tanulmányok 126/1981

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITASTECHNIKAI ÉS AUTOMATIZÁLASI KUTATÓ INTÉZETE

MULTIMIKROSZÁMÍTÓGÉP-RENDSZEREK

Irta:

VÁRSZEGI SÁNDOR

Tanulmányok 126/1981

A kiadásért felelős:

DR VÁMOS TIBOR

ISBN 963 311 124 2

ISSN 0324-2951

Készült a KSH SZÁMOK Nyomdában
195/1981

TARTALOM

Oldal

I. rész: Hardware és software

1. Hardware 10
1.1 A rendszerek osztályozása 11
1.2 Rendszer-architektúrák 13
1.2.1 Fix rendszer-architektúrák 13
1.2.2 Programozható rendszer-architektúrák 23
1.3 Interrupt jelkapcsolatok 30
1.4 Speciális hardware igények 31

2. Software 33
2.1 Program-modellek 33
2.2 Párhuzamos programok jelenségei 40
2.3 Szinkronizáció 44
2.4 A multimikroszámítógép-rendszer használatának kiterjesztése 50

3. Irodalom 55

II. rész: Konverzió szekvenciális és párhuzamos programok között

1. Paralel programok szekvenciális kompozíciója .. 60
1.1 Multiprogramozás 60
1.2 Példák kompozícióra - multiprogramozás nélkül 64

2. Szekvenciális programok paralel dekompozíciója 69
2.1 Dekompozíció tördeléssel 69
2.2 Dekompozíció átalakítással 77
2.3 Dekompozíció redundáns számításokkal 81

3. Konklúzió 90

I. rész:

Hardware és software

Kivonat

A tanulmány I. részében a multimikroszámítógép-rendszerek hardware és software kérdéseit tárgyaljuk.

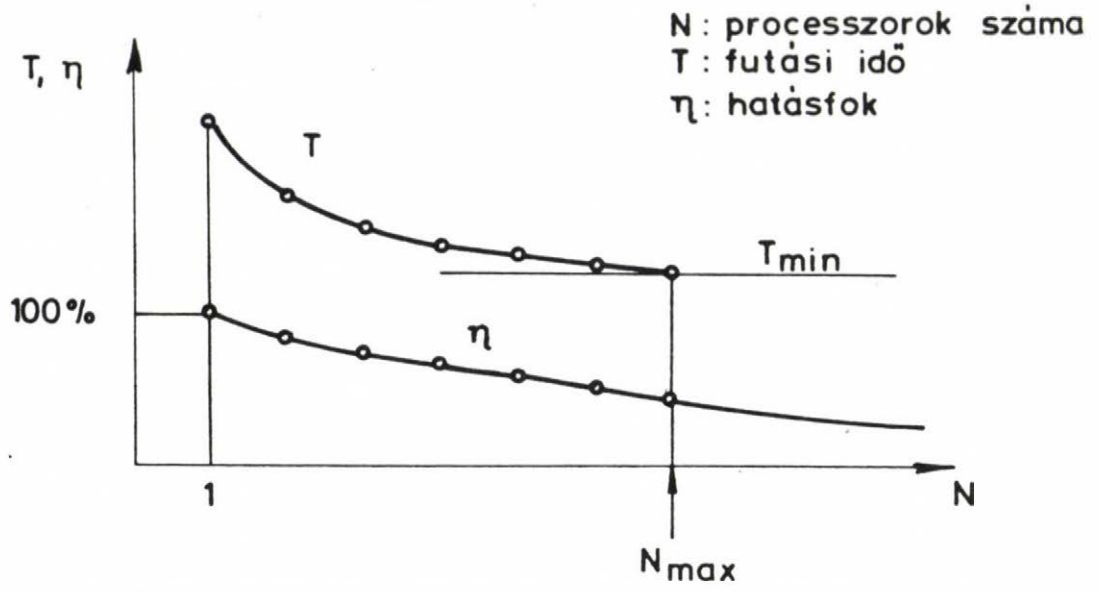
A hardware fejezet a rendszerek osztályozásával, az architektúrákkal, az interrupt jelkapcsolatokkal és a speciális hardware igényekkel foglalkozik. A fejezet hangsúlyos része az architektúrák témája, amelyet fix és programozható architektúrák bontásban tárgyalunk.

A software fejezet a program modellek tárgyalásával kezdődik. Erre épít az I. rész folytatása és a tanulmány II. része. A software téma kérdései között a program jelenségekkel, a szinkronizációval és a rendszer alkalmazásának kiterjesztésével foglalkozunk.

A tanulmány első részében a multimikroszámítógép-rendszerek hardware és software kérdéseit tárgyaljuk. A téma a digitális elektronika egyik aktuális témája; számos folyóiratcikk és egy, a közelmúltban megrendezett konferencia anyaga [1], foglalkoznak vele. A terület még nem teljesen kiforrott, van lehetőség újabb gondolatok hozzáadására.

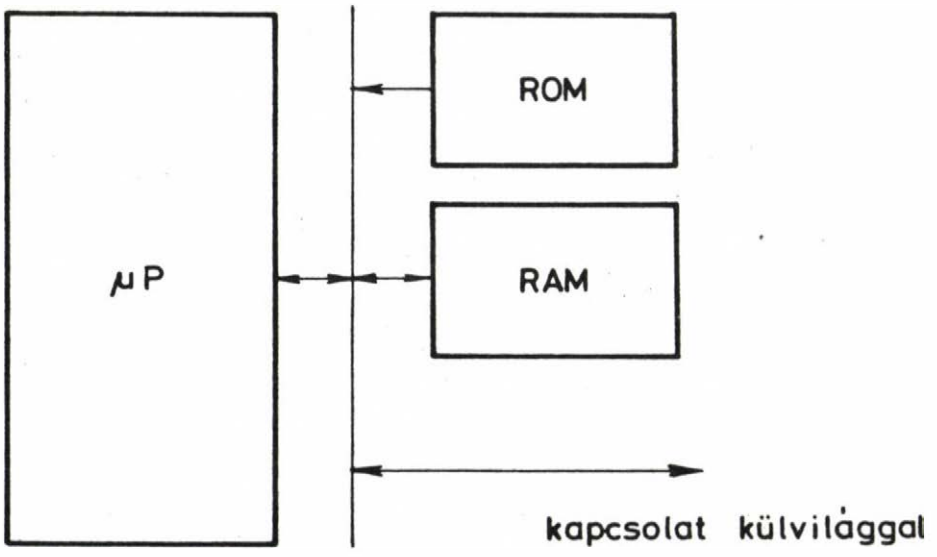
A multimikroszámítógép-rendszerek hardware és software kérdései összefüggnek egymással. Kérdés, hogy melyik területet tekintjük elsődlegesnek. A hardware megszabja, hogy mit és hogyan programozhatunk; a software bizonyos hardware felépítést igényel. A hagyományos megközelítés szerint a hardware konfiguráció adott, és erre kell programokat írni. A modernebb megközelítésben - elkerülve a másik alternatívát, tehát, hogy a program adott - azt vizsgáljuk, hogy a feladat adott, és milyen hardware/software eszközökkel lehet azt optimálisan megoldani. (Ez a programozható architektúrák témájához vezet, amely az irodalomban még kevéssé tárgyalt.)

A multimikroszámítógép-rendszer legegyszerűbb paramétere a benne szereplő processzorok száma. Egy feladat megoldásának optimalitását vizsgálhatjuk a processzorok számának függvényében. Két vizsgálható változó: a feladatot megoldó paralel program futási ideje, ill. hatásfoka. (Hatásfokon itt a processzorokon mért hasznos futásidő és teljes futásidő hányadosainak átlagát értjük.) A két változó alakulását - adott feladat esetén - az 1. ábra mutatja be. A futási idő a program paralelizáltságának növekedtével csökken (egy elképzelhető maximális pro-



Véges számitási feladatok parallelizálása

1. ábra



Mikroszámitógép modell

2. ábra

cesszor számig), a hatásfok - a várakozási idők és az overhead munkák növekedtével - sajnos úgyszintén. Ezért egy feladat paralel megoldásának processzorok számában kifejezhető optimuma van (amely speciális esetben a konvencionális szekvenciális programokat preferálhatja).

A multimikroszámítógép-rendszer méretei - a mikroprocesszorok számát tekintve - tág határok között változhatnak. Ipari automatizálásra kifejlesztett rendszerekben rendszerint kevés mikroprocesszor van (az Intézetben kidolgozott GD80 grafikus display 5, a DIALOG CNC-M szerszámgép vezérlő berendezés 5, a BARETEST nyomtatott áramköri kártyákat mérő berendezés 2 mikroprocesszort tartalmaz). Általános számítástechnikai célokra kifejlesztett konfigurációkban néhány száz vagy ezer mikroprocesszor is lehet. Mi a cikkben a közepes számú mikroprocesszort tartalmazó rendszereket preferáljuk.

1. Hardware

A multimikroszámítógép-rendszer építőkövei a mikroszámítógépek, amelyek a következő funkcionális elemekből épülnek fel: mikroprocesszor, ROM és RAM. A mikroszámítógép kapcsolatát a külvilágával a (kétirányú adatforgalmat lebonyolító) adatbus és az (egyirányú) címbus biztosítja. (A mikroszámítógép blokkvázlatát a 2. ábra tünteti fel.) Speciális esetben a mikroszámítógép memóriája elmarad (helyette önálló egységként jelenik meg), és a mikroszámítógép-rendszer átmegy mikroprocesszor rendszerbe.

1.1 A rendszerek osztályozása

A rendszerek hardware alapú osztályozása több szempontból történhet, azonban ez nem pótolja a rendszer architektúrával való jellemzését, amit később adunk meg.

A rendszerek - céljuk szerint - lehetnek speciálisak vagy általános rendeltetésűek. A speciális rendszerek rendszerint ipari automatizálásra készülnek és kisebbek mint az általános rendeltetésű rendszerek, amelyek általános számítási feladatok megoldására szolgálnak. A speciális rendszerek egy vagy több feladat végrehajtására szolgálhatnak. Az egy és állandó feladatot megoldó rendszerek programjait a mikroszámítógépek ROM-okban tárolják. A több feladatot megoldó rendszerek programjait RAM-okba töltjük.

Az alkalmazott mikroszámítógépek szerint a rendszer homogenitásának különböző fokai lehetnek. A mikroszámítógépekbe épített mikroprocesszorok lehetnek azonos vagy eltérő típusúak. Eltérő típusok esetén a mikroprocesszorok lehetnek azonos vagy eltérő szóhosszúságúak. Az általános rendeltetésű rendszerek rendszerint azonos típusú mikroprocesszorokat tartalmaznak.

A mikroprocesszorok órajelei szerint szinkron és aszinkron rendszerekről beszélhetünk. A szinkron rendszerekben minden mikroprocesszor azonos órajelet kap. Az aszinkron rendszerekben a mikroprocesszorok órajelei egymástól függetlenek (és egyező vagy eltérő frek-

venciájúak). Az aszinkron-rendszerek valamivel gyorsabb működésűek lehetnek, mint a szinkron-rendszerek, de a gyakorlatban kevésbé elterjedtek. (Fokozottabb aszinkronitást jelentene, ha a mikroprocesszorok órajel nélküliek lennének, [2], de ilyen eszközök ma nincsenek piacon.) Az aszinkron-rendszerek két hátránya az alkalmazott órajelek frekvenciájának instabilitásából ered: a) az aszinkron-rendszer nem reprodukálható viselkedésű, b) nincsenek megfelelő eszközeink a frekvencia instabilitás és így a rendszer viselkedésének szimulálására.

A mikroszámítógépekben alkalmazott reset jel lehet szinkron vagy aszinkron. A szinkron reset jel órajellel kapuzott. Az aszinkron reset jelű rendszerek viselkedése nem reprodukálható, mivel a jel hatásosságának időpontjában egy (vagy több) órajelciklus időnyi bizonytalanság lehet.

A rendszereket Flynn 1966-ban a mikroszámítógépek kapcsolatai, pontosabban utasítás- és adatáramlatai szerint osztályozta, [3]. Az utasításáramlatok száma szerint vannak egy (SI) és több (MI) áramlatú rendszerek; az adatáramlatok száma szerint is vannak egy (SD) és több (MD) áramlatú rendszerek. A négy lehetséges kombináció háromra redukálható, mivel a MISD konfigurációnak nincs gyakorlati jelentősége.

A rendszer mikroszámítógépeinek csatolásai szerint vannak lazán és szorosan csatolt rendszerek. A laza csatolású rendszerben nincs, a szoros csatolású rendszerben van közös memória, amelybe a mikroszámítógépek egyaránt írhatnak, és amelyből olvashatnak.

A rendszer az elemek telepítésének fizikai közelsége szerint lehet lokális és földrajzilag elosztott. Az elosztott rendszer mikroszámítógépei között modemek és adatátviteli vonalak teremtenek kapcsolatot.

1.2 Rendszer-architektúrák

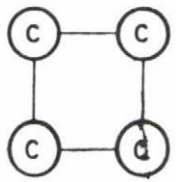
A rendszer-architektúrák a rendszerben szereplő mikroszámítógépek és más elemek kapcsolatait írják le. A rendszer-architektúráknak két alapvető típusuk van: fix és programozható. A fix rendszer-architektúra olyan, hogy jellemzőit a tervezéskor rögzítették, míg a programozható rendszer-architektúra futásidőben, a kiszolgált taskok igényei szerint dinamikusan rekonfigurálható.

1.2.1 Fix rendszer-architektúrák

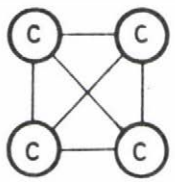
Míg a rendszer legfontosabb elemei, a mikroszámítógépek jól jellemezhetők funkcionális egységeikkel, szerkezetük egységes (2. ábra), addig kapcsolataik és az azokat megvalósító funkcionális elemek változatosak.

a) A rendszerelemek kapcsolatai

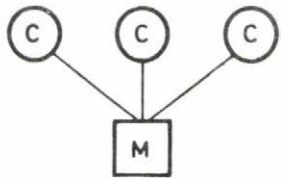
A legismertebb architektúrákat a 3. és 4. ábrák foglalják össze. A 3. ábra a [4] irodalmi forráson alapul. A közölt tíz architektúra reguláris és irreguláris struktúrákat tartalmaz. Általános rendeltetésű rendszer számára természetesen csak reguláris architektúrák jönnek számításba. A 4. ábrát az [5] irodalmi forrásból vettük. Architektúrái mind regulárisak, és elsősorban nagy rendszerek felépítésére szolgálnak. A cikk részletesen elemzi



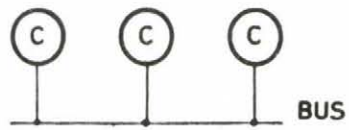
a) DDL: Direct
Dedicated Loop



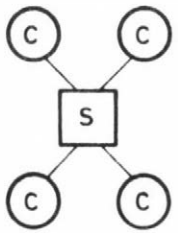
b) DDC: Direct
Dedicated Complete



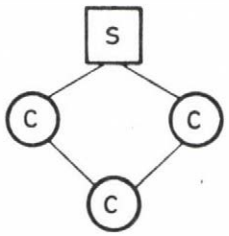
c) DSM: Direct
Shared Memory



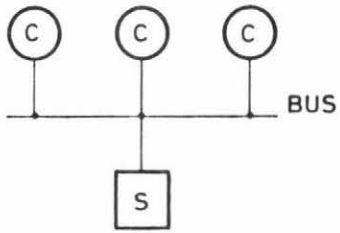
d) DSB: Direct
Shared Bus



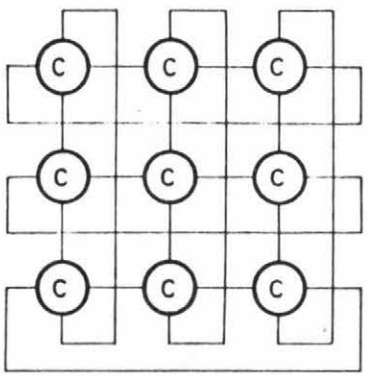
e) ICDS : Indirect
Centralized Dedicated Star



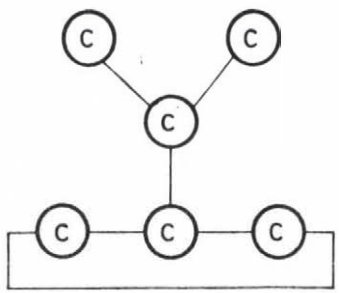
f) ICDL : Indirect
Centralized Dedicated Loop



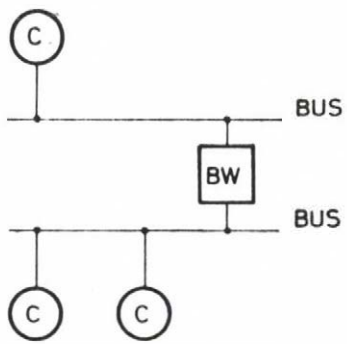
g) ICSB: Indirect
Centralized Shared Bus



h) IDDR: Indirect
Decentralized Dedicated
Regular



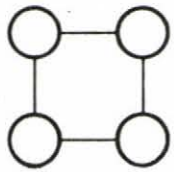
i) IDDI : Indirect
Decentralized Dedicated
Irregular



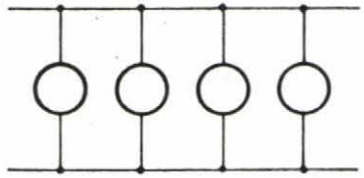
j) IDS: Indirect
Decentralized Shared Bus
- Bus Window

Graphical summary of the ten architecture types of the Anderson and Jensen taxonomy

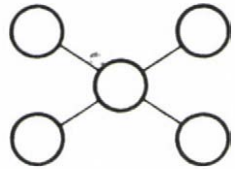
3. ábra



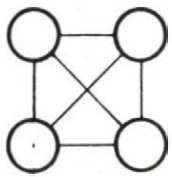
RING



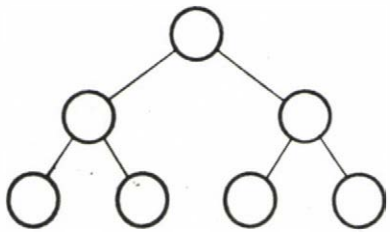
MULTIPLE GLOBAL BUSES



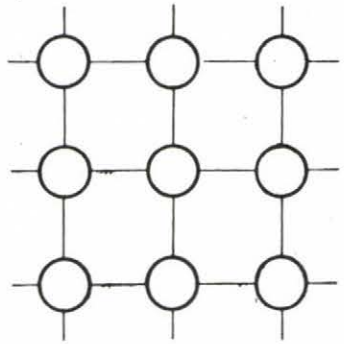
STAR



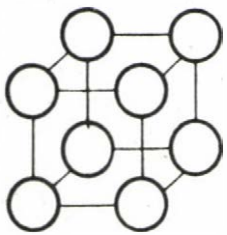
COMPLETELY CONNECTED



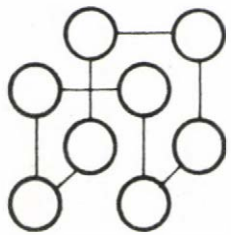
TREE
(B = 2, L = 3)



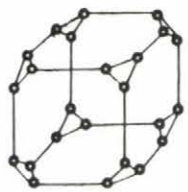
NEAR-NEIGHBOR MESH
(D = 2, W > 3)



SPANNING BUS HYPERCUBE
(D = 3, W = 2)



DUAL-BUS HYPERCUBE
(D = 3, W = 2)



CUBE-CONNECTED-CYCLE
(D = 3)

Examples of connection topologies for multicomputer networks

4. ábra

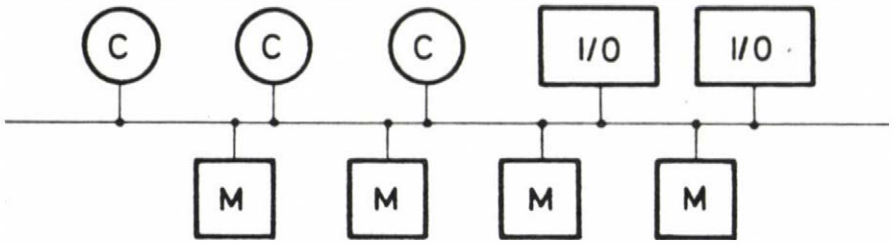
és összehasonlítja a közölt architektúrákat.

További három ismert architektúrát mutat be az 5. ábra, amelyet a [6] irodalmi forrásból vettünk, kis módosítással. Ezek az architektúrák kisebb rendszerekben (köztük több megépített rendszerben) kedveltek.

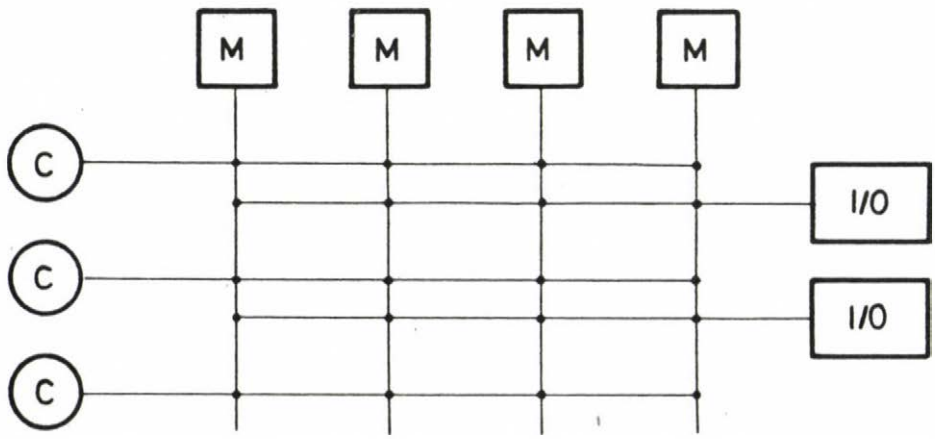
Mint látható a 3.-5. ábrákból, a rendszertervező számára számos, többnyire jól analizált architektúra kínálkozik választásra. A téma azonban nem lezárt: mind nagyobb és nagyobb rendszereket kívánunk építeni, a mikroszámítógépeket összekötő rendszer szerepe jelentős, a méretek növekedésével mennyiségi problémák jelentkeznek. Egy architektúra elfogadásánál számos - többnyire egymás ellen ható - szempontot kell figyelembe venni. Ilyenek például:

- két mikroszámítógép közötti kommunikáció ideje (worst case és átlag);
- lehetséges egyidejű párhuzamos kommunikációk száma;
- a rendszerben az elemek közötti kapcsolatok összes száma;
- két rendszerelem közötti kapcsolat hardware-jének bonyolultsága;
- a rendszer hibatűrő képessége (hibás rendszerelemek söntölhetősége);
- a rendszer bővíthetősége.

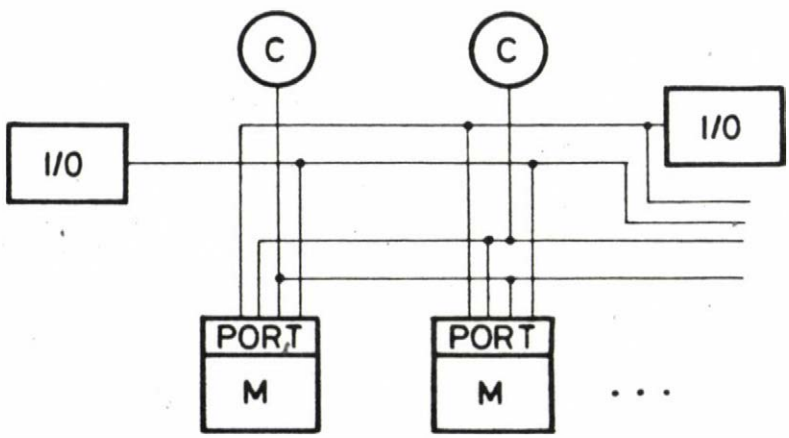
A fenti szempontokat figyelembe véve, egy száz körüli mikroszámítógépet tartalmazó, általános rendeltetésű rendszer felépítésére a szerző két architektúrát tart kedvezőnek, amelyet a 6. ábra mutat be. Mindkét rendszer gyors kommunikációt biztosít, viszonylag kevés hardware-t



a) Közös sinrendszerű számítógép rendszer

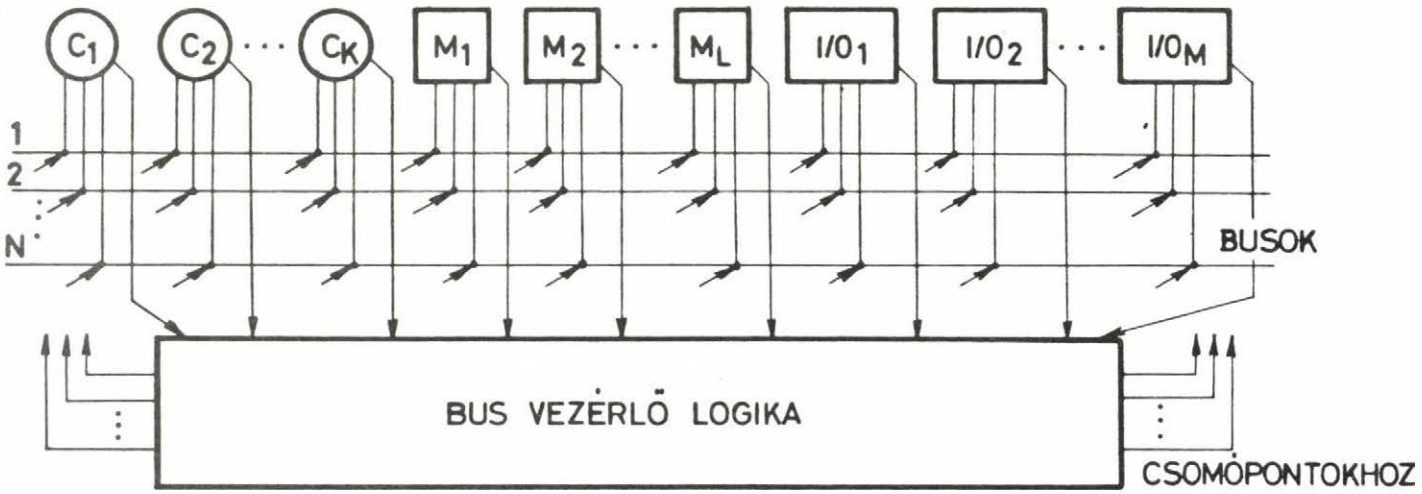


b) Crossbar kapcsolású számítógép rendszer

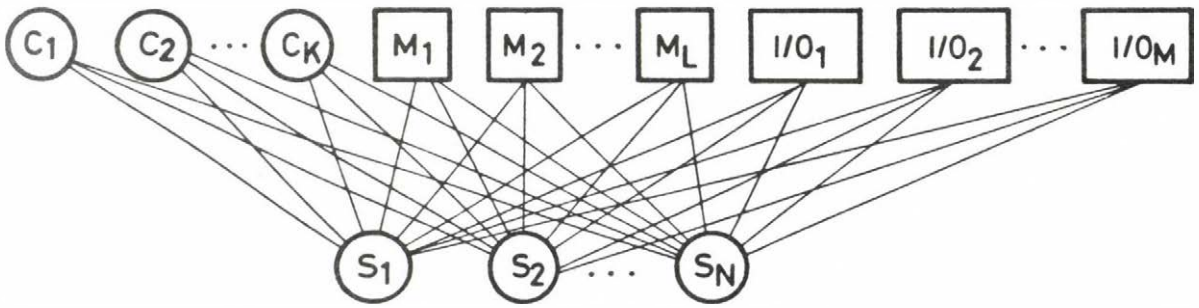


c) Multiport memória rendszerű számítógép rendszer

5. ábra



Multibus architektúra



Multistar architektúra

6. ábra

igényel, egyidejűleg sok paralel kommunikációt biztosít, könnyen bővíthető és jó a hibatűrő képessége. A multibus architektúra a - kisebb rendszerek számára egyébként kitűnő - crossbar architektúra módosítása. Előnye, hogy kevesebb hardware-t igényel, és azt maximálisan kihasználja (míg a crossbar architektúrában a kapcsolatok zöme adott időpontban kihasználatlan). Hátránya, hogy a busok kapcsolására külön vezérlő hardware-t igényel. A multistar architektúra az egyszerű star architektúra módosítása. A star architektúrában egyidejűleg csak egy élő kommunikációs út létezik, a multistar architektúrában viszont annyi, ahány kommunikációs számítógépet (S) alkalmazunk. A multibus architektúrához képest előny, hogy a bus-vezérlő hardware szerepét itt mikroszámítógépek veszik át - bár ez lassítja valamit a kommunikáció sebességét.

Valamennyi ismert architektúra a páronként teljes architektúra (3b. ábra) redukált változataként fogható fel. A redukció rendszerint addig történik, amíg valamilyen egyszerű, de még összefüggő összekötési gráfot nem kapunk. (Speciális esetben az összekötő gráf egy kifeszítő fa, amely - egyirányú adatáramlást engedve meg - hierarchikus rendszert eredményez.)

b) Az összekötő alrendszer

A rendszer architektúrájának vizsgálatakor csak a rendszerelemek kapcsolataira koncentráltunk. Most azt vizsgáljuk, hogy a kapcsolatokat milyen hardware valósíthatja meg. Ezt a témát tárgyalja a [7] irodalmi forrás, amelyből idézünk.

Valamiről azt állítani, hogy az egy összekötési mechanizmus, először is azt kívánja meg, hogy tisztázzuk az a szintet, amelyet használunk. Legalább három szint alkalmazható, éspedig (1) hardware, (2) rendszer funkcionális és (3) software funkcionális.

A hardware szintjén a mikroszámítógép-rendszerekben a rendszerelemek összekötésére csak négy primér mechanizmus létezik: (1) bit soros átvitel egy soros kapcsolaton keresztül, egy processzor vezérlése segítségével, (2) szó párhuzamos átvitel egy párhuzamos kapcsolaton keresztül, egy processzor vezérlése segítségével, (3) szó párhuzamos átvitel egy buson (azaz egy mikroszámítógép belső busán), és (4) szó párhuzamos átvitel egy közös memórián keresztül (az I/O eszközöket itt mint közös memóriát tekinthetjük).

A hardware összekötő mechanizmusokat elemekre particionálhatjuk, amelyek legnagyobb lebontás esetén hardware összekötő primitíveket (HIP) eredményeznek. Példaképpen egy mikroszámítógép alapú rendszer számára a HIP elemek következő halmazát definiálhatjuk:

- mikroszámítógép bus (MB): primer, csomóponton belüli kapcsolat a mikroprocesszor és a csomóponton belüli minden más funkcionális elem között;
- mikroszámítógép bus ablak (MBW): elektronikus leképzés a mikroszámítógép busok között;
- közös memória (SM): olyan memória, amelyet egynél több mikroszámítógép vagy mikroprocesszor érhet el;
- közös mikroszámítógép bus (SMB): belső bus, amelyet dinamikus időosztással két vagy több mikroprocesszor

használhat;

- közös kommunikációs bus (SCB): külső bus, amelyet két vagy több mikroszámítógép megosztva használ csomópontok közötti kommunikációra;
- soros kapcsolat (LS): bit soros interface, amelynek a műveleteit egy mikroprocesszor vezérli. Általában csak információátvitelre használják;
- paralel kapcsolat (LP): szó párhuzamos interface, amelynek a műveleteit egy mikroprocesszor vezérli. Egyszerű üzenetátvitelre használható, vagy mint egy vezérlő mechanizmus része.
- átviteli processzor (PT): olyan mikroprocesszor, amely az átviteli réteg feladatát látja el egy csomóponton belül vagy csomópontok között. Az átviteli processzor két típusának funkcionális megkülönböztetése szükséges, ha a csomóponton belüli és a csomópontok közötti átviteli műveletek alapvetően különböznek.
- DMA vezérlő (KDMA): csomóponton belüli DMA átvitelt valósít meg.

A HIP-ek fenti halmazát a szerző két elemmel látja szükségesnek gyarapítani: a kommunikációs pufferral és az adatszelektorral.

A kommunikációs pufferra a mikroszámítógépek közötti adattranszferek során szinte mindig szükség van, akár busolt, akár közvetlen (csomóponttól-csomópontig) adatkapcsolatról van szó. A 2. ábrában megadott mikroszámítógép-modell szerinti mikroszámítógépek közvetlen (puffer nélküli) busolása három okból nem tartható szerencsésnek:

- az adatforrás és adatfelhasználó mikroszámítógépek

rendszerint egymástól függetlenül dolgoznak, így nem garantált, hogy ugyanabban az időpontban készek a transzferra;
- az adattranszferra való várakozás alatt mindkét mikroszámítógép dolgozhatna a maga programján;
- forrásként egyidejűleg csak egy mikroszámítógép birtokolhatja a bust, a többi mikroszámítógépnek eközben fel kell függesztenie a működését, legalábbis a bus felé.

A problémákra megoldás, ha mikroszámítógépek pufferon keresztül csatlakoznak az összekötő busra. Hasonlóképpen, a csomóponttól csomópontig terjedő közvetlen adatátvitelnél a forrás és felhasználó mikroszámítógépek eltérő időben készek az adatforgalomra. Az átmeneti adattárolást itt is pufferral oldhatjuk meg. (A fentiek képzik a "mikroszámítógépek pufferon keresztüli kapcsolatát", amelyre a software-rel foglalkozó 2. fejezetben hivatkozunk.)

Adatszelektorra minden olyan esetben szükség lehet, amikor több forrás küldhet adatot egy felhasználónak. (Ez egy alternatíva az átviteli processzor alkalmazása mellett. Több hardware-t igényel, de gyorsabb kommunikációt biztosít.) Az adatszelektor alkalmazható mikroszámítógép, közös memória vagy output egység, mint felhasználó mellett. Az adatszelektor komplex egység, mert több adatforrás esetén a prioritás-probléma is felmerül, és az egységnek ezt is meg kell oldania. (Az egyszerű, hardware-be épített prioritásnál bonyolultabb a programozható prioritás, amit kezelni kell tudni. Még általánosabb

a "lebegő prioritás", ahol egy rendszerelem prioritását futás közben átprogramozzuk. Erre példát a 2. Software részben láthatunk.)

1.2.2 Programozható rendszer-architektúrák

A programozható rendszer-architektúrák olyan architektúrák, amelyeknek egy-egy állapota a már ismertetett fix architektúrák valamelyike. A programozhatóságnak két elsődleges fajtája lehet: mindkettő azt szolgálja, hogy az architektúra lehető legjobban megfeleljen a megoldandó feladat természetének, [8]:

a) a rendszer alapját képző, homogén (mondjuk 16 bites) mikroszámítógépeket csoportokba foglaljuk, hogy nagyobb szóhosszúságú (32,48,64...bites) komplex mikroszámítógépeket képezzenek, a megoldandó feladatok precizitás igényei szerint;

b) a mikroszámítógépeket úgy csatoljuk, hogy az a feladathoz közel álló, hatékony architektúrát biztosítson (tömb-elrendezés, pipeline,...)

A programozhatóság biztosítása elsősorban a rendszerelemeket összekötő alrendszert érinti. Csak olyan architekturális állapot programozható be a rendszerbe, amelyhez a hardware háttér létezik - egy programozható architektúra tehát gazdag összekötő alrendszert igényel. Ez bizonyos mértékben ellentmond a nagy rendszerek építése igényének, ugyanis mint azt a fix architektúráknál láttuk, nagy rendszert viszonylag kevés, de jól megalapozott rendszerelemek közötti kapcsolattal lehet építeni.

A programozhatóság nem áll messze a fix architektúráktól, ugyanis azok összekötő alrendszerét az adattranszferek során rendszerint programozni kell. Például a star architektúra S eleme számára meg kell mondani, hogy mely mikroszámítógép lesz az adatok felhasználója, azaz utat kell definiálnunk két csomópont között.

Egy programozott architektúra előnye abban áll, hogy a programozási és az adat jellegű információt szétválasztjuk. Programozási információt egyszer (vagy ritkán) kell közölni egy feladat futtatásakor; ez beállítja az adatutakat, és azokon az adatok rendeltetési helyükre áramlanak. (Ez természetesen nem zárja ki, hogy egy programozott architektúrában egy adatforrás mikroszámítógépnek felhasználót kelljen választani.)

Nézzük most meg közelebbről a programozhatóság fenti a) és b) fajtáját.

A szóhosszt bővítő programozhatóságnak következményei vannak a rendszerelemeket illetően. Ezt a programozhatóságot úgy képzelhetjük el, hogy

- két vagy több mikroszámítógépen ugyanaz a program fut, egymással szinkronban;
- egy-egy mikroszámítógép a bővített szóhossz egy-egy szegmensét kezeli;
- minden mikroszámítógép programjában a teljes hosszúságú címek szerepelnek (tehát egy összetevő mikroszámítógépnek meg kell tudni címezni a komplex egység teljes memória mezejét);

- feltételes utasítás végrehajtásakor az összetevő mikroszámítógépek egyes feltételi jeleiből eredőt kell képezni, és az eredőt kell egyformán érvényesíteni a mikroszámítógépekben.

Az eredő feltételi jel képzése a zero, az előjel, a carry és az overflow jeleken valósul meg. Ezek a mikroprocesszorok státusz jelei. Problémát okoz, hogy ezek a jelek a kereskedelmi mikroprocesszorokon rendszerint kívülről nem hozzáférhetőek - így az eredő képzése nem megvalósítható. Ugyanígy, a shift utasításoknál az akkumulátor tartalmak legkisebb, ill. legnagyobb helyértékű bitjeinek hozzáférhetőségére lenne szükség a szóhosszbővítéshez.

Ha a státusz jelek és az akkumulátor LSB,MSB jelek hozzáférhetőek, akkor az összekötő alrendszernek a megfelelő helyre kell őket kapcsolnia. Általános rendszert tételezve fel, ezeknek a kapcsolatoknak is programozhatóknak kell lenniük. A carry jellel külön probléma, hogy még külön alkatelemeket (look-ahead carry generátorokat) is kezelni kell az összekötő alrendszernek.

A szóhossz-bővítésnek a közös memória blokkok címző hardware-jére is kihatása van. A memória blokkokat ugyanúgy csoportba fogjuk össze, mint a mikroszámítógépeket. Az összefogott blokk egy címére minden összetevő blokknak szimultán kell címződnie - azonban minden összetevő blokk i-edik szava más címre aktiválódik. Ezen úgy segíthetünk, hogy minden összetevő blokk címzőjel vektorára egy összeadót helyezünk el, amelyet egy programozható kezdőcímmel előfeszítünk. (A kezdő-

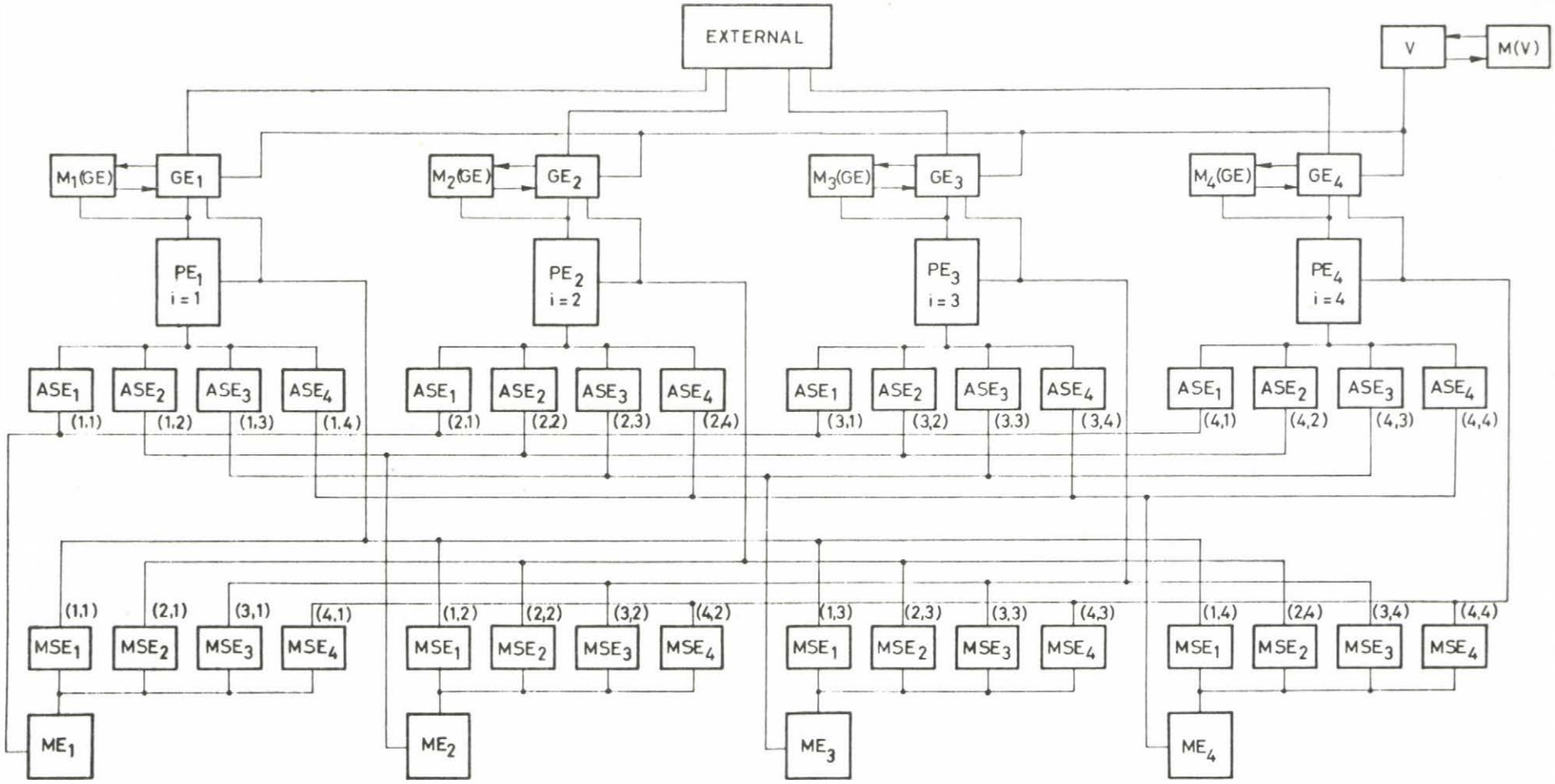
címeket egyszer, az architektúra definiálásakor kell megadnunk.) Így az összefogott memória blokkot például a legkisebb című összetevő blokk címeivel aktiválhatjuk. (Elkerülhetjük az összeadók használatát, ha a mikroszámítógépek programjaiban eltérő, azaz az összetevő memória blokkoknak megfelelő címeket használunk.)

Mind az a), mind a b) fajtájú programozhatóság realizálására a [8] irodalmi forrás egy új építőelemet, a "Universal Dynamic Computer" (UDC) csoportot javasolja. Ezt ismertetjük itt röviden. Az UDC csoport diagramját a 7. ábra mutatja be.

Az UDC csoportot n darab CE számítógépelem, rekonfigurálható memóriaprocesszor bus és M(V) memóriával ellátott V-monitor alkotja. Mindegyik CE 16 bites szavakat dolgoz fel párhuzamosan, és egy PE processzor elemet, egy ME memóriaelemet, valamint egy GE I/O elemet tartalmaz, amelyek mind 16 bit szóhosszúságúak. Mindegyik GE-nek egy kis M(GE) memóriája van, amelynek a szóhosszúsága ugyanannyi.

A rendszer C(k) változó méretű számítógépeket (processzorokat) képezhet. C(k)-ban a k egész szám 1-től n-ig változhat.

A rekonfigurálható memória-processzor bus kétféle összekötő modult tartalmaz: cím összekötő elemeket (ASE) - amelyek memóriacímet és READ, ill. WRITE jeleket közvetítenek PE-től ME-hez -, és információs összekötő ele-



Hardware diagram for the UDC group

7. ábra

meket (MSE) - amelyek h bit hosszúságú byte-okat váltanak ME és PE között.

Egy memória elem címzése a következőképpen történik. A PE egy SEL aktivációs kódot küld az ASE-jainak (ez a csatorna az ábrában nem szerepel), és amelyik ASE aktiválódik, azon a cím és a READ,WRITE jelek az ME-hez áramlanak.

Egy 16 bites byte váltás egy PE és ME pár között az MSE-n keresztül történik. Az MSE elem szelektív aktiválása a READ jellel (w_1) történik, ha ME-ből olvasunk, vagy a WRITE jellel (w_2), ha ME-be írunk. Ezt a két jelet az ASE-ből nyerjük, amely címet és w-ket közvetít ehhez az ME-hez. Ebből az következik, hogy a szelektív aktiváláshoz mindegyik ASE_j-t (amely címet közvetít ME_j-hez) össze kell kötni mindegyik MSE_i-vel (amely h bites byte-ot közvetít PE_i és ME_j között), egy kétvezetékes kapcsolattal (ahol egy-egy kapcsolat w_1 és w_2 jelekhez tartozik). A memória-processzor busnak tartalmaznia kell az összes (i,j) kapcsolatot.

Az UDC csoportban a multiszámítógép és a multiprocesszor architektúrák összeolvadnak, mivel minden funkcionális modul (PE,GE,ME) összeköthető minden más funkcionális modullal, az I/O eszközök használata nélkül. Így például a memória-processzor bus paralel kommunikációt tesz lehetővé processzorok, processzorok és memóriák, és egyik számítógép processzora, valamint másik számítógép primer memóriája között.

Az UDC csoport (egy vagy több) tömbprocesszorként is

funkcionálhat. Minden tömbben egy 16k bites processzor látja el a supervisor processzor feladatát, amely lokális memóriájából hozza az utasításokat, és azokat továbbadja azoknak a processzoroknak, amelyek a tömbben dolgoznak.

Az UDC csoport dinamikus pipeline architektúrát is felvehet, amely több szempontból is adaptálódik a végrehajtott algoritmushoz. Jelenleg [8] szerzői azon dolgoznak, hogy architekturális átmeneteket találjanak pipeline-ról más architektúrákra (tömb, multiszámítógép, multiprocesszor), amelyeket UDC csoportok valósítanak meg.

Az UDC csoport koncepcióját a jelen cikk szerzője a következőképpen értékeli:

- az UDC csoport voltaképpen egy multistar konfiguráció, amelynek terminális csomópontjai processzorok, belső csomópontjai pedig memóriák;
- ha a processzorok száma n, akkor az UDC csoport n^2 számú összekötő logikát igényel. Ez kedvezőtlenné teszi az alkalmazását nagy rendszerekben;
- az UDC csoportban kell, hogy legyen még prioritás logika (több PE fordul egy ME-hez adatért) és felfüggesztő logika (egy ME bemenetén keresztül kapcsolódik össze két PE, és egy harmadik PE ugyanehhez az ME-hez fordul utasításért). Ezek nem szerepelnek a 6. ábrában, és tovább fokozzák az UDC csoport hardware-jének bonyolultságát;
- n kis értékei mellett az UDC csoport realizálható, és architekturálisan igen flexibilis.

1.3 Interrupt jelkapcsolatok

A rendszerelemek eddig tárgyalt (adat- és címjeles) kapcsolatai mellett lehetőség van egy interrupt jeles kapcsolatrendszer kiépítésére is. Az interrupt jeleket - szokványos alkalmazásban - I/O eszközök küldik mikroprocesszoroknak. Mi itt arra hívjuk fel a figyelmet, hogy interrupt jelekkel mikroprocesszorok (mikroszámítógépek) kommunikálhatnak egymással, és ez az információs kapcsolat jól kiegészítheti az adat- és címjeles kapcsolatokat.

A mikroszámítógépek közötti adat- és címjeles kommunikáció hardware eszközei a pufferok és a közös memória. Ezekbe az adatforrás mikroszámítógép akkor ír, ill. a felhasználó mikroszámítógép ezekből akkor olvas, amikor azt saját (rendszerint csak kommunikációval korrelált) programjuk előírja - és ez gyakran kommunikációs várakozáshoz vezet. Gyorsítaná a kommunikációt (azaz csökkentené a várakozási időket), ha az adatforrás mikroszámítógép, amint adatát generálta, értesítené a felhasználó mikroszámítógépet az "adat kész" állapotról, és az az adatot rögtön a lokális memóriájába vinné. Ezt az értesítést közvetítheti az interrupt jel. Természetesen így csak az adatforrás várakozásának csökkentésére van mód.

Az interrupt jeles kapcsolatrendszernek kiterjedtnek kell lennie (minden mikroszámítógépnek kell tudnia interruptot küldenie minden más mikroszámítógép számára). Ez kiterjedt hardware alapot, interrupt összekötő alrendszert igényel, ami a rendszer komplexitását - mint

a működési gyorsaság növekedésének az ára - növeli. Az interrupt jelekkel kapcsolatban is felmerül a prioritás kérdése, amit az adatjeles prioritással összhangban meg kell oldani, és ami külön hardware hátteret igényel.

Az interrupt jel megérkezésekor a mikroszámítógép a programjában egy interrupt kiszolgáló rutinra ugrat. A futó program abbahagyása, az interrupt kezelése overhead munkát igényel, a maga időszükségletével, ami ha kicsi, megtérül az adatkommunikációs hardware várakozástól való felszabadítása által.

A mikroprocesszoroknak az interrupt jel fogadására rendszerint csak egy-két kapocspontjuk van, és azokon keresztül sincs bit-soros feldolgozás. Ezzel szemben az interruptos kapcsolatrendszer nagy mennyiségű információt szállít: tudatja az interrupt forrását és az okát (amely más is lehet, mint adatküldés jelzése). Következésképpen az interruptos információt az adat- és címjeles pályákon kell a mikroszámítógépbe bejuttatni, azaz a kétféle kapcsolatrendszer össze kell hogy fonódjon.

1.4 Speciális hardware igények

A multimikroszámítógép-rendszerben való alkalmazás a konvencionális alkatelemek (mikroprocesszorok, memóriák) iránt speciális igényeket is támaszt. Itt azokat az igényeket gyűjtöttük össze, amelyeket a kereskedelemben kapható alkatelemek nem elégítenek ki.

a) Több mikroprocesszor közös regisztermezővel. Az ilyen

alkatelem a sok változóval kommunikáló programok végrehajtását, így elsősorban a bonyolult kifejezések értékelését és értékadó utasítások végrehajtását könnyítené meg. Az alkatelemben a regiszterek mindegyikéhez mindegyik mikroprocesszor hozzáférhetne, bármelyik írhatna vagy olvashatna (természetesen az egy regiszterbe való többes beírást kizárva). Ennek a megoldásnak gátja a tokonkénti kapocspontszám limitáltsága, amely megakadályozza a többes integrálást. Elképzelhető lenne a mikroprocesszorok adat- és címjeleinek sorosítása melletti integrálás, de ez nagyon lassítaná a működést.

b) Több helyről egyidejűleg címezhető memória. Az ilyen eszköz a közös memóriákban kaphatna elsősorban alkalmazást. Ilyen alkatelemet nem gyártanak; helyette multiport áramkörök választhatnak ki egyidejűleg egy aktív kommunikáló (író vagy olvasó) eszközt. (Egy ilyen alkatelem csírája található meg az Am2900 típusú mikroprocesszor belső regiszter file-jában, amely egy csatornán írható, de egyidejűleg két csatornán olvasható.) Alternatív megoldás lehet, amely a gyakorlatban is kivitelezhető: blokkokra (külön tokokra) particionáljuk a memóriát, amelynek blokkjai már egyidejűleg címezhetők.

c) FIFO memória. A párhuzamos, szinkronizált programok (ld. 2. Software fejezet) működésének gyorsítására a közös memóriában software eszközökkel FIFO memóriát képezhetünk ki. Még gyorsabb megoldást képezne, ha olyan memória tokok állnának rendelkezésre, amelyek minden címével egy-egy FIFO memória volna elérhető.

2. Software

A multimikroszámítógép-rendszer paralel számítások végzésére alkalmas. Software-je egyrészt támaszkodik az uniprocesszoros gépek programjaira (szekvenciális programok), másrészt sajátos új software fogalmakat (csatolás, várakozás, szinkronizáció stb.) kell, hogy számításba vegyen.

A paralel programok tárgyalásához nyelvet kell választanunk. A [9] irodalmi munka a "Concurrent Pascal" nyelvet és a gépfüggetlen megközelítést választotta. Mi a cikkben a magasszintű eszközök helyett a közép szintű modelleket, és a részleges gépfüggőséget részesítjük előnyben. Ennek előnye a paralelitások árnyaltabb kezelhetősége, hátránya viszont a nehezebb programozhatóság.

2.1 Program-modellek

A paralel programok egymással kommunikációt folytató szekvenciális programokból állnak. Modelljeiket megalkothatjuk, ha a szekvenciális programokat és a kommunikációt modellezni tudjuk. Mi a cikk ezen részében csak adatkommunikációt veszünk figyelembe.

A továbbiakban a programokat közép szintű nyelven megírtaknak képzeljük. E nyelv előnye, hogy segítségével minden programozható, ami magas szintű nyelven programozható. A nyelv megengedett eszközei:

- változók: egész típusú változók; indexes változók;
- utasítások: START,STOP; aritmetikai értékadó utasítás; (két vagy három elágazásos) aritmetikai IF utasítás; GO TO; READ, WRITE.

Programmodelljeinket két szinten írjuk le, amelyek kiegészítik egymást. A makroszkópikusabb modell a programgráf, a mikroszkópikusabb a csomópont-gráf.

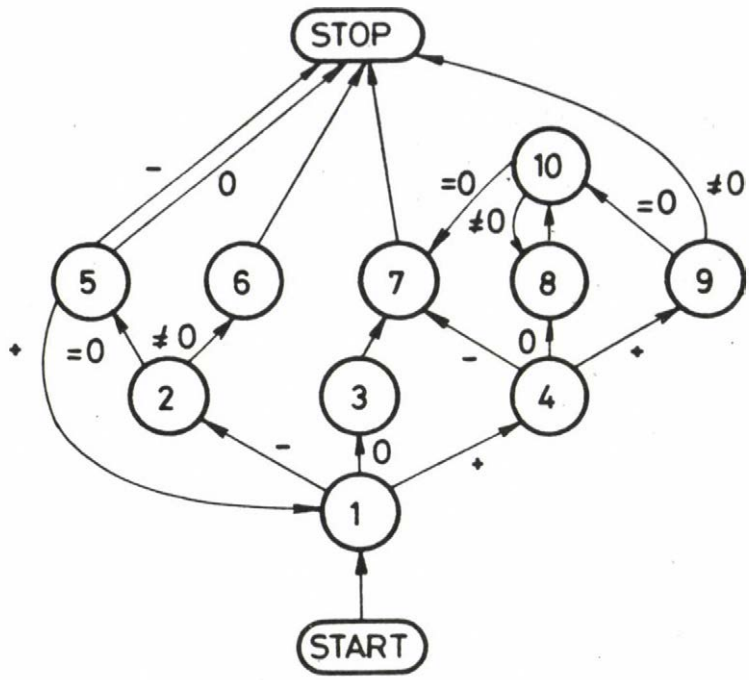
a) Szekvenciális programgráf

A szekvenciális programgráf egy általános irányított gráf, amelynek
- a csomópontjai: aritmetikai értékadó, READ és WRITE utasítások sorozatai, egyetlen IF vagy GO TO utasítással lezárva; START,STOP csomópontok;
- az élei: a vezérlésátadás útjai.

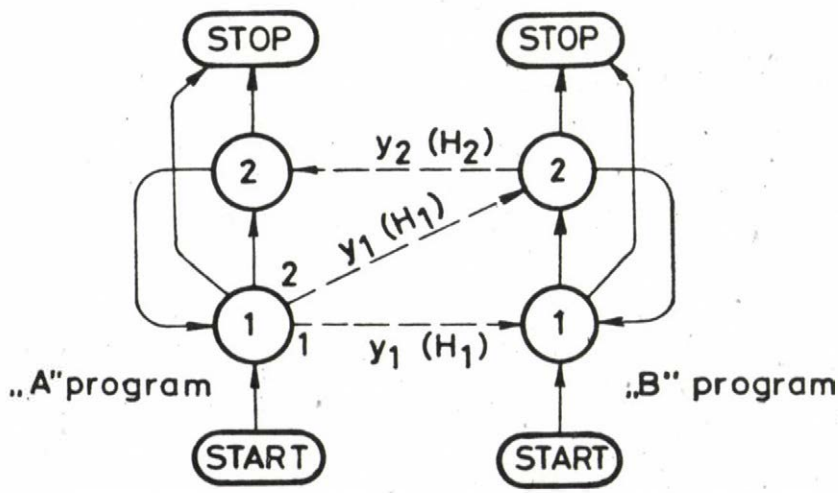
Ennek megfelelően
- a gráfnak van egy START csomópontjai és nulla vagy egy STOP csomópontja (ezek a gráf terminális csomópontjai),
- egy belső csomópont kimenő fokszáma 1,2 vagy 3 lehet,
- egy belső csomópont bemenő fokszáma 1 vagy több lehet,
- ha egy belső csomópont kimenő fokszáma 1, akkor annak a csomópontnak, amelyre mutat, a bemenő fokszáma legalább 2,
- a csomópontokból kimenő élek (ahol 2 van) =0/≠0, (ahol 3 van) 0/+/- jelek valamelyikével vannak cimkézve.

A szekvenciális programgráfra példát a 8a. ábra mutat.

A szekvenciális programgráf hurkairól nem tudjuk, hogy hányszor kerülnek végrehajtásra. (Erre választ csak a teljes program alapos analízise adhat.) Paralel számítási környezetben természetes jelenség a végtelen ciklus - amely számítási feladatokban hiba lenne, de



a) Szekvenciális programgráf



b) Parallel programgráf

8. ábra

folyamatszabályozási feladatban előfordulhat. (Ennek megfelelően programokról és folyamatokról, mint szinonímákról beszélünk a cikkben.) A folyamatszabályozási végtelen ciklus lehet

- hasznos: ha WRITE (és READ) utasítás van benne; (ha csak WRITE utasítás van benne, akkor folyamatvezérlésről, ha READ utasítás is van benne, akkor folyamatszabályozásról van szó),
- haszontalan: nincs WRITE utasítás benne.

A szekvenciális programgráf egy szekvenciális hálózatként fogható fel:

- a program mindig egyetlen állapotban van, egy állapot: egy csomópont,
- a csomópontokból kifutó élek cimkéi a vezérlő jelek (ezek azonban nem primer jelek, mivel a csomópontok számítják ki őket, beolvasott adataikból).

b) Paralel programgráf

A paralel programgráf szekvenciális programgráfok rendszere, amelyben a programok közötti kommunikációt is feltüntetjük. A kommunikáció eszközei a globális változók, amelyekkel egy program adatot küld egy vagy több másik programnak. (Ezzel szemben lokális változók azok, amelynek adatát csak az a program hasznosítja, amelyik definiálta.)

A paralel programgráf csatolásait szaggatott vonallal jelöljük, és a vonalakat a globális változó azonosítójával és kiegészítő információval cimkézzük. A kiegészítő információ:

- a forrásprogram adott csomópontjából hányadszor küldünk (különböző) adatot a globális változóval,
- a kommunikáció milyen hardware eszközöket vesz igénybe.

(Egyszerűsített megadásnál a hardware eszközök megadását el is hagyhatjuk.)

A paralel programgráfra példát a 8b. ábra mutat.

A paralel programgráf csatolásai általában nem lehetnek tetszőlegesek - egy megadott adatkommunikációnak rendszerint következményei vannak (ld. 2.3 Szinkronizáció, g) pont).

c) Csomópont logikai modelljei

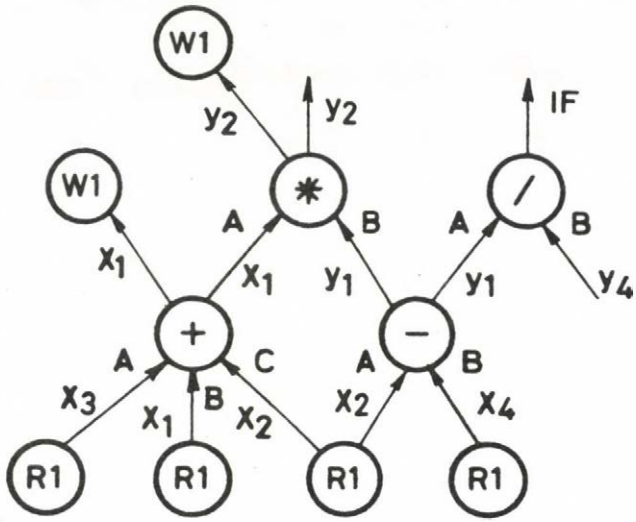
A szekvenciális és a paralel programgráfok csomópontjainak logikai modelljét egy olyan hurokmentes irányított gráffal írhatjuk le, amelynek

- csomópontjait az értékadó és IF utasításokban szereplő műveleti jelek vagy READ,WRITE jelek cimkézik;
- éleit változó azonosítók vagy IF jel cimkézik;
- a csomópontjaiba befutó éleket azonosítóval láttuk el (A,B,C stb...)

Míg a programgráf szekvenciális hálózattal volt ekvivalens, a csomópontgráf többszintű, többértékű kombinációs hálózattal adható meg.

A csomópont logikai modelljére példát a 9. ábra mutat be, ahol az adott gráf a következő programot írja le:

READ(1) x_1
READ(1) x_2
READ(1) x_3



Csomópont logikai modellje

9. ábra

T	Művelet	Változó azonositója
T_1	R	x_1
T_2	R	x_2
T_3	R	x_3
T_4	W	y_2
T_5	A	y_3
T_6	W	y_3
T_7	E	-

R: READ
W: WRITE
A: Assignment
E: Ending

Csomópont időbeli modellje

10 ábra

READ(1) x_4
$x_1=x_1+x_2+x_3$
$y_1=x_2-x_4$
$y_2=x_1*y_2$
WRITE(1) y_2
IF(y_1/y_4)...,...,...

(A példában a READ és WRITE utasítások zárójeles paraméterei a hardware eszközöket azonosítják.)

d) Csomópont időbeli modellje

A programgráf csomópontjaihoz tartozó programrészletek lineáris lefutásúak (maximum egy IF utasítást tartalmaznak, a programrészlet végén). Így az egyes utasítások végrehajtásának időpontjai a csomópont programrészlete megkezdésének időpontjához képest egyértelműen meghatározhatók. A csomópont időbeli modellje a programok szimulációja szempontjából érdekes utasítások végrehajtásának időpontjait adja meg. Ilyen utasítások: a záró utasítás, a periféria kezelő utasítások (READ,WRITE), és az adatkommunikációs utasítások (értékadás csatoló változóknak). A teljes program szimulációjához természetesen ismerni kell a perifériák működésének időparamétereit, a csatoló hardware mechanizmusokat és időparamétereiket.

Egy csomópont időbeli modelljét a 10. ábrában adtuk meg.

A programgráffal és a csomópont logikai modelljével kezelhetők az olyan makroszkópikus utasítások is, mint a ciklus utasítás, a szubrutin vagy function hívás. A ciklus utasítást a középszintű reprezentációban értékadó és IF utasításokra bontjuk le (I=1,...,I=I+1,IF(I-N)...,...,

I:ciklusváltozó). A szubrutin és function hívást a csomóponton belül vagy csak jelöljük (egy csomóponttal, kimenő paraméterekkel cimkézett kifelé mutató nyilakkal és bemenő paraméterekkel cimkézett befelé mutató nyilakkal), vagy a megfelelő programrészletet behelyettesítjük.

A programgráf és a csomópont logikai modellje segítségével három szinten tárgyalhatók a programok paralelitásai:

- paralelitások a programgráf végrehajtásában (csomópontok között),
- paralelitások a csomópont utasításainak végrehajtásában,
- paralelitások az utasítások műveleteinek végrehajtásában.

Amit a csomópont logikai modellje nem tükröz, az a különböző utasításokban szereplő, azonos operandusokon végzett azonos műveletek egyszeri végrehajtásának kérdése. (Ez közel áll a paralelitások kérdéséhez, de szigorúan véve nem az, hanem műveletszám minimalizáció kérdése.) Például az

$y_1=(x_1+x_2)+x_3$
$y_2=x_4*(x_2+x_3)$

értékadó utasításokban (x_2+x_3)-at elegendő egyszer kiszámítani.

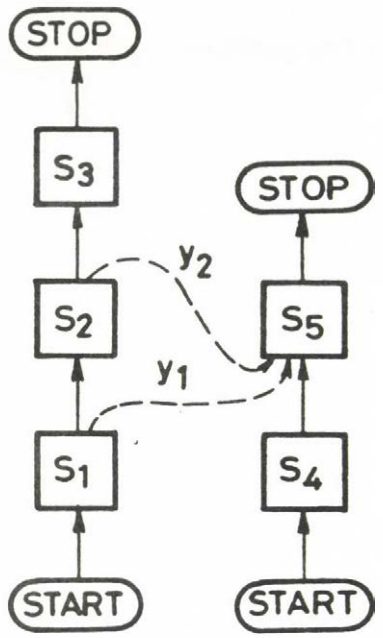
2.2 Párhuzamos programok jelenségei

A párhuzamos programozás három jelensége, amely a szek-

venciális programoknál nem fordul elő, a várakozás, a dead-lock és az adatsorrendezés.

a) A várakozás a csatolt párhuzamos programokban léphet fel, több okból is. Az adatforrás és az adatfelhasználó programok eltérő futási sebessége a kommunikáció létrejöttére való várakozást teheti szükségessé (ld. 2.3 Szinkronizáció fejezetben részletesen). Véges programfutási sebességeket feltételezve, szükséges lehet a várakozás strukturális okokból is. Erre mutat példát a 11. ábra. Az ábrában az S_i szövegű négyzetek tetszőleges részprogramot jelölnek. Az ábra "A" jelű programja y_1 és y_2 adatokat küldi a "B" jelű programnak; az y_1 változó értékének megkapása után a "B" jelű programnak várakoznia kell az S_2 részprogram lefutására, és csak ezután kaphatja meg az y_2 változó értékét.

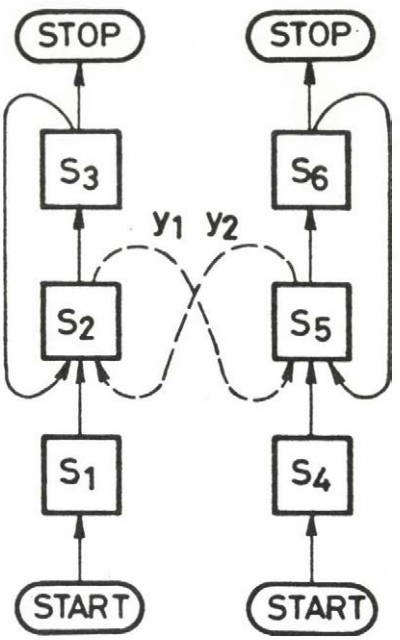
b) A "dead-lock" a párhuzamos rendszerben a végtelen várakozás esetét jelöli. Egyik fellépése az erőforrások lefoglalásával kapcsolatos. Tegyük fel, hogy két, "A" és "B" jelű párhuzamos programunk van, és hogy mindkét program ugyanazt a két perifériát - például sornyomtatót és kártyalyukasztót - kívánja használni. Azonban a sornyomtatóra az "A" program prioritása legyen nagyobb, mint a "B" programé, és a kártyalyukasztóra "B" program prioritása legyen nagyobb, mint "A" programé. A programok futása során mindkét program le tudja foglalni a szükséges perifériái egy részét, de kölcsönösen okai annak, hogy periféria igényük egy része kielégíthetetlen marad. Így tehát kölcsönösen várakozniuk kell egy-



„A" program „B" program

Várakozás

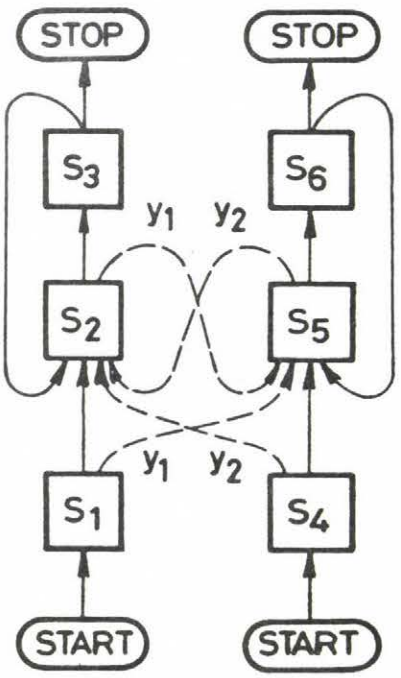
11. ábra



„A" program „B" program

Dead - lock helyzet

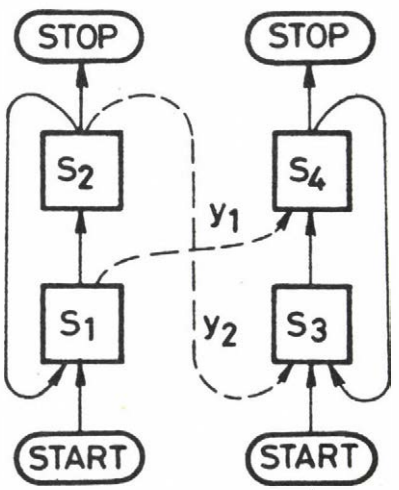
12. ábra



„A" program „B"program

Alternativ adatdefiniálás

13. ábra



„A"program „B" program

Keresztcsatolás

14. ábra

másra - végtelen ideig, vagy amíg a helyzetet beavatkozással fel nem oldjuk.

A dead-lock másik fellépése a hurkos adatcsatolással függ össze. Erre példát a 12. ábra szolgáltat. Az ábrában az S_2 részprogram akkor tud futni, ha S_5 részprogram már lefutott; S_5 futásának előfeltétele viszont, hogy S_2 már lefutott. Így tehát mind S_2, mind S_5 arra vár, hogy a másik részprogram adatot küldjön neki. A helyzetet kétféleképpen is feloldhatjuk. Az egyik módszer feltételezi, hogy kezdetben mind S_2, mind S_5 érzéketlen az adathurokra. Ilyen eset áll fenn, ha például a csatoló y változó egy A*(B+y) kifejezésben hasznosul. Itt ha A=0, a kifejezés értéke független y értékétől. Ilyen módon, ha S_2 és S_5 részprogramok egyszer már lefutottak, utána mindig tudnak adatot küldeni egymásnak. A másik módszer az alternatív adatdefiniálás, amelyre példát a 13. ábra mutat. Itt kezdetben az y_1 és y_2 változók értékét az S_1 és S_4 részprogramok definiálják, majd ezt követően az S_2 és S_5 részprogramok veszik át az adatforrás szerepét. Ha így a paralel programokat kezdetben át tudjuk segíteni az egymás blokkolásán, akkor azok utána mindig tudnak adatot küldeni egymásnak. (Hurokmentes paralel programokban el tudjuk kerülni a dead-lock-ot, ha a csatolások sem okoznak hurkokat.)

c) Az adatsorrendezés igénye pufferral vagy FIFO-val csatolt paralel processzorok közötti adatkapcsolat esetén merülhet fel. Erre egyszerű példát a 14. ábra mutat, amely a keresztcsatolás esete. Itt S_1 részprogram előbb generálja csatoló adatát (y_1), mint S_2 részprogram (y_2),

de y_2 előbb hasznosul, mint y_1. Az adatközlést a betáplálás sorrendje szerint megvalósító hardware csatolóeszközök jelenléte esetén a generált adatokat a felhasználás sorrendje szerint át kell rendeznünk. Erre eszköz lehet a paralel programok lokális memóriája, amelynek csak egyszerű átmeneti tárolási feladatot kell ellátnia. (Az adatsorrendezés közös memória esetén automatikusan megvalósul.) Az adatsorrendezés igénye szekvenciális programokban is felléphet, de ott nem kapcsolódik adatkommunikációval.

2.3 Szinkronizáció

Eddig csak azt mondtuk, hogy a globális változók adatkapcsolatot teremtenek a paralel programok között: az egyik program valamely csomópontja adatot küld, a másik program pedig fogadja. Most ennek az adatkapcsolatnak a mechanizmusát vizsgáljuk.

Az adatkapcsolat alapvetően kétféle lehet: szinkronizálatlan vagy szinkronizált. Mindkétféle adatkapcsolatnak megvannak a maga alkalmazásai.

A szinkronizálatlan adatkapcsolatban a küldő és a fogadó programcsomópontok definiáltak, de a két (vagy több) program a maga sebességével fut, és így nem garantált, hogy egy küldött adat pontosan egyszer éri el a fogadót. Tekintsük a hőmérsékletszabályozás következő példáját. Két paralel programunk van. Az egyik program N ponton elhelyezett hőmérsékletérzékelők adatait begyűjti, azok

átlagát kiszámítja, és az átlag értékét egy változóban a másik program rendelkezésére bocsátja. A második program a mérési átlag, az előírt hőmérséklet és a beavatkozó szerv (fűtőtest) karakterisztikája ismeretében kiszámítja a szükséges fűtőáramot. Ha az első program számítási folyamata a gyorsabb, akkor küldött adatok elveszhetnek (a második program minden m-edik mért adatra szabályoz). Ha a második program számítási folyamata a gyorsabb, a szabályozó beavatkozás többször is ugyanazon mért adat alapján történik. Ebben a példában a két program relatív sebességére szigorú előírásunk nincs.

A szinkronizált adatkapcsolatban a küldő és a fogadó programcsomópontok definiáltak, és a két (vagy több) program sebességét úgy szabályozzuk - várakoztatással -, hogy egy küldött adat pontosan egyszer érje el a fogadót. Tekintsük a lyukkártyamásolás következő példáját [9]. Két paralel programunk van. Az egyik program (az egyszerűség kedvéért) egy változóba beolvassa egy lyukkártya szövegét, és azt a lyukasztó második program rendelkezésére bocsátja; végjel (speciális lyukkártya szöveg) esetén a program leáll. A második program az első programtól átvett adatot egy lyukkártyára kilyukasztja; végjel átvételekor a program leáll. Nyilvánvaló, hogy a két program futási sebessége között szigorú viszony áll fenn, mivel egyetlen beolvasott lyukkártyának egyetlen kilyukasztott lyukkártya kell, hogy megfeleljen. Ha a beolvasó program a gyorsabb, akkor annak várnia kell a lyukasztásra; ha a lyukasztó program a gyorsabb, akkor annak várnia kell a beolvasásra.

A szinkronizáció megfelelő hardware hátteret igényel. Tekintsük először a puffereken keresztüli adatkapcsolatot. Egy kapcsolat adatszava mellett álljon rendelkezésre egy flag bit, amelyet mind a küldő, mind a fogadó program írhat és olvashat. E flag bit értéke szabályozza az adatkapcsolatot, a következőképpen:

- ha a forrás adatot küld, akkor a flag bitet bebillenti 1-be;
- ha a felhasználó az adatot elveszi, akkor a flag bitet reseteli 0-ra;
- ha a forrás adatot küldene, és a flag bit értéke 1, akkor a forrás várakozik;
- ha a felhasználó adatot elvenne, és a flag bit értéke 0, akkor a felhasználó várakozik.

Ha egy küldött adatszóra több paralel program a felhasználó, akkor a flag bitet flag szóvá kell bővítenünk. A flag szó minden bitjét egyszerre billenti be a forrás, de a felhasználó programok bitenként resetelnek. Tekintsük most a közös memórián keresztüli adatkapcsolatot. Itt ugyanaz a mechanizmus szükséges, mint amit előbb leírtunk, azonban a memória készen kínálja a flag szót. Ha például egy N bit szóhosszúságú memória M-edik címén tároljuk a globális változó értékét, akkor az M-1-edik címet flag szónak használhatjuk, és az N darab felhasználót tud kiszolgálni.

A szinkronizáció leírt módja gyors programfutást biztosít, mivel a felhasználó csak addig vár, amíg a forrás nem képzi adatát, ill. a forrás csak addig vár, amíg a felhasználó a megelőző adatot el nem vitte. Felmerül a

kérdés, hogy nem lehetne-e a várakozási időket valahogy csökkenteni. A forrás várakozási idejének redukálására van mód, mégpedig úgy, hogy ez a program (tetszőlegesen) "előreszaladhat". Ehhez a globális változó értékének tárolásához FIFO memória alkalmazása szükséges. A memóriát a forrás a maga sebességével tölti, a felhasználók pedig a maguk idejében olvassák és léptetik. Puffereken keresztüli adatkapcsolat esetén kereskedelemben kapható FIFO memóriát használhatunk. Közös memórián keresztüli adatkapcsolat esetén a memóriában software eszközzel képezhetünk FIFO mezőt, amelyet mindig az időben éppen utolsó felhasználó léptet.

A szinkronizáció fenti módjai automatikusan biztosítják, hogy a felhasználók a források adatait helyesen és egyszer kapják meg, a program futásának kezdetétől a végéig. A szinkronizáció ezen módjai nem igénylik, hogy az adatok koordinátáit megadjuk (ilyen koordináták lennének mind a forrás, mind a felhasználók részéről: melyik csomópont vesz részt az adatforgalomban, a csomóponton belül hányadszor történik adatforgalom, a csomóponton való hányadik áthaladás során lép fel az adatforgalom). A koordináták megadásának szükségtelensége előnyös tulajdonság, viszont igényli, hogy a programozó át tudja tekinteni a paralel programok teljes futását és adatkapcsolataikat. Ehhez eszköz a csatolt programok már tárgyalt programgráfja.

Most tekintsük a szinkronizáció alkalmazásának néhány kérdését.

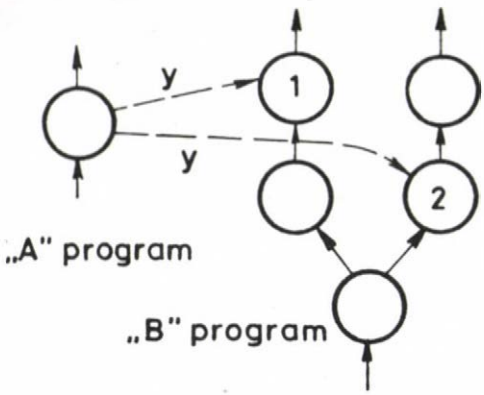
a) A flag bitek kezelésére mind a forrás, mind a felhasználók megfelelő csomópontjainál egy programhurkot kell kialakítanunk, amely az esetleg szükséges várakozást végrehajtja.

b) A közös memórián keresztüli adatkapcsolatnál szerepet kap a lebegő prioritás. Ha a forrás adatot küldene, de azt találja, hogy a felhasználók a megelőző adatot még nem vették el, vagy a felhasználók adatot kérnek, de azt találják, hogy a forrás még nem küldött adatot, akkor prioritásukat alacsony szintre szállítják, hogy ezzel is gyorsítsák az adatforgalmat.

c) Tekintsük a 15. ábrán megadott programgráf-részletet. Itt természetszerűleg a B program 1 és 2 jelű csomópontjainak egyaránt resetelniük kell a flag bitet.

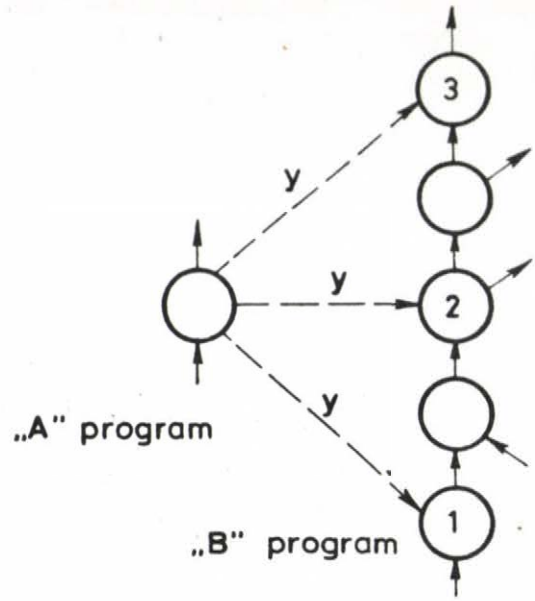
d) Tekintsük a 16. ábrán megadott programgráf-részletet. Itt az "A" program y adatát a "B" program három csomópontja is felhasználja. Az ilyen többszörös adatfelhasználás esetén egyszerűbb, ha csak egy (az 1-es jelű, kezdő) csomópont kommunikál a forrásprogrammal, és az átvett adatot a lokális memóriájába írja.

e) Tekintsük a 17. ábrán megadott programgráf-részletet. Itt az előző, d) pontbeliek érvényesek, azonban tegyük fel, hogy az y adatot a hurkon való kétszeri áthaladás során kívánjuk hasznosítani. Ehhez a lokális memóriában képezhetünk flag biteket, amelyek számláló funkciót töltenek be (hatszori adatfelhasználás esetén nullázódnak teljesen).



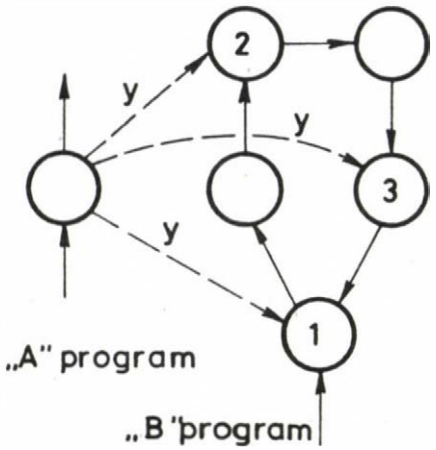
Szinkronizáció

15. ábra



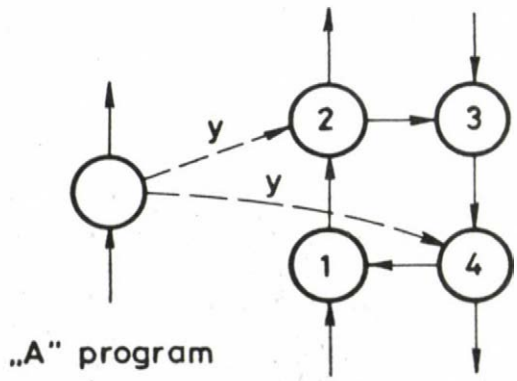
Szinkronizáció

16. ábra



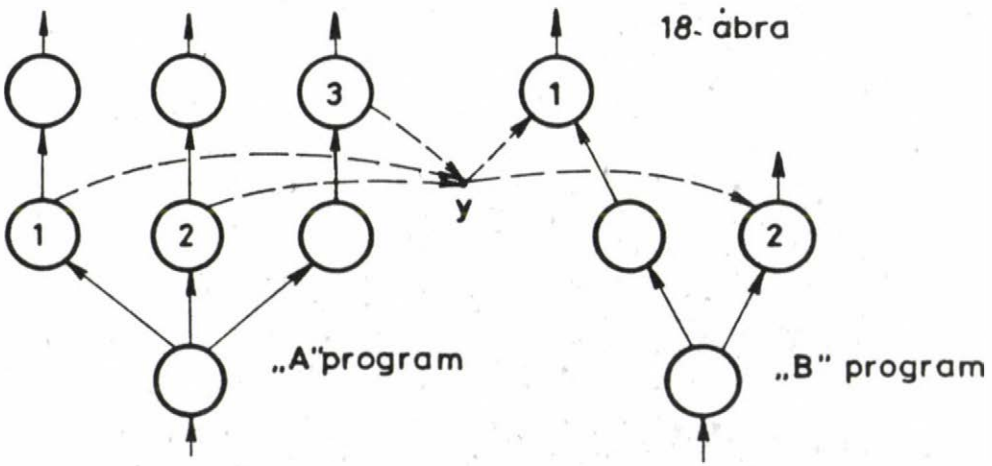
Szinkronizáció

17. ábra



Szinkronizáció

18. ábra



Szinkronizáció

19. ábra

f) Tekintsük a 18. ábrán megadott programgráf-részletet. Itt a "B" programban olyan hurkunk van, amelybe több ponton léphetünk be (1, 3 jelű csomópontok), amelyet több ponton hagyhatunk el (2, 4 jelű csomópontok), és ahol többszörös az adatfelhasználás (2,4 jelű csomópontok). A d) pontban leírtak nem érvényesek, mivel nincs egyértelmű kezdőcsomópont. Ehelyett a forrásnak két flag bitet kell setelni, és a 2 és 4 jelű csomópontoknak egy vagy két flag bitet kell resetelni, megkövetelve annak előzetes ismeretét, hogy a program meddig tartózkodik a hurokban.

g) Tekintsük a 19. ábrán megadott programgráf-részletet. Ez példája annak, hogy a szinkronizáció a programok csatolásaira általában megkötéseket állít. Tegyük fel, hogy az "A" program 3 jelű csomópontja adatot küld a "B" program 1 jelű csomópontjának. Ennek az adatküldésnek akkor is le kell játszódnia, ha az "A" vagy a "B" program más ágára fut (ezt mutatja be az ábra), különben a flag biten "fennakad" a rendszer. Ezen enyhíthetünk annyit, hogy a "más" ágakon a programok csak a flag bitet kezelik, de adatot nem küldenek vagy vesznek.

A szinkronizáció iránt tovább érdeklődő olvasó figyelmét a [9], [10] és [11] irodalmi munkákra hívjuk fel, amelyek a szinkronizációt más szemszögből tárgyalják.

2.4 A multimikroszámítógép-rendszer használatának kiterjesztése

A multimikroszámítógép-rendszer software-jéről eddig

elmondottak elsősorban arra az esetre vonatkoznak, amikor a rendszeren egyidejűleg egy program fut. Nagyobb rendszereket szem előtt tartva, egy program futása nem nagyon terheli le a rendszert - ezért kiterjesztjük a rendszer használatát arra az esetre, amikor a rendszeren egyidejűleg több program fut.

Tasknak nevezünk egy szekvenciális vagy paralel programot, amely a rendszeren futtatható. Egy job a taskok paralel rendszere. A jobokból is egyszerre több futtatható, azonban a jobok függetlenek egymástól, azaz futás közben nem kommunikálnak egymással. Egyetlen task, mint program egyidejű futtatása esetén a rendszer az operátori pultról kényelmesen vezérelhető. Multitask és multijob esetben a vezérlés (és a kommunikáció) csak operációs rendszer használatával tehető kényelmessé. Az operációs rendszer multitask kezelő szerepe elég bonyolult lehet, mivel a taskok rendszerének paralelitásait - magas szinten, az eddig leírtaktól eltérően - kell kezelni.

A következőkben a multitask operációs rendszer néhány jellemzőjét tárgyaljuk.

Az operációs rendszer számára meg kell adnunk a taskok néhány paraméterét: az azonosítót, a prioritást és az erőforrás igényt (I/O perifériák, memóriaterület). Az operációs rendszer ennek megfelelően taskonként allokál: processzorokat, memóriát (sajátot és közöst), valamint I/O egységeket. Futáskor a taskok kommunikálnak egymással. Ez történhet

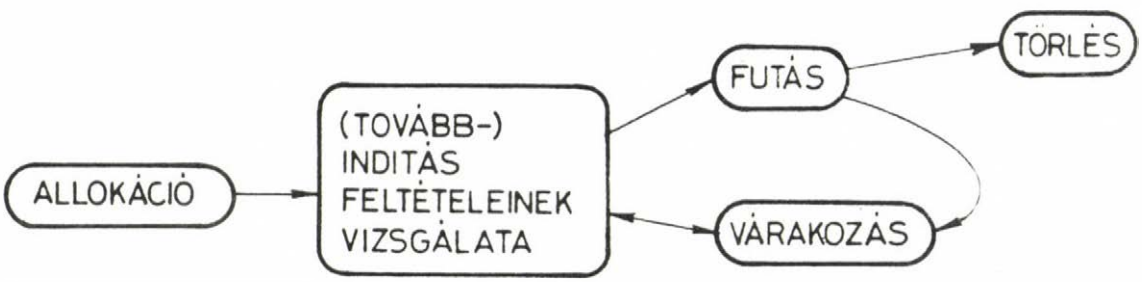
- I/O file-okon keresztül vagy közös memóriamezőkön keresztül,

- futás közben vagy futáseredményeken keresztül,
- az operációs rendszer tudtával vagy anélkül.

Az operációs rendszer fenti feladatainak ellátására célszerű külön processzort felhasználni.

Egy task az operációs rendszer kezelésében, élete folyamán, különböző állapotokban lehet. Az öt lehetséges állapot: az allokáció, a (tovább-) indítás feltételeinek vizsgálata, a futás, a várakozás és a törlés. Az állapotokat és kapcsolataikat a 20. ábra tünteti fel. Az allokáció során az operációs rendszer a task paramétereit tudomásul veszi, és erőforrásokat jelöl ki számára. A (tovább-)indítás feltételeinek vizsgálata során az operációs rendszer ellenőrzi, hogy a (tovább-) futás előfeltételei teljesülnek-e. A futás során a processzorok a task utasításait hajtják végre. A várakozás egy felfüggesztett állapot, amelyben a task arra vár, hogy továbbfutásának feltételei teljesüljenek. A törlés során a task irreverzibilisen megszűnik létezni.

Az öt állapot közül a legérdekesebb a (tovább-) futás feltételeinek vizsgálata. Ez a jobon belül egy task-vezérlő nyelv létezését igényli. A task indításának feltételéül azt szabhatjuk meg, hogy a taskok egy bizonyos csoportja már lefutott állapotba kerüljön. A továbbfutás feltételeinek vizsgálata a taskok szinkronizációjával kapcsolatos. A taskoknak, mint a paralel programoknak is, arra kellhet várakozniuk, hogy adatokat küldhessenek vagy kaphassanak. A kommunikáció lebonyolítását - amelyet az operációs rendszer végez - események bekövetkezéséhez köthetjük. Bizonyos események bekövetkezése véget vethet a task várakozás-állapotának. Ilyen események lehetnek:



Egy task állapotai

20. ábra

bizonyos task bizonyos memóriaterületet (vagy perifériát) felszabadított vagy bizonyos task meghatározott azonosítóval adathalmazt képezett.

A várakozás állapotában egyidejűleg több task létezhet. Itt az operációs rendszer prioritás szerint választ taskokat futásra - ha a futás feltételei teljesülnek. A taskok közötti paralelitás esetén - a taskon belüli paralelitásokhoz képest (paralel programok) - a várakozás hosszabb idejű, a kommunikáció ritkább, és a kommunikáció nagyobb adatmezőkre terjed ki általában.

Ha egy task a futás állapotából kimozdul (ki/belép), akkor processzorai regisztereinek adatát az operációs rendszer kezeli: kimenti vagy betölti. Ez overhead munkát jelent, amely azonban általában tetemesen megtérül azáltal, hogy a várakozni kényszerülő taskok processzorait felszabadítjuk. (Az operációs rendszer a hardware konfiguráció terhelésének fokozódásával először a szabad processzorcsoportokra allokálja az újonnan belépő taskokat, majd a várakozó taskok processzorait igyekszik hasznos munkával ellátni.)

A multitask operációs rendszer problémáival foglalkoznak a [12] és [13] irodalmi források.

Több program egy hardware konfiguráción való egyidejű futtatásának másik következménye a program- és adatvédelem szükségessége. Ez azt jelenti, hogy meg kell akadályozni, hogy egy program szándékosan vagy tévesen más programok program- és adatmezejéhez férhessen. A szándékos (illetéktelen) hozzáférés nem hiba, a téves hozzáférés azonban az (olvasás esetén csak a saját

program számára okoz hibát; írás esetén azonban a másik programban is hibát okoz).

A program- és adatvédelem hardware és software eszközökkel valósítható meg. Tiltott memóriahozzáférés ellen (akár lokális, akár közös memóriában) hardware védelem képzelhető el. (A védelem a processzor egy I/O ciklusán belül kell, hogy megtörténjen. Azt, hogy a processzor a számára kiosztott memóriához fordul-e, cím range határokra beállított komparátorok ellenőrizhetik.) A hardware védelmet a konfiguráció tervezésekor már figyelembe kell venni. Tiltott file-hozzáférés ellen software védelem alkalmazható (például kulcsszavak ismerete).

3. Irodalom

[1] Microprocessor Systems (Software, Firmware and Hardware), North-Holland Publishing Company, 1980.

[2] Manwaring, M.L., Meador, J.L., "A High-Speed All-Asynchronous Microprocessor", 13th Asilomar Conf. on Circuits, Systems and Computers, Nov. 5-7, 1979, pp. 449-454.

[3] Garcia, O.N., "Parallel Processing: an Introduction - Guest Editor's Comments", Journal of Digital Systems, Vol. IV, Issue 2, pp. 107-113.

[4] Anderson, C.A. and Jensen, E.D., "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", ACM Computing Surveys, Dec. 1976, pp. 197-213.

[5] Wittie, L.D., "Communication Structures for Large Networks of Microcomputers", IEEE Trans. on Computers, Vol. C-30, No.4, Apr. 1981, pp.264-273.

[6] Domán, A., "Párhuzamos számítási rendszerek, párhuzamos számítógépek általános vonásai", Számítástechnika, 1981. márc., pp. 4-5.

[7] Carey, B.J., McCoy, E.E., "Basic Hardware Interconnection Mechanisms for Building Multiple Microcomputer Systems", Department of Computer Science, Naval Postgraduate School, Monterey, Calif. 93940, Nov. 1979, Report No. ADA081036.

[8] Kartashev, S.P., Kartashev, S.J., "Architectures for supersystems of the '80s", National Computer Conference, 1980, pp. 165-180.

[9] Per Brinch Hansen, "The Architecture of Concurrent Programs", Prentice-Hall Series in Automatic Computation, Prentice-Hall, Inc., 1977.

[10] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, Aug. 1978., Vol.21, No.8, pp. 666-677.

[11] Gentleman, W.M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software-Practice and Experience, Vol.11, 1981, pp. 435-466.

[12] Irvin, J.M., "Multitasking executive speeds 16-bit micros", Electronic Design, March 5, 1981, pp. 131-135.

[13] Vincze, S., "A DIALOG CNC-M szerszámgépvezérlő berendezés multitask supervisora", Mérés és Automatika, 1981. márc., pp. 91-94.

II. rész:

Konverzió szekvenciális és párhuzamos programok között

Kivonat

A tanulmány II. részében a paralel programok szekvenciális programmá történő egyesítését (kompozíció) és a szekvenciális programok paralel programokká történő lebontását (dekompozíció) tárgyaljuk.

A kompozíció változataiként a multiprogramozásos és a multiprogramozás nélküli módszereket mutatjuk be.
A dekompozíció módszereiként a tördeléses, az átalakításos és a redundáns számításos változatokat tárgyaljuk.

A szekvenciális és paralel programok konverziója kétféle programtranszformációt foglal magába: paralel programok átalakítását szekvenciálissá (kompozíció) és szekvenciális programok átalakítását paralellé (dekompozíció). Mindkét átalakítás során a kiindulási és az eredmény programok inputjai és outputjai azonosak (minőségileg, mennyiségileg), csak a programok struktúrája változik.

A kompozíciós átalakítás elsősorban programfejlesztési célokat szolgál. Egy multiprocesszoros rendszer kifejlesztésekor, már a hardware tervezés fázisában is írhatunk és kipróbálhatunk paralel programokat, ha a rendszerkonfigurációt (és vele a paralel programokat) a bárhol rendelkezésre álló egy-processzoros szekvenciális gépeken szimulálni tudjuk.

A dekompozíciós átalakítás kétféle célra szolgálhat. Egyrészt az egy-processzoros szekvenciális gépekre írt programokat (a létező számítógép-programok döntő többsége ilyen) átültethetjük az átalakítással multiprocesszoros rendszerekre, például a végrehajtás gyorsítása céljából. Másrészt a paralel programok tervezésében a programozóknak ma még kevés gyakorlatuk van, és vannak feladatok, amelyeknek a paralel programozhatósága nem szembetűnő. Ilyenkor a feladatot először szekvenciális programmal oldhatjuk meg, majd a dekompozíciót alkalmazhatjuk.

A soron következő tárgyalásban a tanulmány I. részének anyagát és terminológiáját (pl. programgráf, szinkronizáció,...) alkalmazzuk. Mind a kompozíciót, mind a

dekompozíciót elsősorban a programgráf csomópontjainak szintjén vizsgáljuk (ez az érdekesebb feladat), alacsonyabb szintet csak ott érintünk, ahol az szükséges.

1. Paralel programok szekvenciális kompozíciója

A következőkben szükségünk lesz a független és függő paralel programok fogalmára. A független programok egyrészt nem csatoltak (sem közös memórián, sem pufferekeresztül nem kommunikálnak), másrészt minden programnak saját I/O eszközei vannak. Függőek a paralel programok, ha az előző két feltétel legalább egyike nem teljesül.

1.1 Multiprogramozás

A paralel programok kompozícióját független és függő programok csoportosításban tárgyaljuk.

a) Független programok

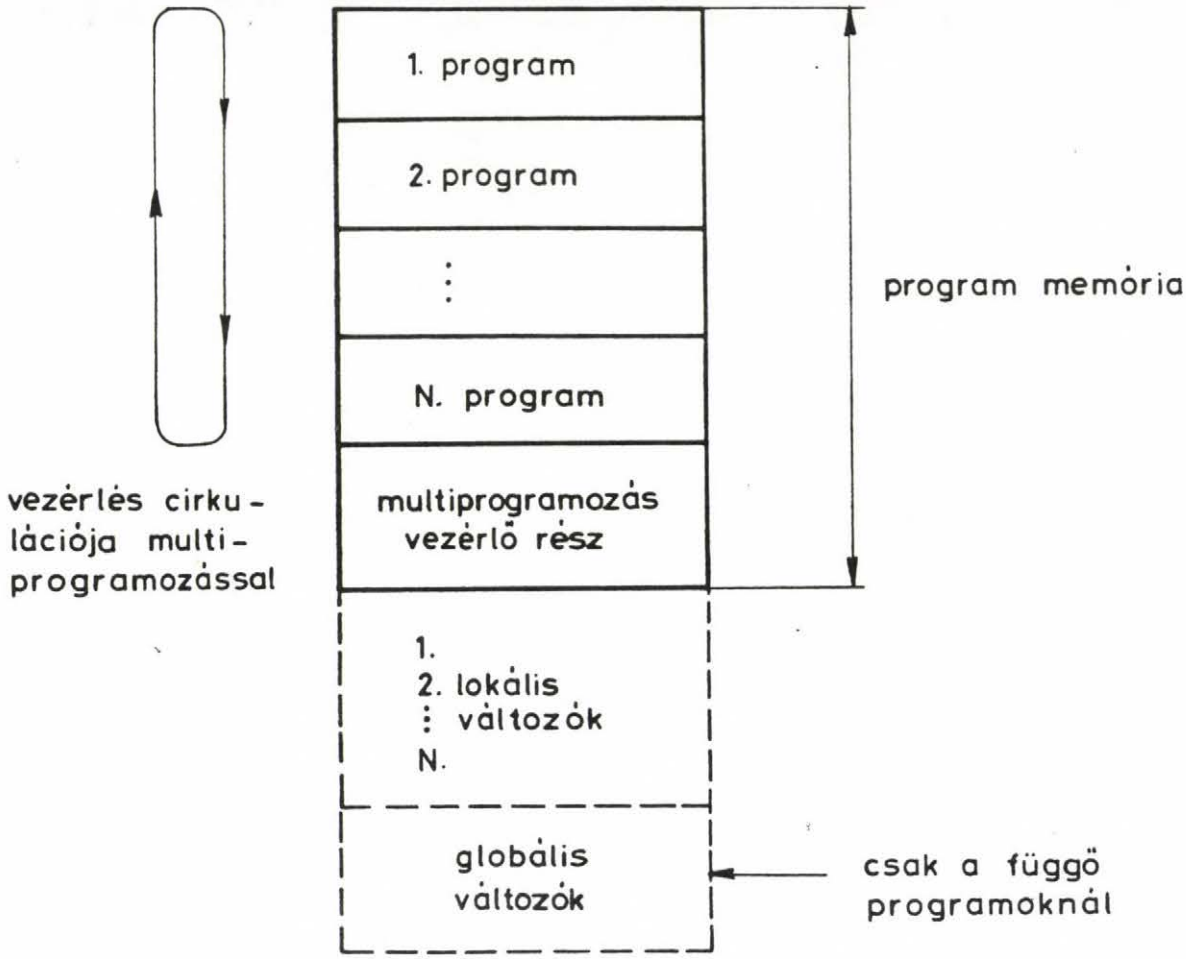
A véges paralel programok kompozíciója egyszerű. Legegyszerűbb megoldás, hogy a programokat egymás után (egymástól függetlenül, tetszőleges sorrendben) végrehajtjuk. Bonyolultabb megoldás, hogy a programokat részekre vagy akár csomópontokra tördeljük, és azokat alternálva, egymás után hajtjuk végre (erre példa az 1.2 pontban következik).

A végtelen programok kompozíciójának kulcsa: a programokat feldaraboljuk, és a részeket multiprogramozással végrehajtjuk. A programok feldarabolásának olyannak

kell lennie, hogy egyetlen programban se maradjon hurok (mert rendszerint nem tudjuk, hogy az véges vagy végtelen lenne-e). A programok legkevesebb részre történő darabolásakor - a programgráf metszett éleinek eliminálásával - a programgráf egy kifeszítő hurokmentes gráfját kapjuk. A multiprogramozás vázlatát az 1. ábra mutatja be. Az ábrából látható, hogy a szekvenciális program zömmel a paralel programok lineáris egymás után írásával áll elő. Azonban a paralel programokban megszakítási pontok vannak programozva. A megszakítások kezelése a következő: a folytatási címet (a megszakítás sorszámát) megadjuk egy ADDRESS-I változóban (ahol I az I-edik paralel programra utal), a program sorszámát megadjuk egy PROGNO változóban, és a multiprogramozás vezérlő részre ugratunk. Ott kijelöljük a végrehajtásban következő program sorszámát, és a már előzőleg beállított ADDRESS-I-je szerint egy kiszámított GO TO-t hajtunk végre. Az ADDRESS-I változók kezdőértéke: a START pontok címe.

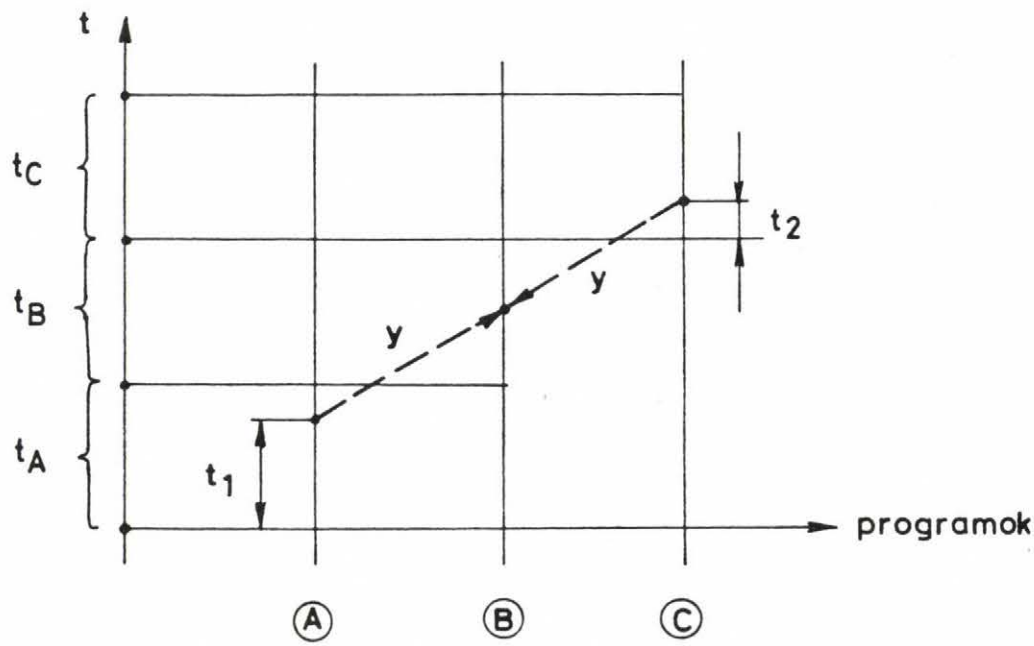
b) Függő programok

A függő programoknál különbség a független programokhoz képest, hogy itt a kapcsolatot teremtő eszközöket (pufferek, közös memória, I/O eszközök), ill. software képüket (globális változók) is kezelni kell. Ezek miatt még a véges paralel programokat sem hajthatjuk végre függetlenül, egymás után: a programoknak az adatkapcsolatok mentén rendszerint várakozniuk kell egymásra.



Multiprogramozás memória - kiosztása

1 ábra



Vagylagos adatdefiniálás

2. ábra

A véges és végtelen programok szekvenciális végrehajtásának módja itt is a feldarabolás és multiprogramozás. A feldarabolásnak nem csak a programok hurkait kell érintenie: feltételes megszakításokat is be kell iktatnunk ott, ahol adat küldésre vagy vételre várni kellhet (ADDRESS-I változók használata). A multiprogramozás az 1. ábra szerint tovább érvényes. Azonban a multiprogramozás végrehajtásakor, ha egy programban adatkommunikációra várni kell, a vezérlést továbbadjuk a következő programra.

Függő programoknál a multiprogramozásban két probléma lép fel:

- vagylagos adatdefiniálás. Tekintsük a 2. ábra szerinti szituációt. Itt három paralel programunk van. Az A és C program vagylagosan adatot (y) küld a B programnak. A multiprogramozásos sorrend (A,B,C) beleszól az adatok keletkezésének természetes sorrendjébe: a C program adata időben előbb keletkezik ($t_2 < t_1$), de az A program adata jut el előbb a B programhoz.
- prioritások kezelése az interface-eken (pufferek, közös memória, I/O eszközök). A multiprogramozott sorrend itt is meghamisíthatja a valóságos paralel viszonyokat: nem feltétlenül a legmagasabb prioritás érvényesül adott időpontban (ennek következtében egy program idegen adatot olvashat, több program outputja összekeveredhet stb....)

A két problémára közös megoldást jelent az explicit időszimuláció bevezetése - amit a multiprogramozással kombinálunk. A multiprogramozás így két passzos lesz:

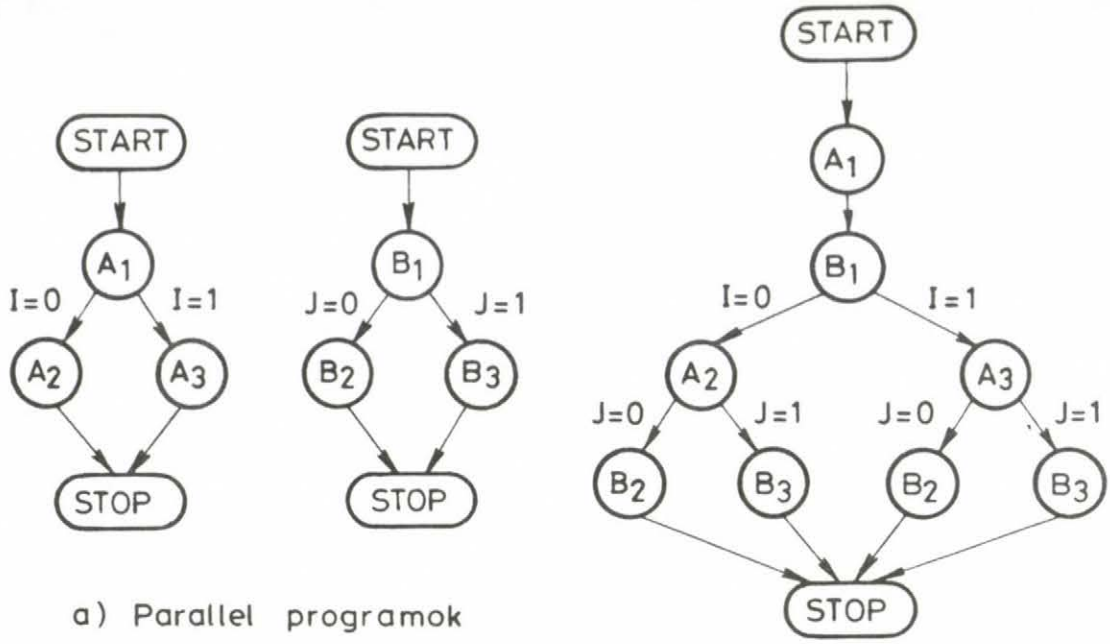
- először minden programon végigmegyünk a következő megszakítási pontig; regisztráljuk, hogy a programok mely időpontokban képzik adataikat,
- utána újra végigmegyünk minden programon, és időhelyesen érvényesítjük az előbb képzett adatok hatását.

A szimuláció és multiprogramozás haladhatnának fix időnövekménnyel (pl. 1 órajelciklus), de ekkor túl nagy lenne a kezelési overhead. Előnyösebb eseményekig (feltétlen és feltételes megszakításokig) haladni egy löketben; fölösleges viszont olyan hosszan foglalkozni egy programmal, hogy az egy globális változóra több értéket képezzen, mert utána kényszerű várakozás következik (bár ez nem jelent a szekvenciális program végrehajtásában várakozást).

1.2 Példák kompozícióra - multiprogramozás nélkül

Ebben a részben néhány speciális példát mutatunk be szekvenciális programok kompozíciójára. A példák azért speciálisak, hogy néhány jelenséget megmutathassunk.

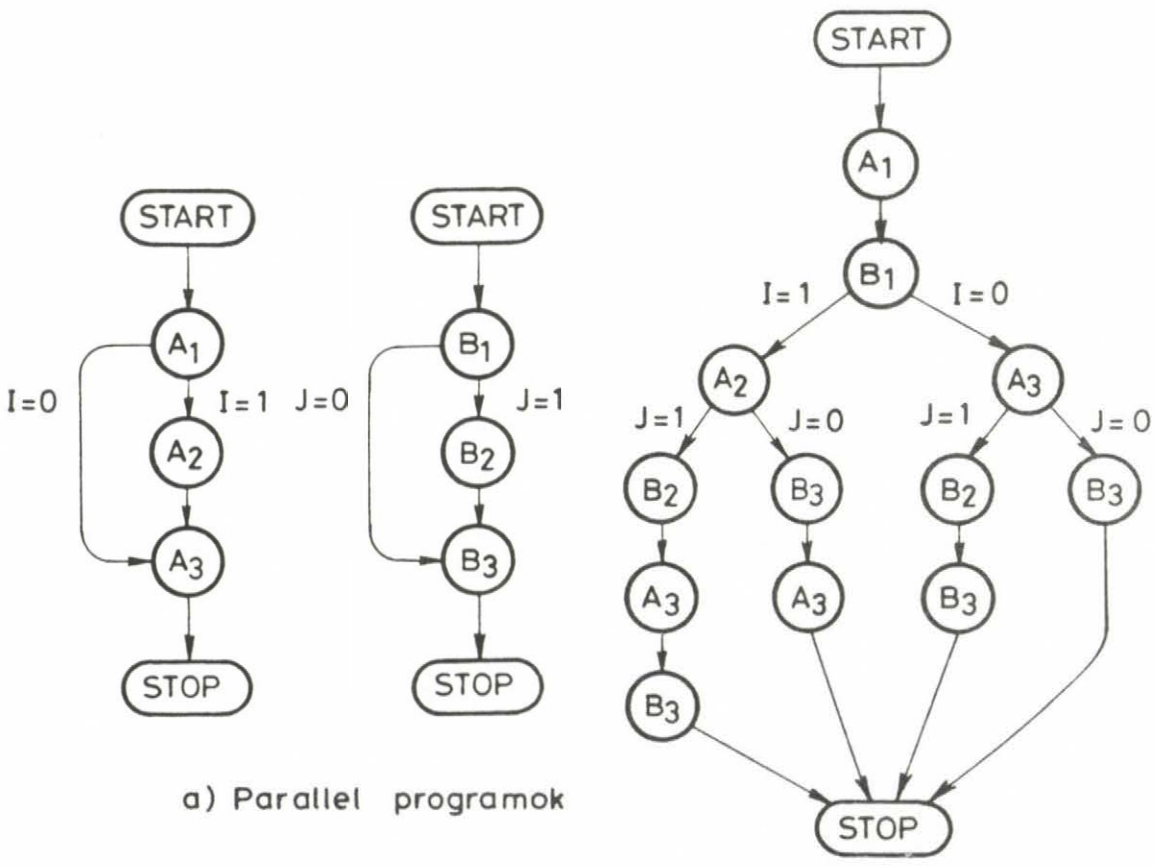
A 3., 4. és 5. ábrákban független programokat komponálunk csomópontról csomópontra. A 3. és 4. ábra paralel programjai (fagráf és rekonvergenciás gráf) véges gráffal jellemezhetők, az 5. ábra programjai (hurkos gráf) lehetnek véges vagy végtelen gráfúak. A 4. ábra paralel gráfjait a kompozició során fagráfokká fejtettük ki. Az 5. ábra gráfjainak kompozíciója során el tudtuk kerülni a hurkok felvágását, és így multiprogramozni sem kellett (viszont a szekvenciális prog-



a) Parallel programok

b) Szekvenciális program

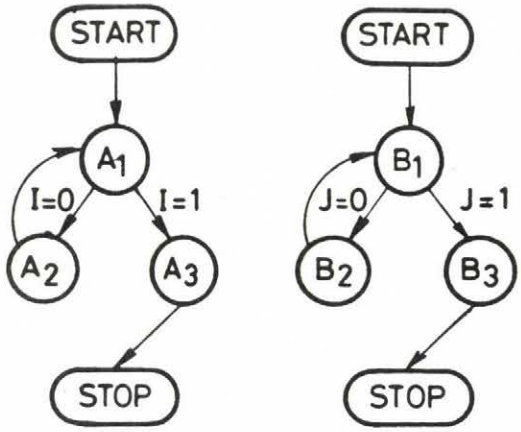
3. ábra



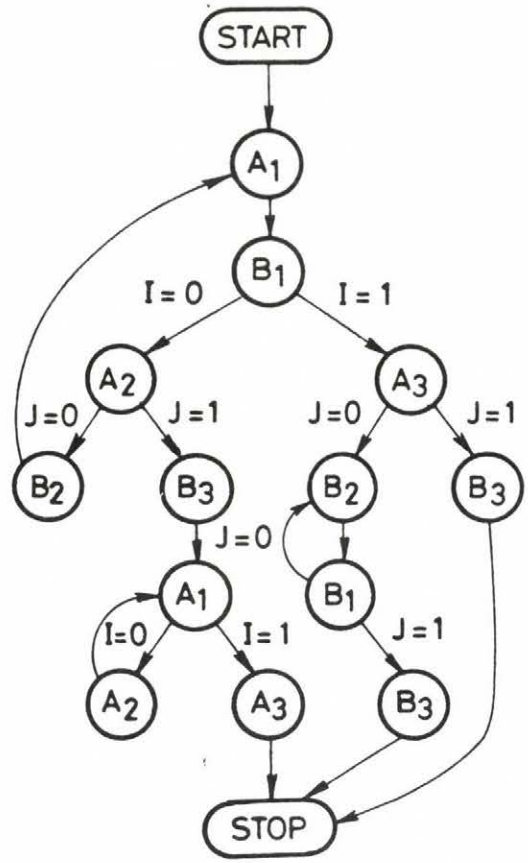
a) Parallel programok

b) Szekvenciális program

4. ábra

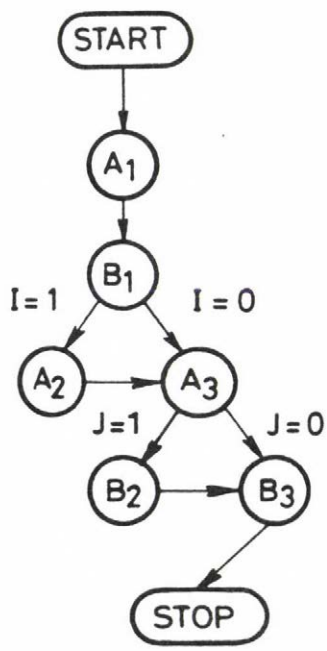


a) Parallel programok



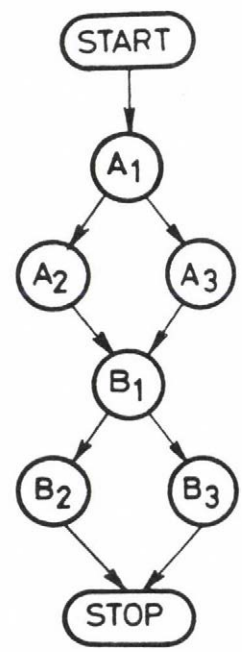
b) Szekvenciális program

5. ábra



Részgráfok kompoziciója

6. ábra



Teljes gráfok kompoziciója

7. ábra

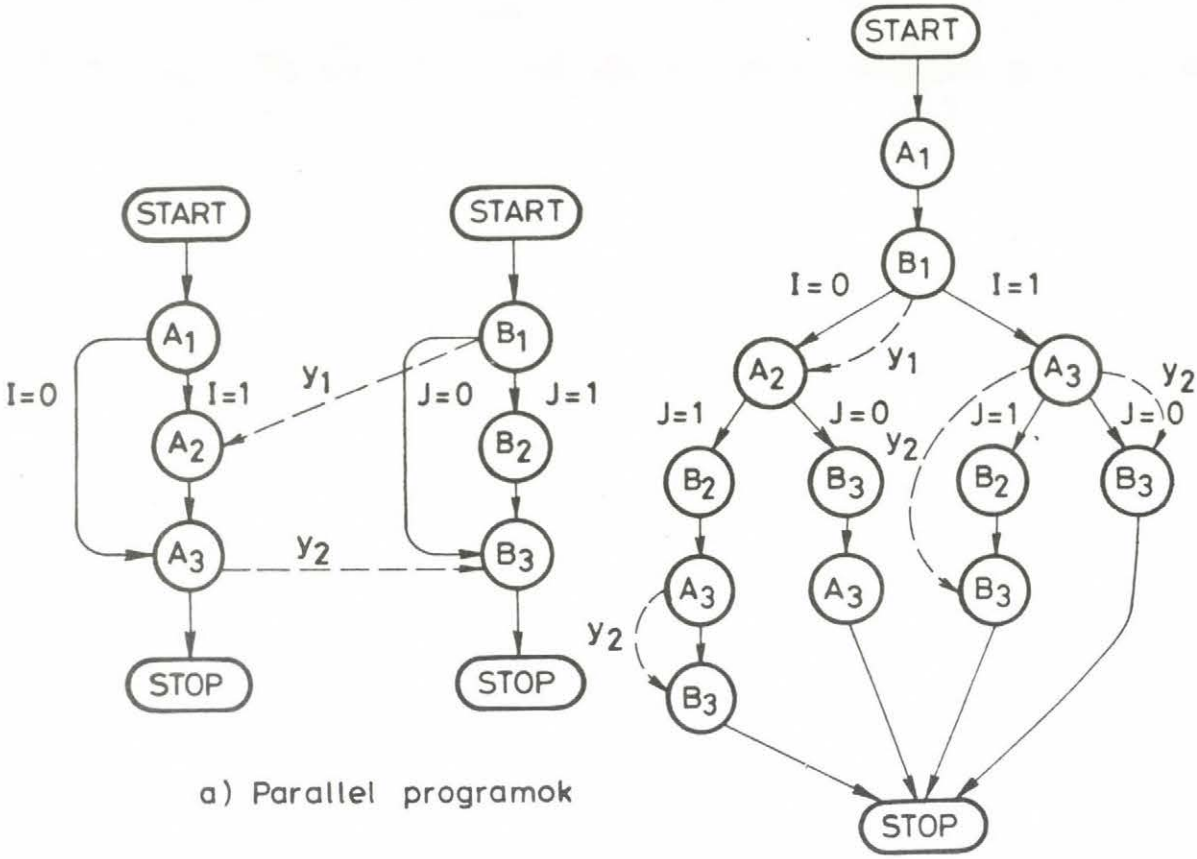
ramgráfban nem is alternálnak az A és B programok csomópontjai).

A 6. és 7. ábrában továbbra is független gráfokat komponálunk, azonban a 6. ábra a 4. ábra paralel gráfjait részgráfonként egyesíti, a 7. ábra pedig a 3. ábra paralel gráfjainak teljes gráfokként történő kompozíciójára ad példát.

A 8. és 9. ábra alapján függő véges paralel gráfokat komponálunk. A 8. ábrában a kompozíció csomópontról csomópontra halad; a 9. ábrában a kompozíciót úgy oldottuk meg, hogy részgráfokat egyesítettünk (a 8b. ábrához hasonló csomópontról csomópontra történő kompozíció azt eredményezte volna, hogy dead-lock képződik: a B_2 csomópontot előbb kell végrehajtani, mint A_3-at, de A_3 képzi az y_2 adatot B_2 számára). A probléma úgy is elkerülhető lett volna, hogy a 8b. ábrában a baloldali ágon A_3 és B_3 csomópontok sorrendjét megcseréljük. Mind a 8., mind a 9. ábrában el tudtuk kerülni a multiprogramozást.

A 3., 4., 6., 7., 8. és 9. ábrákban a szekvenciális programokban az A_1, B_1 csomópont megkülönböztetés tulajdonképpen indokolatlan (egyetlen csomóponttal kellett volna őket reprezentálni). Mi itt csak a szemléletesség kedvéért választottuk szét őket.

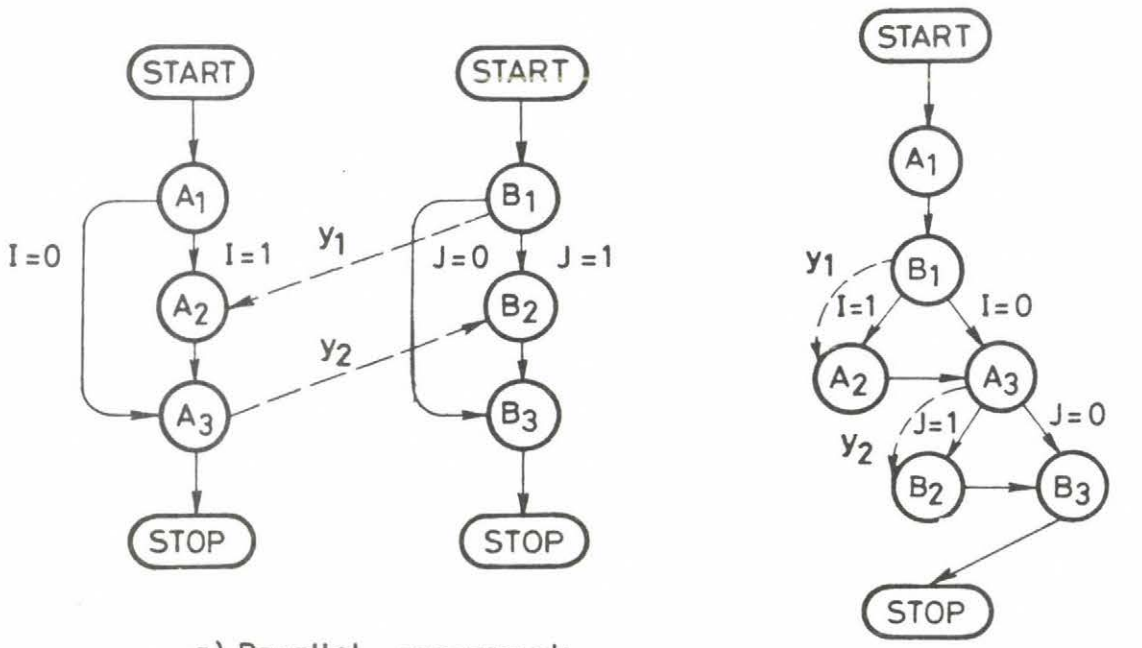
Fel kell hívnunk a figyelmet, például a 3. ábra alapján, egy fontos jelenségre. A paralel gráfokban az A_1 és B_1 csomópontok az I és J változók szerint ágaznak



a) Parallel programok

b) Szekvenciális program

8. ábra



a) Parallel programok

b) Szekvenciális program

9. ábra

el (tehát aszerint, amit a csomópontok kiszámítottak) - ezért IF utasításaikat "azonnali IF-eknek" nevezzük. A szekvenciális programgráfban viszont B_1 csomópont I szerint, A_2 és A_3 csomópontok J szerint ágaznak el - csakhogy I-t A_1, J-t pedig B_1 számította ki. Ezért a kompozit gráf csomópontjaiban az IF utasítások a paralel gráfokhoz képest eltolódnak, IF-jeik "késleltetett IF-ek".

2. Szekvenciális programok paralel dekompozíciója

A szekvenciális gráfok paralel dekompozíciója nehezebb és változatosabb feladat, mint a paralel gráfok kompozíciója. A dekompozíciónak három módszerét mutatjuk be a következőkben.

2.1 Dekompozíció tördeléssel

A dekompozíció tördeléses módszerével megpróbáljuk visszafelé csinálni azt, amit a paralel gráfok kompozíciójánál csináltunk. Ehhez alapot a 3.-9. ábrák anyaga szolgáltat. Ezeknek az ábráknak a szekvenciális gráfjai tudottan paralel gráfokból vannak komponálva, dekompozicióval tehát ezeket a paralel gráfokat kell visszakapnunk. Meg kell jegyeznünk, hogy a gyakorlatban előforduló szekvenciális programgráfok többsége nem olyan, mint az ábrákban megadottak. Az utóbbiaknak ugyanis van két speciális tulajdonságuk:

- IF utasításaik többségükben késleltetettek,
- nagy számú azonos szövegű csomópont található bennük.

Ha a 4. és 5. ábrák gráfjait tekintjük, megállapíthatjuk, hogy a dekompozíció jelentős csomópontszám csökkenéssel járhat (a szekvenciális gráfok rendre 12 és 13 csomópontot, a paralel gráfok rendre összesen 6 csomópontot tartalmaznak). Ez megkönnyíti a paralel gráfok állapotainak megtalálását.

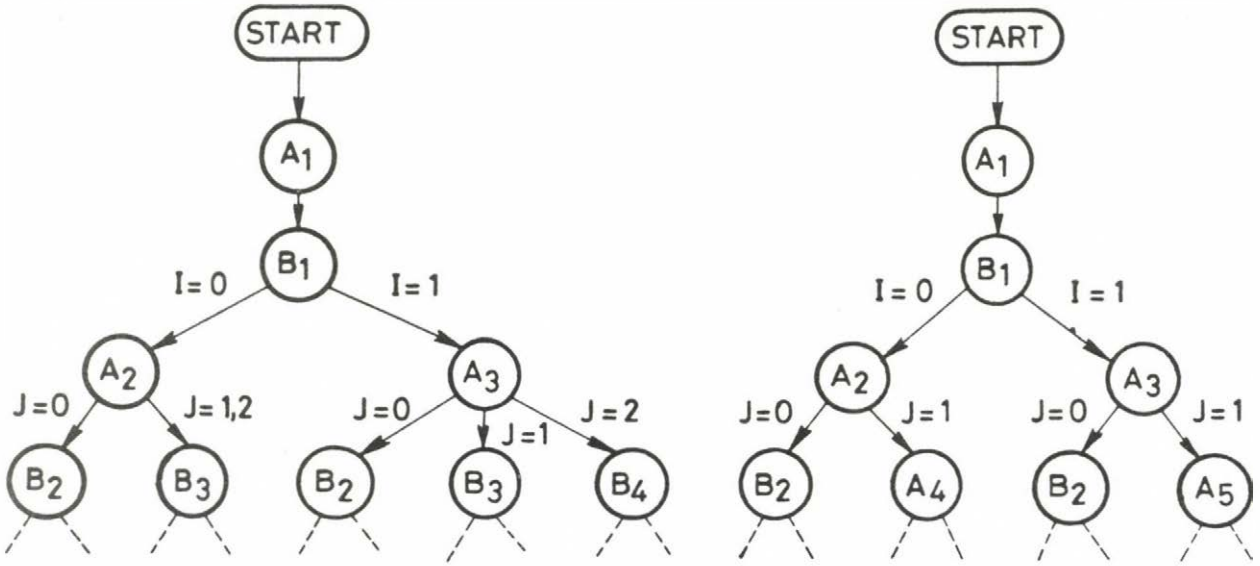
A tördeléses dekompozíció során a következőképpen járhatunk el:

- a szekvenciális gráf azonos szövegű (azonos programrészt tartalmazó) csomópontjaiban valamely paralal gráf valamely csomópontját gyaníthatjuk. Ha az azonos szövegű csomópontokat azonos betűjellel látjuk el, az nem feltétlenül végleges, mivel egy vagy több paralel gráf tartalmazhat azonos szövegű csomópontokat. Másrészt viszont, ha a szekvenciális gráf azonos szövegű csomópontjaiból túl sok paralel csomópontot képezünk, a paralel gráfokban csomópontszám minimalizációt kellhet végrehajtanunk ("ekvivalens állapotok").
- megállapítjuk a szekvenciális gráf csomópontjainak egymástól való függéseit (erre egy reprezentációt, a "függésgráfot", később mutatjuk be). Az egymástól erősen függő csomópontok feltehetőleg ugyanabba a paralel gráfba kerülnek.
- sor kerülhet a szekvenciális gráf csomópontjainak soros dekomponálására (pl. ha a 3. ábra szerinti A_1 és B_1 csomópontok a szekvenciális gráfban egyetlen csomópontként voltak adva).
- ha a szekvenciális gráfban egy csomópontot és minden azonos betűs megjelenését a függések mentén - elágazási feltételek szerint is helyesen - ugyanolyan csomó-

pontok követnek, akkor az adott csomópont, a követői és az elágazás feltételei egy paralel gráf részei lehetnek. (Például az 5b. ábrában a B_1 csomópontot J=0 feltétellel mindig B_2, J=1 feltétellel mindig B_3 csomópont követi a függések mentén - ezek a B paralel gráf részei, 5a. ábra.)

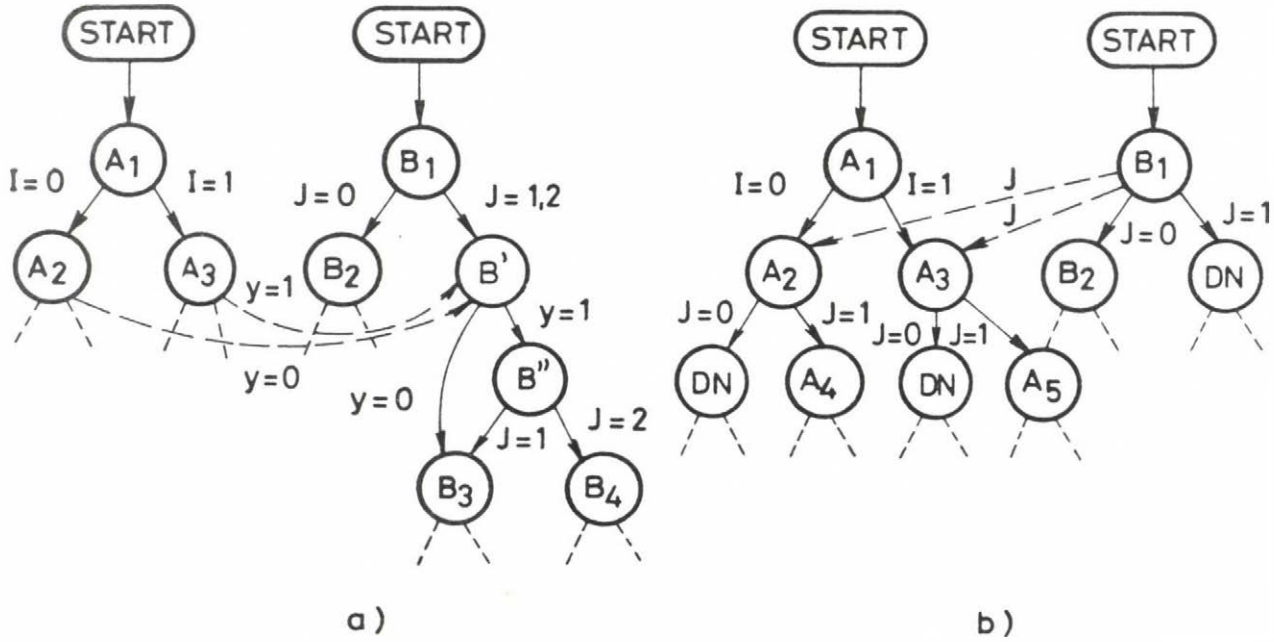
- ha egy paralel gráf egy csomópontja több globális változótól függ, elvileg mindegy, hogy melyik paralel gráfba kerül. Oda kerüljön, ahol a legtöbb a lokális változója.

A tördeléses dekompozíció tárgyát képző szekvenciális programgráfok - szerkezetüket tekintve - igen változatosak lehetnek. A 3b. ábra gráfja teljesen szabályos. A 4b. ábrában a szabályosságot már megtöri, hogy A_3 csomópont után (J=1 feltétellel) két B csomópont következik. Az 5b. ábra gráfja már sok szabálytalanságot tartalmaz, pedig a paralel programok teljesen szabályosak. A 8b. és 9b. ábrákban a globális változók nem változtatják meg a független paralel programok szekvenciális gráfjának szerkezetét - ez azonban nem általánosítható. Más esetekben különösen a globális változók jelenléte okoz szabálytalanságot a szekvenciális gráfban. Erre példát mutatnak a 10a. és 10b. ábrák. A 10a. ábrában az A_2 és A_3 csomópontokban a B_1 csomópont által kiszámított J változó értéke szerint ágazunk el; csakhogy A_2-ből két, A_3-ból három elágazás indul ki. B_1-ből továbblépve, a B program nemcsak J-től, hanem az A program aktuális csomópontjától is függ, ami egy globális változó bevezetését igényli a dekompozíció során (11a. ábra). Hasonló a helyzet a 10b. ábrában is. Itt az A_2 és A_3 csomópontokban a B_1 csomópont által kiszámított J változó értéke



Szekvenciális gráfrészletek

10. ábra



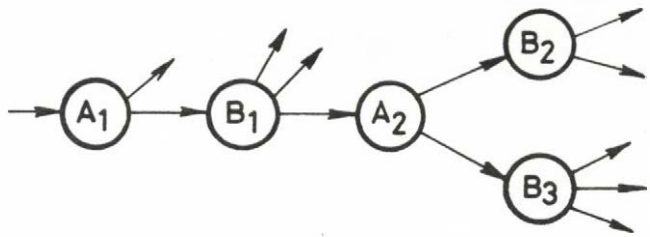
Parallel gráfrészletek

11. ábra

szerint ágazunk el; csakhogy az elágazás A és B program csomópontjaiba egyaránt irányul. Ez egy globális változó (J) és "do nothing" (DN) csomópontok bevezetését igényli a dekompozíció során (11b. ábra).

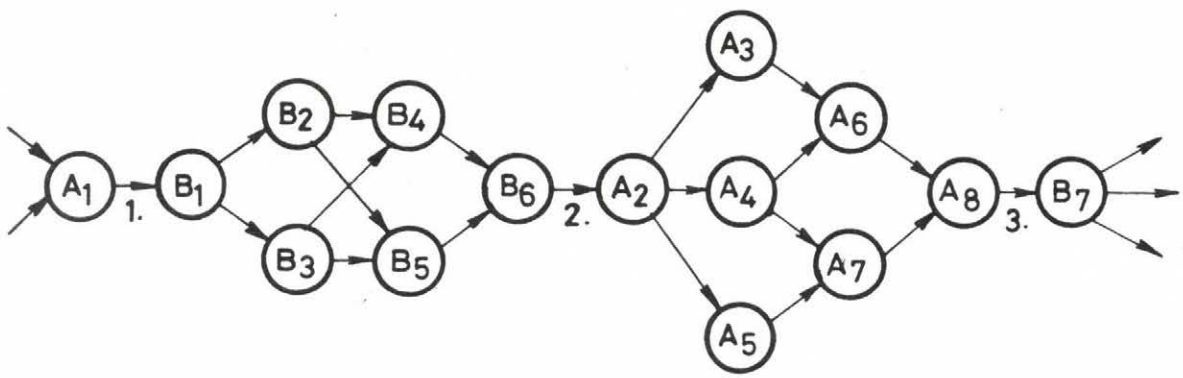
Beszélnünk kell még az azonnali IF utasítások hatásáról. A 12. ábrában egy olyan szekvenciális gráf részletét mutatjuk be, amely A és B paralel gráfokra dekomponálható, és összes elágazása azonnali IF utasítás. A szekvenciális gráf a paralel gráfok csomópontonkénti kompozíciójával áll elő. Tegyük fel, hogy a dekomponált rendszerben éppen A_2 csomópont kiszámítása folyik. Ekkor a B program sem nem haladhat (B_2 vagy B_3 számítása) - mert arra vár, hogy az A_2 csomópont mit számít ki -, sem megelőző számítást nem végezhet (B_1 számítása) - mert ez meg feltétele volt annak, hogy az A program dolgozhasson. Következésképpen a B program állni kényszerül, és nincs értelme a paralelizálásnak. (Ebből a helyzetből mutat kiutat a cikk 2.3 pontja.) Hasonlóak a viszonyok a 13. ábra szekvenciális gráfjában, azonban itt az 1, 2., 3. pontokban vagy nincs elágazás, vagy késleltetett IF van. A szekvenciális gráf a két paralel gráf részgráfonkénti kompozíciójával áll elő. A dekomponált rendszerben az azonnali IF-es részgráfok számítása egyidejűleg történhet, a paralelizálásnak van értelme.

Függésgráf. Mint már hivatkoztunk rá, a függésgráf a szekvenciális programgráf dekomponálásának segédeszköze. A függésgráf feltünteti a szekvenciális program utasításainak, változóinak egymástól való függését, és a programgráf finomításának tekinthető.



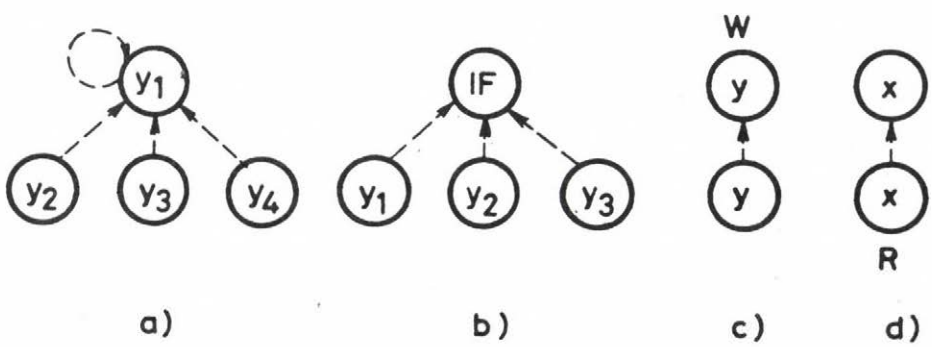
Szekvenciális gráf részlete

12. ábra



Szekvenciális gráf részlete

13. ábra



Utasitások függésgráfjai

14. ábra

A függésgráf csomópontjai a szekvenciális program utasításainak felelnek meg, élei a függéseket fejezik ki. Példaképpen a 14. ábrában egy értékadó utasítás (14a. ábra: $y_1=f(y_1,y_2,y_3,y_4)$), egy feltételes utasítás (14b. ábra: $IF(y_1+y_2*y_3)$), egy output utasítás (14c. ábra: WRITE y) és egy input utasítás (14d. ábra: READx) gráfrészletét adtuk meg. Meg kell jegyezni, hogy a GO TO utasításnak nem felel meg csomópont, és az output és input utasítások csomópontjai (14c. és 14d. ábrák W és R cimkéjű csomópontjai) a függések szempontjából elhagyhatók.

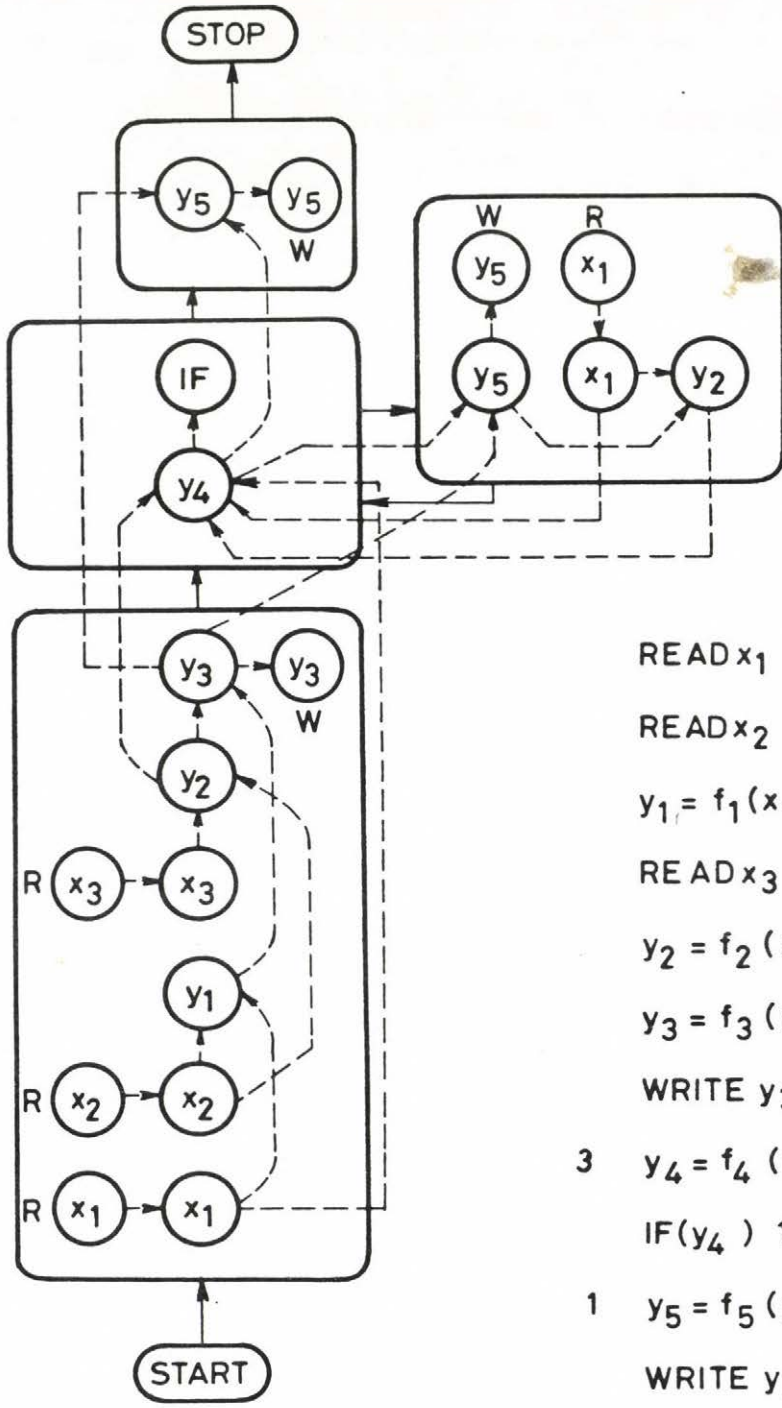
A függésgráf előbbi csomópontjait mikrocsomópontoknak tekinthetjük, mivel a gráf egyes részleteit is csomópontokká - makrocsomópontokká - foglaljuk össze. Egy makrocsomópont a ráugratási ponttól az elugró utasításig tart (amely IF vagy GO TO utasítás lehet). A makrocsomópontok közötti élek a vezérléselágazás lehetőségeit fejezik ki. A makrocsomópontok azonosak a programgráf csomópontjaival.

Példaképpen egy minta program függésgráfját a 15. ábrában adtuk meg (a program szövegével együtt). A függéseket szaggatott, a vezérléselágazásokat folytonos élekkel rajzoltuk meg. A függésgráf néhány jellemzője:

- az y_2 és y_5 változóknak két-két utasítás is ad értéket, ami példa a változó azonosítók újrafelhasználására,
- az y_4 változó értékének definiálása alternatív (két helyről történhet értékadás),
- a gráf egy ismeretlen ismétlésszámú hurkot tartalmaz.

A függésgráf felhasználásának néhány jellemzője általában:

- tükrözi, hogy egy makrocsomóponton belül mely mikrocsomópontok végrehajtását lehet paralelizálni,




```
      READ x1
      READ x2
      y1 = f1(x1 , x2 )
      READ x3
      y2 = f2 (x2 , x3 )
      y3 = f3 (y1 , y2 )
      WRITE y3
3     y4 = f4 (x1 , y2 )
      IF(y4 ) 1, 2, 1
1     y5 = f5 (y3 , y4 )
      WRITE y5
      READ x1
      y2 = f6 (x1 , y5 )
      GO TO 3
2     y5 = f7 (y3 , y4 )
      WRITE y5
      STOP
```

Minta program függésgráfja

15. ábra

- az egymástól nem függő makrocsomópont rendszerek csatolatlan paralel programokra való dekomponálás lehetőségére utalnak,
- az IF utasításokból kiolvashatók, hogy azonnaliak vagy késleltetettek-e, és hogy mely (más) makrocsomópontoktól függnek,
- meghatározhatók a makrocsomópontok közötti erős függések, ill. dekomponálás után a globális változók.

Problémát jelent a függésgráfban a hurkok jelenléte az egymástól nem függő makrocsomópont rendszerek (és a csatolatlan paralel programok számának) meghatározásában. Képzeljünk el ugyanis egy egyetlen hurokból álló függésgráfot, amelynek a programja N darab beolvasott egész szám hármas átlagát számítja és nyomtatja ki. Itt N darab csatolatlan paralel programunk lehet. Ez akkor válik nyilvánvalóvá, ha a függésgráf hurkát kifejtjük (N darab iteratív képpé). Ha a hurok ismétlésszáma ismeretlen (például a szám hármasokat egy végjelig kell feldolgoznunk), az iteratív képek számát limitálnunk kell, például a rendelkezésre álló processzorok száma szerint.

2.2 Dekompozíció átalakítással

Azt keressük, hogy késleltetett IF utasítások nélkül hogyan lehet dekomponálni egy szekvenciális programot. Erre módszer a dekompozíció átalakítással. A módszer alkalmazási körének határait nem ismerjük, de bemutatunk három példát, ahol a módszert alkalmazni lehetett. A módszer lényege mindhárom példában, hogy az azonnali IF utasításokat tartalmazó szekvenciális programrészletet átszervezzük úgy, hogy adat-eredményei azonosak marad-

janak, és paralelizáció legyen végrehajtható.

1. példa. Ciklus, a ciklusmagban IF utasítás nélkül: $y=x^8$ kiszámítása. Tegyük fel, hogy a feladat megoldásának szekvenciális megoldása a következőképpen adott:

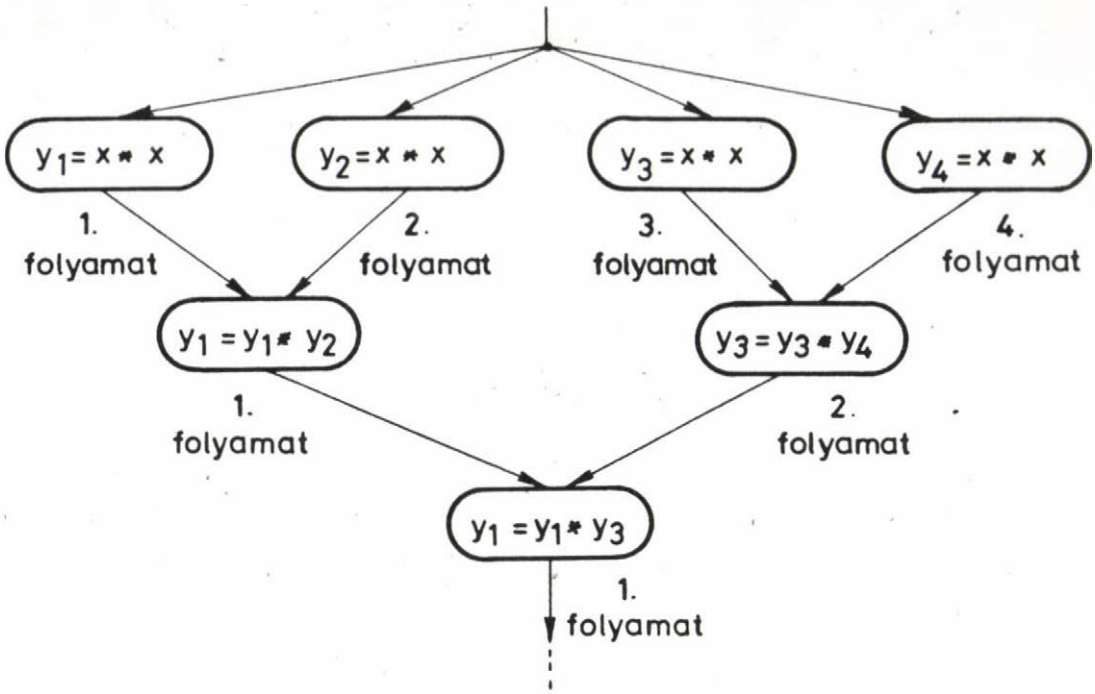
```
      y=x
      I=-7
  1   y=y*x
      I=I+1
      IF(I)1,2,1
  2   CONTINUE
      :
```

A szekvenciális megoldás a hatványozást a következő csoportosításban végzi el: y=((((((x*x)*x)*x)*x)*x)*x)*x). Ha most a szorzásra vonatkozó asszociatív azonosságot alkalmazzuk, a hatványkifejezés kiszámításának paralelizálható változatához jutunk: y=(((x*x)*(x*x))*((x*x)*(x*x))). A párhuzamos programváltozatot a 16. ábra mutatja be.

2. példa. Ciklikus keresés. A feladatban azt kérdezzük, hogy egy y változó aktuális értéke egyenlő-e $C_1, C_2, C_3, (...)$ konstansok valamelyikével. Ha egyenlő, I változó értéke jelezze a konstans sorszámát (indexét).

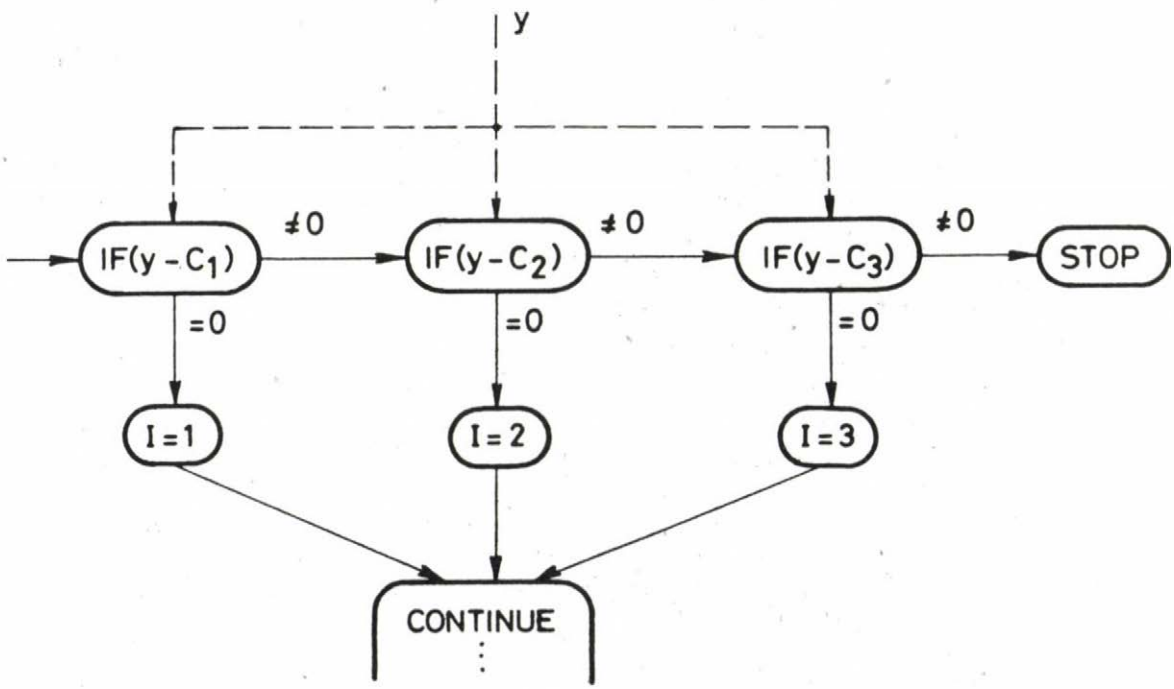
A feladat szekvenciális megoldását, az azonnali IF utasításokkal, a 17. ábra mutatja be. A programrészlet paralelizált változatát a 18. ábrában adtuk meg. A paralel megoldásváltozat nagyobb számú konstans esetén még értékesebb.

3. példa. Elágazásfa három kimenetelű IF utasításokkal.



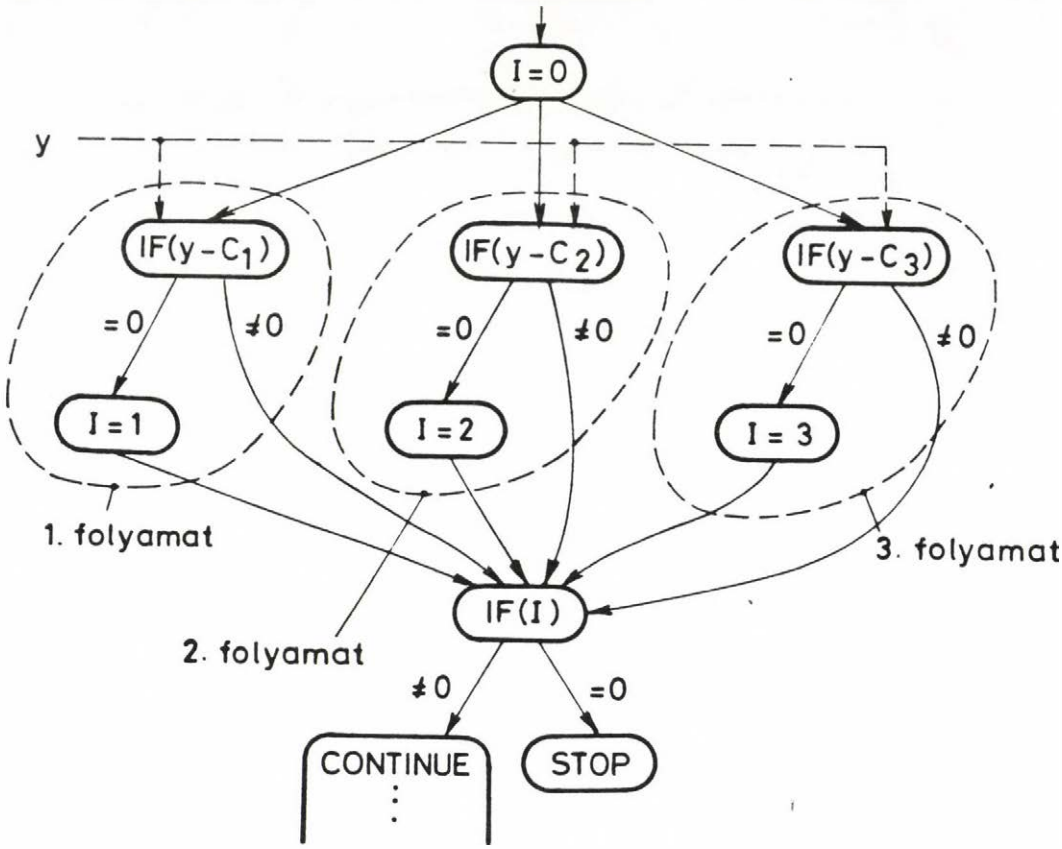
$y = x^8$ parallel kiszámitása

16. ábra



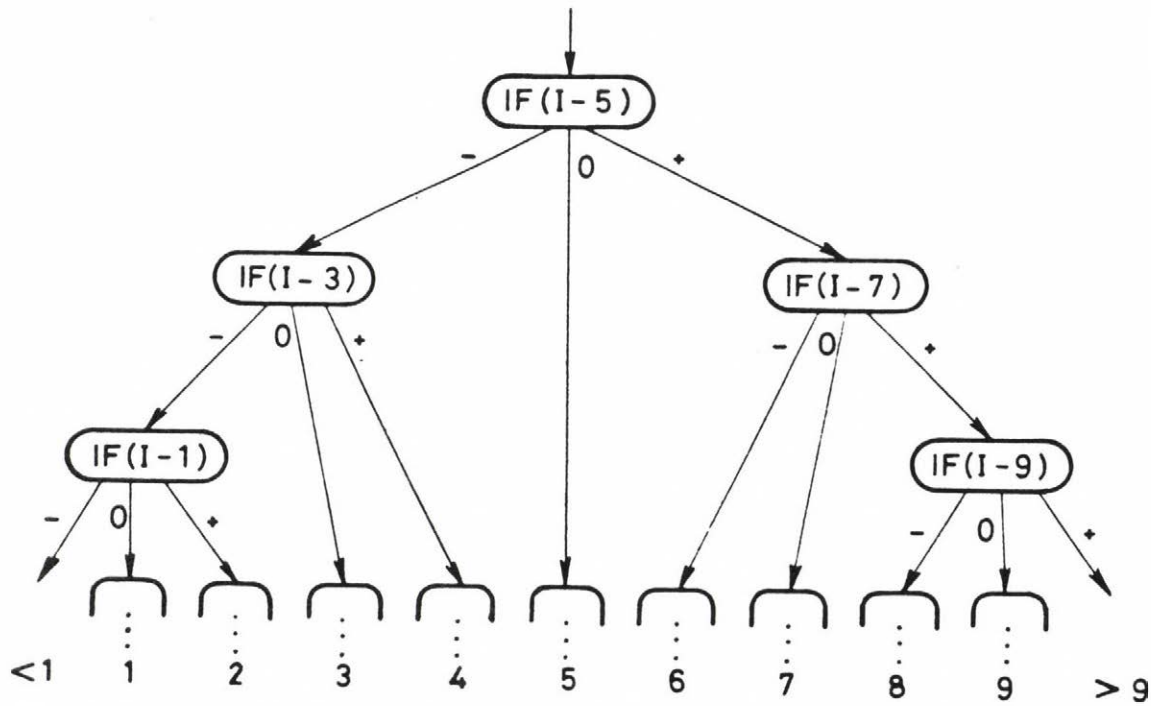
Ciklikus keresés szekvenciális változata

17. ábra



Ciklikus keresés parallel változata

18. ábra



Elágazásfa szekvenciális változata

19. ábra

A feladatban adott egy I változó aktuális értéke, amely - tegyük fel, hogy - 1...9 tartományba eső egész szám. Programelágazást kell végrehajtanunk I aktuális értéke szerint.

A feladatot megoldó szekvenciális programváltozatot a 19. ábrán adjuk meg; a paralel átalakított programváltozat a 20. ábrán látható. A paralel változat időben még előnyösebb, ha I aktuális értékének tartománya még nagyobb.

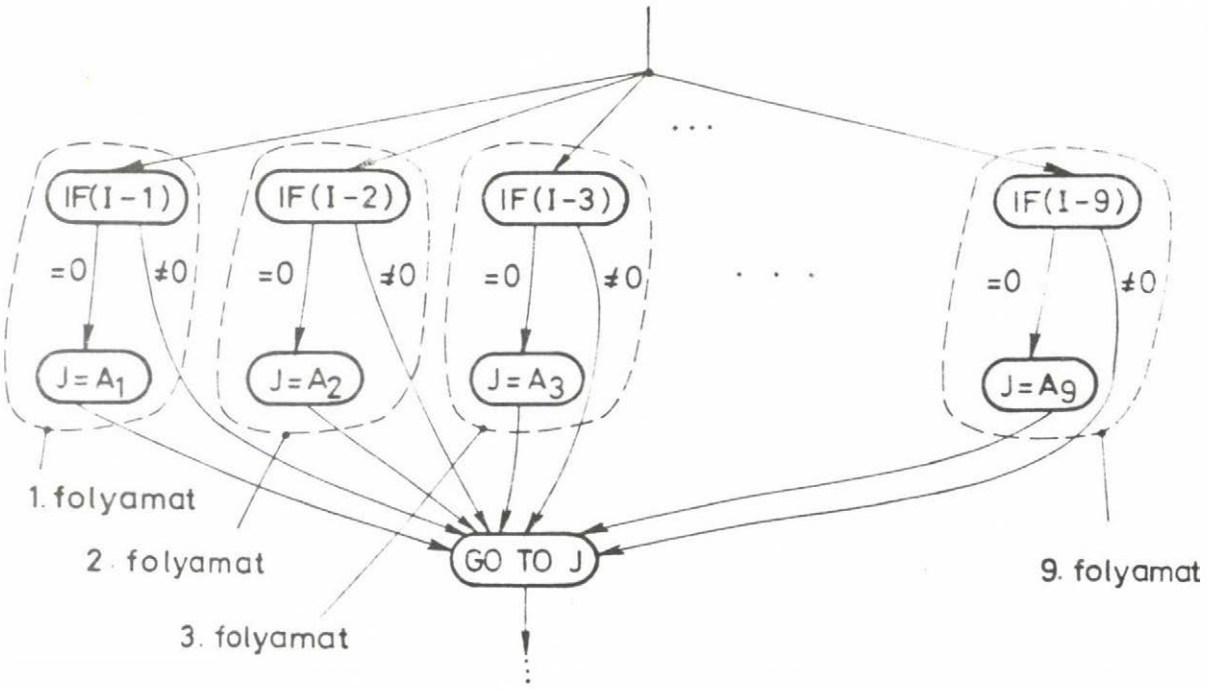
A paralel változat ábrájában A_i címet jelöl. A GO TO J utasítás assemblerben programozható (a $J=A_i$ utasítás a programmezőbe ír).

2.3 Dekompozíció redundáns számításokkal

A dekompozíciónak ez a módja kedvező, mert általánosan alkalmazható a szekvenciális programgráfokra, és azok IF utasításaitól nem kell megkövetelnünk, hogy késleltetettek legyenek. A módszer alkalmazásának viszont az az ára, hogy a módszer a minimálisnál nagyobb számú paralel processzort igényel - viszont olyan gyors, amennyire az egyáltalán lehetséges.

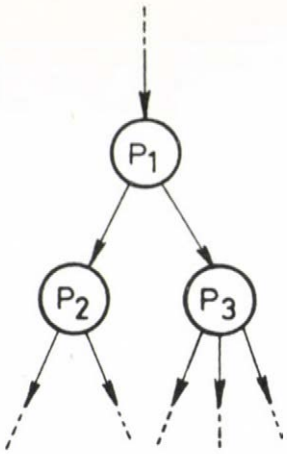
A módszert a szekvenciális gráf jellege (fagráf, rekonvergenciás gráf, hurkos gráf) szerint tagoltan mutatjuk be.

a) Fagráfok. Legyen adott egy szekvenciális gráfnak a 21. ábrán látható részlete. P_1,P_2 és P_3 kimeneti elágazásai: azonnali elágazások. A gráfrészletet 3 paralel folyamatra dekomponáljuk - P_1,P_2 és P_3 programját szi-



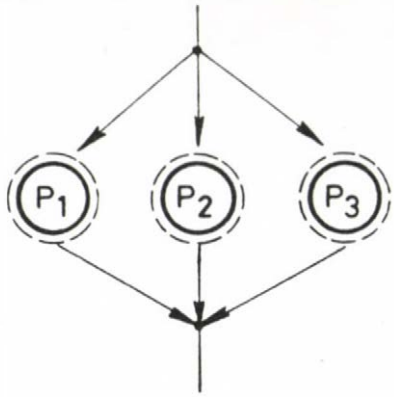
Elágazásfa parallel változata

20. ábra



a)

Szekvenciális program-gráf részlet

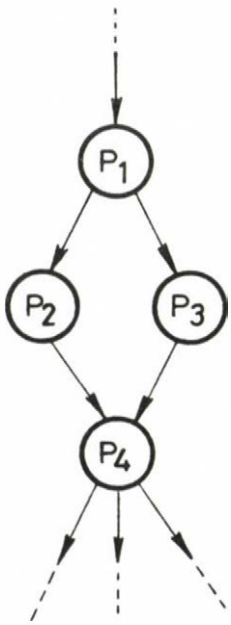


párhuzamos folyamatok

b)

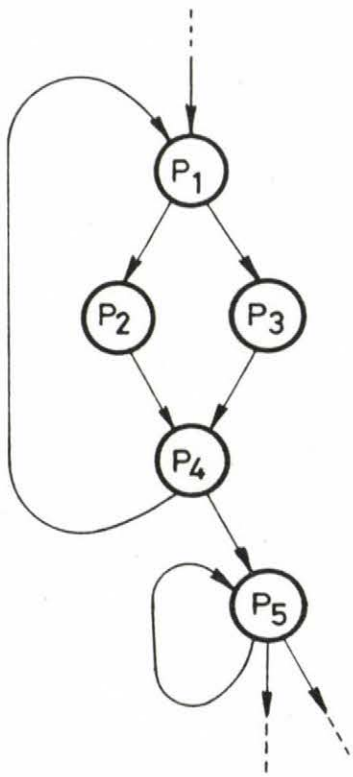
Parallel dekompozíció

21. ábra



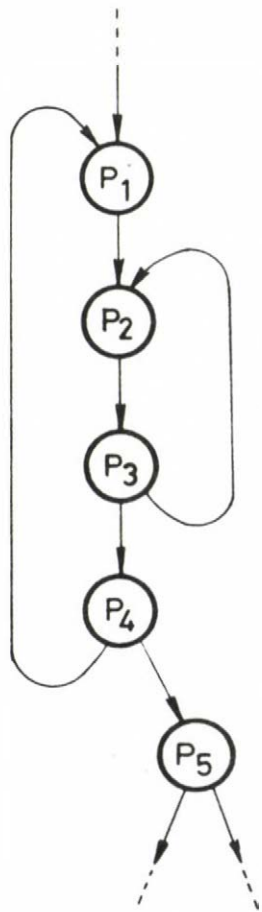
Rekonvergencia

22. ábra



a)

Diszjunkt hurokrendszer



b)

Csatolt hurokrendszer

23. ábra

multán számítjuk.

Először, az egyszerűség kedvéért, tegyük fel, hogy P_2 és P_3 nem függ P_1-től. Amíg P_1 számítása folyik, a paralel rendszerben P_2-t és P_3-t is számítjuk; P_1 befejeződése (a záró IF kiértékelése) után döntjük el, hogy P_2-t vagy P_3-t tartjuk meg, a másik paralel számítást a továbbiak számára elvetjük (az ugyanis redundáns). P_2 és P_3 végrehajtása során műveleti paralelitások is felléphetnek: pl. P_2 és P_3 ugyanazon változókra is tartalmazhat értékadást, vagy P_2 és P_3 egyaránt akarhat olvasni. Ezeket a problémákat egyszerűen megoldhatjuk: pl. a mindkét folyamat által számított változók értékeit először csak a lokális memóriában képezzük, az olvasás számára az adatokat előpufferelјük. Meg kell jegyezni, hogy P_2 és P_3 output adatokat is képezhetnek. Ezek kivitelét (print, punch) késleltetnünk kell addig, amíg eldől, hogy P_2-t vagy P_3-t tartjuk meg. Egyszerűen adódik, hogy P_2 és P_3 követői is (tehát egy teljes gráf) paralel számíthatók a fenti módon.

Ha P_2 és P_3 függnek P_1-től legalább egy változón keresztül, akkor a paralel folyamatokat szinkronizálni kell:

- amíg a végrehajtás során P_2 és P_3 nem függnek P_1-től, addig akadálytalanul számíthatók,
- azokon a pontokon, ahol P_2 és P_3 függnek P_1-től, ott esetleg várnak P_1-re (ha az még nem számította ki az összekötő változó értékét).

b) Rekonvergenciás gráfok. A rekonvergenciás gráfokban új jelenség a változó értékek alternatív definiálása, amely a rekonvergencia helyén fekvő csomópontban, a re-

konvergáló ágak szerint lép fel.

A rekonvergenciás gráfokban a dekompozíció során a szekvenciális csomópontoknak külön-külön folyamatokat feleltetünk meg, azonban a rekonvergencia helyén fekvő és az azt követő csomópontoknak lehet, hogy több folyamatot kell megfeleltetni.

Tekintsük a 22. ábrán megadott szekvenciális gráfrészletet, amely egyszerű rekonvergencia minta. A gráfrészlet dekompozíciójához a következő megjegyzéseket fűzzük:

- Ha P_4 nem függ P_2-től és P_3-tól, akkor P_4-et csak egyszer számítjuk ki.
- Ha egy I változónak P_2 és P_3 is ad értéket (alternatív definiálás), és P_4 függ I-től, P_4-re két számítást indítunk a kétféle I értékkel, és P_1 kiszámításának befejezése után P_4 egyik számított változatát elvetjük.
- Ha P_2 és P_3 más-más változóknak ad értéket, és P_4 függ függ P_2-től és P_3-tól, P_4-et akkor is kétszer kell kiszámítani.

Általában igaz a rekonvergenciás gráfok dekompozíciójára, hogy

- ha a rekonvergencia csomópontja függ valamelyik ágtól, a rekonvergáló gráfot fává kell kifejteni a rekonvergencia csomópontjának megsokszorozásával,
- ha a rekonvergencia csomópontjának valamelyik követője függ valamelyik ágtól, akkor a csomópontban (azaz a függés helyén) multiplikálni kell a számítást,
- ha minden csomópont függ minden megelőzőtől, akkor az összes programút mentén az összes csomópontot kiszámítjuk.

A következő tárgyalás az a) és b) esetre közösen (hurokmentes szekvenciális gráf) érvényes:

- A szekvenciális programgráf leggyorsabb paralel végrehajtása úgy történik, hogy minden szekvenciális csomópontnak megfeleltetünk egy paralel folyamatot, figyelembe véve, hogy rekonvergencia és függések esetén egy csomópontnak több folyamatot kellhet megfeleltetni, valamint, hogy ha egy csomópont számítása (a függések miatt) csak egy másik csomópont kiszámítása után kezdődhet meg, akkor folyamataikat ugyanarra a processzorra allokálhatjuk.

- A szekvenciális programgráfban (ugyanazon vagy különböző csomópontokban) ugyanazon változó többször is szerepelhet értékadás bal oldalán (ha a változó használata egyszer befejeződött, azt más célra újra felhasználhatjuk). A paralel dekompozícióban a változó különböző, egymás utáni használatait át kell betűznünk, hogy a függésekben a változó értéke akkor hasznosuljon, amikor az a szekvenciális programban ténylegesen megtörténik.

- A szekvenciális programgráf egy csomópontjában egy változó többször is kaphat értéket. Ha más csomópont függ ettől a változótól, a szinkronizációban a flag bitet (szót) a kívánt (rendszerint legutolsó) értékadás billentse be. Pl. az $I=3I^2+1$ utasítást 3 alutasításra bontva programozva:

```
I=I**2
I=3*I
I=I+1 ... erről szinkronizálunk.
```

A más csomóponttal csatoló változó-megjelenést eltérően is betűzhetjük, ugyanis míg a megelőző változó-megjelenések a lokális memóriába, a csatoló megjelenés a közös memóriába írja értékét.

- Ha a hurokmentes szekvenciális programgráf paralel folyamatai közel egyszerre fejeződnek be, akkor a paralel programozott tevékenység 3-passzos:
  -- paralel folyamatok végrehajtása (print, punch outputok kivételével, ezeket felfüggesztjük),
  -- a szekvenciális programgráfnak (IF utasítások elágazásának) megfelelő aktuális folyamatok kijelölése (redundáns számítások elvetése),
  -- felfüggesztett output tevékenységek végrehajtása sorban egymás után (a szekvenciális gráfban végrehajtott egyetlen programút mentén).

Ha a paralel folyamatok nem egyszerre fejeződnek be (pl. mert nem egyforma hosszúak a folyamatok, vagy mert várniuk kell más folyamatokra) - és ez az általános eset -, akkor esetleg már a végrehajtás során (közben) letilthatunk redundáns számításokat, amelyek a teljes végrehajtási időt feleslegesen növelnék. Ehhez egy kisegítő processzort alkalmazhatunk, amely egy-egy IF elágazás kiszámítása után mindig letiltja az éppen redundánssá váló programszámításokat (pl. interrupttal).

c) Hurkos gráfok. A hurkos szekvenciális programgráfnak két típusa lehet aszerint, hogy hurkaiknak van-e közös csomópontjuk:
- diszjunkt (23a. ábra),
- csatolt (23b. ábra)
hurokrendszerű gráfok.

A hurkoknak, a rajtuk való áthaladások száma szerint, négyféle típusuk lehet:

1. véges, fix hosszúságú (pl. DO ciklus, a magjában IF nélkül),
2. véges, limitált hosszúságú (pl. DO ciklus a magjában IF utasítással),
3. véges, lebegő hosszúságú (pl. input adatoktól függő),
4. végtelen hosszúságú.

Tekintsük először a diszjunkt hurokrendszerű programgráfokat. Az 1. és 2. típusú hurkokat véges részgráffá kifejthetjük (iteráció), és kezelésük olyan, mint a hurokmentes gráfoké. (Minden iteratív hurokképben a többihez képest új változókat kell alkalmaznunk.) Előfordulhat, hogy az iterációk száma túl nagy lenne, és így a módszer nem realizálható.

Az 1., 2., 3. és 4. típusú hurkokat véges, fix (limitált) számú processzorra kell allokálnunk. (Fix számú processzorra, mert már dekomponáláskor eltervezünk mindent, és a paralel folyamatok végrehajtásakor processzorokon programot nem akarunk sem generálni, sem módosítani.) Ezt legegyszerűbben úgy tehetjük meg, hogy minden hurokcsomópontnak egy processzort feleltetünk meg (ill. a rekonvergencia és függések miatt esetleg többet) - annak megfelelően, hogy a hurkon való egyszeri áthaladást paralelizáljuk. (Lehetne még kétszeri, háromszori,... áthaladást paralelizálni.) Így a lehetséges maximális párhuzamosságot szükségszerűen korlátozzuk. (A rekonvergencia és függés a hurkon való áthaladások számát tetemesen felduzzaszthatja: az egy hátraugrási pont helyett k darab

potenciális pont jelentkezik párhuzamosan. Szerencsére a hurkon való egyszeri áthaladás paralelizálásánál ilyen jelenség nem lép fel, mert rögtön kiszámítjuk az aktuális hátraugrási pontot.)

Tekintsük most a csatolt hurokrendszerű programgráfokat. A csatolt hurokrendszerek típusa - hurkaik szerint - vegyes lehet. A tisztán 1-es és 2-es típusú csatolt hurokrendszereket (vagy a vegyes típusú hurokrendszerekben az 1-es és 2-es típusú hurkokat) teljesen kifejthetjük (iteráció), például skatulyázott ciklusok esetén. A módszer azonban itt sem mindig realizálható.

Általános esetben a csatolt hurokrendszerek egy kifeszítő hurokmentes részgráfját vihetjük egyidejűleg processzorokra. (Azért nem kifeszítő fagráfot választunk, mert rekonvergenciát is megengedhetünk.) Ez egyenlősíti az egyes programágak végrehajtására tett kísérletek számát (a rekonvergenciától eltekintve, a szekvenciális programgráf minden útját egyszer hajtjuk végre), holott a valóságban a különböző programágakat különböző számúszor kellhet végrehajtani (még potenciálisan is). Mást azonban nem tehetünk az IF-ek végrehajtása előtt.

A kifeszítő hurokmentes részgráf paralel programozása annak felel meg, hogy egyszer áthaladunk a hurokrendszeren. Lehetne a párhuzamosítást azzal fokozni, hogy a kétszeri, háromszori,... áthaladást vesszük paralel programozási egységnek, de ez bonyolultnak tűnik. (Tulajdonképpen ugyanezt csináltuk a diszjunkt hurokrendszerű programgráfoknál is, de ott a kifeszítő részgráf egy lineáris programrészletet jelentett.)

A kifeszítő hurokmentes részgráf (egyszeri) végrehajtása után aktualizálódik a folytatási pont, amely vagy egy visszacsatolási pont lehet (ahol felvágtuk a hurokrendszert, hogy kifeszítő részgráfot kapjunk), vagy egy exit pont (ahol elhagyjuk a hurokrendszert). Ha visszacsatolási ponton folytatjuk a program végrehajtást, képeznünk kell az adott pontra is a kifeszítő hurokmentes részgráfot - ezt is egyszer végrehajtjuk, és így tovább... Az összes kifeszítő hurokmentes részgráfot már a szekvenciális program dekomponálásakor képezzük, és nem a párhuzamos programok végrehajtásakor. (Egyszerű a helyzet, mert minden kifeszítő részgráf ugyanazokat a csomópontokat tartalmazza, csak a gráf struktúrája, ill. gyökere változik.) A kifeszítő részgráfok halmazát ugyanarra a processzorhalmazra allokálhatjuk; a processzorok minden kifeszítő részgráfban ugyanazon programot hajtják végre, csak a processzorok kapcsolatai mások.

3. Konklúzió

A cikkben a szekvenciális és paralel programok konverziójának kérdéseit vizsgáltuk. A témával a szerző a szakirodalomban a mai napig nem találkozott, így a cikk anyaga úttörő munka, amely további folytatást igényel.

A konverzió két fajtájára módszereket mutattunk be.
A kompozíciónak multiprogramozásos és multiprogramozás nélküli változatait tárgyaltuk. A dekompozícióra a törzdeléses, az átalakításos és a redundáns számításos módszereket adtuk meg.

Konverziós módszereinket elsősorban mint elveket és útmutatásokat mutattuk be. Nem adtuk meg általában az alkalmazhatóságuk határait és az egzakt algoritmusukat - ezek tisztázása még hátra van. Annyi azonban bizonyos, hogy a tárgyalt módszerekkel a konverzió mindig elvégezhető, sokszor nem is egyféleképpen.

A TANULMÁNYSOROZATBAN 1980-BAN JELENTEK MEG:

101/1980 Gerencsér László - Hangos Katalin:
Diszkrét lineáris sztochasztikus rendszerek
önhangoló szabályozása

102/1980 Pásztorné Varga Katalin: Rekurziv eljárás

103/1980 Gerencsér Piroska - Szép Endre - Zilahy Ferenc
Marton Zsolt: Robotmegfogók adaptivitása I.

104/1980 Knuth Előd - Radó Péter - Tóth Árpád:
A SDLA előzetes ismertetése

105/1980 E. Knuth - P. Radó - Á. Tóth:
Preliminary description of SDLA

106/1980 Prékopa András: Sztochasztikus programozási
modellek és alkalmazásuk

107/1980 Kelle Péter: Megbizhatósági készletmodellek
és alkalmazásuk

108/1980 Almásy Gedeon: Mérlegegyenletek és mérési hibák

109/1980 Békéssy A. - Demetrovics J. - Gyepesi Gy.:
Relációs adatbázis logikai szintü vizsgálata
funkcionális függőségek szempontjából

110/1980 Gaál A. - Soltész J. - Ruda M. - Ratkó I.:
Tanulmányok a statisztikai adatfeldolgozásról

111/1980 Benedikt Szvetlána: Nem ismételhető döntéshozatal
analizise kockázattal járó esetekben

112/1980 Verebély Pál: Többprocesszoros, osztott intel-
ligenciáju grafikus rendszerek tervezési és meg-
valósitási kérdései

113/1980 V. Visegrádi Téli Iskola

114/1980 Demetrovics János: Relációs adatmodell logikai és strukturális vizsgálata

115/1980 Gergely József: Program package for sparse matrices

1981-BEN JELENTEK MEG:

116/1981 Siegler András: Egy 6 szabadságfoku antropomorf manipulátor kinematikája és számitógépes vezérlése

117/181 Knuth Előd - Radó Péter: Principles of Computer Aided System Description

118/1981 Demetrovics János - Gyepesi György: Általános függések és lekérdezéssel kapcsolatos algoritmusok relációs adatmodellekben

119/1981 Sztanó Tamás: REAL-TIME programrendszerek esemény-vezérelt szervezése

120/1981 Szentgyörgyi Zsuzsa: A számitástechnika müszaki fejlődése és társadalmi hatásai

121/1981 Vicsek Tamásné (Strehó Mária): Vizsgálatok a kezdeti érték problémák numerikus megoldásával kapcsalatban

122/1981 Andó Györgyi-Lipcsey Zsolt: Sztochasztikus Ljapunov módszerek és alkalmazásaik

123/1981 Márkusz Zsuzsanna: Intelligens interaktiv rendszerek elvi problémái

124/1981 Márkusz Zsuzsanna: Logikai alapu programozási módszerek és alkalmazásaik számitógéppel segitett épitészeti tervezési feladatok megoldásához

125/1981 Fabók Julianna: Software implementációs nyelvek