OSNOVNA NAČELA DF SISTEMOV

J. ŠILC IN B. ROBIČ

INSTITUT „JOŽEF STEFAN"
LJUBLJANA

UDK: 681.519.7

Koncept krmiljenja s tokom podatkov predstavlja bistveno spremembo v načinu izvajanja instrukcij napram klasičnemu sekvenčnemu izvajanju. Ti sistemi ne delujejo na podlagi krmilnega toka, zato tudi ne potrebujejo programskega števca in pomnilnika s klasično funkcijo. V DF računalnikih postanejo instrukcije izvršljive v trenutku, ko prispe zadnji med zahtevanimi operandi, kar omogoča vzporedno izvrševanje instrukcij. Logična posledica tega je visoka stopnja izkoriščenosti inherentne paralelnosti, prisotne v algoritmih.

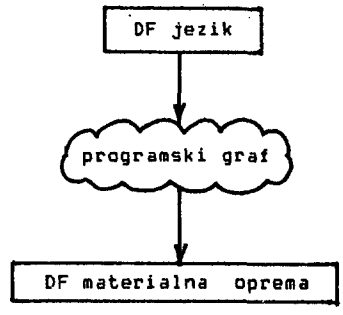
Basic Principles of Data Flow systems. The data flow concept is a fundamentally different way of looking at instruction execution in machine-level programs - an alternative to sequential instruction execution. In a data flow computer an instruction is ready for execution when its operands have arrived. There is no concept of control flow, and data flow computers do not have program location counters. A consequence of data - activated instruction execution is that many instructions of a data flow program may be available for execution at once. Thus, highly concurrent computation is a natural consequence of the data flow concept.

Uvod

Vse von Neumannove arhitekture računalniških sistemov imajo dve pomembni karakteristiki. Prvič, imajo globalni naslovljivi pomnilnik, ki pomni programe ter podatke in katerega vsebina se v skladu s programskimi instrukcijami pogosto ažurira. In drugič, vsebujejo programski števec, katerega vsebina je naslov naslednje instrukcije, ki naj se izvrši. Programski števec se bodisi implicitno ali eksplicitno ažurira in s tem določa sekvence instrukcij, ki se izvajajo. Takšno krmiljenje poteka programa je temeljna omejitev von Neumannovih arhitektur, še posebno pri vzporednem procesiranju.

DF računalniki (data flow computers) nimajo nobene od obeh zgoraj omenjenih značilnosti. Prvič, njihova struktura je zasnovana tako, da poteka procesiranje na osnovi vrednosti in ne naslovov spremenljivk, kar pomeni, da ni potreben globalni naslovljivi pomnilnik, saj ni nikakršnega naslavljanja. Drugič, v DF sistemih ni ničesar, kar bi bilo podobno programskemu števcu. Torej temeljijo ti sistemi na načelih asinhronosti in funkcionalnosti, ki pravita, da so vse operacije funkcije (funkcionalnost), ki postanejo izvršljive takoj, ko so na voljo vrednosti vseh vhodnih operandov (asinhronost). Iz načela funkcionalnosti sledi, da se katerikoli operaciji, ki sta izvršljivi, lahko izvršita po kakršnemkoli zaporedju ali konkurenčno.

Zgornji načeli logično vodita do programskega grafa (data flow program graph), v katerem točke ponazarjajo operacije, usmerjene povezave pa so nosilke vrednosti vhodnih operandov ter včasih še dodatnih informacij o delih izračunov, katerim pripadajo operandi. Programskemu grafu, ki je rezultat prevajanja programa v višjem DF jeziku (data flow language), se dinamično prilagodi materialna oprema (data flow hardware), ki je sposobna izvajanja vsega inherentnega paralelizma, opisanega s tem grafom , kar ponazarja slika 1 [1].



Slika 1.

* Raziskavo financira RSS po pogodbi C2-0124-106-84

Dinamična vzporednost

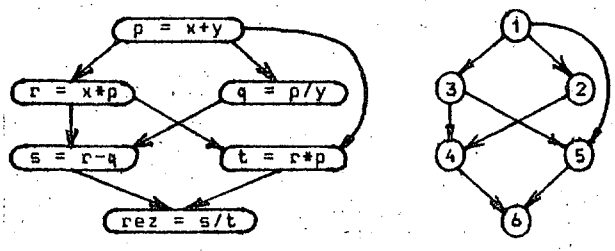
Veliko število algoritmov (npr. pri NP problemih) vsebuje inherentni paralelizem, ki pa v von Neumannovih arhitekturah ni izrabljen v največji možni meri. Arhitekture, ki bi to omogočale, bi npr. vse NP probleme prevedle na P probleme, kar pomeni, da bi se eksponentna časovna kompleksnost znižala na polinomsko. Zakaj ? Eksponentna časovna kompleksnost determinističnih algoritmov za reševanje nekaterih NP problemov izvira iz števila potencialnih rešitev, katere mora algoritem sekvenčno generirati in preveriti. S paralelnimi arhitekturami pa bi bilo moč realizirati nedeterministične algoritme, ki bi bili sposobni sočasnega generiranja potencialnih rešitev in izbire najustreznejše med njimi.

V ta namen je potrebno definirati programski jezik, ki bi dovolj eksplicitno izražal inherentno paralelnost algoritma, definirati strukturo (strojni jezik), ki je rezultat prevajanja programa v tem jeziku in ki omogoča dovolj enostavno realizacijo z materialno opremo; predvsem pa naj ohranja vso inherentno paralelnost algoritma.

Oglejmo si segment programa, zapisanega v enem od von Neumannovih jezikov.

```
1)  p := x+y ;
2)  q := p/y ;
3)  r := x*p ;
4)  s := r-q ;
5)  t := r*p ;
6)  rez := s/t ;
```

Standardni prevajalniki generirajo kodo, ki ustreza zaporednemu izvajanju stavkov segmenta, torej 1,2,3,4,5,6 . Vendar pa se lahko nekateri med njimi izvršijo paralelno, npr. 1,[2,3],4,5,6, kjer z [2,3] opišemo paralelno izvršitev stavkov 2 in 3. Vprašamo se, ali je to vsa inherentna paralelnost zgornjega segmenta. Očitno je, da je iz sekvenčnega zapisa paralelnost težko razvidna, zato se poslužujemo programskih grafov. Programski graf, ki ustreza zgornjemu segmentu, je prikazan na sliki 2.



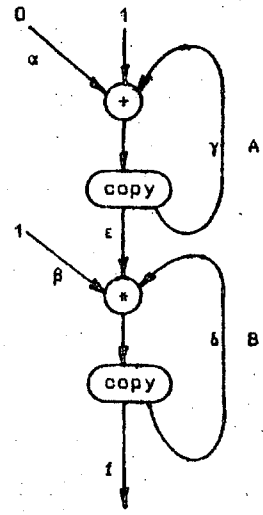
Slika 2.

Točke programskega grafa ponazarjajo stavke (operacije), usmerjene povezave pa sovpadajo s prehajanjem vrednosti operandov. Ker je graf na sliki 1 aciklični, se lahko operacije, ki so na istem nivoju, izvajajo paralelno. Operacija w je na nivoju i, če je dolžina najdaljše poti od začetne operacije do operacije w enaka i. V programskem grafu na sliki 1 so operacije porazdeljene po nivojih na sledeč način:

nivo 0 : 1
nivo 1 : 2,3
nivo 2 : 4,5
nivo 3 : 6.

Zaporedje izvrševanja stavkov 1,[2,3],[4,5],6 izkorišča inherentno paralelnost v največji meri. Operacije, ki so na istem nivoju, lahko kljub temu postanejo izvršljive v različnih časih - tedaj, ko so na voljo vrednosti vseh vhodnih operandov. Predpostavimo, da trajata operaciji seštevanja in odštevanja po eno, množenja dve in deljenja tri časovne enote. Čeprav sta operaciji 4 in 5 na istem nivoju, torej se lahko izvajata paralelno, bo postala operacija 4 izvršljiva po štirih, operacija 5 pa že po treh časovnih enotah. Torej je izvajanje programskega grafa popolnoma asinhrono.

Pri cikličnih grafih lahko nastanejo dodatne težave. Kadar v posamezni ponovitvi zanke naletimo na podatkovne odvisnosti med operacijami, to ne ustavi nadaljnih ponovitev, dasiravno predhodna ponovitev ni popolnoma končana, saj so vrednosti vhodnih operandov akumulirane v določenih vejah grafa. Oglejmo si primer intuitivno konstruiranega cikličnega programskega grafa za izračun funkcije f(i) = i! (slika 3).



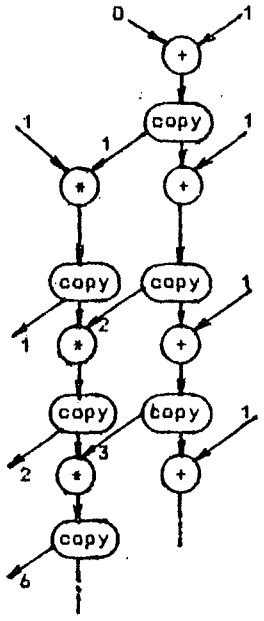
Slika 3.

Poseben selekcijski mehanizem poskrbi, da se ob prvi izvršitvi blokov A in B uporabita povezavi α in β ,ob vsaki naslednji pa namesto njiju povezavi γ in δ. Blok A inkrementira vrednost operanda za 1, s katerim nato blok B pomnoži prejšnji rezultat. Predpostavimo, da je za operacijo kopiranja (copy) potrebno pol, operacijo seštevanja ena in za operacijo množenja dve časovni enoti. Tedaj bi bile funkcijske vrednosti v točki f zaporedoma 1! , 2! , 4! ... Vzrok za izostanek rezultata 3! je v tem, da se zaradi hitrejšega izvajanja bloka A vrednost vhodnega operanda v povezavi ε prekrije z novo vrednostjo, ne da bi blok B uporabil predhodno vrednost. V primeru, ko bi se blok B izvajal hitreje kot blok A, pa bi se ista vrednost vhodnega operanda v povezavi ε uporabila večkrat, kar bi imelo za posledico, da bi se nekatere funkcijske vrednosti v točki f ponavljale, npr. 1!, 2! , 2! , 3! ...

Tako ni mogoče s prisotnostjo katerekoli vrednosti vhodnega operanda deklarirati vozlišča kot izvršljivega, saj lahko pripadajo vrednosti vhodnih operandov popolnoma različnim delom izračuna. Obstaja nekaj različnih rešitev nastalega problema oziroma načinov realizacije DF sistema [2].

(i) Ciklični programski graf transformiramo v aciklični tako, da vsako ponovitev opišemo z acikličnim podgrafom. Tako transformiran graf iz slike 3 ima obliko kot je prikazana na sliki 4.

Ta rešitev zahteva velike količine programskega pomnilnika, zahteva pa tudi dinamično generiranje koda v primeru, ko je pogoj za izstop iz zanke izračunan šele v času njenega izvajanja. Obe zahtevi se odražata kot pomembna pomankljivost v praktičnih sistemih.



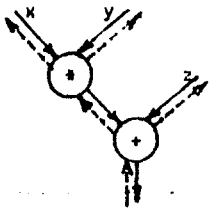
Slika 4.

(ii) Ponovitve opišemo z enim grafom, vendar se lahko ponovitev prične šele tedaj, ko je predhodna končana.

Takšna rešitev ne dovoljuje vzporednosti med ponovitvami in zahteva posebne ukaze ali posebno materialno opremo, ki testira konec ponovitve.

(iii) Uporaba grafov je omejena tako, da je v veji grafa prisotna istočasno samo ena vrednost operanda [3]. To pomeni, da se operacija lahko izvrši le tedaj, ko so prisotne vrednosti vseh vhodnih operandov in ni v izhodni veji nobene vrednosti. Torej vrednost spremenljivega operanda v neki povezavi ne sme biti spremenjena vse dokler ni uporabljena. Ko pa je enkrat uporabljena, mora postati neuporabljiva. Kljub sekvenčnosti izvajanja je prednost tega načina pred prvima dvema v možni izrabi cevljenja (pipelining). Vsaka operacija po svoji izvršitvi obvesti svoje "starše", da je pripravljena od njih sprejeti naslednje vrednosti vhodnih operandov. To stori s pomočjo dodatnih povezav - nosilk potrditvenih (aknowledge) signalov, kar kaže primer na sliki 5.

——— nosilka vrednosti operanda
– – – – – nosilka potrditvenega signala



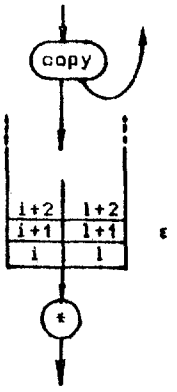
Slika 5.

Z uvedbo potrditvenih signalov se število povezav, in s tem promet v grafu, podvoji.

(iv) Poleg vrednosti operandov nosijo povezave še dodatno informacijo o delih izračunov, katerim pripadajo operandi [4]. Pravimo, da so povezave nosilke znakov (token). Znaki imajo dodatne labele, ki vsebujejo indeks in nivo ponovitve. Te labele običajno imenujemo barva. Operacije se lahko izvrši le tedaj, ko imajo vsi vhodni znaki enako barvo. Ta metoda uporablja popolnoma statično generiran kod in omogoča maksimalno vzporednost izvajanja.

Takšna rešitev zahteva povečan pretok informacij po grafu in dodatna vozlišča za spreminjanje ter primerjanje label, kar ima za posledico bodisi dodaten čas za izračun label ali pa uporabo posebne materialne opreme.

(v) Tudi tu so povezave nosilke znakov, poleg tega pa opravljajo še funkcijo vrst (queue), kar pomeni, da so v njih znaki razvrščeni v istem vrstnem redu, kot so vanje prihajali. Povezava ε s slike 3 ima tedaj obliko, kot jo prikazuje slika 6.



Slika 6.

Ta rešitev omogoča enako stopnjo vzporednosti kot pri labeliranju, zahteva pa čakalne vrste, ki so prostorsko zahtevne.

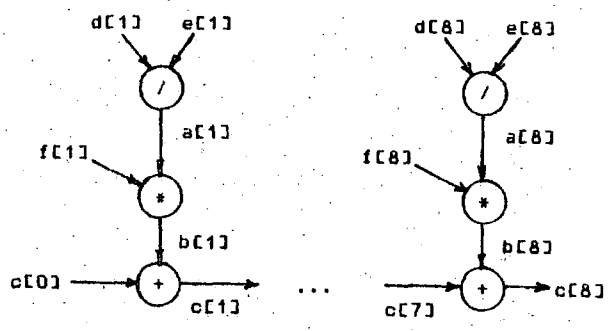
Programski graf, zgrajen po enem od zgornjih načel, je struktura, ki učinkovito povezuje visok programski jezik z materialno opremo in popolnoma ohranja inherentno paralelnost (slika 1).

Primerjavo opisanih petih realizacij DF sistema si oglejmo na primeru naslednjega programa:

```
input d,e,f
   c[0] = 0
for i from 1 to 8 do
   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 zahteva deljenje tri, množenje dve in seštevanje eno časovno enoto. Namišljeni DF računalnik naj ima štiri procesne enote P1, P2, P3 in P4, od katerih lahko vsaka izvaja katerokoli od operacij. Idealizirajmo DF računalnik tako, da bodo zakasnitve pomnilnika in med povezavami nič. Program se bo izvrševal po programskem grafu, prikazanem na sliki 7.

Iz programskega grafa na sliki 7, je razvidno, da je kritična pot a[1], b[1], c[1], c[2], ... , c[8], katere minimalni čas izvajanja je 13 časovnih enot.



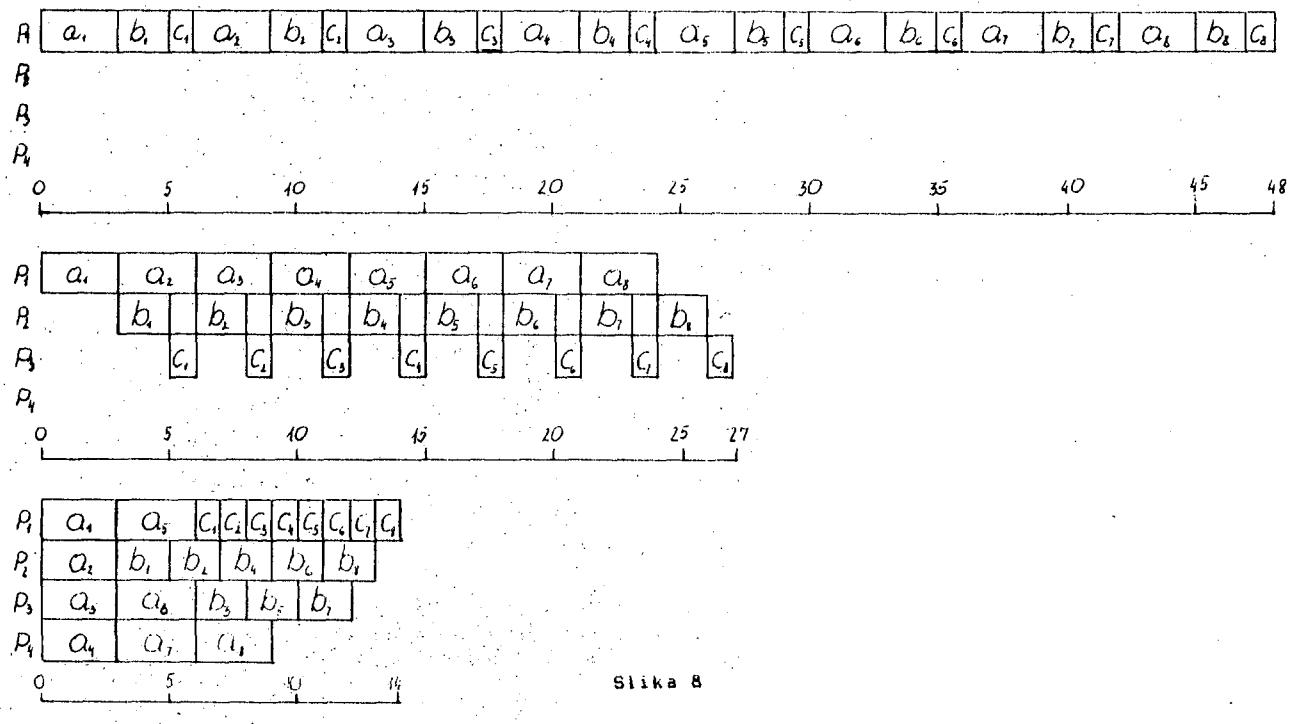
Slika 7.

Če uporabimo strategijo (ii), po kateri se naslednja ponovitev prične šele ko se predhodnja konča, dobimo popolnoma sekvenčno izvajanje programa. Ker zahteva vsaka ponovitev po 6 časovnih enot, je skupni čas izvajanje programa enak 6 X 8 = 48 časovnih enot. Kot rečeno, je izvajanje programa sekvenčno, torej bi zadoščal en procesor; vendar je praktično izračun porazdeljen preko vseh štirih procesorjev, tako da postane uporabljenost procesorjev 12 / 48 = 0.25 (slika 8.a). Strategija (iii), ki dovoljuje istočasno prisotnost enega znaka v veji programskega grafa, vodi do cevljenja bloka prireditvenih stavkov znotraj zanke (slika 8.b). Čas izvajanja zanke narekuje časovno najzahtevnejša operacija v zanki (deljenje), tako je čas izvajanja programa enak 3 X 8 + 3 = 27 in uporabljenost procesorjev 12 / 27 = 0.44. Strategije (i), (iv) in (v) so povsem enakovredne in omogočajo najhitrejše izvajanje programa in sicer 14 časovnih enot, pri najboljši uporabljenosti procesorjev 12 / 14 = 0.86 (slika 8.c).

Iz primera vidimo, da imajo sekvenčni sistemi najslabše in DF sistemi z labeliranimi znaki najboljše performance, medtem ko so sistemi, ki uporabljajo koncept cevljenja nekje vmes.

DF jeziki

Osnovni cilj vseh izboljšav klasične von Neumannove arhitekture je v čimvečji izrabi paralelnosti. Hkrati z izboljšavami v arhitekturi so potekale raziskave tudi na področju optimizacije prevajalnikov, ki so programe, pisane v konvencionalnih von Neumannovih jezikih, prevedli v obliko, prirejeno izboljšani arhitekturi. Poleg tega pa je bilo konstruiranih tudi nekaj novih visokih programskih jezikov, kot sta Concurrent Pascal in Glypnir, prirejenih izboljšanim von Neumannovim arhitekturam. Ti jeziki vsebujejo sintaksne konstrukte, s pomočjo katerih postanejo arhitekturne lastnosti računalnika vidne za progra-



Slika 8

merja. Le-ta z njihovo uporabo olajša delo prevajalniku pri odkrivanju paralelnosti. Večina teh jezikov pa je nenaravnih v smislu, da v preveliki meri odražajo arhitekturo, na podlagi katere so oblikovani, manj pa način, na katerega programer razmišlja pri reševanju nekega problema.

Tako kot ostale oblike paralelnih računalnikov zahtevajo tudi DF računalniki (zaradi čimboljše izrabe svojih lastnosti) posebne visoke programske jezike, t.i. DF jezike, ki pa ne sodijo med von Neumannove jezike [5].

Za razliko od konvencionalnih jezikov, kateri delujejo nad podatki s pomočjo stranskih učinkov, DF jeziki le-teh ne poznajo. Znana je namreč zveza med učinkovitim paralelnim računanjem in odsotnostjo stranskih učinkov. To lastnost imajo funkcionalni jeziki, ki delujejo izključno na podlagi uporab funkcij nad vrednostmi [6]. Sledeča lastnost, katero imajo DF jeziki, je lokalnost učinka, kar pomeni, da ukazi nimajo nepotrebnih, daleč segajočih podatkovnih odvisnosti. Omejitve glede izvajanja ukazov temeljijo izključno na podatkovnih odvisnostih. Posledica te zahteve je, da je vsa informacija, potrebna za izvajanje programa, vsebovana v programskem grafu.

DF jeziki imajo v sintaksnem smislu veliko skupnih lastnosti s konvencialnimi jeziki, saj uporabljajo podobne programske konstrukte, kot so prireditev, aritmetične izraze, pogojne stavke, iteracije in rekurzijo. Bistveno pa se razlikujejo v semantiki. Za razliko od konvencionalnih jezikov, pri katerih identifikator predstavlja naslovljivo enoto pomnilnika, pa pri DF jezikih predstavlja povezavo. Posledica takšne semantike DF jezikov je, da se lahko nekemu identifikatorju priredi vrednost samo enkrat, kar imenujemo pravilo o enkratni prireditvi. S preimenovanjem identifikatorjev lahko v neiterativnih delih programa problem večkratne prireditve uspešno rešimo. Težave pa lahko nastopijo v zankah. Zato je potrebno pri definiranju zanke navesti (1) vhodne vrednosti vseh začetnih identifikatorjev, (2) pogoj ustavljanja, (3) vrednost, ki naj bo rezultat ob koncu izvajanja zanke in (4) pravila po katerih se začetnim identifikatorjem (ob vsaki izvršitvi telesa zanke) priredijo nove vrednosti.

Za ilustracijo si oglejmo primer algoritma za izračun n!, zapisanega v DF jezikih VAL (Value-oriented Algorithmic Language)

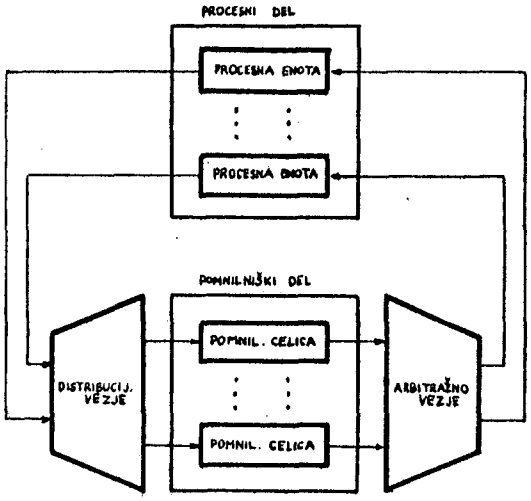
```
for j,k := n,1 ;                          (1)
do if j = 0                               (2)
     then k                               (3)
     else iter j,k := j-1,k*j ;           (4)
   end
end
```

in ID [5]

```
(initial j ← n; k ← 1                     (1)
 while j <> 0 do                          (2)
   new j ← j-1; new k ← k*j;              (4)
 return k)                                (3)
```

Arhitektura DF sistema

DF računalniki nimajo centralnega procesorja, temveč imajo procesni del, ki ga sestavlja množica nekaj deset, sto ali tisoč procesnih enot. Vsaka procesna enota je enaka enostavni aritmetično logični enoti ali vhodno/izhodnemu procesorju. Nimajo niti pomnilnika, kakršnega uporabljajo von Neumannove arhitekture, zato pa imajo pomnilniški del, ki ima potencialno veliko število pomnilniških celic, v katerih se nehajajo vsi podatki o programskem grafu. Za DF računalnike je značilno tudi to, da ne uporabljajo sinhronizacijske ure, programskega števca in registrov. Zato pa ima arbitražno vezje, ki usmerja izhode iz celic pomnilniškega dela v ustrezne procesne enote procesnega dela in distribucijsko vezje, ki povezuje procesne enote s pomnilniškimi celicami. Arhitektura DF sistema je prikazana na sliki 9 [7].



Slika 9.

Arbitražno vezje ugotovi, katere operacije so godne za izvršitev in jih posreduje procesnemu delu. Le-ta jih izvrši in pošlje delne rezultate distribucijskemu vezju. To vezje pa ugotovi katere operacije iz pomnilniškega dela čakajo na dobljene rezultate in jim jih posreduje v obliki vhodnih operandov. Sedaj zopet nastopi arbitražno vezje s čimer se cikel ponovi.

Zaključek

Osnovne ideje, na katerih temelji DF koncept, segajo v pozna šestdeseta leta. Z razvojem sodobne VLSI tehnologije je postala obstoječa (von Neumannova) arhitektura glavna ovira pri izkoriščanju paralelnosti v algoritmih. VLSI tehnologija pa je omogočila smiselno uporabo materialne opreme v arhitekturah non-von (ne von Neumannovih) sistemov. DF koncept šestdesetih let je tako postal uresničljiv, saj omogoča ta tehnologija izdelavo integriranih vezij s celo 100 nožicami. Tisoč takih vezij, ki opravljajo usmerjevalno-povezovalno funkcijo (router), omogoča uporabo celo 512 procesnih enot ali celičnih blokov (cell block). Če vsaka od procesnih enot izvrši milijon instrukcij v sekundi, potem nove arhitekture (ob pravilni izrabi paralelizmov) omogočajo skoraj milijardo inštrukcij v sekundi.

Večji DF projekti v svetu potekajo :

- na MIT, kjer skupina pod vodstvom J.Dennisa razvija DF sistem z rezinastimi procesorji tipa Am2903,
- tudi na MIT, kjer skupina pod Arvindovim vodstvom gradi VLSI 64 procesorski DF računalnik z labeliranimi znaki (tagged-token),
- na univerzi Utah deluje skupina pod vodstvom A.Davisa, ki je sestavil prvi delujoči DF računalnik,

- na CERT v Toulousu, kjer deluje skupina pod vodstvom J.C.Syra na projektu LAU,
- na univerzi v Manchesteru gradijo eksperimentalni multiprocesorki DF sistem z labeliranimi znaki pod vodstvom J.Gurda in I. Watsona in
- na univerzi Tokyo, računalnik Topstar, pod vodstvom T.Suzukija in J.Motooke.

Analize učinkovitosti, ki so jih izvršili na realiziranih DF sistemih, so pokazale občutno časovno izboljšanje pri reševanju nekaterih znanih algoritmov, kot sta FFT (40:1) in Gaussova eliminacijska metoda (80:1). Ker so ti sistemi šele v razvoju, obstaja tudi nekaj nerešenih problemov, kot sta problema vhodno/izhodnih aktivnosti in začasnega pomnenja [2].

Literatura

[1] T.Agerwala, Arvind : 'Data Flow Systems', Computer, Vol.15, No.2,Feb.1982, pp.10-13

[2] D.D.Gajski, D.A.Padua, D.J.Kuck & R.H. Kuhn : 'A Second Opinion on Data Flow Machines and Languages', Computer,Vol.15, No.2, Feb.1982, pp.58-69

[3] J.B.Dennis : 'Data Flow Supercomputers', Computer,Vol.13,No.11,Nov.1980, pp.48-56

[4] I.Watson, J.Gurd : 'A Practical Data Flow Computer', Computer,Vol.15,No.2,Feb.1982, pp.51-57

[5] W.B.Ackerman : 'Data Flow Languages', Computer,Vol.15,No.2,Feb.1982, pp.15-25

[6] J.Backus : 'Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs', Comm. of the ACM, Vol.21,No.5,Aug.1978, pp.613-641

[7] J.B.Dennis, W.Y.-P.Lim & W.B.Ackerman : 'The MIT Data Flow Engineering Model', Information Processing 83, R.E.A.Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), pp.553-560