligence (razpoznavanje slik in govora, ekspertni sistemi, avtomatsko dokazovanje itd.) in pri zahtevnih numeričnih izračunih (simulacija, modeliranje, grafika itd.). Zato je v zadnjih letih poseben poudarek na raziskavah novih načinov vzporednega procesiranja in programiranja [4, 21, 22] ter na vzporednih računalniških arhitekturah [19, 23, 24, 25, 26].

Eden obetavnejših načinov vzporednega procesiranja izvira iz modela *podatkovno vodenega računanja* [6], ki ga podpirajo pripadajoče *podatkovno pretokovne* arhitekture [3, 27, 28]. Ker bo podatkovno vodeno računanje predmet nadaljnje obravnave, določimo najprej mesto pripadajočih podatkovno pretokovnih arhitektur v množici računalniških arhitektur.

2.1. Razvrstitev arhitektur

Razvrščanja se lahko lotimo z zelo različnih zornih kotov. Tako predlaga *Feng* [29] razvrščanje na osnovi stopnje vzporednosti ter *Händler* [30] razvrščanje na podlagi stopnje vzporednosti in cevljenja[1]. Najpogosteje se uporablja *Flynnova* [31] razvrstitev računalniških arhitektur, ki temelji na ukaznih in podatkovnih tokovih . Arhitekture razvršča v štiri skupine: en ukazni in podatkovni tok (SISD), en ukazni in več podatkovnih tokov (SIMD), več ukaznih in en podatkovni tok (MISD) ter več ukaznih in podatkovnih tokov (MIMD). *Skillicorn* [32] je predlagal razširjeno Flynnovo razvrstitev. Na osnovi števila ukaznih in podatkovnih procesorjev, ukaznih in podatkovnih pomnilnikov ter načina njihovega povezovanja je razdelil računalniške arhitekture v 28 razredov. V razredih od 1 do 5 so podatkovno pretokovne in redukcijske arhitekture, v razredu 6 so zaporedne von Neumannove arhitekture, razredi od 7 do 10 obsegajo procesorska polja, razreda 11 in 12 sta MISD arhitekture in končno v razrede od 13 do 28 razvrsti večprocesorske arhitekture. *Dasgupta* [33] je odpravil nekatere pomankljivosti v Skillicornovi razvrstitvi in predlagal hierarhični sistem razvrstitve arhitektur. *Treleaven* [4] razvršča računalniške arhitekture glede na: organizacijo računanja, programsko organizacijo in organizacijo stroja. V slednji razvrstitvi je mesto podatkovno pretokovnih arhitektur zelo jasno opredeljeno, zato bomo v nadaljevanju to razvrstitev v grobem tudi predstavili.

2.1.1 Organizacija računanja

Faze računanja. Računanje je proces, ki ga sestavlja zaporedno ponavljanje treh faz, in sicer: *izbiranje*, *preverjanje* in *izvrševanje*.

Izbiranje: Določi se množica ukazov programa, ki so možni kandidati za izvrševanje. V splošnem izbira ukaza še ne pomeni njegove takojšnje

[1] Cevljenje iz angl. "pipelining"

izvršitve. Pravilo, po katerem se ukazi izbirajo, imenujemo *pravilo izbiranja*. Poznamo tri pravila izbiranja: *brezpogojno* (ukaz se izbere ne glede na njegov položaj v programu), *notranje* (izberejo se najglobje gnezdeni ukazi izraza) in *zunanje* (izbirajo se samo negnezdeni ukazi).

Preverjanje: Ugotavlja se izvršljivost ukaza, kar pomeni, da se preverja prisotnost vseh potrebnih vrednosti operandov v izbranih ukazih. Pravilo za preverjanje imenujemo *pravilo izvršitve*. Izvršljivi ukazi preidejo v tretjo fazo računanja. Če ukaz ni izvršljiv, se bodisi odloži ter se izbere kasneje ali pa se zahtevajo manjkajoče vrednosti operandov.

Izvrševanje: Izvršljivi ukazi se dejansko izvrše.

Vodenje računanja. Ločimo tri osnovne načine vodenja računanja: *krmilno* vodeno računaje, *z zahtevo* vodeno računanje in *podatkovno* vodeno računanje.

Krmilno vodenje: Izbiranju ukazov, ki je brezpogojno, sledi izvrševanje ukaza. Torej se pri krmilnem vodenju ne izvaja preverjanja ukazov.

Vodenje z zahtevo: Ukaz se izbere (navadno po pravilu zunanjega izbiranja) tedaj, ko prej izbrani ukaz zahteva njegov rezultat. Nato se ugotovi, če so izbrani ukazi izvršljivi, če so torej znane vrednosti njihovih operandov; če vrednost operanda še ni znana, se posreduje zahteva tistemu ukazu, ki ga more izračunati. Izvršijo se vsi tisti ukazi, katerih potrebni operandi so znani.

Podatkovno vodenje: Izbiranje poteka tako, da se vsakemu ukazu dodeli procesor. Preverjanje sestoji iz sprotnega ugotavljanja vrednosti vseh vhodnih operandov, sledi mu izvrševanje vseh preverjenih ukazov.

2.1.2 Programska organizacija

Krmilni mehanizem. Krmilni mehanizem določa vpliv izvršitve nekega ukaza na izvrševanje ostalih ukazov ter je bodisi *zaporedni* (ukazi se izbirajo drug za drugim in se po vrsti izvršujejo), *rekurzivni* (posreduje se zahtevo po operandih ter sproži izvrševanje tistega ukaza, katerega rezultat je bil zahtevan) ali *sočasni* (ugotavlja se prisotnost vhodnih operandov ter sproži izvrševanje tistih ukazov, ki imajo na voljo vse potrebne operande).

Podatkovni mehanizem. Podatkovni mehanizem določa način, po katerem si ukazi delijo skupne operande. Ukazi si lahko delijo skupne operande s pomočjo: *vstavljene vrednosti* (če je vrednost operanda znana že pred pričetkom računanja, se takoj vpiše v vse ukaze, ki jo potrebujejo), *sprotne vrednosti* (kopije operanda, ki se je izračunal, se posredujejo vsem ukazom, ki jim je ta operand skupen) in *reference* (vsi ukazi vsebujejo tudi referenco na skupen operand).

Modeli računanja. Vodenja računanja realiziramo z različnimi *modeli* računanja. V vsakem modelu se zrcali izvirni način vodenja računanja v obliki *krmilnega* ter *podatkovnega* mehanizma. Opredelitev različnih modelov računanja glede na podatkovni in krmilni mehanizem prikazuje slika 1.

Krmilno vodenje: Realizirati ga je mogoče z naslednjimi modeli: z zaporednim krmilnim tokom, s sočasni krmilnim tokom in s tokom krmilnih paketov.

Vodenje z zahtevo: Uporabljata se dva modela, in sicer: redukcija izrazov in redukcija grafov.

Podatkovno vodenje: Pripada mu le model računanja s tokom podatkovnih paketov.

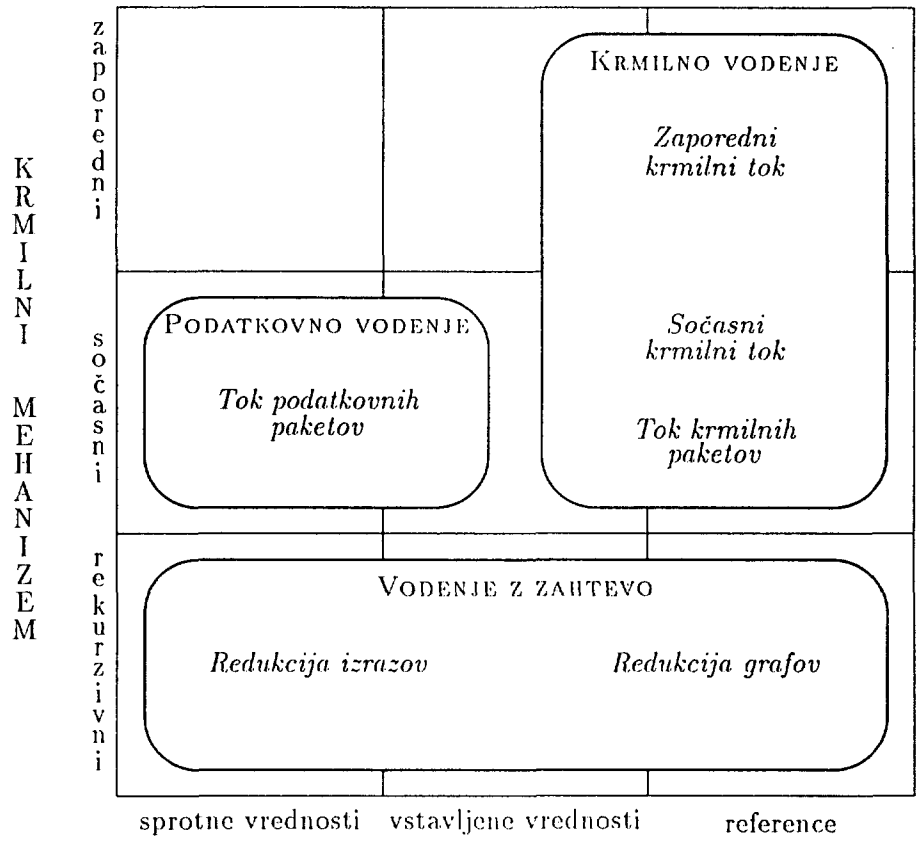
2.1.3 Organizacija stroja

Pod izrazom "organizacija stroja" razumemo način sestavljanja in medsebojnega povezovanja osnovnih računalniških materialnih virov: procesorjev, pomnilnikov in komunikacijskih enot. Ločimo tri osnovne organizacije.

Centralizirana organizacija. Sestavljajo jo procesor, pomnilnik in vodilo med njima. To je tradicionalna von Neumannova arhitektura.

Organizacija za obravnavo izrazov. Osnovni gradniki, ki so sestavljeni iz procesorja, pomnilnika in komunikacijske enote, so povezani v pravilno strukturo.

Organizacija za posredovanje paketov. Procesna enota (več procesorjev), pomnilniška enota in komunikacijska enota so preko čakalnih vrst krožno povezane. Izvrševanje programa poteka cevljeno v obliki enosmernega pretoka paketov skozi omenjene enote, ki delujejo sočasno. Vrste opravljajo nalogo začasnega skladiščenja paketov: paketi, ki so pripravljeni za obdelavo v eni od enot, čakajo v ustrezni vrsti, dokler jih ta enota ni pripravljena sprejeti.



Slika 1: Modeli računanja.

2.2 Podatkovno pretokovne arhitekture

Podatkovno pretokovni računalniki [2] se bistveno razlikujejo od krmilno vodenih računalnikov. Prvič: njihova struktura je zasnovana tako, da poteka procesiranje na osnovi *vrednosti*, ne pa naslovov spremenljivk. In drugič: v podatkovno pretokovnih računalnikih ni ničesar, kar bi bilo podobno programskemu števcu, saj postanejo operacije izvršljive takoj, ko so na voljo vsi potrebni vhodni operandi. Torej temeljijo ti računalniki na načelih *asinhronosti* in *funkcionalnosti* [34].

2.2.1 Podatkovno vodeno računanje

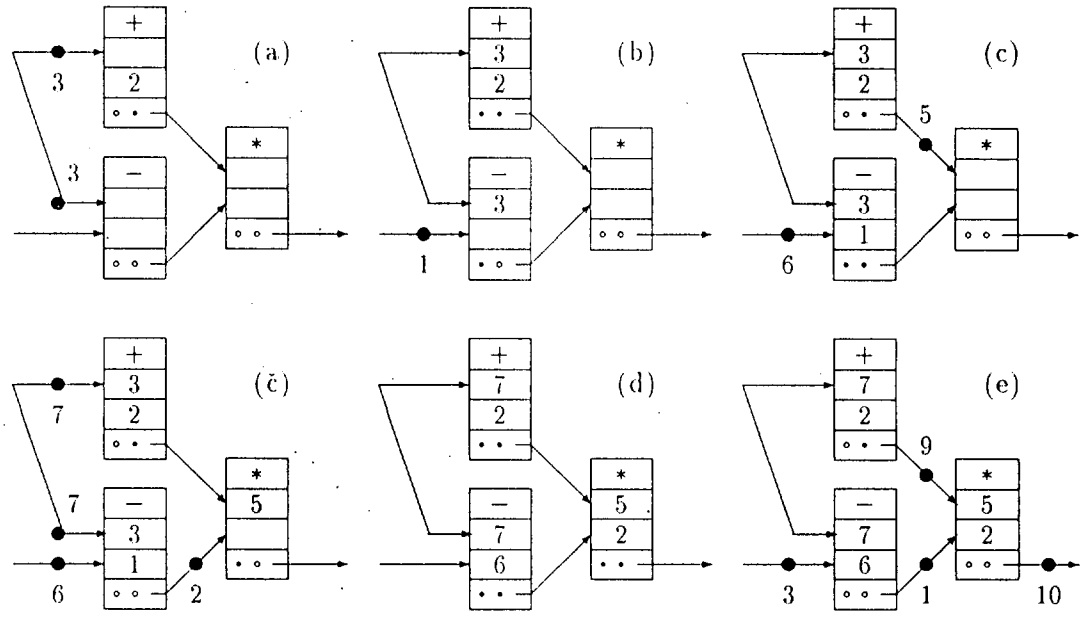
Model, s katerim realiziramo podatkovno vodeno računanje, je tok podatkovnih paketov. Opredeljujeta ga sočasni krmilni mehanizem ter podatkovni mehanizem s sprotnimi in vstavljenimi vrednostmi. Računanje se odvija s pomočjo podatkovnih paketov[2]. Ukazi sestojijo iz operacij, operandov (vstavljene ali sprotne vrednosti) in kazalcev na neposredne naslednike. Ukaz se prične izvrševati takoj, ko mu je zadnji od neposrednih predhodnikov poslal podatkovni paket.

Podatkovno vodeno računanje si ilustrirajmo s primerom $z = (x + 2) * (x - y)$, $x = 3,7\ldots$ in $y = 1,6,3\ldots$, ki je prikazan na sliki 2.

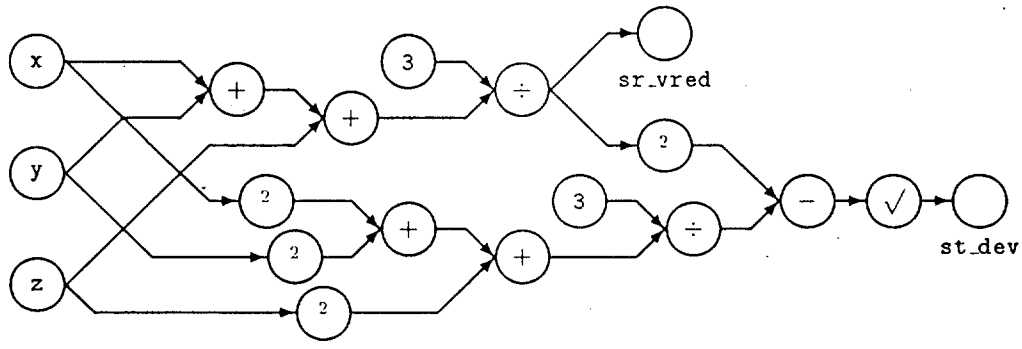
2.2.2 Graf pretoka podatkov

Strojni jezik podatkovno pretokovnih računalnikov je *graf pretoka podatkov* (*GPP*) [8], ki je posebna oblika *Petrijeve mreže* [35]. Točke v *GPP* ponazarjajo operacije, usmerjene povezave pa so nosilke podatkovnih paketov (vrednosti operandov). *GPP* je rezultat prevajanja visokega *podatkovno pretokovnega jezika* (VAL, ID, LUCID, SISAL, VALID itd.) [9]. Osnovni jezik strojnega nivoja v podatkovno pretokovnih arhitekturah je torej *GPP*. Izvrševanja programa si zatorej predočimo s pretokom podatkov v *GPP*. Ko v neko točko stečejo

[2]Uporablja se tudi izraz žeton (iz angl. "token").



Slika 2: Podatkovno vodeno računanje $z = (x + 2) * (x - y)$.



Slika 3: *GPP* funkcije `Statistika`.

podatki po vseh njenih vhodnih povezavah, postane točka *izvršljiva* in prične takoj z izvrševanjem pridružene operacije. Pravimo, da se je točka *sprožila*. Ko točka izvrševanje konča, pošlje podatkovne pakete (rezultat) v vse svoje izhodne povezave. Samoumevno je, da stoji za vsako točko *procesni element* in za vsako povezavo *komunikacijski kanal*.

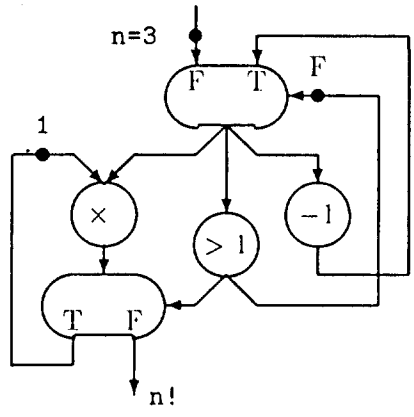
Za ilustracijo vzemimo še fukcijo `Statistika`, ki vrne srednjo vrednost `sr_vred` in standardni odklon `st_dev` vhodnih spremenljivk `x`, `y` in `z`. Zapis funkcije v visokem podatkovno pretokovnem jeziku VAL [10] je naslednji:

```
function Statistika (
         x,y,z: real
         returns real, real);
   let
      sr_vred := (x + y + z)/3;
      st_dev := SQRT((x**2 + y**2 +
                z**2)/3 - sr_vred**2);
   in
      sr_vred, st_dev
   endlet
endfun
```

V primeru izračuna funkcije `Statistika` smo dobili *aciklični GPP*. Tedaj ko v algoritmih nastopajo ponovitve (*iteracije*), pa se srečujemo s *cikličnimi GPP*. Vzemimimo npr. izračun fakto-

riele števila n. Ustrezen program za izračun n! bi se zapisal v visokem podatkovno pretokovnem jeziku ID kot (*GPP* prikazuje slika 4):

```
( initial j <- n; k <- 1
  while j > 1 do
     new j <- j - 1; new k <- k * j;
  return k )
```



Slika 4: *GPP* za izračun n!.

Če primerjamo *GPP* za izračun funkcije Statistika (slika 3) z *GPP* za izračun n! (slika 4), opazimo, da se v slednjem poleg zank pojavljajo tudi *pogojni konstrukti* ter da se po nekaterih povezavah pretakajo tudi podatki, ki predstavljajo logični vrednosti T in F. V splošnem so povezave nosilke različnih tipov podatkov: *integer, real, boolean* itd..

Pri cikličnih *GPP* pogosto nastanejo dodatne težave. Kadar v posamezni ponovitvi zanke naletimo na podatkovne odvisnosti med operacijami, to ne ustavi nadaljnjih ponovitev, čeprav predhodna ponovitev ni v celoti končana. Tako se v določenih povezavah podatki prekrivajo. Vzemimo za primer naslednji segment programa:

```
input a
      a[0] = 1
      b[0] = 1
for i from 1 to n
   begin
      a[i] = i + a[i-1]
      b[i] = a[i] * b[i-1]
   end
output b
```

Predpostavimo, da je za seštevanje potrebna 1 časovna enota in za množenje 2 časovni enoti. Pričakovani rezultat bi bil: b[0] = 1, b[1] = 2, b[2] = 8, b[3] = 56 itd. Dejansko pa bi dobili naslednje rezultate: b[0] = 1, b[1] = 2, b[2] = 14, b[3] = 308 itd. Oglejmo si nekaj prvih korakov izvajanja programa. Zaradi hitrejšega izvrševanja zanke pri seštevanju se nekateri elementi vektorja a[i] preprosto prekrijejo. Tako se a[2], ki je bil v trenutku $t = 2$ še 4, prekrije z vrednostjo 7, še preden se uporabi kot faktor pri izračunu b[2].

2.2.3 Statični in dinamični modeli računanja

Iz zadnjega primera vidimo, da ni mogoče s prisotnostjo kateregakoli podatka razglasiti točke za izvršljivo, saj v splošnem pripadajo vhodni podatki različnim delom izračuna. Obstaja nekaj rešitev nastalega problema oz. modelov podatkovno vodenega računanja, ki vplivajo tudi na izvedbo podatkovno pretokovnega računalnika.

Model 1: Vsaka ponovitev zanke se sme pričeti šele tedaj, ko se je predhodna končala. Ta model [1-1] ne dovoljuje vzporednosti med ponovitvami in zahteva dodatno ugotavljanje konca ponovitve.

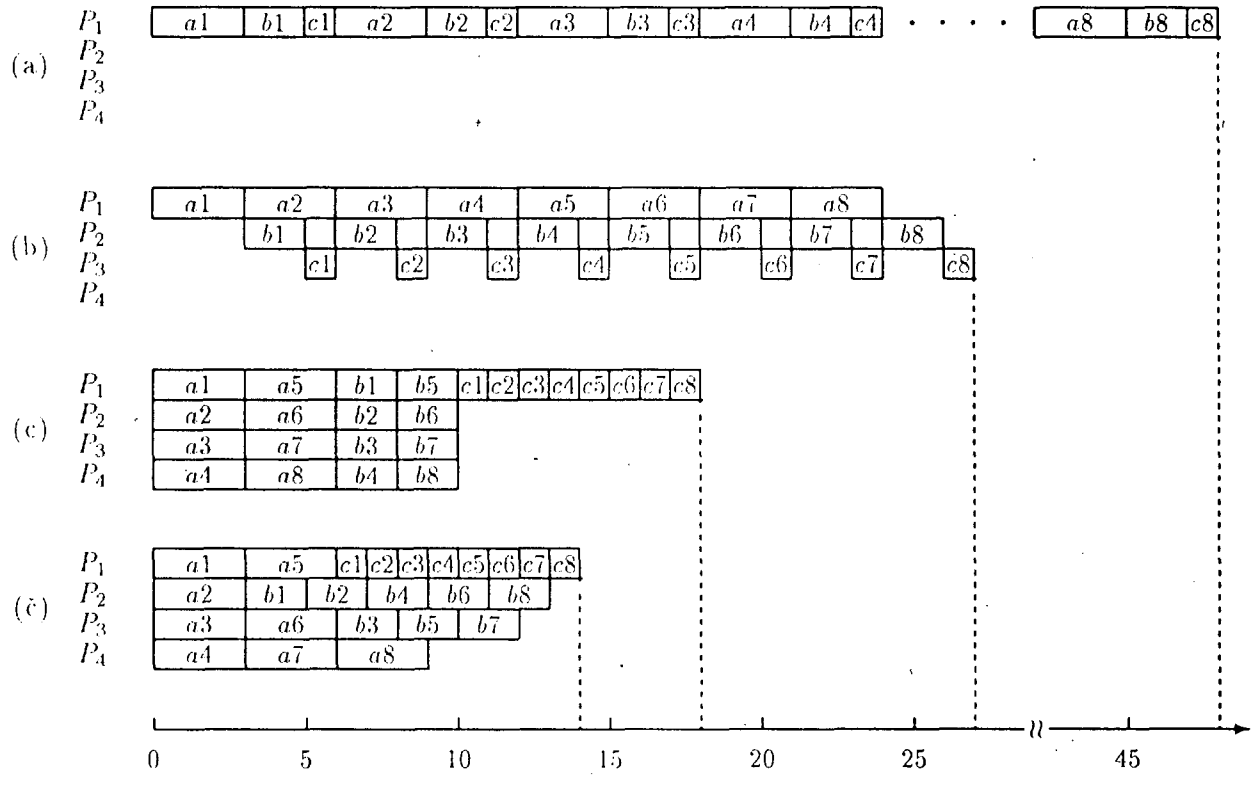
Model 2: V vsaki povezavi *GPP* dovoljujemo istočasno le en podatkovni paket [1]. To pomeni, da se sme točka sproži le tedaj, ko so prisotni podatkovni paketi v vseh vhodnih povezavah in ni v izhodnih povezavah nobenega podatkovnega paketa. Takšno pravilo izvršitve uresničimo s pomočjo *potrditvenih signalov*. Prednost tega modela je, da omogoča cevljenje, slabost pa, da se število povezav podvoji.

Model 3: Ciklični *GPP* transformiramo tako, da vsako ponovitev opišemo z acikličnim podgrafom. Takšen model zahteva večje programske pomnilnike in sprotno generiranje koda v primeru, ko je izstop iz zanke izračunan šele v času njenega izvajanja. Ti zahtevi se odražata kot resna pomankljivost v praktičnih sistemih.

Model 4: Podatkovne pakete opremimo z dodatnimi oznakami[3] [15, 36], ki vsebujejo indeks oz. nivo ponovitve. Točka se sme sproži le tedaj, ko imajo vsi vhodni podatkovni paketi enako oznako. Ta model zahteva povečan pretok po *GPP* ter mehanizme za primerjanje oznak.

Model 5: Povezave v *GPP* opravljajo tudi funkcijo vrste [37], kar pomeni, da so v njih podatkovni paketi ravrščeni v istem vrstnem redu, kot

[3] V angl. literaturi imenujejo tak paket "tagged token".



Slika 5: Primerjava modelov podatkovno vodenega računanja.

so vstopali v povezave[4]. Takšen model omogoča enako stopnjo vzporednosti kot model z označenimi paketi, le da je izvedba čakalnih vrst cenovno zelo zahtevna.

Primerjavo navedenih petih modelov podatkovno vodenega računanja si oglejmo na naslednjem testnem programu [5]:

```
input d, e, f
      c[0] = 0
for i from 1 to 8
   begin
      a[i] = d[i] / e[i]
      b[i] = a[i] * f[i]
      c[i] = b[i] + c[i-1]
   end
output a, b, c
```

Predpostavimo, da so potrebne za deljenje tri, za množenje dve in za seštevanje ena časovna enota. Zamislimo si podatkovno pretokovni računalnik s štirimi procesorji P_1, P_2, P_3 in P_4, ki izvajajo katerokoli od potrebnih operacij. Idealizirajmo računalnik tako, da ne bo zakasnitev v pomnilniku in pri medprocesorskih komunikacijah. Nadalje definirajmo *pospešitev* računanja S_p in *izkoriščenost* procesorjev E_p, s katerima bomo merili kvaliteto računalnika. Če je T_1 čas potreben za zaporedno izvrševanje programa in T_p čas izvrševanja istega programa na p procesorjih, potem je

$$S_p = \frac{T_1}{T_p} \text{ in } E_p = \frac{S_p}{p}.$$

Za naš testni program bi dobili $T_1 = 48$. V tabeli 1 so povzeti rezultati primerjave posameznih modelov podatkovno vodenega računanja. Podrobnejša analiza je prikazana tudi na sliki 5.

Model 1: Tak model nas pripelje do zaporednega izvajanja programa, saj se naslednja ponovitev sme pričeti šele, ko se predhodna konča. Za takšno izvrševanje zadošča en procesor (slika 5a).

Model 2: Ker dovoljujemo v povezavah istočasno le en podatkovni paket, nas to privede do cevljenja prireditvenih stavkov znotraj zanke. Za takšno izvrševanje bi zadoščali že trije procesorji (slika 5b).

[4] Povezave so FIFO vrste.

$p = 4$	T_p	S_p	E_p
Model 1	48	1	.250
Model 2	27	1.778	.444
Modeli 3,4,5	14	3.429	.857

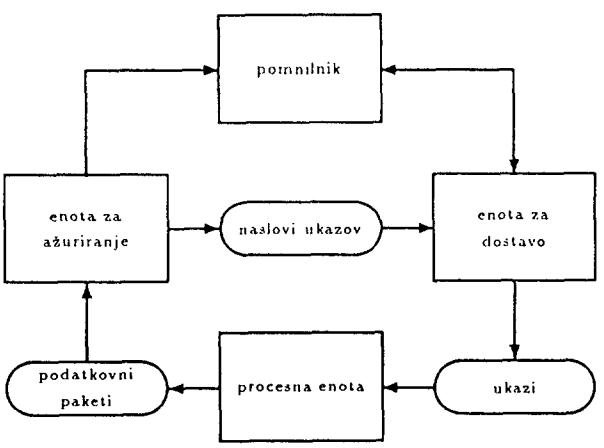
Tabela 1: Primerjava modelov podatkovno vodenega računanja.

Modeli 3, 4, 5: Najboljše rezultate dobimo pri statičnem računanju z razvitjem zank ter pri obeh dinamičnih modelih (slika 5č).

Zaradi primerjave je na sliki 5c prikazan tudi potek računanja na vektorskih računalnikih [19]. Med prevajanjem testnega programa bi se prva dva prireditvena stavka v zanki vektorizirala, tako da bi dobili $T_p = 18, S_p = 2.667$ in $E_p = 0.667$.

2.2.4 Modeli računalnikov

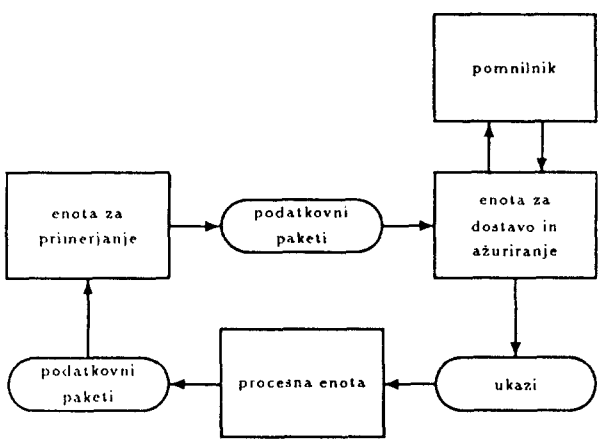
Z razvojem VLSI tehnologije so postale uresničljive mnoge ideje v računalniških arhitekturah non-von Neumannovega tipa. Tako je bilo v zgodnjih osemdesetih letih mogoče realizirati tudi prve podatkovno pretokovne računalnike. Dober pregled in primerjavo zgodnjih podatkovno pretokovnih arhitektur je podal *Srini* v delu [27]. Nekatere novejše arhitekture pa so predstavljene v [38, 28, 39, 40, 7]. Podatkovno pretokovno računanje (statično in dimanično) običajno uresničimo na računalnikih, ki temeljijo na organizacijah za posredovanje paketov. Oglejmo si arhitekturi, ki sta značilni za statični in dinamični model podatkovno vodenega računanja.



Slika 6: Statična arhitektura.

Statična arhitektura. V statični arhitekturi (slika 6) je uporabljen sinhronizacijski mehanizem, ki temelji na *shranjevanju paketov*. Podatkovni paketi se shranjujejo v ukaze, ki se izvrše, ko prejmejo vse vhodne operande. Ukazi se nahajajo v *pomnilniku* in vsebujejo: operacijo, prostor za vhodne operande in kazalce na neposredne naslednike. Enota za *ažuriranje* hrani naslov ukaza ter števec manjkajočih vhodnih operandov (podatkovnih paketov). Podatkovni paket, ki zapusti *procesno* enoto, vsebuje poleg rezultata tudi naslov ukaza, ki mu je namenjen. Enota za ažuriranje posreduje rezultat ustreznim ukazom v pomnilniški enoti ter zmanjša števec manjkajočih vhodnih operandov za ena. V primeru, ko pade števec na nič, posreduje naslov ukaza enoti za *dostavo*. Leta pošlje naslovljeni ukaz iz pomnilnika v *procesno* enoto.

Značilni prestavniki statičnih arhitektur so: SDFM[5] (Tehnološki institut Massachusetts, ZDA) [1], LAU[6] večprocesorski sistem (CERT -- Toulouse, Francija) [14], DDP[7] (Texas Instruments, ZDA) [13], μPD7281[8] (NEC, Japonska) [11] itd..



Slika 7: Dinamična arhitektura.

Dinamična arhitektura. Za razliko od statične arhitekture je v dinamični arhitekturi (slika 7) uporabljen sinhronizacijski mehanizem, ki temelji na *primerjanju paketov*. Ustrezno označeni podatkovni paketi, ki zapustijo *procesno* enoto, se zbirajo v množicah v enoti za *primerjanje*. Množice so opremljene z naslovi pripadajočih ukazov v *pom-*

[5] Static Dataflow Machine.

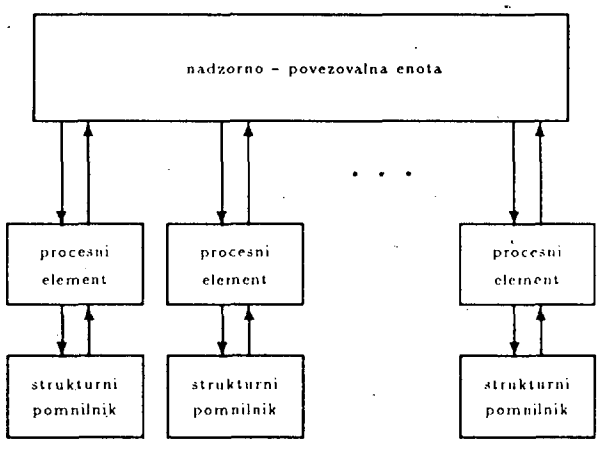
[6] "Language á assignation unique" pomeni jezik z enkratno prireditvijo.

[7] Distributed Data Processor.

[8] Prvi VLSI podatkovno pretokovni mikroprocesor.

nilniku. Enota za primerjanje razporeja prispele pakete v množice glede na njihove oznake. Ko se množica napolni, se vsi paketi prenesejo v enoto za *dostavo in ažuriranje.* Le-ta združi prispele pakete s pripadajočim ukazom iz pomnilnika ter jih posreduje procesni enoti.

Med dinamične arhitekture uvrščamo: TTDFM[9] (Tehnološki institut Massachusetts, ZDA) [41], MDFC[10](Univerza v Manchestru, Velika Britanija) [16], PIM-D[11] (ICOT, Japonska) [42], PATTSY[12] (Univerza v Queenslandu, Avstralija) [43] itd..



Slika 8: Hibridna arhitektura.

Hibridna arhitektura. Poleg omenjenih statičnih in dinamičnih arhitektur se predvsem v zadnjem času pojavljajo mnoge izvedbe arhitektur, ki jih ni mogoče uvrstiti v omenjeni dve skupini. Za te arhitekture je značilna organizacija, ki jo prikazuje slika 8. *GPP* se porazdeli med *strukturne pomnilnike* ob posameznih *procesnih elementih.* Procesni elementi izvršujejo pripadajoče dele *GPP* ter preko *nadzorno - povezovalne* enote izmenjujejo podatkovne pakete, pripadajoče delom *GPP*, ki se nahajajo v drugih strukturnih pomnilnikih. Izvrševanje *GPP* poteka bodisi statično bodisi dinamično.

Nekateri novejši računalniki iz tega razreda so: HDFM[13] (Hughes Aircraft Co., ZDA) [17], Sigma-1[14] (Electrotechnical Laboratory, Japonska) [18], EPSILON-2[15] (Sandia National Laboratories, ZDA) [44], Monsoon[16] (Tehnološki institut Massachusetts, ZDA) [45] itd..

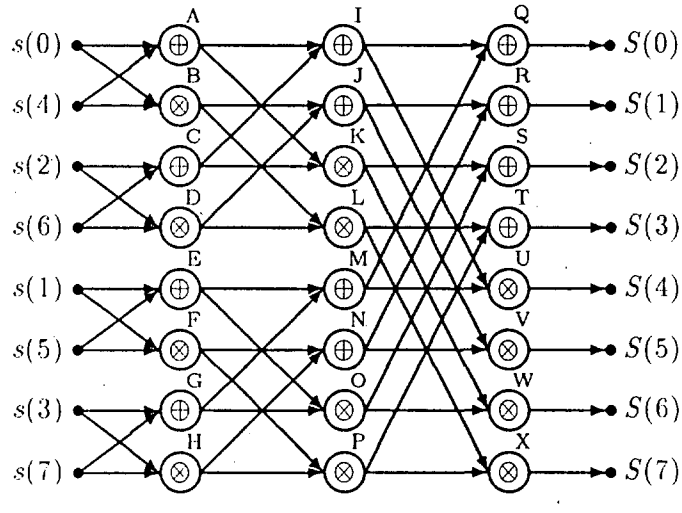
[9]Tagged-Token Dataflow Machine
[10]Manchester Dataflow Computer.
[11]Parallel Inference Machine.
[12]Processor Array Tagged-Token System.
[13]Hughes Dataflow Multiprocessor (statična arhitektura).
[14]Dinamična arhitektura.
[15]Hibridno krmilno-podatkovno vodeno računanje.

3. Motivacija in opis problema

Maloprej smo videli, da se tako v statičnih kot v dinamičnih arhitekturah izvršljivi podatkovni paketi (ukazi) preko vrste dodeljujejo procesorjem. Žal pa ima procesna enota omejeno število procesorjev, zato se v primeru, ko so vsi procesorji zasedeni, paketi kopičijo v vrsti pred procesno enoto, kjer čakajo na svojo izvršitev. Posledica takšnega čakanja je lahko občutno upočasnjeno izvrševanja *GPP*. Da se temu izognemo, je potrebno bodisi povečati število procesorjev v procesni enoti ali pa zagotoviti ustrezen način vstopanja paketov v čakalno vrsto (ali izstopanja iz vrste) pred procesno enoto in ustrezeno dodeljevanje paketov prostim procesorjem.

3.1 Primer računanja FFT

Problem upočasnjenega izvrševanja *GPP*, ki nastane zaradi pomanjkanja procesorjev, osvetlimo s primerom izračuna hitre Fourierjeve transformacije (FFT) na 8 točkah (slika 9).



Slika 9: *GPP* za FFT z 8 točkami.

Osnovni operaciji sta seštevanje (označeno z $\oplus$) ter odštevanje in množenje (označeno z $\otimes$). Če sta I_m in I_n vhodna podatka, potem velja

$$I_m \oplus I_n \equiv I_m + In$$

[16]Dinamična arhitektura.

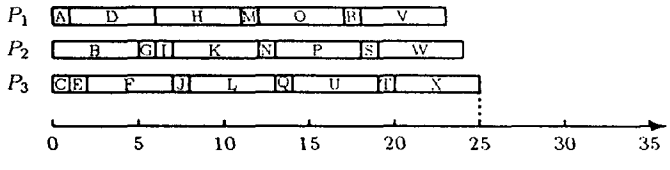
in

$$I_m \otimes I_n \equiv (I_m - I_n) \times e^{2\pi ik/8}, 0 \leq k \leq 7.$$

Predpostavimo, da je za izvršitev operacije $\oplus$ potrebna 1 časovna enota in za izvršitev operacije $\otimes$ 5 časovnih enot. Potem je čas potreben za zaporedno izvrševanje FFT algoritma $T_1 = 72$ časovnih enot. Vzporedno izvrševanje algoritma bi ob predpostavki, da imamo na voljo neomejeno mnogo procesorjev, zahtevalo le $T_\infty = 15$ časovnih enot. Torej je idealna pospešitev FFT algoritma $S_\infty = 4.8$.

3.1.2 Primer 1: Premalo procesorjev

Naj bo dan podatkovno pretokovni računalnik s tremi procesorji P_1, P_2 in P_3, ki morejo izvajati katerokoli od operacij $\oplus$ in $\otimes$. Idealizirajmo računalnik tako, da ne bo zakasnitev v pomnilniku in pri medprocesorskih komunikacijah. Predpostavimo statično arhitekturo (slika 6), ki podpira podatkovno pretokovno računanaje opredeljemo z modelom 3. Točke GPP bi se porazdelile med procesorje P_1, P_2 in P_3, kot je prikazano na sliki 10. Za ta primer dobimo $T_p = 25, S_p = 2.88$ in $E_p = 0.96$. Pospešitev S_p je manjša od S_∞.

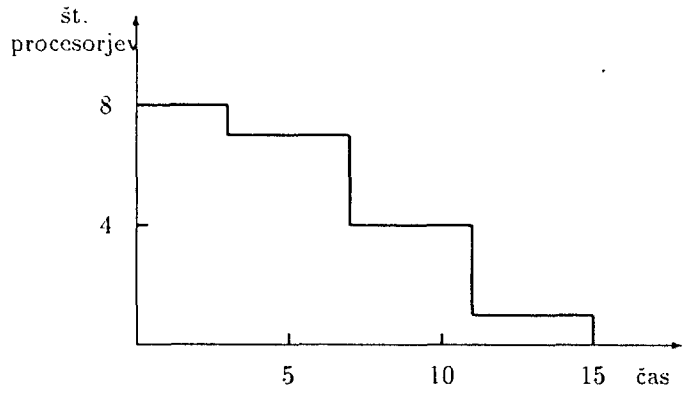


Slika 10: Izvrševanje FFT pri $p = 3$.

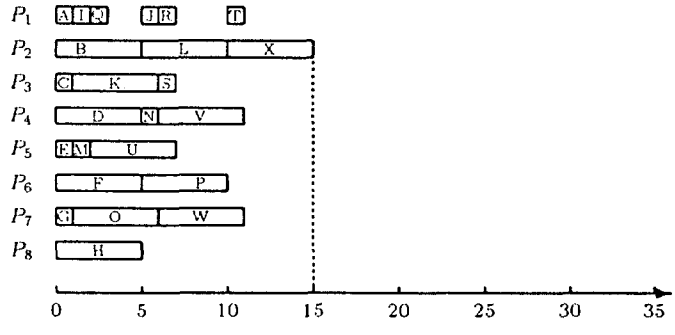
3.1.2 Primer 2: Optimizacija procesorjev

Predpostavimo enako arhitekturo kot v primeru 1, le da vsebuje procesna enota neomejeno mnogo procesorjev. Naj nam bo število delujočih procesorjev v trenutku t merilo vzporednosti algoritma v tem trenutku. Za FFT algoritem je prikazan časovni potek vzporednost na sliki 11. Opazimo, da vzporednost niha med 1 in 8 ter da je povprečna vzporednost enaka idealni pospešitvi algoritma $S_\infty = 4.8$. Postavlja se vprašanje, koliko procesorjev p bi zadoščalo, da bi veljalo $S_p = S_\infty$. Če bi se odločili za število procesorjev, ki je določeno z maksimalno vzporednostjo, tj. $p = 8$, bi dobili rezultat, ki je prikazan na sliki 12.

Dobimo $T_p = 15$ in $S_p = 4.8$. S tem smo zagotovili, da je $S_p = S_\infty$, žal pa je izkoriščenost procesorjev le $E_p = 0.6$. E_p je mogoče povečati le tako,



Slika 11: Dinamična vzporednost FFT algoritma.



Slika 12: Izvrševanje FFT pri $p = 8$.

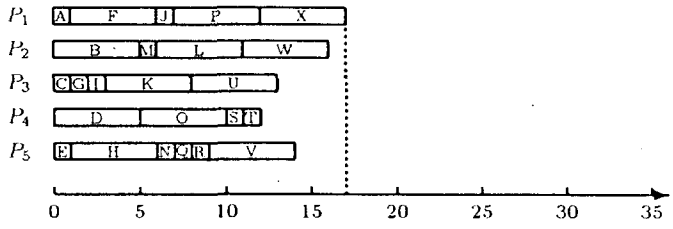
da uporabimo manj procesorjev. Sedaj je mogoče uvesti prvi problem, ki ga želimo rešiti, tj. *optimizacija procesorjev.*

Optimizacija procesorjev: *Za dani GPP moramo določiti takšno število procesorjev p, da bo $S_p = S_\infty$ in da bo E_p čim bliže 1.*

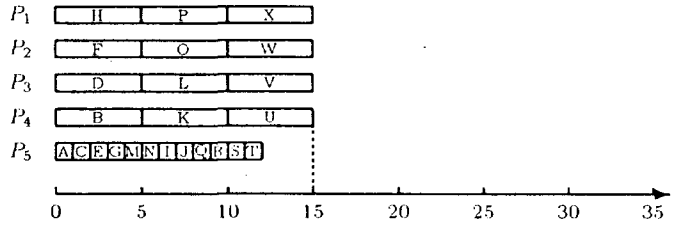
Morda bi zadoščalo, če bi vzeli toliko procesorjev, kot jih določa povprečna vzporednost S_∞. V našem primeru bi bilo $p = \lceil S_\infty \rceil = 5$. To možnost prikazuje slika 13. V tem primeru je rezultat naslednji: $S_p = 4.235$ in $E_p = 0.847$, kar pomeni, da je $S_p < S_\infty$.

Pojavi se vprašanje, ali bi bilo mogoče zagotoviti ustrezen način vstopanja paketov v čakalno vrsto (ali izstopanja iz vrste) pred procesno enoto in s tem takšno dodeljevanje paketov prostim procesorjem, da bi uspeli v primeru $p = 5$ zagotoviti tudi $S_p = S_\infty$. V našem primeru je to mogoče, kar prikazuje slika 14. Rezultat, ki ga dobimo po takšni optimizaciji, je: $S_p = 4.8$ in $E_p = 0.96$.

Postopek *optimizacije procesorjev* bomo opisali v



Slika 13: Izvrševanje FFT pri $p = 5$.



Slika 14: Optimizirano izvrševanje FFT pri $p = 5$.

nadaljevanju.

3.1.3 Primer 3: Optimizacija časa

Tudi sedaj predpostavimo enako arhitekturo kot v primeru 1; torej podatkovno pretokovni računalnik s $p = 3$ procesorji P_1, P_2 in P_3. Upeljimo najprej *upad idealne pospešitve* računanja D_p, ki ga definiramo kot

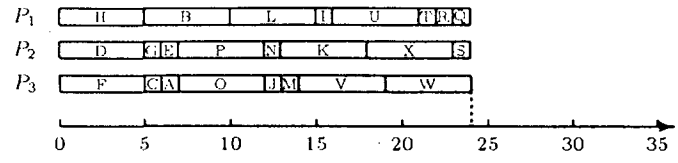
$$D_p = \frac{S_\infty - S_p}{S_p} = \frac{T_p - T_\infty}{T_\infty}.$$

V primeru 1 dobimo $D_p = 0.667$, kar pomeni 66.7% upad idealne pospešitve. Tudi tokrat je mogoče zagotoviti ustreznejši način vstopanja paketov v čakalno vrsto (oz. izstopanja iz vrste) pred procesno enoto in s tem zagotoviti manjši upad idealne pospešitve. Rešitev prikazuje slika 15. Rezultat, ki ga dobimo po takšni optimizaciji, je: $S_p = 3, D_p = 0.6$. V tem primeru dobimo tudi $E_p = 1$, kar pomeni, da smo dosegli tudi največjo izkoriščenost procesorjev. Ker se bomo v nadaljevanju še sklicevali na to rešitev, jo poimenujmo "optimalna".

Uvedimo sedaj drugi problem, ki ga želimo rešiti, tj. *optimizacija časa.*

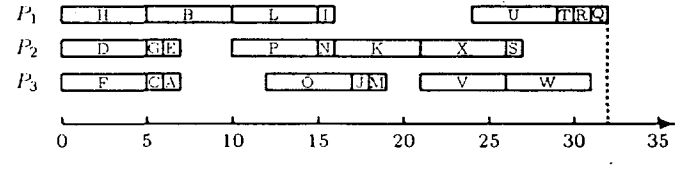
Optimizacija časa: *Za dani GPP in dano število procesorjev p, moramo določiti takšno dodeljevanje, da bo D_p čim bliže* 0.

Poizkusimo problem optimizacije časa še nekoliko zaplesti. Vzemimo sedaj hibridno arhitekturo



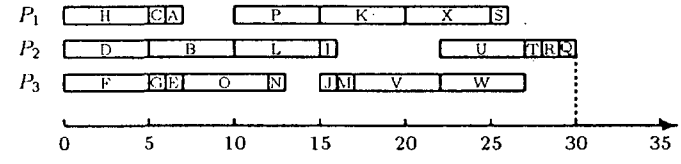
Slika 15: Optimalno izvrševanje FFT pri $p = 3, t_c = 0$.

(slika 8). Število procesorjev naj ostane $p = 3$, predpostavimo pa, da je za *medprocesorsko komunikacijo* t_c potrebnih 5 časovnih enot. Dodeljevanje procesorjev, kot je bilo v primeru "optimalne" rešitve, rezultira v $S_p = 2.25, D_p = 1.133$ in $E_p = 0.75$ (slika 16).



Slika 16: Izvrševanje FFT pri $p = 3, t_c = 5$.

S primernim dodeljevanjem procesorjev je mogoče doseči določene izboljšave. Tako npr. z rešitvijo, ki je podana na sliki 17, dosežemo $S_p = 2.4, D_p = 1$ in $E_p = 0.8$. Za izboljšanje D_p gre zahvala predvsem zmanjšanju medprocesorskih komunikacij. Uporabljenih je le 14 medprocesorskih komunikacij, medtem ko jih je bilo v prejšnjem primeru kar 24.



Slika 17: Optimizirano izvrševanje FFT pri $p = 3, t_c = 5$.

Tudi postopek *optimizacije časa* bomo opisali v nadaljevanju.

3.2 Primerjava rešitev

Iz dosedanje analize (tabela 2) vidimo, da tako problem optimizacije procesorjev kakor tudi optimizacije časa uspešno rešimo le, če zagotovimo ustrezen način dodeljevanja paketov prostim pro-

cesorjem (preko čakalne vrste), oz. zagotovimo ustrezno porazdelitev GPP med procesorje. Kot bomo videli kasneje, je to moč doseči le s preoblikovanjem podatkovno vodenega računanja.

p	t_c	T_p	S_p	D_p	E_p	vodenje računanja
8	0	15	4.8	0	0.6	podatkovno
5	0	17	4.235	0.133	0.847	podatkovno
3	0	25	2.88	0.667	0.96	podatkovno
5	0	15	4.8	0	0.96	mod. podatkovno
3	0	24	3	0.6	1	mod. podatkovno
3	5	33	2.182	1.2	0.727	podatkovno
3	5	30	2.4	1	0.8	mod. podatkovno

Tabela 2: Primerjava računanja FFT.

3.3 NP-polnost problema

Sedaj bomo spoznali, da je problem *optimizacije časa* NP-poln. Neposredna posledica tega je tudi NP-polnost problema *optimizacije procesorjev*. Ti dejstvi sta pomembni, saj opravičujeta uporabo hevrističnih metod pri reševanju obeh problemov.

Kot vemo, je $\mathcal{NP}$ razred problemov, ki so nedeterministično rešljivi v polinomskem času (in zato deterministično rešljivi v eksponentnem času). Nekateri med njimi so deterministično rešljivi celo v polinomskem času in tvorijo razred $\mathcal{P}$. Seveda je $\mathcal{P} \subseteq \mathcal{NP}$. Odprto pa je vprašanje, ali je $\mathcal{P} \neq \mathcal{NP}$. Zato ne vemo, ali obstaja problem, ki je deterministično rešljiv v eksponentnem, ne pa tudi polinomskem času.

Nekateri problemi iz razreda $\mathcal{NP}$ imajo še posebej zanimivo in pomembno lastnost, ki se kaže v tem, da moremo reševanje kateregakoli problema iz $\mathcal{NP}$ v polinomskem času prevesti na reševanje kakega od teh odlikovanih problemov, imenovanih NP-polni problemi. V tem smislu so NP-polni problemi med najtežjimi v razredu $\mathcal{NP}$. Če bi nam uspelo pokazati za vsaj enega izmed njih, da je deterministično rešljiv v polinomskem času, bi s tem pokazali, da je v polinomskem času rešljiv vsak problem iz $\mathcal{NP}$. Kljub velikim naporom pa doslej kaj takega še nikomur ni uspelo, čeprav zbirka znanih NP-polnih problemov šteje že nekaj sto problemov. Zato prevladuje prepričanje, da NP-polni problemi v resnici niso deterministično polinomsko rešljivi. Praktična posledica tega pa je, da se v primeru, ko naletimo na NP-poln problem, vedemo pragmatično, torej ne iščemo polinomskega algoritma (zaradi majhne verjetnosti, da bi ga našli), ampak raje pričnemo sestavljati hevristični algoritem.

Problem optimizacije časa ter problem optimizacije procesorjev spadata v razred NP-polnih problemov! Prvi se v literaturi pojavlja pod imenom DODELJEVANJE UREJENIH OPRAVIL (PCS[17]) v naslednji obliki

Naj bo dana množica $\mathcal{V} = \{v_1, \ldots, v_n\}$ *opravil dolžine* 1, *ki so urejena z relacijo* $\longmapsto$ *stroge delne urejenosti,* $p \in \mathbb{N}$ *procesorjev ter mejni čas* $T \in \mathbb{N}$. *Ali obstaja takšna preslikava* s *množice* $\mathcal{V}$ *v množico* $\{0, 1, \ldots, T\}$, *da za vse* $\tau \in \{0, 1, \ldots, T\}$ *ter* $u, v \in \mathcal{V}$ *velja* $| s^{-1}(\tau) | \leq p$ *in hkrati* $u \longmapsto v \Rightarrow s(u) < s(v)$?

Dokaz NP-polnosti problema PCS je podan v [46]. Ker imajo vsa opravila enako časovno zahtevnost, je PCS le posebna oblika problema optimizacije časa - torej je slednji gotovo NP-poln. NP-polnost problema optimizacije procesorjev pa pokažemo tako, da nanj polinomsko časovno prevedemo problem PCS [47]. Če namreč nek NP-poln problem A polinomsko časovno prevedemo na problem $B \in \mathcal{NP}$, potem je (zaradi tranzitivnosti relacije polinomske prevedljivosti) tudi B NP-poln.

3.3.1 Reševanje NP-polnih problemov

Kot smo uvodoma omenili, imajo najboljši znani algoritmi za reševanje NP-polnih problemov vsi eksponentno časovno kompleksnost. Zaradi hitre rasti eksponentne funkcije to običajno pomeni, da ne znamo dobiti optimalnih rešitev že pri relativno majhnih problemih. Eden od pristopov k reševanju NP-polnega problema je zato pospešitev danega algoritma, tj. pospešitev pregledovanja množice možnih rešitev. Takšen pristop ponuja delno rešitev ter sproti preverja, ali le-ta vodi do optimalne rešitve. Kadar temu ni tako, delno rešitev opusti ter sestavi kako drugo delno rešitev. Med takšne pristope spada npr. *razveji in omeji.*

Drug pristop pri reševanju NP-polnega problema je razvoj algoritma, ki v sprejemljivem času vrne rešitev, ki usreza danim zahtevam. Takšni algoritmi se imenujejo *hevristični.* Dve družini hevrističnih algoritmov obsegata *aproksimacijske* ter *verjetnostne* algoritme. Aproksimacijski algoritmi vračajo rešitve, za katere je znano, koliko največ utegnejo odstopati od optimalne reštve. Verjetnostni algoritmi [48] pa ob polinomski časovni zatevnosti dovolj pogosto (z določeno verjetnostjo) vračajo optimalne rezultate. Slednji bodo

[17] Precedence Constrained Scheduling

uporabljeni pri reševanju problemov optimizacije časa in optimizacije procesorjev.

Literatura

[1] J. B. Dennis. The Variaties of Data Flow Computers. In *Proc. First Int'l Conf. Distributed Comp. Sys.*, pages 430–439, October 1979.

[2] J. B. Dennis.. Data Flow Supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[3] S. Ribarić. Računari upravljani tokom podataka. *Informatica*, 6(4):3–11, 1982.

[4] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-Driven and Demand-Driven Computer Architectures. *Computing Surveys*, 14(1):93–143, March 1982.

[5] J. Šilc in B. Robič. Osnovna načela DF sistemov. *Informatica*, 9(2):10–15, 1985.

[6] J. B. Dennis. Dataflow Computation: A Case Study. In V. M. Milutinović, editor, *Computer Architecture - Concepts and Systems*, pages 354–404. North-Holland, 1988.

[7] L. Bic and J. L. Gaudiot. Special Issue on Data-Flow Computing. *Journal of Parallel and Distributed Computing*, 10(4):277–385, December 1990.

[8] A. L. Davis and R. M. Keller. Data Flow Program Graphs. *IEEE Computer*, 15(2):26–41, February 1982.

[9] W. B. Ackerman. Data Flow Languages. In *Proc. National Comp. Conf.*, pages 1087–1095, June 1979.

[10] J. R. McGraw. Data-Flow Computing: The VAL Language. *ACM Trans. Prog. Languages and Systems*, 4(1):44–82, January 1982.

[11] T. Jeffery. The μPD7281 Processor. *Byte*, pages 237–246, November 1985.

[12] S. Weiss, I. Spillinger, and G. M. Silberman. Architectural Improvements for Data-Driven VLSI Processing Arrays. In *Proc. Functional Programming Languages and Computer Architecture*, pages 243–259, September 1989.

[13] M. Cornish. The TI Dataflow Architecture: The Power of Concurrency for Avionics. In *Proc. 3rd Conf. Digital Avionics Systems*, pages 19–25, November 1979.

[14] D. Comte, N. Hifdi, and J. C. Syre. The Data Driven LAU Multiprocessor System: Results and Perspectives. In *Proc. World Comp. Congress IFIP'80*, pages 175–180, October 1980.

[15] J. Gurd and I. Watson. Preliminary Evaluation of a Prototype Dataflow Computer. In *Proc. 9th World Comp. Congress IFIP'83*, pages 545–551, September 1983.

[16] J. R. Gurd C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Comm. ACM*, 28(1):34–52, January 1985.

[17] R. Vedder, M. Campbell, and G. Tucker. The Hughes Data Flow Multiprocessor. In *Proc. 5th Int'l Conf. Distributed Computing Systems*, pages 2–9, May 1985.

[18] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi. Evaluation of a Prototype Data Flow Processor of the SIGMA–1 for Scientific Computations. In *Proc. 13th Annual Int'l Symp. Comp. Arch.*, pages 226–234, June 1986.

[19] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, 1984.

[20] B. Robič in J. Šilc. Razvrstitev novogeneracijskih računalniških arhitektur. *Informatica*, 10(4):18–32, 1986.

[21] N. Gehani and A. D. McGettrick, editors. *Concurrent Programming*. Addison-Wesley Publishing Comp., 1988.

[22] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Comp., 1989.

[23] V. M. Milutinović, editor. *Computer Architecture - Concepts and Systems*. North-Holland, 1988.

[24] L. M. Patnaik, R. Govindarajan, M. Špegel, and J. Šilc. A Critique on Parallel Computer Architectures. *Informatica*, 12(2):47–64, 1988.

[25] V. M. Milutinović, editor. *High-Level Computer Architecture*. Computer Science Press, 1989.

[26] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2):5–16, February 1990.

[27] V. P. Srini. An Architectural Comparison of Dataflow Systems. *IEEE Computer*, 19(3):68–88, March 1986.

[28] J. Šilc and B. Robič. The Review of Some Data Flow Computer Architectures. *Informatica*, 11(1):61–66, 1987.

[29] T. Y. Feng. Some Caracteristic of Associative/Parallel Processing. In *Proc. 1972 Sagamore Computing Conf.*, pages 5–16, August 1972.

[30] W. Händler. The impact of classification schemas on computer architecture. In *Proc. 1977 Int'l Conf. Parallel Processing*, pages 7–15, August 1977.

[31] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers*, C-21(9):948–960, September 1972.

[32] D. B. Skillicorn. A Taxonomy for Computer Architectures. *IEEE Computer*, 21(11):46–57, November 1988.

[33] S. Dasgupta. A Hirarchical Taxonomic System for Computer Architectures. *IEEE Computer*, 23(3):64–74, March 1990.

[34] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A Second Opinion on Data Flow Machines and Languages. *IEEE Computer*, 15(2):58–89, February 1982.

[35] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[36] Arvind and D. E. Culler. Dataflow Architectures. *Ann. Rev. Comput. Sci.*, 1:225–253, 1986.

[37] A. L. Davis. A Dataflow Evaluation System Based on the Concept of Recursive Locality. In *Proc. 1979 Nat. Computer Conf.*, pages 1079–1086, 1979.

[38] J. Šilc and B. Robič. Data Flow Based Parallel Inference Machine. *Informatica*, 11(4):27–34, 1987.

[39] A. V. S. Sastry, L. M. Patnaik, and J. Šilc. Dataflow Architectures for Logic Programming. *Elektrotehniški vestnik*, 55(1):9–19, januar 1988.

[40] J. Šilc and B. Robič. Efficient Dataflow Architecture for Specialized Computations. In *Proc. 12th World Congress on Scientifics Computation*, pages 4.681–4.684, July 1988.

[41] Arvind and V. Kathail. A Multiple Processor Dataflow Machine That Supports Generalized Procedures. In *Proc. 8th Annual Int'l Symp. Comp. Arch.*, pages 291–302, May 1981.

[42] N. Ito, M. Sato, E. Kuno, and K. Rokusawa. The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D. In *Proc. 13th Annual Int'l Symp. Comp. Arch.*, pages 149–156, June 1986.

[43] V. L. Narasimhan and T. Downs. Operating System Features of a Dynamic Dataflow Array Processing System (PATTSY). In *Proc. 3rd Annual Parallel Processing Symp.*, pages 722–740, March 1989.

[44] V. G. Grafe and J. E. Hoch. The Epsilon-2 Multiprocessor System. *Journal of Parallel and Distributed Computing*, 10(4):309–318, December 1990.

[45] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, December 1990.

[46] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[47] B. Robič. Minimizacija števila procesorjev v podatkovno pretokovni arhitekturi. Magistersko delo, Fakulteta za elektrotehniko, Ljubljana, 1987.

[48] J. Žerovnik. *Verjetnost v kombinatorični optimizaciji*. Doktorska disertacija, Fakulteta za elektrotehniko in računalništvo, Ljubljana, 1992. V pripravi.