Tartalomjegyzék

Tartalomjegyzék ... 1

Bevezetés ... 3

1. Absztrakt soros program párhuzamosítása ... 5

   1.1. Függőségi programgráf ... 6

   1.2. Élekkel kiterjesztett függőségi programgráf ... 9

   1.3. Állapotokkal kiterjesztett függőségi programgráf ... 12

2. Objektum-elvű közös modell felépítése ... 14

   2.1. Statikus modell ... 14

   2.2. Dinamikus modell ... 17

3. Párhuzamosítás ... 24

   3.1. Program refaktorálása ... 24

   3.2. Közvetett jobb- és baloldali változók meghatározása ... 26

   3.3. Függőségi programgráf felépítése ... 29

4. Párhuzamosító módszerek összehasonlítása ... 37

   4.1. Programábrázolás ... 37

   4.2. Párhuzamosítási különbségek ... 39

      4.2.1. Író/Olvasó probléma ... 39

      4.2.2. Elágazás, ciklus probléma ... 39

      4.2.3. Függvények ki- és bemenetei ... 40

5. Programoptimalizálás ... 41

6. Fejlesztői dokumentáció 44
6.1. Objektum-elvű program modell 44
6.2. Párhuzamosítás 50
6.3. Szimulációs programrész 53
7. Szimuláció 55
7.1. Szimuláció eredménye 57
Összefoglaló 66
Irodalomjegyzék 67
Csatolmányok 69

Bevezetés

Az informatika fejlődésével egyre gyakoribbá válnak a többprocesszoros számítógépek. Igény van arra, hogy a mai programok kihasználják ezeknek a lehetőségeit. Meg lehet közelíteni ezt a problémát úgy, hogy több processzorra tervezzük az algoritmusainkat. Ilyen párhuzamos algoritmusokról *Iványi Antal: Párhuzamos algoritmusok,*[1] és *Horváth Zoltán: Párhuzamos és elosztott programozás*[2] című munkáiban olvashatunk. Én másfajta megközelítést használok. A célom az, hogy egy meglévő soros programot átalakítsunk párhuzamossá. A különböző programozási nyelvekhez különféle megoldások szükségesek. Funkcionális nyelvekkel *Bazhanov, Kutepov és Shestakov: Functional Parallel Typified Language and Its Implementation on Clusters*[3] című cikk foglalkozik. A dolgozat az objektum-elvű programokat dolgozza fel. A soros program párhuzamosítása azzal az előnnyel jár, hogy az eddig megírt soros programok képesek lennének kihasználni a többprocesszoros gépek által nyújtott lehetőségeket az átalakítás után. Másik előnye, hogy nem kéne eleve párhuzamos programokat írni, az erőforrások hozzáférése áttetsző lenne a programozó számára.[4] Ilyen jellegű párhuzamosító megközelítésről *Halász Attila Péter: Objektumorientált, szekvenciális programok átalakítása párhuzamos környezet számára hatékony implementációval*[5] munkájában szintén olvashatunk.

---

1 Iványi Antal: Párhuzamos algoritmusok, ELTE Eötvös Kiadó, Budapest, 2003.
http://compalg.inf.elte.hu/~tony/Oktatas/Parhuzamos-algoritmusok/Parhuzamos-algoritmusok.pdf

2 Horváth Zoltán: Párhuzamos és elosztott programozás, ELTE Informatikai Kar, Elektronikus kézirat, 2005. http://www.inf.elte.hu/karunkrol/digitkonyv/Jegyzetek2004/ParhProg.pdf

3 S. E. Bazhanov, V. P. Kutepov, and D. A. Shestakov: Functional Parallel Typified Language and Its Implementation on Clusters, Programming and Computer Software, Vol. 31, No. 5, 2005, pp. 237–269. Translated from Programmirovanie, Vol. 31, No. 5, 2005.
http://www.springerlink.com/content/u22q05g62q251702/fulltext.pdf

4 Andrew S. Tanenbaum - Maarten Van Steen: Elosztott Rendszerek Alapelvek és Paradigmák, Panem Kft., 2004.

5 Halász Attila Péter: Objektumorientált, szekvenciális programok átalakítása párhuzamos környezet számára hatékony implementációval, 2006.
http://compalg.inf.elte.hu/~tony/Oktatas/Diplomamunka-Szakdoli-Nagyprogram/Halasz%20Attila%20-%20Diplomamunka.zip

Minden objektum-elvű nyelv szintaktikailag máshogy épül fel, ezért szükség van egy közös modellre, ahol ezeket egységesen tudjuk kezelni. Ezt a közös modellt felhasználva a következő lépésekből áll majd a párhuzamosítás folyamata:

1. **Objektum-elvű programnyelvben megírt program átalakítása a közös modellre.**
2. **A közös modellben végrehajtjuk a programon a párhuzamosítást.**
3. **A közös modellben elkészített párhuzamos programot visszaalakítjuk egy objektum-elvű programnyelvre.**

Az első és a harmadik lépés nyelvfüggő, míg a második lépés minden nyelv esetén ugyanaz. A dolgozatban a második lépéssel foglalkozok részletesen, vagyis a közös modellben végrehajtott párhuzamosítással. A munkám első fele a következő fő részekből tevődik össze:

1. **A párhuzamosító módszer alapelvének a bemutatása absztrakt programon.**
2. **Az objektum-elvű nyelvekhez felépítem az általam használt közös modellt.**
3. **A közös modellen végrehajtom a párhuzamosítást.**

A dolgozat második részében összehasonlítást végzünk Halász Attila módszerével, majd megmutatjuk, hogy a dolgozatban tárgyalt párhuzamosító módszernek egy része hogyan használható programoptimalizálásra. A párhuzamosító módszer bemutatására egy szimuláló programot készítettem, ennek a fejlesztői dokumentációjával és szimulációs eredményeivel zárom a dolgozatot.

A dolgozatban sok ábrával találkozunk, ezeket mind magam készítettem.

1. Absztrakt soros program párhuzamosítása

Ebben a részben a párhuzamosító módszer alapelvét fogom bemutatni egy absztrakt soros programon. A programot direktszorzat[6] segítségével írjuk le.

$$S = A \times A^{*},\ A = A_1 \times A_2 \times \ldots \times A_n,\ \ A^{*} = (A^1, A^2, \ldots, A^m),$$

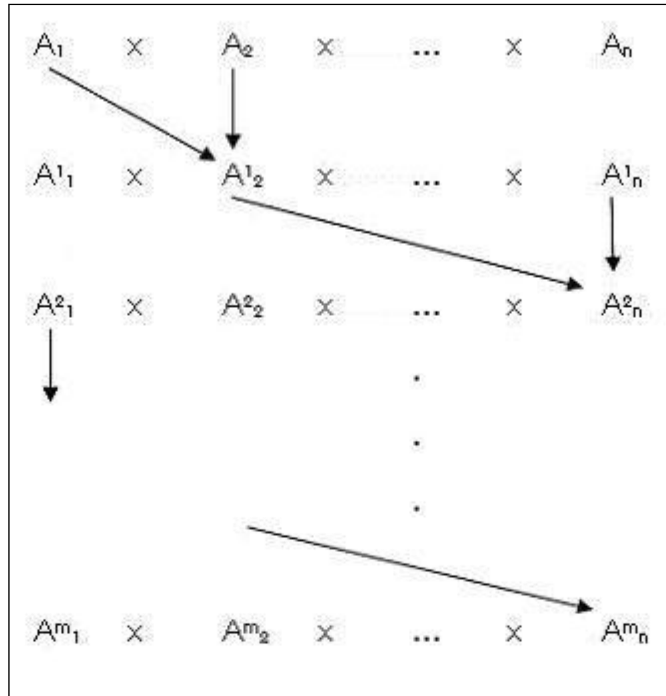
ahol *S* az *A* állapottér felett értelmezett programunk. A reláció első tagja a program kezdő állapota, a második tagja a program állapotainak véges sorozata. A dolgozatban csak ilyen programokkal foglalkozunk, olyanokat nem foglal magába, ahol a második tag egy végtelen sorozat.

Bevezetem a **programgráf** fogalmát. Az *1. ábrán* látható programgráfban az alsó index jelöli az állapot alterének a sorszámát, a felső index a végrehajtási lépések, másképpen nevezve az utasítások sorszámát. Minden egyes lépésben értéket adunk valamelyik altérnek, azaz egy utasítást hajtunk végre. Soros esetben egy lépésben csak egy altér kaphat értéket. Akkor húzunk be egy-egy nyilat, ha az adott alteret meghatározza az előző állapotból származó altér, azaz függ tőle. Ezt a gráfot nevezzük *S* programgráfjának. Mivel nagyobb sorszámú lépésből kisebb sorszámú lépésbe nem mutathat él, biztosan körmentes lesz a gráfunk. Ezt az absztrakt programot úgy párhuzamosítjuk, hogy megvizsgáljuk, hogy mely utasítások azok, amelyeket ha egyszerre hajtunk végre, nem változtatják meg a program programfüggvényét[7]. Azaz a kezdőállapotok halmaza megegyezik, és ha az eredeti programban „*a*" kezdőállapothoz „*B*" végállapot halmaz tartozik, akkor a párhuzamosított programban is „*a*" kezdőállapothoz „*B*" végállapot halmaz tartozik.

---

[6] Fóthi Ákos, Horváth Zoltán: Bevezetés a programozásba, Budapest, ELTE Eötvös Kiadó, 2005., 22. oldal http://www.inf.elte.hu/karunkrol/digitkonyv/Jegyzetek2004/BevProg.pdf

[7] Fóthi Ákos, Horváth Zoltán: Bevezetés a programozásba, Budapest, ELTE Eötvös Kiadó, 2005., 23. oldal http://www.inf.elte.hu/karunkrol/digitkonyv/Jegyzetek2004/BevProg.pdf



1. ábra: Programgráf

A párhuzamosításhoz egy új gráfot építünk a programgráfból, amelyet **függőségi programgráfnak** neveztem el. Ezzel az új gráffal foglalkozik a következő rész.

1.1. Függőségi programgráf

Egy olyan gráfot állítunk elő, ahol a csúcsok a következők: *egyrészt* minden végrehajtási lépés egy csúcs, *másrészt* a kezdőállapotokból azok az alterek csúcsok, amelyekből vezet kifelé él. A csúcsokat megcímkézzük rendezett párokkal oly módon, hogy a párok első tagja az altér neve, amely értéket kap, a második tag pedig egy szám, amely azt jelöli, hogy az adott altér hányadszor kap értéket a program futása során. Ezt a számot **előfordulás számlálónak** nevezzük a továbbiakban. A kezdőállapotból származó csúcsokat úgy kell tekinteni, mintha az az altér kapott volna értéket, amelyből származik. A gráf felépítése a következő algoritmussal történik:

1. Felvesszük a kezdőállapotokhoz tartozó csúcsokat.
2. **for**(i=1; i<végrehajtási lépések száma; i++)
3. létrehozzuk az *i.* végrehajtási lépéshez tartozó csúcsot (*A,b*) címkével, ahol *A* az alteret jelöli, amelyet az utasítás megváltoztat, *b*

pedig azt, hogy hányadik változása ez az altérnek a függőségi programgráf felépítése során.

4. Az *i*-edik végrehajtási lépésben a megváltoztatott altérbe élek mutattak korábbi alterekből. Kiválasztjuk ezeket a korábbi altereket.
5. **for**(j=1;j<=kiválasztott alterek száma;j++)
6. Megkeressük az eddig felépített függőségi programgráfunkban a j-edik altérrel címkézett csúcsot. Ha több ilyen van, akkor azt vesszük, amelyikhez a legnagyobb előfordulás számláló tartozik. Egy élt húzunk ebből a csúcsból az új csúcsba.
7. Az (*A,b-1*) címkéjű csúcsból egy élt húzunk az új csúcsba. Későbbiekben ezt **triviális függőségnek** nevezzük.

A 7. lépés azért szükséges, hogy megakadályozzuk, hogy a változó helytelen sorrendben vegye fel az értékeit. A végrehajtási lépésekhez rendelt csúcsok a programnak azokat az állapotait reprezentálják, amelyek az adott utasításkor végrehajtódtak. Ezután a csúcsokat állapotoknak is nevezzük. A függőségi programgráfot a következő példa szemlélteti:

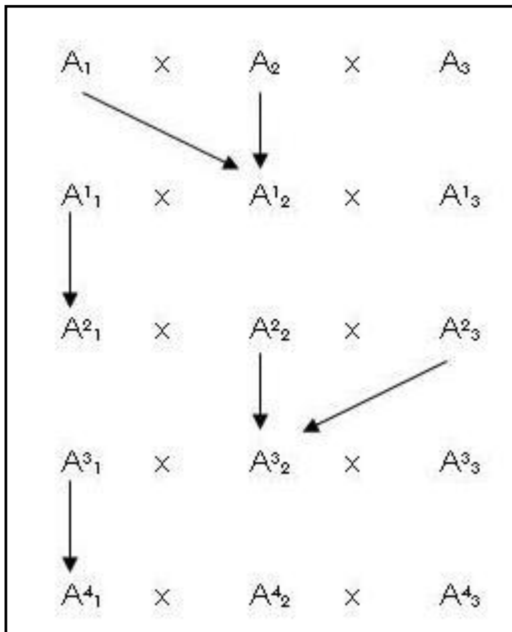
$A=A_1 x A_2 x A_3$

$S=Ax(A^1,A^2,A^3,A^4)$

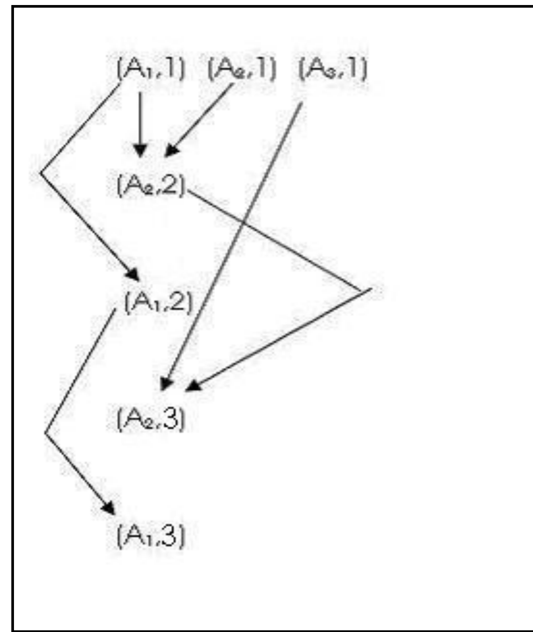
$A_1=a,\ A_2=b,\ A_3=c$

```
Kezdetben: a=1 b=2 c=3
b=a+b;
a=a+1;
b=b+c;
a=a+2;
```

A *2. és 3. ábrán* láthatjuk a példa programgráfját, és függőségi programgráfját.



2. ábra: A példa programgráfja



3. ábra: A példa függőségi programgráfja

Egy állapot lehet nem kiszámított, kiszámítható, és kiszámított. Egy állapot közvetetten függ egy másik állapottól, ha a másik állapotból vezet felé út. Jól látható, hogy a példában vannak egymástól teljesen független állapotok, és mi ezt kihasználva párhuzamosítjuk ezeknek a kiszámítását. Egy állapot kiszámítható, ha az összes olyan állapot, amelyből él vezet bele, kiszámított. Ha több kiszámítható állapot van egyszerre, akkor ezek kiszámítása párhuzamosan elvégezhető.

A függőségi programgráf önmagában még nem biztosít helyes megszorításokat az állapotok kiszámításához. A probléma bemutatásához ütemezzük a példa állapotainak kiszámítását két processzorra. Az egyszerűség kedvéért most feltételezzük, hogy az állapotok kiszámítása egységnyi idejű. Ekkor a LEV algoritmussal[8] a következő ütemezést kapjuk két processzorra:

[8] Iványi Antal: Processzorütemezés, Budapest, Elektronikus kézirat, ELTE Informatikai Kar, 2009., 18. oldal http://compalg.inf.elte.hu/~tony/Oktatas/Utemezesi-algoritmusok/utemezes-2010_marcius_23.pdf

P1: $(A_2,2)$, $(A_2,2)$, azaz: b=a+b; b=b+c;

P2: $(A_1,2)$, $(A_1,3)$, azaz: a=a+1; a=a+2;

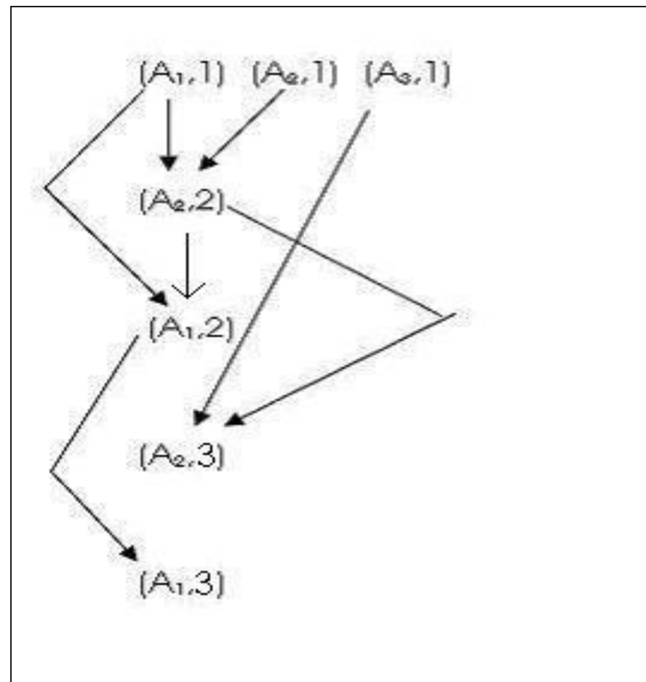
Ugyan a *b=a+b* értékadás csak az '*a*' és '*b*' első állapotától függ, de ha az *a=a+1* előbb végrehajtódik, akkor az utasításban már az '*a*' második állapotával számolunk, és hibás eredményt kapunk. Ezt a jelenséget a továbbiakban **korai állapotváltás hibának** nevezzük. Kétféle megoldás lehetséges a korai állapotváltás hibára: vagy figyelünk arra, hogy csak akkor lépünk át egy változó következő előfordulásába, amikor már minden más állapotot kiszámítottunk, amely tőle függött (ez további élek behúzását jelentené), vagy pedig tároljuk a változó szükséges előfordulásainak értékét ameddig szükséges. Mindkét módszernek megvan az előnye, és hátránya is. *Első esetben* lassítanánk a program végrehajtását, hiszen addig nem számoljuk ki egy változó új értékét, ameddig van olyan más állapot, amely függ tőle. *Második esetben* plusz tárhelyre van szükség, hiszen egy változó több előfordulását is tároljuk egyszerre. A következő két részben ezt a két megoldást tárgyaljuk.

1.2. Élekkel kiterjesztett függőségi programgráf

Azt a függőségi programgráfot nevezzük **élekkel kiterjesztett függőségi programgráfnak**, amelyben új éleket húzunk be annak érdekében, hogy a korai állapotváltás hibát kiküszöböljük. Ha egy altérrel címkézett valamely csúcs meghatároz akármilyen más, egy vagy több csúcsot, és létezik ugyanilyen altérrel és egyel nagyobb előfordulás számlálóval címkézett csúcs a gráfban, akkor a meghatározott csúcsoktól függ ez a címkéjű csúcs. Ezekkel az új élekkel olyan őrfeltételeket állítottunk be, hogy nem engednek egy állapotot tovább billenni mindaddig, amíg a tőle függő állapotok kiszámítása nem zajlott le. Mivel az eredeti gráfban minden egyes él legfeljebb egy másik élt állít elő, ezért éleink száma legfeljebb az eredeti gráfunk élei számának a kétszerese. Felmerül a kérdés, hogy ez a művelet rekurzív-e, azaz kell-e az új élek miatt további éleket behúzni. A válasz nem, mert az új élek miatt nem léphet már fel a

korai állapotváltás hiba, hiszen az új függőségek nem meghatározzák az állapotot, amelybe mutatnak, csak késleltetik annak kiszámítását.

A korábbiakban tárgyalt példához a *4. ábrán* látható ez az élekkel kiterjesztett függőségi programgráf.



4. ábra: Élekkel kiegészített függőségi programgráf

Az (*A2,2*)-be két él is mutat, és mindkettő olyan altérből, amelynek van rákövetkezője. Beállítunk egy függőséget (A_2,*2*)-ből (A_1,*2*)-be, viszont (A_2,*3*)-ba már nem kell plusz él, mert már eredetileg is volt. A gráfban a további függőségeket megvizsgálva szintén azt kapjuk, hogy ahova kellene az őrfeltétel miatt él, már eredetileg is van. Ebben az esetben a kiterjesztett függőségi gráfunkhoz egyetlen új él behúzása kellett. Az új gráffal a LEV ütemezés a következőképpen módosul 2 processzorra:

P1: (A_2,2), (A_1,2), (A_1,3) azaz: b=a+b; a=a+1; a=a+2;

P2: (A_2,2) azaz: b=b+c;

A P2 processzor az első lépésben nem dolgozik, a 2. lépésben hajtja végre *(A_2,2)-t.* A befejezési időnk eggyel nőtt az előző ütemezéshez képest, de már helyes megoldást kapunk.

Tétel

Ha egy absztrakt programban, a hozzá felépített élekkel kiterjesztett függőségi programgráf által megengedett párhuzamosan kiszámítható állapotok kiszámítását párhuzamosítjuk, akkor az absztrakt program kezdő és végállapota nem változik. Ez azt jelenti, hogy a párhuzamosított és az eredeti program programfüggvénye megegyezik.

Bizonyítás

Az eredeti program legyen *S,* a párhuzamosított program legyen *PS.* A kezdőállapotot nem érinti az utasítások végrehajtási sorrendje, ezért *PS* és *S* kezdőállapot halmaza megegyezik.

Megmutatjuk, hogy a végállapotok halmaza is egyező kell legyen. A bizonyítás indirekt. Tegyük fel, hogy van olyan eset, hogy *S*-ben „*a*” kezdőállapotból „*B*” végállapot halmazba jutunk, viszont a *PS*-ben „*a*” kezdőállapotból „*C*” végállapot halmazba jutunk, ahol B $\neq$ *C.* Ha a két halmaz nem egyenlő, akkor az alábbi két eset egyikének igaznak kell lennie.

1. eset: $\exists\, b \in B$, hogy $b \notin C$

Ekkor történik olyan végrehajtás, hogy *S*-ben „*a*” kezdőállapotból „*b*” végállapotba jutunk, viszont *PS*-ben „*a*” kezdőállapotból nem tudunk „*b*” végállapotba jutni. Az élekkel kiterjesztett függőségi gráfunkban egy utasítás sosem függ egy későbbi utasítástól, ezért nem fordulhat elő, hogy *PS*-ben egy későbbi utasítás mindenképp előbb kell, hogy lefusson, mint egy azelőtti. Ezért *PS*-ben lehet úgy ütemezni az utasítások lefutását, hogy pont *S*-t kapjuk. Ekkor viszont *PS*-ben „*a*” kezdőállapotból el kellett volna jutni „*b*” végállapotba. Ez ellentmondás, nem igaz, hogy $\exists\, b \in B$, hogy $b \notin C$.

2. eset: $\exists\, c \in C$, hogy $c \notin B$

Ekkor történik olyan végrehajtás, hogy *PS*-ben „*a*” kezdőállapotból eljutunk „*c*” végállapotba, viszont *S*-ben „*a*” kezdőállapotból nem tudunk „*c*” végállapotba jutni. *S*-ben és *PS*-ben pontosan ugyanazok az utasítások vannak,

abban azonban különbözhetnek, hogy *PS*-ben több utasítás is futhat egyszerre, mint *S*-ben, vagy más lehet az utasítások sorrendje:

a. *PS*-ben ha több utasítás fut egyszerre, akkor az utasítások nem ugyanazt az alteret módosítják, ezt a triviális függőségek okozzák. Valamint az egyes állapotokban megváltoztatott alterektől nem függ a többi párhuzamosan kiszámítható állapot, ezt a függőségi gráf és az élekkel való kiterjesztés okozza. Ezért ha ezeket az utasításokat párhuzamosan hajtjuk végre, akkor ugyanazt az eredményt adják, mintha egymás után sorosan futtattuk volna le őket.

b. Ha *PS*-ben két utasítás függ egymástól, akkor azok olyan sorrendben futnak le egymáshoz képest, mint az *S-ben*, mivel későbbi utasítás nem függhet korábbi utasítástól.

Az *a* és *b* pont miatt, mivel *PS*-ben és *S*-ben megegyeznek a végrehajtandó utasítások, ugyanazt a végeredményt kell kapnunk *S*-ben és *PS*-ben is. Ez azt jelenti, hogy *S*-ben is el kellett volna jutni „*a*"-ból kezdőpontból „*c*" végállapotba. Ez ellentmondás, nem igaz, hogy $\exists\ c \in C$, hogy $c \notin B$.

Az 1. és 2. eset miatt beláttuk, hogy B és C halmaz nem lehet különböző, tehát S és PS végállapot halmaza megegyezik. Mivel S és PS kezdőállapot halmaza és végállapot halmaza megegyezik, programfüggvényük azonos.

Ezzel az absztrakt leírással egy irányelvet mutattunk be, hogy mi a párhuzamosítás elképzelésének alapja. Ezt ki lehet terjeszteni egyaránt procedurális és objektum-elvű nyelvekre. A dolgozathoz készített program a korai állapotváltás hiba kiküszöbölésére az élekkel való kiterjesztést alkalmazza. Mielőtt rátérnénk az objektum elvű modellre, egy rövid kitérőt teszünk a korai állapotváltás hiba egy másik megoldására, az **állapotokkal való kiterjesztésre**.

1.3. Állapotokkal kiterjesztett függőségi programgráf

A korai állapotváltás hiba másik kiküszöbölési módja az, hogy a változókat kiterjesztjük tömbbé, és nem csak az aktuális állapotukat tároljuk,

hanem mindent, amit felvettek. Amikor egy értékadás történik, a változó tömbjének éppen aktuális előfordulására hivatkozunk. A fenti példát két processzorra ütemezve a következőt kapjuk:

P1: b[1]=a[0]+b[0]; b[2=b[1]+c[0];

P2: a[1]=a[0]+1; a[2]=a[1]+1;

Ennek a módszernek a hátránya, hogy hatalmassá nőhet a programunk. Ezért finomíthatjuk azzal, hogy azt is tároljuk, hogy egy változó adott előfordulására hivatkoznak-e még, és ha nem, akkor azt törölhetjük. Ennek implementálásához egy változó *i*-edik eleme tárolná a változó aktuális értéket, valamint azt, hogy hányszor hivatkoznak rá összesen. Minden egyes hivatkozásnál ez a számláló csökken eggyel, és ha eléri a nullát, akkor abban az esetben törölhető.

Az eddig tárgyalt példa ezzel a módszerrel:

vizsgál(x)= x.számláló--; Ha x.számláló==0 akkor töröl x;

P1:

- *b[1]=a[0]+b[0]; vizsgál(a[0]); vizsgál(b[0]);*
- *b[2]=b[1]+c[0]; vizsgál(b[1]); vizsgál(c[0]);*

P2:

- *a[0]=a[1]+1; vizsgál(a[0]);*
- *a[2]=a[1]+1; vizsgál(a[1]);*

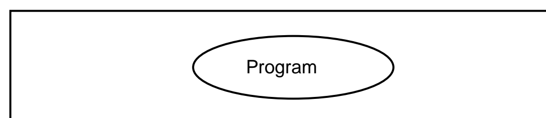
Ezzel a finomítással megnőtt a műveletigényünk, ezért annak függvényében kell dönteni arról, hogy ezt alkalmazzuk-e, hogy mekkora az utasítások futási ideje, valamint mennyit nyerünk azzal, hogy a memóriából felszabadítjuk a már feleslegessé vált változókat.

2. Objektum-elvű közös modell felépítése

Ez a rész az objektum-elvű közös modell felépítésével foglalkozik. Az objektum-elvűség fogalmát ismertnek tekintem, erről bővebben *Grady Booch: Object Orienteted Analysys And Design*[9] című munkájában olvashatunk. A különböző programnyelvek különbségeinek elfedésére egy közös modellt építek. A modell két részből áll, egy **statikus** és egy **dinamikus** részből. A statikus rész felépítése arra szolgál, hogy azonosítani tudjuk az osztályainkat, függvényeinket és változóinkat, míg a dinamikus rész azért felelős, hogy leírja a függvényekben lévő műveleteket, és egymással való kapcsolatukat.

2.1. Statikus modell

A modell statikus része egy kis módosítással megegyezik a Halász Attila Péter[10] által épített gráffal. Az általános gráf adatszerkezetet és alapvető fogalmait ismertnek tekintem. Ezekről bővebben *Láng Csabáné, Gonda János: Bevezetés a matematikába II.*[11] című könyvében olvashatunk. A program gyökere a „Program” csúcs, ez reprezentálja az egész programunkat, ebből indulunk ki, ha képet akarunk kapni a programunkról. *(5. ábra)*



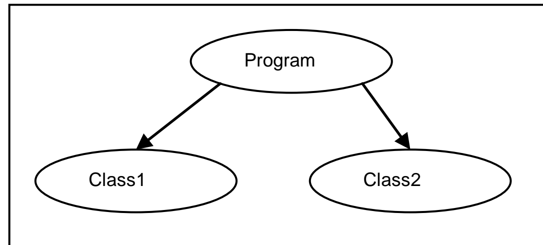
5. ábra: Program csúcs

Az osztályt a „Class” csúcs reprezentálja, ezekbe a csúcsokba a „Program” csúcsból vezetnek élek *(6. ábra)*.

[9] Grady Booch: Object Orienteted Analysys And Design, 1998.
http://bib.tiera.ru/dvd37/Booch%20G.%20-%20Object-oriented%20analysis%20and%20design%20with%20applications%281993%29%28Second%20Edition%29%28608%29.pdf

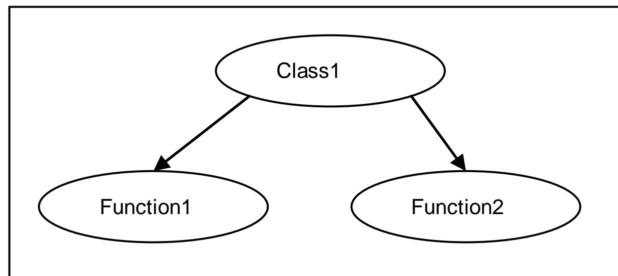
[10] Halász Attila Péter: Objektumorientált, szekvenciális programok átalakítása párhuzamos környezet számára hatékony implementációval, 2006.
http://compalg.inf.elte.hu/~tony/Oktatas/Diplomamunka-Szakdoli-Nagyprogram/Halasz%20Attila%20-%20Diplomamunka.zip

[11] Láng Csabáné, Gonda János: Bevezetés a matematikába II., Budapest, ELTE Eötvös Kiadó, 1995.



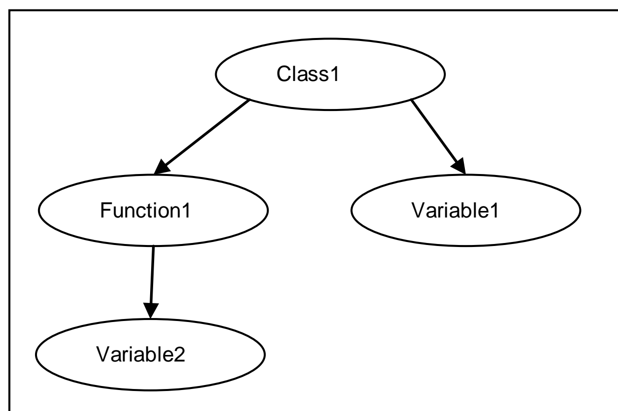
6. ábra: Class csúcsok

A függvényt a „Function" csúcs reprezentálja, ezekbe a csúcsokba abból a „Class" csúcsból vezet él, amelyik azt az osztályt reprezentálja, amely tartalmazza a függvényt. *(7. ábra)*



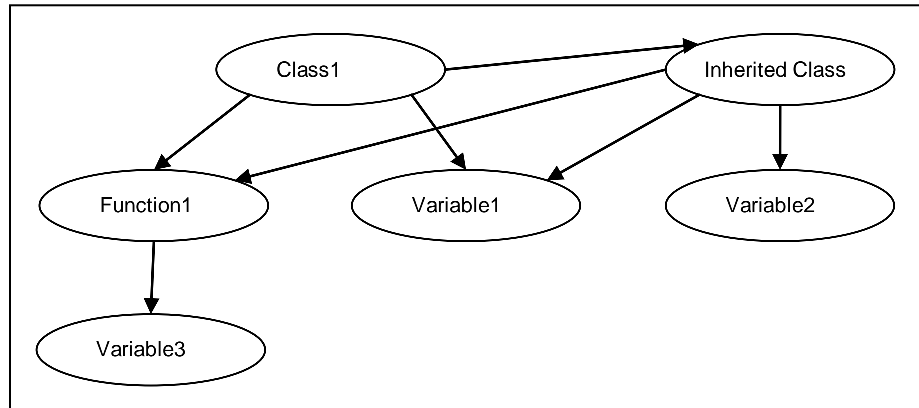
7. ábra: Function csúcsok, és a beléjük vezető élek

A változót a „Variable" csúcs reprezentálja. Osztály és függvény tartalmazhat változókat, ezért a „Class" és „Function" csúcsokból vezet beléjük él. Osztály esetében a hozzá tartozó változó tagváltozó, függvény esetében a hozzá tartozó változó lokális, vagy paraméterváltozó. A változó fajtáját a gráfban itt külön nem jelezzük. *(8. ábra)*



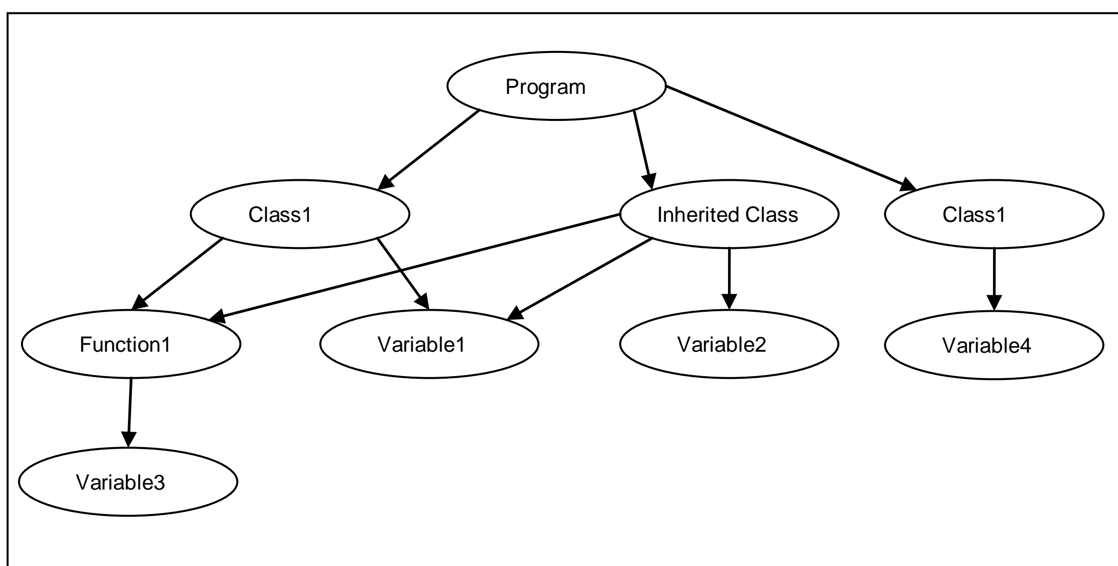
8. ábra: Variable csúcsok, és a beléjük vezető élek

Öröklődés esetén előfordul, hogy több osztály is ugyanazt a változót és függvényt tartalmazza. Ilyen esetben minden ilyen osztályból vezet az adott változóba él, továbbá az örökölt osztályba a szülő osztályból vezet él. *(9. ábra)*



9. ábra: Öröklődéssel ábrázol osztályok

A program statikus leírása véget ért, a programban szereplő összes változót azonosítani tudjuk. A párhuzamosító algoritmusunk számára ez az információ lesz értékes, viszont ahhoz, hogy pontosan definiáljuk a programunkat még sok egyéb információra is szükségünk van, ilyen például a változók típusa, vagy a függvények, változók hatóköre. Ezeket is tároljuk természetesen, erről részletesebben a fejlesztői dokumentációban olvashatunk. A statikus modell összefoglalását a *10. ábrán* láthatjuk.



10. ábra: Statikus modell összefoglaló

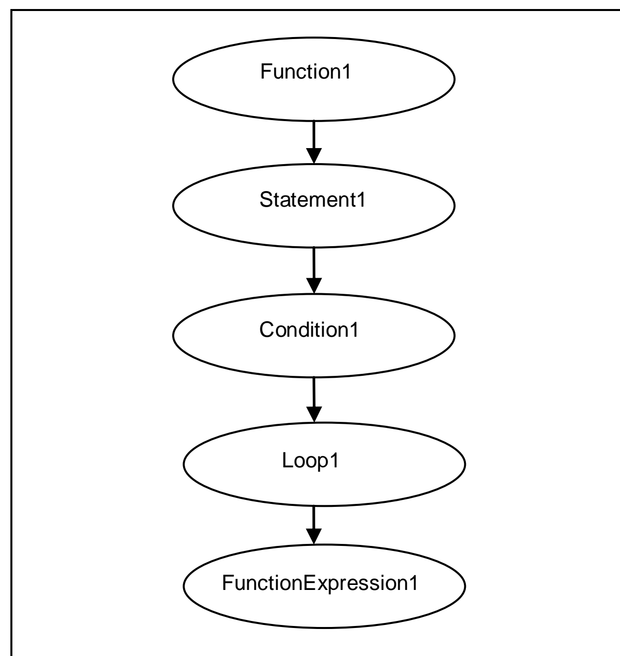
2.2. Dinamikus modell

Most, hogy azonosítani tudjuk a változóinkat, meghatározzuk, hogy az egyes függvények milyen műveletekből állnak, továbbá meghatározzuk, hogy ezeknek a műveleteknek milyenek a kapcsolatatai egymással és a változókkal.

2.2.1. Műveletek fajtái és egymással való kapcsolatai

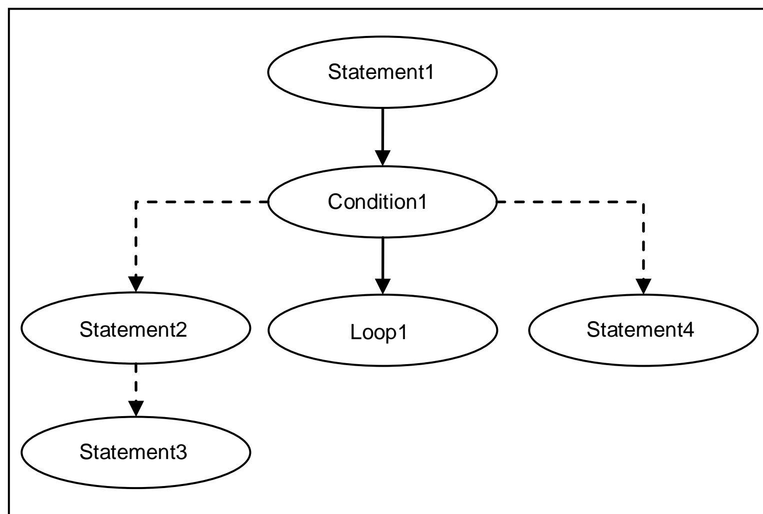
Négy alapvető műveletet különböztetünk meg: **értékadás, elágazás, ciklus, függvényhívás**. Ezek egymás utáni szekvenciája ír le egy függvényt.

Az **értékadás** számára a „Statement”, az elágazás számára a „Condition”, a ciklus számára a „Loop”, valamint a függvényhívás számára a „FunctionExpression” csúcsot vezettem be. A függvény egymás utáni műveletei között folyamatos élt húzunk be. *(11. ábra)*



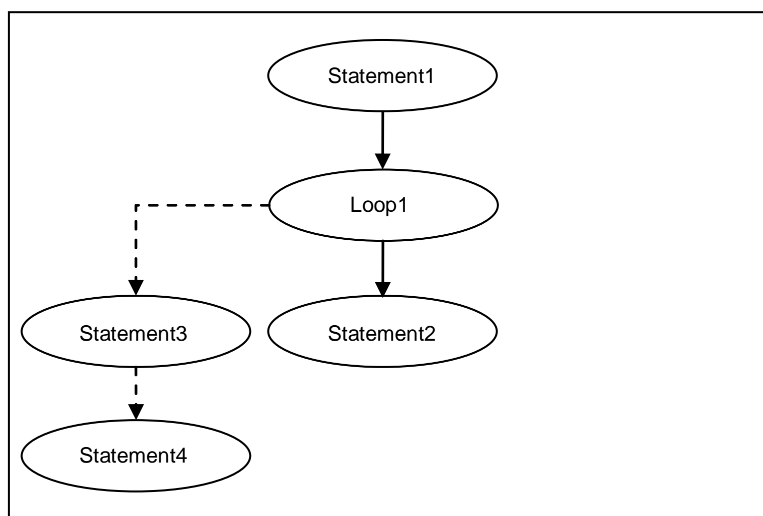
11. ábra: Függvényt ábrázoló gráf példa

Elágazás esetén attól függően, hogy az elágazás értéke igaz, vagy hamis, különböző műveleteket hajtunk végre. A gráfban ezt úgy jelöljük, hogy a gráfban az elágazást jelölő „Condition” csúcsból szaggatott élek vezetnek ezek felé a műveletek felé. A hivatkozott műveletek között is szaggatott él vezet. *(12. ábra)*



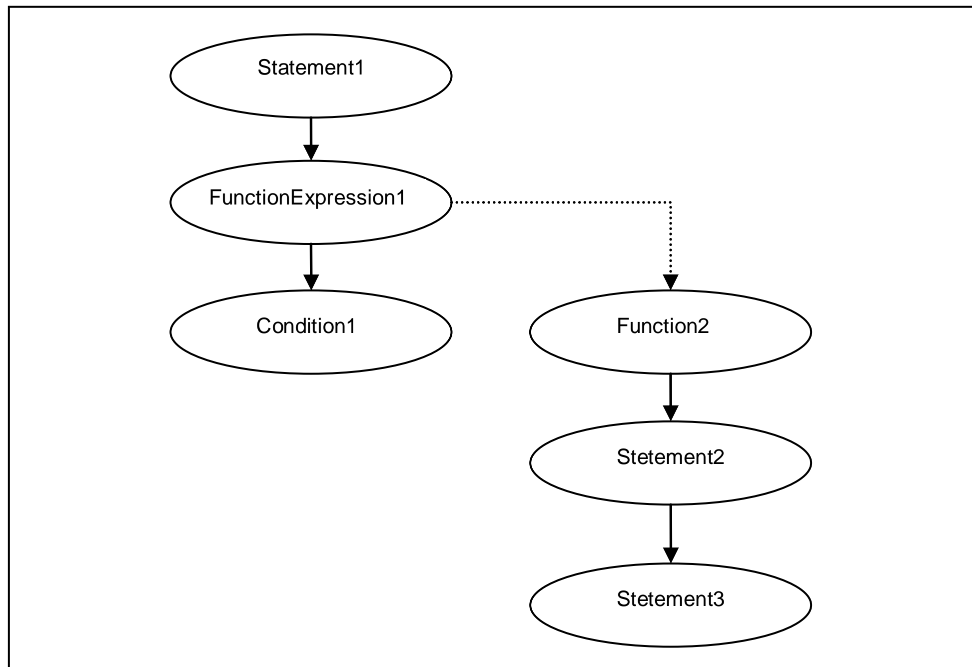
12. ábra: Elágazás ábrázolása az igaz ágon két utasítással, a hamis ágon egy utasítással

Ciklus esetén a ciklusmag tartalmaz műveleteket, ezt a gráfban úgy jelöljük, hogy a gráfban a „Loop" csúcstól szaggatott vonalakat húzunk a műveletei felé. A hivatkozott műveletek között is szaggatott él vezet. (*13. ábra*)



13. ábra: Ciklus ábrázolása két művelettel a ciklusmagban

Függvényhívás esetén a hivatkozott függvénybe pontozott élt húzunk. *(14. ábra)*

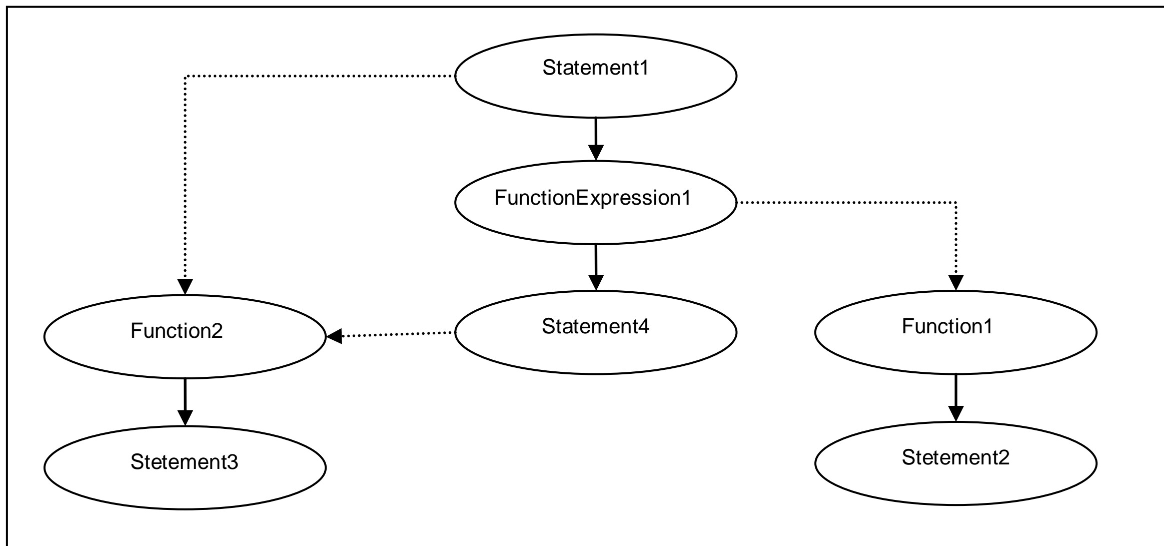


14. ábra: Függvényhívás a Function2 függvénybe

Függvényhívás nem csak külön műveletként szerepelhet. Ha a függvénynek van visszatérési értéke, akkor tekinthető egy művelet részének is. A műveleteknél a következő helyeken szerepelhet függvényhívás:

- **Értékadás** esetén a függvényhívás szerepelhet az értékadás bal oldalán.
- **Elágazás** esetén a függvényhívás szerepelhet a feltételben.
- **Ciklus** esetén a függvényhívás szerepelhet a ciklus feltételben.
- **Függvényhívás** esetén függvényhívás szerepelhet a paraméterként átadott értékek között.

Az olyan műveleteknél, ahol a művelet része egy függvényhívás, egy pontozott élt húzunk a művelettől a hivatkozott függvénybe. *(15. ábra)*



15. ábra: Statement1 és a Statement4 értékadásban szerepel a Function2 függvény a jobb oldalukon

A kapcsolatok leírása során három fajta élt használtunk: **simát, szaggatottat és pontozottat**. Az élek fajtáinak a párhuzamosítás folyamán lesz jelentősége. Azoknál a műveleteknél, ahol **sima él** van köztük, ott fogja a párhuzamosító algoritmusom megvizsgálni, hogy ezek a műveletek futtathatóak-e párhuzamosan. A **szaggatott élekkel** összekötött műveletek azt jelzik, hogy ezek a műveletek mindenképp együtt futnak le, nem végezhető el párhuzamosítás közöttük. A párhuzamosító algoritmusomat függvényenként külön-külön egymástól függetlenül futtatom le. A **pontozott él** mindig egy függvénybe mutat, ez azt jelzi, hogy ott a párhuzamosító algoritmusom egy újabb futtatása fog párhuzamosítani. A *15. ábrán* például három függvényt írunk le, háromszor fogjuk lefuttatni a párhuzamosító algoritmusomat.

A függvényünk műveleteinek és egymással való kapcsolatuk leírásával végeztünk. A következő fejezetben azt tárgyaljuk, hogy az egyes műveleteknek milyen kapcsolata van a program változóival.

2.2.2. Műveletek kapcsolata a változókkal

A párhuzamosító módszer alapja, hogy minden egyes műveletnél ismerjük azt, hogy mely változókat változtat meg, illetve mely változókat használ fel. Azokat a változókat, amelyeket megváltoztat, nevezzük bal oldali változóknak, azokat a változókat, amelyek értékét felhasználja, nevezzük jobb oldali változóknak. Mindkét változófajtából van közvetlen, és közvetett. Közvetlen változók közé soroljuk azokat a változókat, amelyeket a művelet függvényhívás nélkül használ jobb, vagy bal oldali változónak. Közvetett változók közé soroljuk azokat a változókat, amelyeket a művelet függvényhíváson keresztül használ jobb, vagy bal oldali változónak. A közvetlen változók műveletenként a következők:

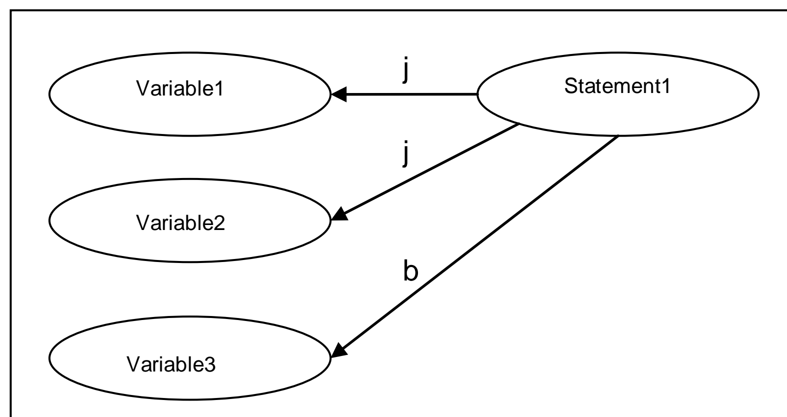
- **Értékadás** esetében az értékadás bal oldala a bal oldali változó, az értékadás jobb oldalán szereplő változók a jobb oldali változók.
- **Elágazás** esetén nincs közvetlen bal oldali változó, a jobb oldali változók az elágazás feltételében szereplő változók.
- **Ciklus** esetén nincs közvetlen bal oldali változó, a jobb oldali változók a ciklus feltételben szereplő változók.
- **Függvényhívás** esetén nincs közvetlen bal oldali változó, és nincs közvetlen jobb oldali változó sem.

Objektum-elvű nyelvben egy változóra, vagy függvényre hivatkozhatunk simán, vagy más változókon keresztül. Azokat a változókat, amelyeken keresztül hivatkozunk a változóra, vagy függvényre, **elérési láncnak** neveztem el. Erre példát a *16. ábrán* láthatunk. Amennyiben egy műveletben egy változóra, vagy függvényre hivatkozási láncon keresztül hivatkozunk, akkor a művelet közvetlen jobb oldali változói közé hozzávesszük az elérési lánc változóit.

```
class A
{
		int b;
		int MyFunc()
		{
				b=3; //Sima hivatkozás az A osztály „b” tagváltozójára.
				A a=new A();
				a.b=3; // A MyFunc függvény „a” lokális változóján keresztül hivatkoztunk az A osztály „b”
					tagváltozójára.
		}
}
```

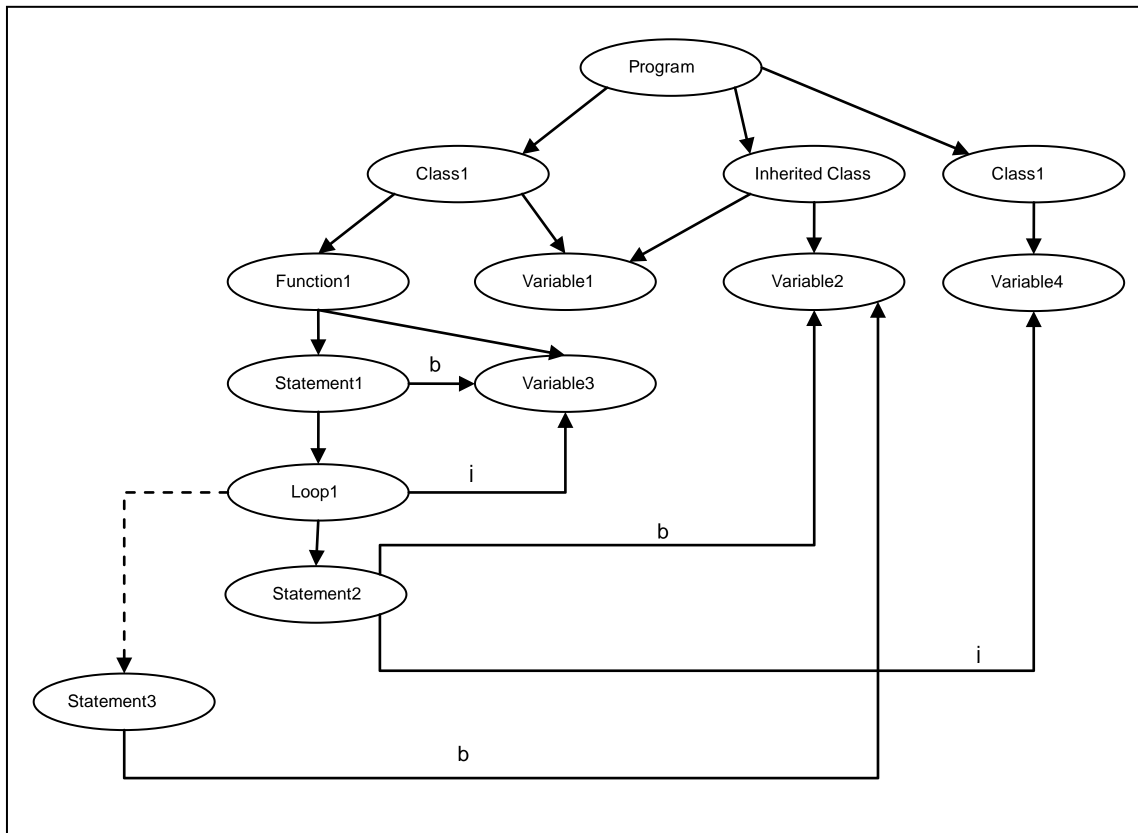
16. ábra: Változóra való sima, és elérési láncon keresztül való hivatkozás példa

Egy műveletből „*j*” címkéjű éleket húzunk a művelet jobb oldali közvetlen változói felé, és „*b*” címkéjű éleket húzunk a közvetlen bal oldali változói felé. A *17. ábrán* láthatunk példát erre.



17. ábra: Egy értékadáshoz tartozó jobb és bal oldali változók kapcsolata

A közvetett változók meghatározásához a függvényeket be kell járni, amelyeket a művelet meghív, ezt a párhuzamosító algoritmus részeként fogjuk részletesen tárgyalni. Az objektum-elvű közös modell összefoglalója a *18. ábrán* látható.



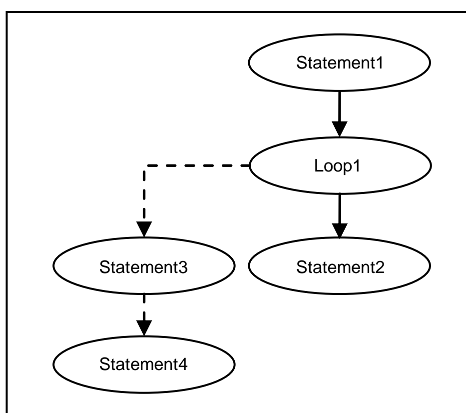
18. ábra: Objektum-elvű közös modell összefoglalás

3. Párhuzamosítás

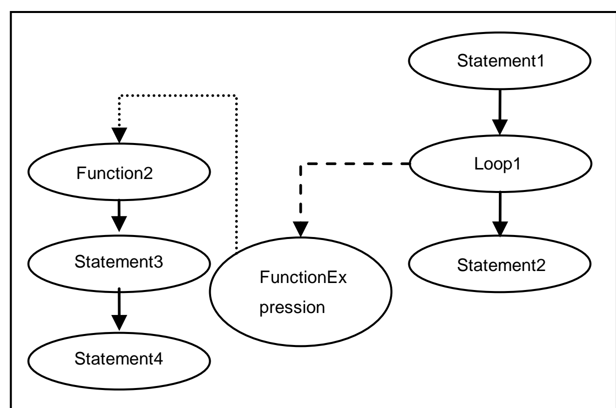
Az előző fejezetben felépített modellhez most elkészítem a párhuzamosító eljárást, amely három fő részből áll. Az első részben a **bemeneti programunkat refaktoráljuk**, a második részben **kiszámoljuk az egyes műveletekhez a közvetett jobb és bal oldali változóit**, majd a harmadik lépésben **felépítünk a programhoz a függőségi programgráfot**, amely meghatározza, hogy mely utasításait lehet párhuzamosan végrehajtani.

3.1. Program refaktorálása

Az előző fejezetben tárgyalt modellben három fajta élt használtunk a műveletek között. A sima él jelezte, hogy végezhetünk párhuzamosítást a műveletek között, a szaggatott jelezte, hogy nem végezhetünk párhuzamosítást, illetve a pontozott él jelezte, hogy a párhuzamosító algoritmusom egy másik futtatása fogja a mutatott függvényt párhuzamosítani. A refaktorálás célja, hogy a programunkat ábrázoló gráfban minél kevesebb szaggatott él legyen, mert így több helyen végezhetünk majd párhuzamosítást. Elágazás esetében az igaz és hamis ágában lévő műveletek között, ciklus esetén a ciklus magjában lévő műveletek között szerepeltek ilyen élek. A programunkat úgy alakítjuk át, hogy ezeket a műveleteket külön függvénybe exportáljuk, így már csak 1 darab szaggatott él lesz, mint ahogy a *19. és 20. ábrán* látható.



19. ábra: Eredeti program



20. ábra: Refaktorált program

Ezt az átalakítást minden függvényre a következő algoritmussal végezzük el:

```
foreach (var expression in function.Expressions)
{
    if (expression is IExpressionContainer)
    {
        if (((IExpressionContainer)expression).Expressions.Count > 1)
        {
            var newFunc = Refactor.ExtractToMethod((IExpressionContainer)expression);
            RefactorFunction(newFunc);
        }
        if (expression is Condition)
        {
            ((Condition)expression).IsTrueBranch =
!((Condition)expression).IsTrueBranch;
            if (((IExpressionContainer)expression).Expressions.Count > 1)
            {
                var newFunc2 =
Refactor.ExtractToMethod((IExpressionContainer)expression);
                RefactorFunction(newFunc2);
            }
        }
    }
}
```

A fenti algoritmusban végigmegyünk a függvény összes műveletén, és amennyiben a művelet függvény, vagy elágazás, akkor a hozzá tartozó műveleteket exportáljuk egy új függvénybe. Kivételt képez ez alól, ha az exportálandó műveletek száma egy, ekkor nem csinálunk semmit, mert ebben az esetben a refaktorálás után nem lenne kevesebb szaggatott élünk. A programban az elágazást és a ciklust **IExpressionContanier** közös interface-el láttam el, az algoritmusban erre a típusra hivatkozunk. Az algoritmusban hivatkozunk a **Refactor.ExtractToMethod** függvényre, ez a függvény végzi el a paraméterben átadott műveletlista exportálását külön függvénybe. Az algoritmusa a következő:

```
public static Function ExtractToMethod(IExpressionContainer expressionContainer)
{
    extractMethodNumber++;
    var programUnit=(ProgramUnit)expressionContainer;
    Function newFunction = new Function(GetParentClass(programUnit),
Primitive.VoidPrimitive, "ExtractedMethod" + extractMethodNumber);


    foreach (var expression in expressionContainer.Expressions)
    {
        expression.ProgramUnitParent = newFunction;
        newFunction.Expressions.Add(expression);
    }


    List<Variable> variables = new List<Variable>();
    foreach (var expression in
Expression.GetAllExpressions(expressionContainer.Expressions))
    {
        variables.AddRange(expression.GetAllUsedVariable());
    }
    variables = variables.Distinct().ToList();

    foreach (var variable in variables)
```

```
    {
        var newVariable=new Variable(variable.MyType, newFunction, variable.DisplayName, 
VariableScope.Parameter);
    }

    expressionContainer.Expressions.Clear();
    FunctionExpression fe = new FunctionExpression(expressionContainer, newFunction, 
variables.Select(v => new Tuple<IValue, bool>(new  VariableInstance(v), 
true)).ToList());
    return newFunction;
}
```

Az algoritmusban létrehozunk egy új függvényt, majd a paraméterben átadott műveleteket felvesszük ebbe a függvénybe. A műveletekben használt összes változót felvesszük az új függvény paraméterváltozói közé. A paraméterben átadott műveleteket kivesszük az eredeti függvényből, és az új függvényünk meghívását tesszük be helyette.

Most, hogy a lehető legkevesebb szaggatott él van a gráfunkban, rátérünk a közvetett változók meghatározására.

3.2. Közvetett jobb- és baloldali változók meghatározása

Minden művelethez tartoznak közvetett és közvetlen jobb- és baloldali változók. A közvetlen változók azok, amelyeket a művelet függvényhívás nélkül használ, a közvetettek azok, amelyeket függvényhíváson keresztül használ. A közvetlen változókat a 2.2.2. részben tárgyaltuk. A közvetett változók meghatározásához először minden egyes függvénynél kiszámítjuk, hogy mely változókat használnak a műveletei, majd minden függvényhívásnál kiszámítjuk, hogy a meghívott függvénnyel milyen változókat használ.

3.2.1. Függvények bal és jobb oldali változói

A függvények bal és jobb oldali változóinak meghatározása több lépésen keresztül történik. *Első lépésben* minden függvény bal és jobb oldali változóihoz hozzáadjuk a műveleteiben használt közvetlen változókat, amelyet az alábbi algoritmus végez el:

```
foreach (var function in functions)
{
    var allExpressions = Expression.GetAllExpressions(function.Expressions);
    foreach (var expression in allExpressions)
    {
        function.InputVariables.AddRange(expression.RightVariables);
        function.OutPutVariables.AddRange(expression.LeftVariables);
    }
}
```

Ha a függvény műveletei között vannak függvényhívások, akkor a meghívott függvény bal és jobb oldali változóit is hozzá kell venni a hívó függvény bal és jobb oldali változóihoz, ez a *második lépés*. Arra figyelni kell, hogy ne a hívott függvény paraméter változóit adjuk hozzá a hívó függvény változóihoz, hanem azokat a változókat, amelyeket paraméterül átadtunk. Ahhoz, hogy biztosan minden függvény ismerje az összes általa meghívott függvény összes bal és jobb oldali változóit, n-1-szer kell lekérdezni a meghívott függvények változóit, mert így a legrosszabb esetet is lefedjük, amikor a függvényhívások láncba vannak. A függvények bal és jobb oldali változóinak meghatározásánál a *második lépést* az alábbi algoritmus végzi el:

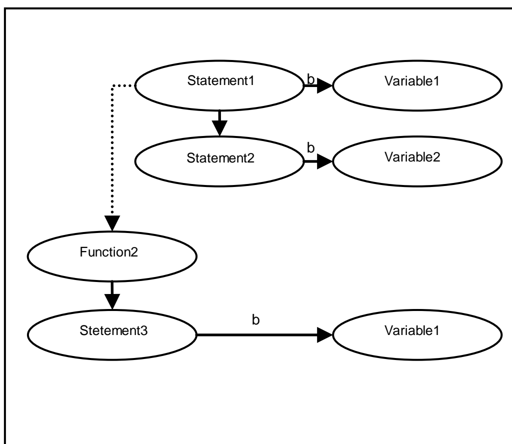
```
for (int i = 0; i < functions.Count - 1; i++)
{
    foreach (var function in functions)
    {
        var allFunctionExpressions =
Expression.GetAllExpressions(function.Expressions).Where(e => e is
FunctionExpression).Select(e => (FunctionExpression)e).ToList();
        foreach (var expression in Expression.GetAllExpressions(function.Expressions))
        {
            allFunctionExpressions.AddRange(expression.RightValues.Where(v => v is
FunctionExpression).Select(v => (FunctionExpression)v).ToList());
        }

        foreach (var functionExpression in allFunctionExpressions)
        {

function.InputVariables.AddRange(functionExpression.Function.InputVariables.Where(v =>
v.VariableScope == VariableScope.Member));

function.OutPutVariables.AddRange(functionExpression.Function.OutPutVariables.Where(v =>
v.VariableScope == VariableScope.Member));

            var funcExpressionParameters = functionExpression.Function.Variables.Where(v
=> v.VariableScope == VariableScope.Parameter).ToList();
            foreach (var parameter in funcExpressionParameters)
            {
                if (functionExpression.Function.InputVariables.Contains(parameter))
                {
                    var index =
functionExpression.Function.Variables.IndexOf(parameter);
                    if (functionExpression.RightValues[index] is VariableInstance)
                    {

function.InputVariables.Add(((VariableInstance)functionExpression.RightValues[index]).Va
riable);
                    }
                }
                if (functionExpression.Function.OutPutVariables.Contains(parameter))
                {
                    var index =
functionExpression.Function.Variables.IndexOf(parameter);
                    if (functionExpression.RightValues[index] is VariableInstance)
                    {

function.OutPutVariables.Add(((VariableInstance)functionExpression.RightValues[index]).V
ariable);
                    }
                }
            }
        }
        function.InputVariables=function.InputVariables.Distinct().ToList();
        function.OutPutVariables = function.OutPutVariables.Distinct().ToList();
    }
```

Végeztünk a függvények jobb és bal oldali változók meghatározásával. Ha meghívunk egy függvényt, akkor a függvényhívás közvetett bal és jobb oldali változói közé felvesszük a meghívott függvény bal és jobb oldali tagváltozóit, valamint a paraméterként átadott változókat, ha az adott paramétert a hívott függvény bal vagy jobb változónak használja. A függvényhívások bal és jobb oldali változóinak meghatározását az alábbi függvény végzi el:

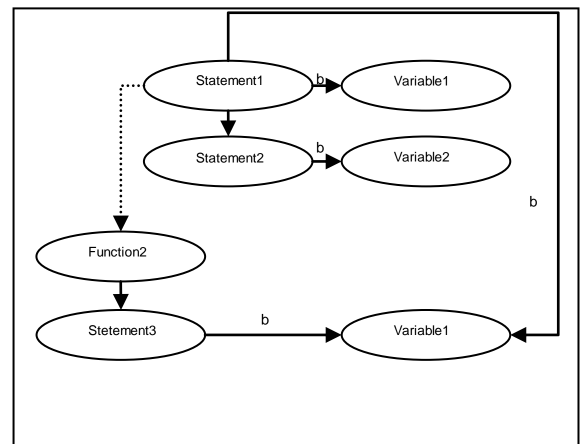
```
foreach (var function in functions)
{
    var allFunctionExpressions =
Expression.GetAllExpressions(function.Expressions).Where(e => e is
FunctionExpression).Select(e => (FunctionExpression)e).ToList();

    foreach (var expression in Expression.GetAllExpressions(function.Expressions))
    {
        allFunctionExpressions.AddRange(expression.RightValues.Where(v => v is
FunctionExpression).Select(v => (FunctionExpression)v).ToList());
    }

    foreach (var functionExpression in allFunctionExpressions)
    {
        functionExpression.Function.OutPutVariables.Where(v => v.VariableScope ==
VariableScope.Member).ToList().ForEach(v =>
functionExpression.LeftVariablesBase.Add(v));
        functionExpression.Function.InputVariables.Where(v => v.VariableScope ==
VariableScope.Member).ToList().ForEach(v =>
functionExpression.RightVariablesBase.Add(v));

        var parameters = functionExpression.RightValues.Where(p => p is
VariableInstance).ToList();

        for (int j = 0; j < functionExpression.RightValues.Count; j++)
        {
            if (functionExpression.RightValues[j] is VariableInstance)
            {
                var funcParameter = functionExpression.Function.Variables.Where(v =>
v.VariableScope == VariableScope.Parameter).ToList();
                if
(functionExpression.Function.InputVariables.Select(v=>v.Id).Contains(funcParameter[j].Id
))
                {

functionExpression.RightVariablesBase.Add(((VariableInstance)functionExpression.RightVal
ues[j]).Variable);
                }
                if
(functionExpression.Function.OutPutVariables.Select(v=>v.Id).Contains(funcParameter[j].I
d))
                {

functionExpression.LeftVariablesBase.Add(((VariableInstance)functionExpression.RightValu
es[j]).Variable);
                }

functionExpression.RightVariablesBase.AddRange(((VariableInstance)functionExpression.Rig
htValues[j]).ReachList);
            }
        }

        functionExpression.RightVariables.AddRange(functionExpression.ReachList);

        functionExpression.LeftVariablesBase =
functionExpression.LeftVariablesBase.Distinct().ToList();
```

```
        functionExpression.RightVariablesBase =
functionExpression.RightVariablesBase.Distinct().ToList();
    }
}
```

Most már meg tudjuk határozni a függvényhívások bal és jobb oldali változóit, ezért minden művelethez meg tudjuk adni, hogy melyek a közvetett bal és jobb oldali változói. Az objektum elvű közös modell gráfjában minden műveletből most már nem csak a közvetlen, hanem a közvetett jobb és bal oldali változókba is mutat él.



19. ábra: Közvetett változók nélküli gráf



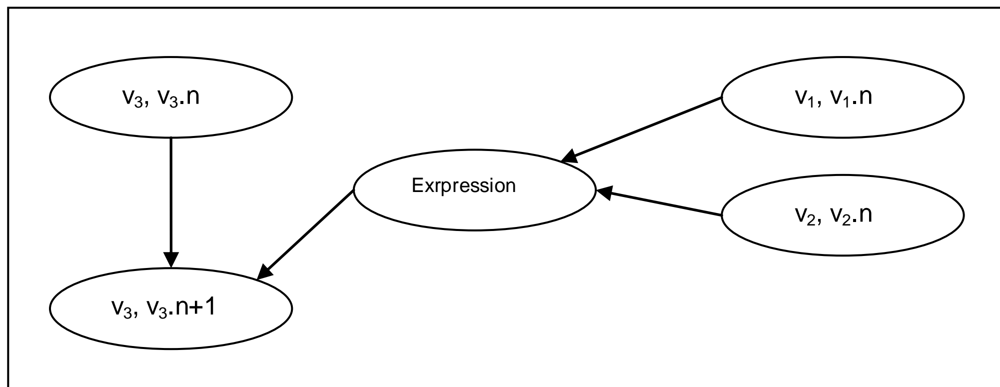
20. ábra: Közvetett változókkal ábrázolt gráf

3.3. Függőségi programgráf felépítése

Az első fejezetben leírtuk a párhuzamosító módszer alapelvét. A második fejezetben bemutattuk az objektum-elvű közös modellt. A harmadik fejezet eddigi részében előkészítettük a programunkat a párhuzamosításhoz. A párhuzamosítást minden egyes függvény azon műveletei között végezzük el, amelyek között sima él vezet. Annak eldöntésére, hogy mely műveletek futtathatóak párhuzamosan egy gráfot építünk fel, amelyet függőségi programgráfnak neveztünk el. A függőségi programgráf felépítésének a részletes leírása következik.

Minden egyes függvényhez külön felépítjük ezt a gráfot. Az első fejezetben az absztrakt programnál a gráf csomópontjait a program által használt alterekkel címkéztük meg. Ezek az alterek a második fejezetben tárgyalt modellben a változók. A most felépítendő gráfunkban a változókhoz tartozó csúcsokat nevezzük normál csúcsoknak. További csúcsok bevezetésére is szükség van, mert van olyan műveletünk, amely nem csak egy változót

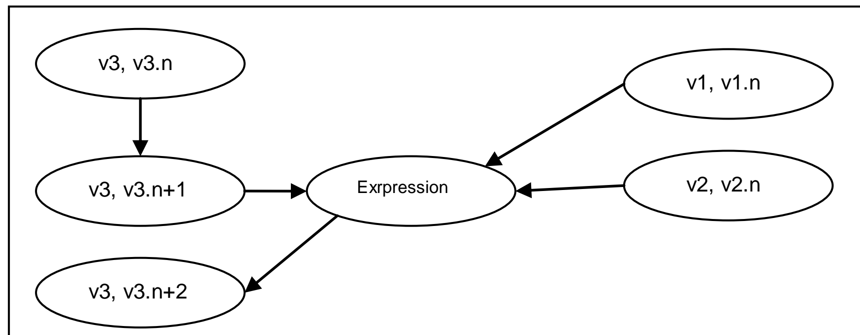
változtat meg, és ezt kezelni kell. Bevezetjük az „expression” csúcsot, amely azt reprezentálja, hogy valami műveletet végzünk benne. A műveletnek nincsen előfordulása a változóval ellentétben. Amikor egy új műveletet veszünk hozzá a függőségi gráfunkhoz, akkor felveszünk egy új „expression” csúcsot, és a művelet jobb oldali változóiból él vezet a művelet csúcsába. Továbbá felveszünk n darab új normál csúcsot a művelet bal oldali változóihoz, és élt húzunk a műveletből beléjük. *(21. ábra)*



21. ábra: Művelet kettő darab jobb oldali, és egy darab bal oldali változóval

Az „expression” csúcs bevezetésével megmarad a gráfnak az a tulajdonsága, hogy ha egy változó függ egy másik változótól, akkor abba él vezet. Mindössze annyiban módosult ez, hogy nem közvetlenül, hanem közvetetten vezet oda él. Egy probléma felmerül az új csúcs bevezetése miatt. Eddig minden változó minden előfordulása egy értéket képviselt. Most a művelet bal oldali változóihoz tartozó csúcsok két értéket viselhetnek. A *23. ábrán* a (v_3, $v_3.n$) csúcs az „expression” csúcs végrehajtása előtt és után más-más értéket vesz fel, azaz nem determinisztikus. Ez akkor jelenthet problémát, ha létezne egy másik csúcs a gráfban, amely a (v_3, $v_3.n$) csúcstól függne. Bevezetjük a függőségi gráfunk harmadik, egyben utolsó típusú csúcsát, a szinkronizációs csúcsot. Ez a csúcs szerkezetileg a normál csúccsal egyezik meg, azaz címkézzük egy változóval, és egy előfordulás számlálóval. Ebből a csúcsból sehova máshova nem vezet él, csak abba az „expression” csúcsba, amely megváltoztatja. Ez azt jelenti, hogy ennek a csúcsnak az értékét sehol sem használjuk fel, ezért nem baj, ha nem determinisztikus értéket jelöl. A normál csúcsoknál kikötjük, hogy ne fordulhasson elő olyan eset, hogy nem determinisztikus értéket jelöljenek. Amikor egy műveletet hozzáveszünk a

gráfunkhoz, akkor a baloldali változók mindegyikéhez felveszünk egy-egy szinkronizációs csúcsot, amelyekbe él vezet a bal oldali változókhoz tartozó utolsó előfordulásukat ábrázoló csúcsokból. Ezekből a szinkronizációs csúcsokból élt húzunk a műveletbe. Létrehozunk egy-egy új normál csúcsot a bal oldali változóknak, és a műveletből élt húzunk feléjük. *(22. ábra)*



22. ábra: Szinkornizációs csúccsal ábrázolt művelet

A *24. ábrán* a (v_3, $v_3.n+1$) szinkronizációs csúcs. Más műveletbe biztosan nem vezet él belőle, mert amelyet a mostani művelet előtt adtunk hozzá a függőségi programgráfunkhoz, az legfeljebb a (v_3, $v_3.n$)-es normál csúcstól függne, amelyet pedig a mostani művelet után adunk hozzá a gráfunkhoz, az legfeljebb a (v_3, $v_3.n+2$)-es csúcstól függne.

Meghatároztuk, hogy egy új művelet felvételekor milyen éleket, és milyen új csúcsokat kell bevezetni. Most megnézzük, hogy egy függvényből hogyan épül fel a függőségi gráfunk teljesen az elejétől.

Első lépésben felvesszük a változók nulladik előfordulását ábrázoló csúcsokat a gráfban. Ezek a változók a függvény összes változói, valamint az összes osztály összes tagváltozója.

```
foreach (var variable in function.Variables)
{
    graph.AddRootNode(variable);
}
foreach (var myClass in ((Program)function.ParentClass.ProgramUnitParent).Classes)
{
    myClass.Variables.ForEach(v=>graph.AddRootNode(v));
}
```

Az algoritmus további részében végighaladunk a függvény minden egyes műveletén, és hozzáadjuk a gráfhoz. Egy művelet lehet értékadás, elágazás, ciklus, függvényhívás, valamint egyéb utasítás, amely az előző

néggyel nem írható le. Ezek mind „expression” csúcsoknak számítanak, az algoritmusban a csúcsok súlyozása miatt más-más nevet kapnak. Ezek a nevek: StatementNode, ConditionNode, LoopNode, FunctionNode, CustomNode.

```
foreach (var expression in function.Expressions)
{
    Node node=null;
    List<Variable> outPuts = new List<Variable>();
    List<Variable> inPuts = new List<Variable>();
    if (expression is Statement)
    {
        node = new StatementNode();
    }
    else if (expression is Condition)
    {
        var condition = (Condition)expression;
        node = new ConditionNode();
        outPuts = condition.TrueExpressions[0].LeftVariables.ToList();
        inPuts = condition.TrueExpressions[0].RightVariables.ToList();
        if (condition.FalseExpressions.Count > 0)
        {
            outPuts.AddRange(condition.FalseExpressions[0].LeftVariables);
            inPuts.AddRange(condition.FalseExpressions[0].RightVariables);
        }
    }
    else if (expression is Loop)
    {
        var loop = (Loop)expression;
        node = new LoopNode();
        outPuts = loop.Expressions[0].LeftVariables;
        inPuts = loop.Expressions[0].RightVariables;
    }
    else if (expression is FunctionExpression)
    {
        node = new FunctionNode();
    }
    else if (expression is CustomExpression)
    {
        node = new CustomNode ();
    }

    outPuts.AddRange(expression.LeftVariables);
    inPuts.AddRange(expression.RightVariables);

    inPuts = inPuts.Distinct().ToList();
    foreach (var input in inPuts)
    {
        graph.AddRelation(input, node);
    }

    node.ProgramCode = expression;


    outPuts = outPuts.Distinct().ToList();
    foreach (var outPut in outPuts)
    {
        var newNode2=graph.GetSyncNode(outPut);
        graph.AddRelation(newNode2, node);
        graph.Nodes.Add(newNode2);

        var newNode3 = graph.GetNextNode(outPut);
        graph.AddRelation(node, newNode3);
        graph.Nodes.Add(newNode3);
    }

    node.ProgramCode = expression;
    graph.AddNode(node);
```

Felépítettük a függőségi programgráfunkat. Az első fejezetben említett korai állapotváltás hiba javítására a felépített gráfunkon lefuttatjuk az élekkel kiegészítés algoritmust, amelyet szintén az első fejezetben részleteztünk:

```
List<Tuple<Node, Node>> newRelations = new List<Tuple<Node, Node>>();

foreach (var node in graph.Nodes)
{
    foreach (var parent in node.Parents.ToList())
    {
        var nextNode=graph.GetNextOccour(parent);
        if (nextNode != null && node!=nextNode)
        {
            if (!(nextNode.Childs.Contains(node)))
            {
                newRelations.Add(new Tuple<Node, Node>(node, nextNode));
            }
        }
    }
}
foreach (var relation in newRelations)
{
    graph.AddRelation(relation.Item1, relation.Item2);
}
```

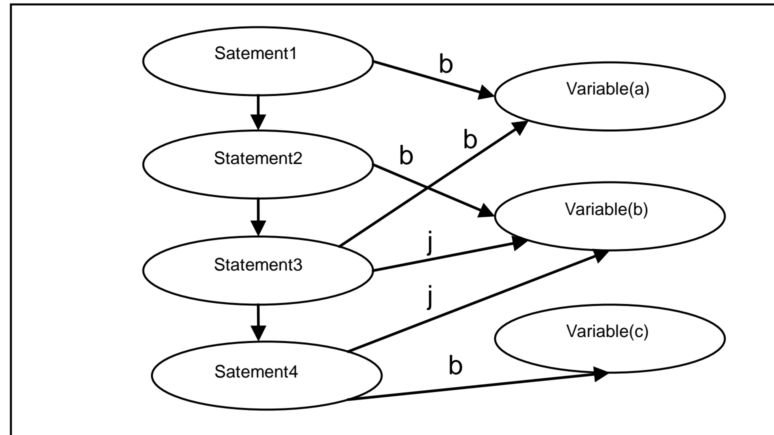
Végeztünk az élekkel kiterjesztett függőségi programgráf felépítésével. A gráfban az „expression” csúcsok tartalmazzák a függvény műveleteit. A többi csúcs csak a szinkronizáció miatt szükséges, ezért amikor ütemezzük a felépített gráf csúcsait, ezeket nulla súllyal számolhatjuk. A gráf működésére a következő fejezetben láthatunk egy egyszerű példát.

A gráf működésére vonatkozó példa

Ez a példa csak a párhuzamosítás bemutatására szolgál, olyan egyszerű programot adunk meg, amelyben nem kell refaktorálni, és nem kell függvények jobb és bal oldali változóit kiszámítani. Legyen a programunk:

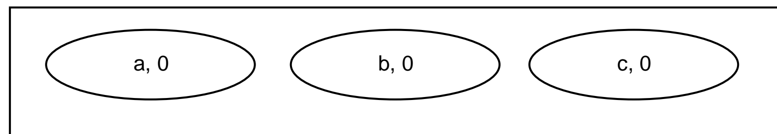
```
a=1
b=2
a=b
c=b
```

A második fejezetben tárgyalt modellben ennek a függvénynek a felépítése a *25. ábrán* látható.



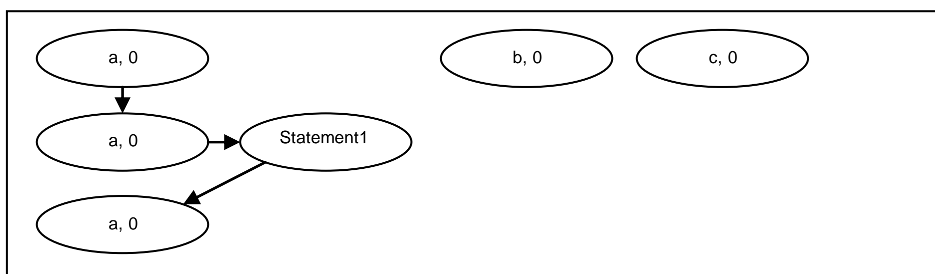
23. ábra: A példához tartozó modell

A műveletek között sima él van, mindegyik között vizsgáljuk, hogy futhatnak-e párhuzamosan. Felépítjük a függőségi programgráfot. Első lépésben felvesszük a használt változók nulladik előfordulását. *(24. ábra)*

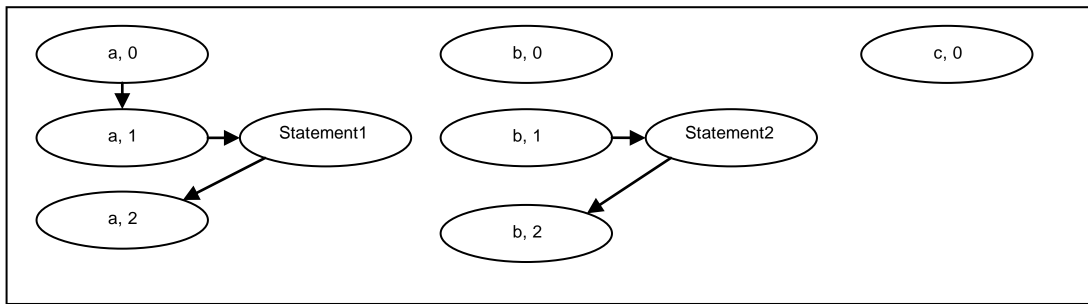


24. ábra: Függőségi programgráf felépítése. 1. lépés

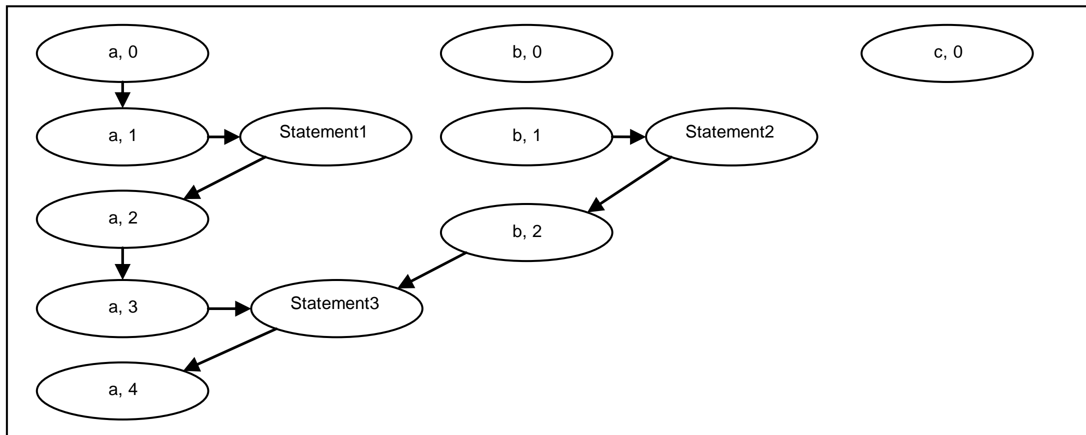
Elkezdjük feldolgozni a műveleteket. Minden további lépésben egy-egy műveletet rendelünk hozzá a gráfhoz. *(25-28. ábra)*



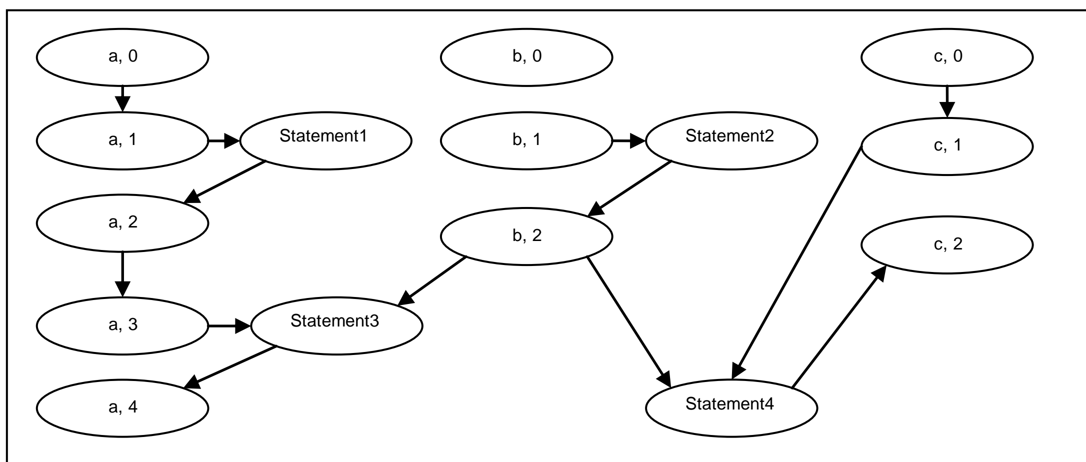
25. ábra: Függőségi programgráf felépítése. 2. lépés



26. ábra: Függőségi programgráf felépítése. 3. lépés



27. ábra: Függőségi programgráf felépítése. 4. lépés



28. ábra: Függőségi programgráf felépítése. 5. lépés

Végeztünk a függőségi programgráf felépítésével. Az élekkel való kiterjesztés nem húz be újabb éleket. Ütemezzük a csúcsok végrehajtását. Az egyszerűség kedvéért minden értékadást tekintsünk a példánkban egységnyi idejűnek. Mivel a normál és a szinkronizációs csúcsok súlya mindig nulla, ezért ezeket minden lépésben előre feldolgozunk, nem rendelünk hozzá processzort. **Első lépésben** tehát az *(a, 0), (b, 0), (a, 0), (a, 1), (b, 1), (c, 1)* csúcsokat előre feldolgozzuk, majd megnézzük melyek azok a csúcsok, amelyek most

futtathatóak. Ez a Statement1, és Statement2 csúcs, ezeket két processzorhoz rendeljük. **Második lépésben** előre feldolgozzuk a *(a, 2), (b, 2), (a, 3)* csúcsokat, majd megnézzük mik azok a csúcsok, amelyek most futtathatóak. Ezek a Statement3 és Statement4 csúcs, feldolgozzuk ezeket. **Harmadik lépésben** előre feldolgozzuk az *(a, 4), (c, 2)* csúcsokat. Minden csúcs fel van dolgozva, a harmadik lépésben tényleges utasításra nem került sor. **Összefoglalva** a következőt kaptuk:

Első lépés:

P1: `a=1`
P2: `b=2`

Második lépés:

P1: `a=b`
P2: `c=b`

Harmadik lépésben nem történik tényleges utasítás, ezért itt nem is ütemezünk semmit.

4. Párhuzamosító módszerek összehasonlítása

Ebben a részben Halász Attila által elkészített párhuzamosító módszert[12] hasonlítom össze a sajátommal. Az összehasonlítás célja, hogy megmutassam, hogy az én módszerem miben más, esetleg több Halász Attila módszerénél.

4.1. Programábrázolás

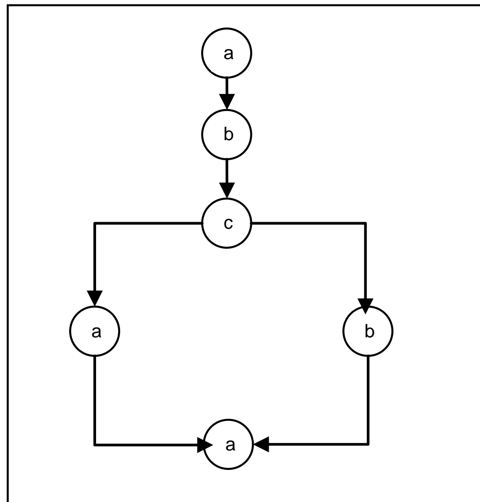
Halász Attila munkájában abból indul ki, hogy adott a soros program folyamatábrája, és a szomszédos utasítások között nézzük, hogy melyek változó függetlenek, és ott végzünk párhuzamosítást.

Dolgozatomban nem a folyamatot nézzük, hanem azt, hogy egyes változók mikor kapnak értéket, és mely változóktól függ az érték, amelyet kapnak. Nagyvonalakban akkor párhuzamosíthatunk két értékadást, ha egyik változó sem függ a másiktól. Első ránézésre hasonlónak tűnhet a két szemlélet, de nézzük a következő példát:

```
a=1
b=1
ha c==1 akkor a=2, különben b=2
a=c
```

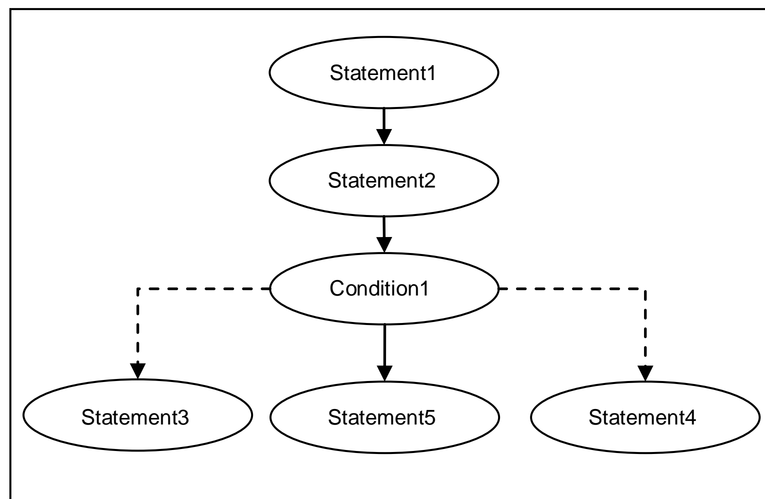
Halász Attila a *29. ábrában* foglaltakból indul ki, amelyet a folyamatábrából nyer ki, és a párhuzamosító algoritmus ebből a gráfból készít egy párhuzamos programot úgy, hogy csak az éleket módosítja.

[12] Halász Attila Péter: Objektumorientált, szekvenciális programok átalakítása párhuzamos környezet számára hatékony implementációval, 2006. http://compalg.inf.elte.hu/~tony/Oktatas/Diplomamunka-Szakdoli-Nagyprogram/Halasz%20Attila%20-%20Diplomamunka.zip



29. ábra Halász Attila programgráfja

Dolgozatomban nem a folyamatábrából indulok ki, hanem felépítettem egy saját modellt, amelyben ábrázolom a függvények műveleteit. A párhuzamosító eljárásom nem egy meglévő gráfban módosítja az éleket, hanem egy újat hoz létre, ahol a csúcsok is különbözőek lesznek. A saját modellemben ábrázolt program a *30. ábrán* látható.



30**. ábra: Saját modellemben ábrázolt példaprogram**

4.2. Párhuzamosítási különbségek

Ebben a részben azt fejtem ki, hogy az egyes módszerekkel párhuzamosított programok között milyen különbségek vannak.

4.2.1. Író/Olvasó probléma

A párhuzamos folyamatoknál szoktuk említeni az író/olvasó problémát. Adott egy közös tárterület, amelyet egyszerre akár többen is olvashatnak, de csak egyszerre egy valaki írhat, illetve írás közben nem lehet olvasni. Halász Attila munkájában az él törés feltétele, hogy a két szomszédos utasítás változó független legyen, ami nála azt jelentette, hogy nem hivatkoznak az utasítások semmilyen módon az adott változóra. Ebből kifolyólag ha a két utasítás csak olvasta a hivatkozott változót, akkor sem futhattak párhuzamosan. Az én dolgozatomban a változók írás/olvasásánál több olvasás megengedett egyszerre, de csak egy írás, illetve írás közben nem megengedett az olvasás. Az alábbi példával illusztrálom a két módszer közötti különbséget:

```
a=a+1
b=a
c=a
```

Halász Attila módszerével ezen a programon nem tudnánk párhuzamosítani, az én módszeremmel az utolsó két utasítás végrehajtható lesz párhuzamosan.

4.2.2. Elágazás, ciklus probléma

Halász Attila munkájában döntési pontként is említette ezeket az eseteket, én is ezt az elnevezést fogom itt használni. A módszerével, ha döntési ponthoz érünk, akkor ott mindenképpen meg kell várni, hogy az addigi utasítások lefussanak, mert döntési pontnál megtiltotta az él törést. Ha a döntési pont után voltak olyan utasítások, amelyek az elágazás előttiektől teljesen függetlenek, akkor sem futhattak velük párhuzamosan. Az én módszeremmel ez a probléma nem áll fenn, ezek az utasítások futhatnak párhuzamosan. Az alábbi példával illusztrálom a két módszer eredményét:

```
a=1
ha a==1, akkor a=2
b=3
```

Halász Attila módszerével itt nem tudunk párhuzamosítani, az enyémmel első lépésben lefut az 1. és 3. utasítás, majd második lépésben a 2. utasítás.

4.2.3. Függvények ki- és bemenetei

Függvényhívásnál a be- és kimenetek meghatározására a dolgozatomban egy algoritmust adtam, hogy hogyan kaphatjuk meg. Halász Attila munkájában nem találtam erre algoritmust, illetve nem találtam példát nála, amelyben függvényhívás volt.

Összegezve, a két módszer más fajta megközelítést használ, két dologban hatékonyabb a módszerem Halász Attila módszerénél. Az író olvasó problémát jobban kezelem, mert csak akkor tiltjuk az olvasást, ha történik írás is közben. Másodszor nálam a döntési pontnál nincs mindenképpen szinkronizáció, ha a döntési pont utáni műveletek függetlenek a döntési pont előttiektől, akkor párhuzamosan végrehajthatók.

5. Programoptimalizálás

Ha elhagyjuk a triviális függőségeket, akkor a felépített függőségi programgráfunk programoptimalizálásra is alkalmas lehet. A szimulációs program ezt már nem tartalmazza, ezért pszeudokóddal írom le az algoritmusokat.

Tétel

Ha a felépített függőségi programgráfból eltávolítjuk azokat a csúcsokat, amelyekből nem vezet ki él, és nem egy változó utolsó előfordulását ábrázoló csúcs, akkor az új gráfból épített program ugyanazt a végeredményt adja, mint az eredeti gráfból felépített program.

Bizonyítás

A felépített gráfban azok a csúcsok, amelyekből nem vezet ki él, azt jelzik, hogy a változó adott előfordulását nem használtuk fel sehol. Ekkor a változónak az ilyen előfordulása felesleges, hacsak nem a végállapotát ábrázolja, amelyből szintén nem vezet ki él, mivel sehol nem fogjuk felhasználni az értékét.

A felesleges csúcsok eltávolítása egy egyszerű algoritmussal megoldható:

```
foreach(csúcs in gráf)
        if csúcs.gyerek=Üreshalmaz and not Végcsúcs(csúcs)
                GráfKiszed(csúcs)
```

Előfordulhat olyan eset is, hogy nekünk csak bizonyos változók értéke kell a program végén, nem az összes. Ekkor feleslegessé válik minden olyan csúcs, amelyből nem vezet út ezen változók utolsó előfordulását ábrázoló csúcsok valamelyikéhez. A továbbiakban értékes csúcsoknak nevezzük ezeket. Minden értékes csúcstól indítunk egy mélységi bejárást, a látogatott csúcsokat egy halmazba tesszük, és azokat a csúcsokat törölhetjük, amelyek nem kerültek be ebbe a halmazba.

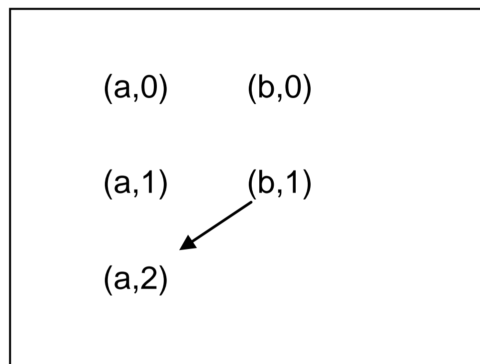
```
Halmaz=Üres
foreach(csúcs in értékes csúcsok)
        Bejár(csúcs)
foreach(csúcs in gráf)
        if csúcs not in halmaz
                GráfKiszed(csúcs)
```

```
Bejár(csúcs)
        halmaz.bepakol(csúcs)
        foreach(szülőcsúcs in csúcs.szülők)
                Bejár(szülőcsúcs)
```

Az alábbi példával szemléltetjük az algoritmus működését:

```
a=1;
b=2;
a=b+2;
```

Könnyen észrevehető, hogy az első *a=1* értékadás felesleges. A példa kódjának a függőségi programgráfja triviális függőségek nélkül a *31. ábrán* látható.



31. ábra: A példa függőségi programgráfja triviális függőségek nélkül

Az optimalizáló algoritmust ezen a gráfon lefuttatva azt kapjuk, hogy csak az ($a,2$), és a ($b,1$) csúcsok maradtak benne, amelyek megfelelnek a $b=2$ és $a=b+2$ utasításoknak.

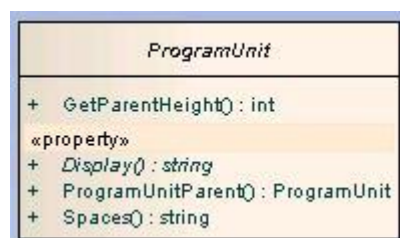
6. Fejlesztői dokumentáció

A párhuzamosító módszer bemutatására egy programot írtam .Net 3.5 keretrendszerben C# nyelven. A program tartalmazza az objektum-elvű modell felépítését, a modellben való refaktorálást, valamint a modellben való párhuzamosítást. Az egyes programnyelvek szintaktikai szabályainak feldolgozását nem foglalja magába a dolgozat, ezért csak a modellben foglalkozunk a párhuzamosítással. A módszer működésének szemléltetéséhez szükséges, hogy valamilyen kódot lássunk, hogy miből indultunk ki, és mi lett belőle. Ezt a kódot a modellből állítjuk elő. A program a párhuzamosításon túl egy szimulációt is magában foglal, amellyel véletlen programokat lehet előállítani különböző paraméterek megadásával.

A fejlesztői dokumentációt az objektum-elvű modell ábrázolásához szükséges entitások[13] leírásával kezdem. A látható osztálydiagramokat az Enterprise Arhitect nevű programmal készítettem.

6.1. Objektum-elvű program modell

Minden egyes az objektum-elvű modellhez tartozó entitás a **ProgramUnit** osztályból származik. *(32. ábra)*



32. ábra: ProgramUnit osztály

---

[13] Entitás fogalma: http://en.wikipedia.org/wiki/Entity

Tagváltozók:

- Display: Adott programegység szöveges formátuma.
- ProgramUnitParent: Adott programegység szülő programegysége
- Spaces: Kód előállításánál a tördeléshez szükséges szóközök.

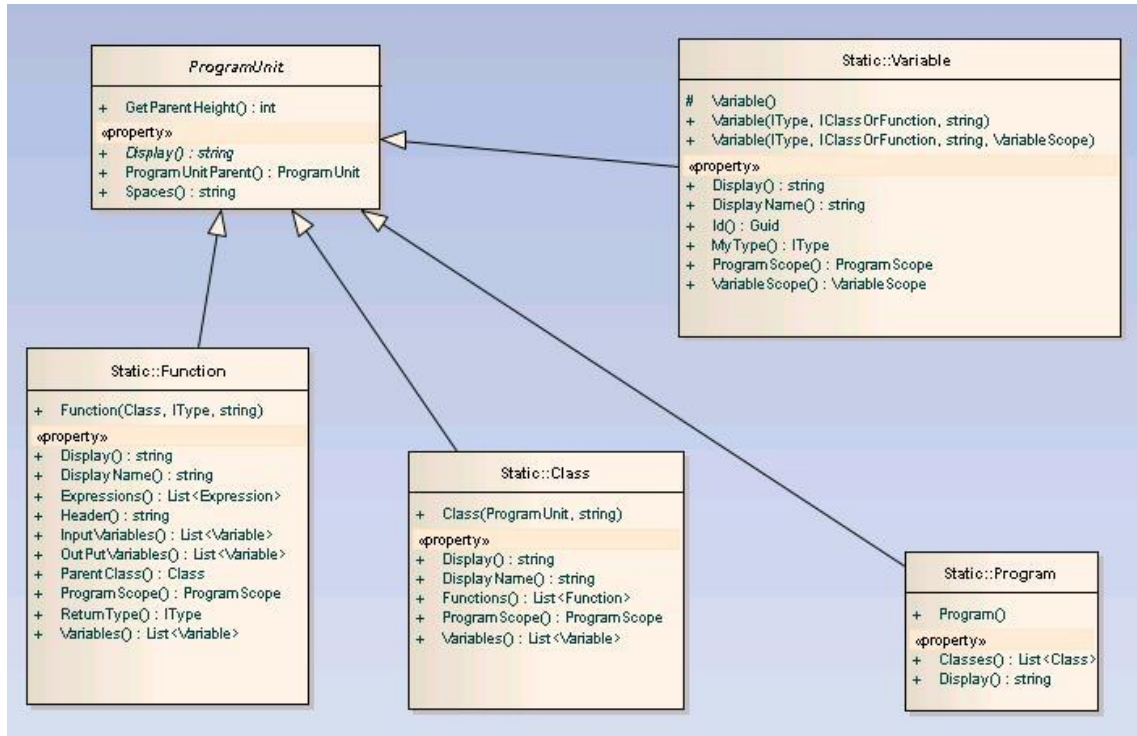
Függvények:

- GetParentHeight: Visszaadja, hogy hányadik gyerek a ProgramUnit hierarchiában.

Az objektum elvű modell három fő részre bontható. Az **első rész** a statikus rész, amelybe az osztályleírások tartoznak, a **második rész** a dinamikus rész, amelybe a függvények műveletei tartoznak, a **harmadik részbe** az értékek és típusok ábrázolása tartozik. Amikor az objektum-elvű modellben lévő elemekről írok, akkor a fogalmakat idézőjelbe teszem, mert a párhuzamosító program is objektum-elvű, ezért a fejlesztői dokumentáció is hasonló nevű fogalmakat használ.

6.1.1. Statikus modell

A **statikus** elemekhez tartozik a Program, Class, Function és a Variable osztályok. Osztálydiagramjuk a *33. ábrán* látható.



33. ábra: Statikus programelemek. Program, Class, Function, Variable osztályok

Program osztály: Ez az osztály foglalja össze az egész programot, ebben vannak az „osztályok”.

Class osztály: Ez az osztály reprezentálja az objektum-elvű környezetben az „osztályt”.

Function osztály: Ez az osztály reprezentálja „függvényeket”. Az „osztályok” ilyen „függvényeket” tartalmaznak.

Variable osztály: Ez az osztály reprezentálja „változókat”. Az „osztályok” és „függvények” ilyen „változókat” tartalmaznak.

6.1.2. Dinamikus modell

A dinamikus modellbe tartoznak a függvények műveletei. A műveletek ősosztálya az **Expression** osztály.

Tagváltozói:

- LeftVariables: A műveletben használt bal oldali változók.
- RightVariables: A műveletben használt jobb oldali változók.
- RightValues: A műveletben használt jobb oldali értékek.

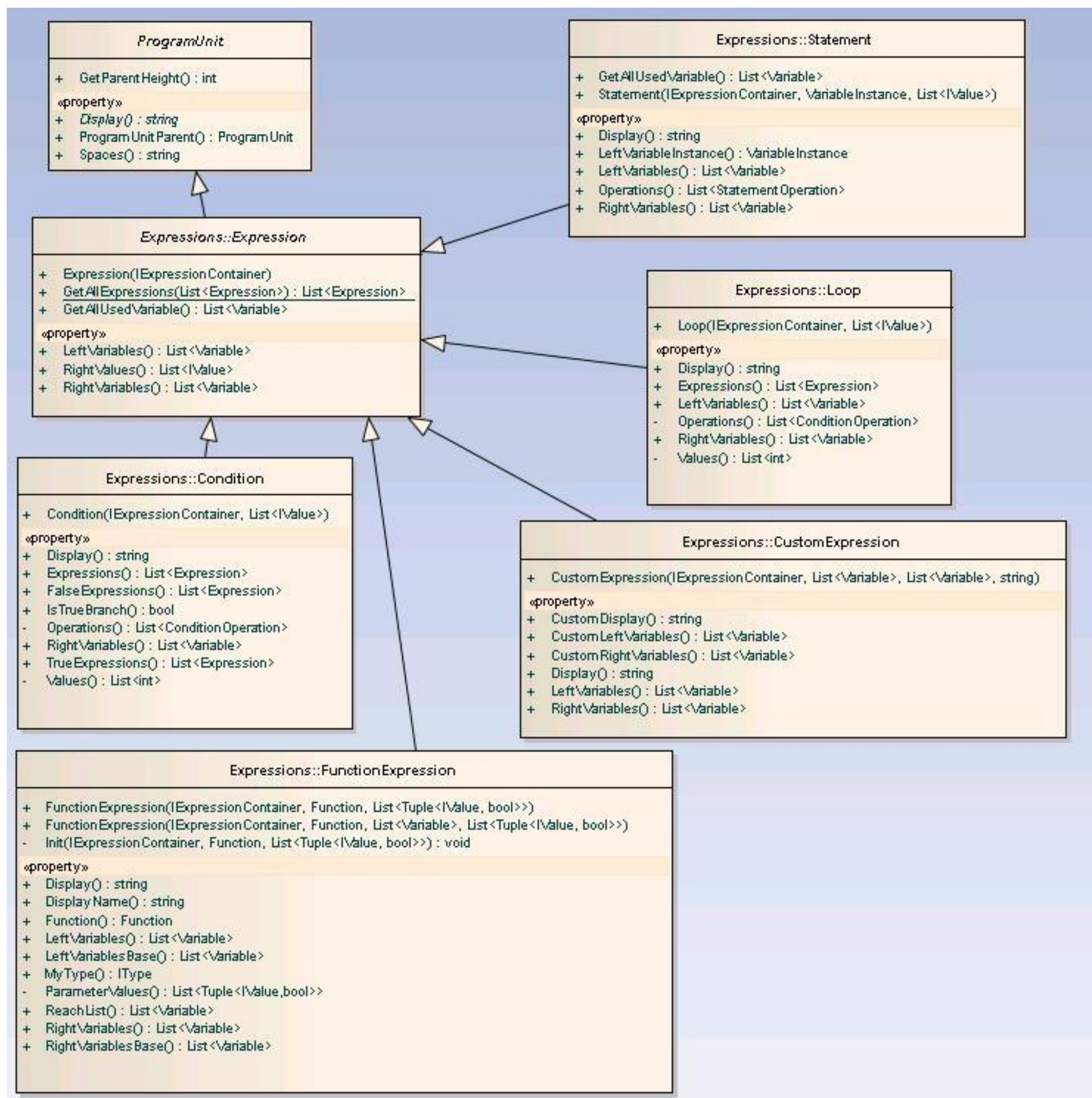
Függvénye:

- GetAllUsedVariable: Visszaadja az összes változót, amelyeket a műveletben használunk.

Statikus függvénye:

- GetAllExpressions(`List<Expression> pexpressions`): Visszaadja az összes műveletet, amelyeket a művelet listából elérhetünk.

Az Expression osztály és leszármazottjainak osztálydiagramja a *34. ábrán* látható.



34. ábra: Dinamikus programelemek. Expression, Statement, Condition, Loop, FunctionExpression, CustomExpression osztályok

Statement osztály: Ez az osztály reprezentálja az „értékadást”.

Condition osztály: Ez az osztály reprezentálja az „elágazást”. Azok a műveletek, amelyek az „elágazás” belsejében vannak, az „elágazás” lesz a ProgramUnitParent-je.

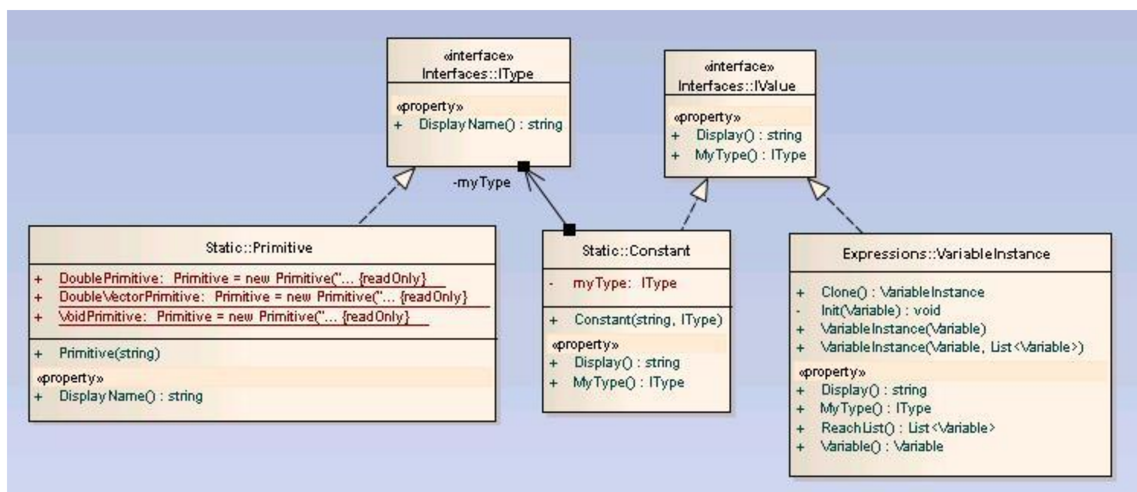
Loop osztály: Ez az osztály reprezentálja a „ciklust”. Azok a műveletek, amelyek a ”ciklus” belsejében vannak, a „ciklus” lesz a ProgramUnitParent-je.

FunctionExpression osztály: Ez az osztály reprezentálja a „függvényhívásokat”. Legfontosabb tagváltozója a Function, amely a hivatkozott „függvényre” mutat.

CustomExpression osztály: Ez az osztály kezeli az olyan eseteket, amelyek nem sorolhatóak be az előzőekbe. Azért fontos egy ilyen osztály használata, hogy a modell rugalmas legyen a speciális nyelvi elemek kezelésére egy későbbi továbbfejlesztés során.

6.1.3. Értékek és típusok

„Típusok” ábrázolására egy interface-t vezettem be **IType** névvel. Ezt az interface-t az „osztály”, és a „primitív típusok” implementálják. Az „érték” típusra az **IValue** interface-t vezettem be. Minden IValue interface-t implementáló osztályhoz tartozik egy IType típus, amely a „típusát” adja meg. *(35. ábra)*



35. ábra: IType és IValue interface. Primitive, Constant, VariableInstance osztályok

Primitive osztály: a „primitív típusok” ábrázolására szolgáló osztály.

Constant osztály: A „konstansok” ábrázolásáért felelős osztály.

VariableInstance osztály: A „függvényekben” amikor egy változóra hivatkozunk, ez az osztály burkolja be ezt a változót. A burkolásra azért van szükség, mert lehet, hogy egy elérési láncon keresztül érjük el a változót, és ezt tárolni kell.

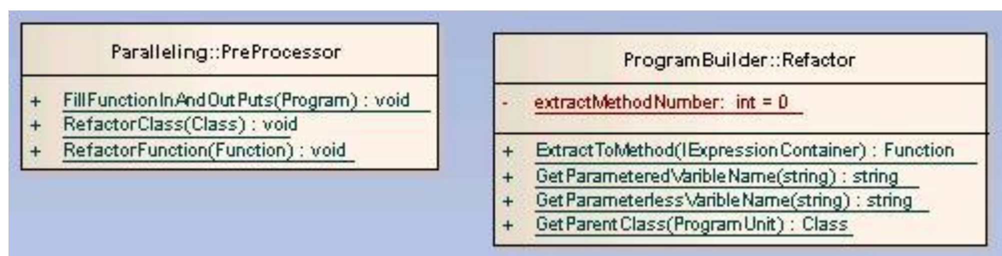
Az objektum-elvű modell ábrázolását befejeztük a programban. A következő fejezetben a párhuzamosító algoritmust megvalósító részt tárgyaljuk.

6.2. Párhuzamosítás

A párhuzamosítási folyamatot egy előkészítő eljárás előz meg. Ebbe az eljárásba tartozik a program refaktorálása, illetve a függvények jobb és bal oldali változóinak meghatározása. Ezután a függőségi programgráf szerkezetét tárgyaljuk, majd végezetül a függőségi programgráf felépítését vizsgáljuk.

6.2.1. Előkészítő eljárás

Az előkészítő eljárásban lévő feladatokat a **PreProcesszor** osztály végzi el a **Refactor** osztály segítségével, amelyeknek az osztálydiagramjait a *36. ábrán* láthatjuk.



36. ábra: PreProcessor és Refactor osztályok

Legfontosabb függvényei:

- FillFunctionInAndOutPuts: Ez a függvény végzi el a „függvények” jobb és bal oldali változóinak kiszámítását. Ennek a függvénynek az algoritmusát tárgyaltuk a 3.2.1-es részben.

- RefactorClass: Ez a függvény végzi el az „osztály” refaktorálását. A 3.1-es fejezetben olvashatunk erről. Ez a függvény minden egyes „függvényre” meghívja a refaktorálást, magát az átalakítást a Refactor osztály ExtractToMethod függvénye végzi el.

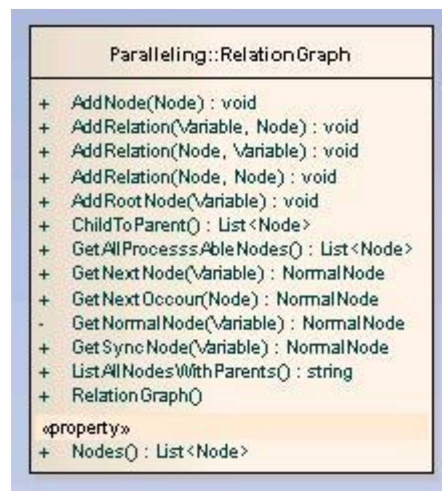
A párhuzamosítás elvégzéséhez szükség van a függőségi gráf ábrázolásához. Ennek a gráfnak a szerkezete következik.

6.2.2. Függőségi programgráf szerkezete

RelationGraph osztály: Ez az osztály felelős a gráfban szereplő csúcsok kezeléséért. Legfontosabb függvényei:

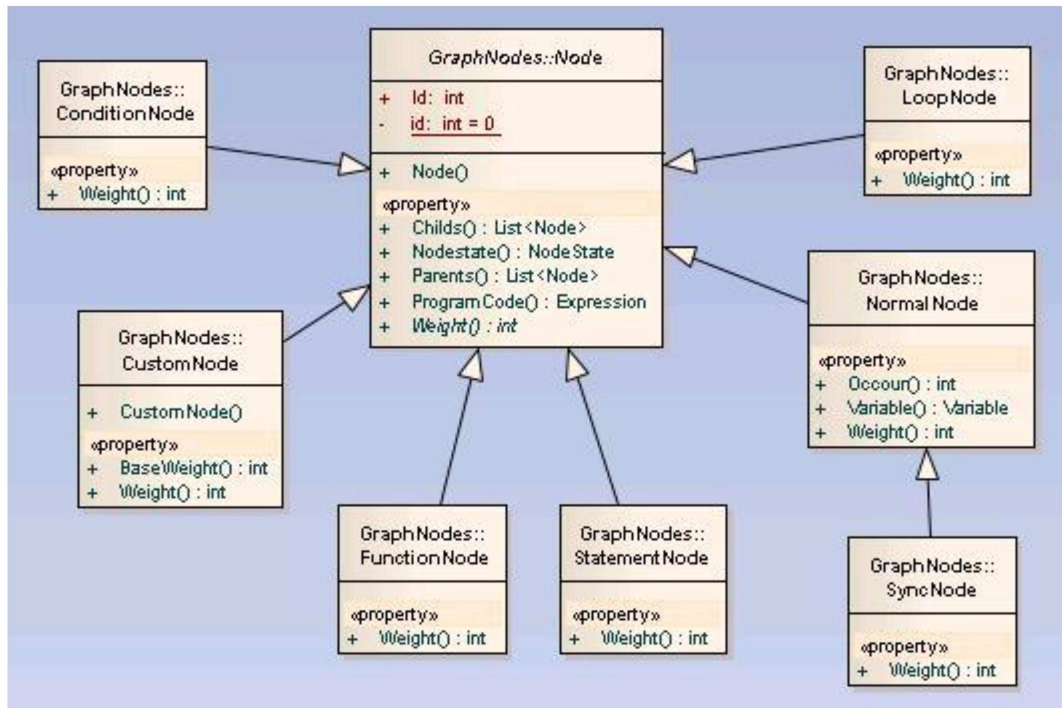
- AddNode: Csúcs hozzáadása a gráfhoz.
- AddRelation: Él hozzáadása a gráfhoz.
- GetNormalNode(Variable variable): Adott változó utolsó előfordulását ábrázoló csúcs visszaadása.
- GetAllProcessAbleNodes: Visszaadja azokat a csúcsokat, amelyek végrehajthatóak a gráfban.

Az **RelationGraph** osztály osztálydiagramja a *37. ábrán* látható.



37. ábra: RelationGraph osztály

A gráfban használt csúcsok tárgyalása következik. A csúcsok kezelésére bevezettem a **Node** ősosztályt, a különböző tulajdonságú csúcsok ebből származnak. A csúcsok osztálydiagramját a *38. ábrán* láthatjuk.



38. ábra: Node, NormalNode, SyncNode, StatementNode, ConditionNode, LoopNode, FunctionNode, CustomNode osztályok

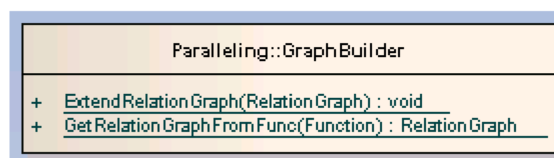
NormalNode, SyncNode osztály: A változók előfordulásaihoz tartozó csúcsok ábrázolásáért felelős osztályok.

StatementNode, **ConditionNode**, **LoopNode**, **FunctionNode**, **CustomNode** osztályok: Értékadást, ciklust, függvényhívást és egyéb utasítást ábrázoló csúcsokhoz tartozó osztályok.

Végeztünk a függőségi programgráf szerkezetével. A következő fejezetben a gráf felépítéséért felelős osztályt tárgyaljuk.

6.2.3. Függőségi programgráf felépítése

A függőségi programgráf felépítéséért a **GraphBuilder** osztály a felelős. Az osztálydiagramja a *39. ábrán* látható.



39. ábra: GraphBuilder osztály

Függvényei:

- GetRelationGraphFromFunc: Ez a függvény végzi el a függőségi programgráf felépítését. A 3.2.2-es fejezetben tárgyaltunk erről.
- ExtendRelationGraph: Ez a függvény végzi el az élekkel való kiterjesztést. Szintén a 3.2.2-es fejezetben tárgyaltunk erről.

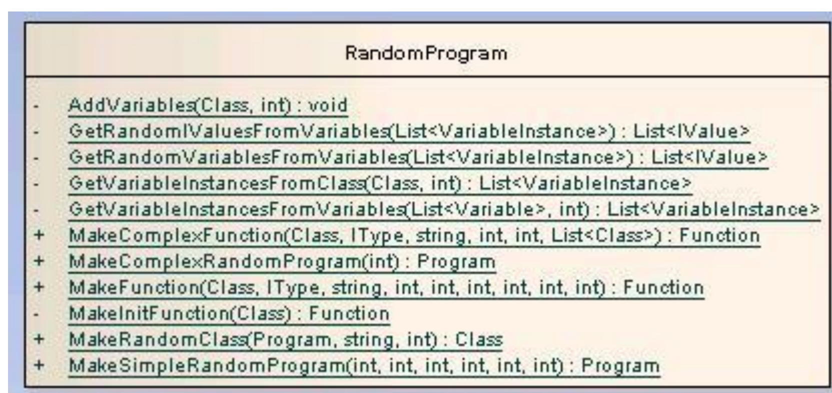
A párhuzamosításért felelős osztályok leírását befejeztük. A következő fejezetben a szimulációs programrész leírása következik.

6.3. Szimulációs programrész

A diplomamunkához készített program harmadik része a szimulációs elemzés. Ebben a részben a szimulációval foglalkozó osztályokat és fontosabb függvényeit tárgyaljuk.

A **RandomProgram** osztály feladata egy véletlenszerű program előállítása. Az osztálydiagramja a *40. ábrán* látható. Fontosabb függvényei a következők:

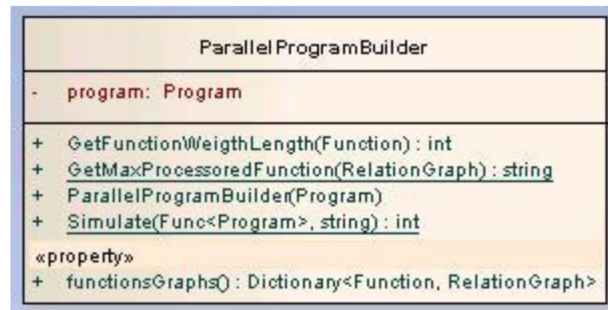
- MakeSimpleRandomProgram: Kapott paraméterek alapján előállít egy osztályból és egy függvényből álló soros programot. A szimulációban ezt a függvényt fogom használni.
- MakeComplexRandomProgram: Kapott paraméterek alapján előállít több osztályból és több függvényből álló soros programot.



40. ábra: RandomProgram osztály

A **ParallelProgramBuilder** osztály feladata egy meglévő soros programhoz felépített párhuzamos program elemzése. Osztálydiagramja a *41. ábrán* látható. Fontosabb függvényei a következők:

- GetMaxProcessoredFunction: A paraméterben átadott függőségi programgráfból visszaad egy olyan programkódot, amely a párhuzamos programot tartalmazza.
- Simulate: A paraméterben átadott függőségi programgráfból kiszámolja, hogy soros és párhuzamos esetben mennyi ideig tart a program végrehajtása.



41. ábra: ParallelProgramBuilder osztály

A fejlesztői dokumentáció véget ért. A következő fejezetben a szimulációs eredményeket elemezzük.

7. Szimuláció

A program szimulációs részében különböző véletlen soros programokat állítok elő. Az előállítást különböző paraméterek alapján hajtom végre, ezek a paraméterek a következők.

- **Műveletszám:** függvény műveleteinek a száma
- **Változószám:** függvényben használt változók száma
- **Értékadás százalék:** egy művelet hány százalék eséllyel legyen értékadás
- **Elágazás százalék:** egy művelet hány százalék eséllyel legyen elágazás.
- **Ciklus százalék:** egy művelet hány százalék eséllyel legyen ciklus.
- **Mélységi mérték:** amikor egy elágazás vagy ciklus belsejében vagyunk, hány százalék eséllyel maradjon a belsejében a következő művelet. Másként megfogalmazva ezzel a paraméterrel azt állítjuk, hogy milyen mélyek legyenek az egymásba ágyazott elágazások, ciklusok.

Az előállított soros programhoz felépítem az élekkel kiterjesztett függőségi programgráfját. A mellékletben három konkrét szimulációt csatoltam, amelyek az előállított programból, a refaktorált programból, és egységes súlyozású műveletek mellett felépített párhuzamos programból állnak. Azért választottam egységes súlyozást, mert így a leginkább áttekinthető, hogy melyik lépésben mely műveleteket hajtjuk végre.

Hatékonyságelemzéshez megvizsgáljuk, hogy a gráffal mennyivel hatékonyabb program állítható elő, mint soros esetben. A gráf egyes pontjait más-más súlyozással láttam el, ezek a következők:

- Értékadás esetében a súly 1+jobb oldali változók száma. A bal oldali változó miatt van ott a plusz egy érték.
- Elágazás esetében a súly a feltételben szereplő értékek száma, továbbá hozzávesszük az igaz és hamis ágában lévő művelet súlyainak az

átlagát. Szándékosan nem tettem többes számba a művelet szót, mert refaktorálás után már biztosan csak egy, vagy nulla darab művelet lesz az igaz, és a hamis ágban is.

- Ciklus esetében a súly a feltételben szereplő értékek száma n-szeresen, továbbá hozzávesszük a ciklusmagban szereplő művelet súlyát szintén n-szeresen. Az n egy 1 és 10 között szereplő véletlenszerű érték.
- Függvényhívás esetén a súly attól függ, hogy párhuzamos, vagy soros súlyokat vizsgálunk. Soros esetben a súly a meghívott függvényben szereplő összes művelet súlyának az összege. Párhuzamos esetben korlátlan számú processzorral számolva a súly a meghívott függvényhez felépített élekkel kiterjesztett függőségi programgráf súlyozott magassága.

Egy függvény végrehajtási idejét úgy számoltam ki soros esetben, hogy az összes benne szereplő művelet súlyát összeadtam. Párhuzamos esetben a függvény végrehajtási idejét korlátlan számú processzorra számoltam. A függvényhez felépített élekkel kiterjesztett függőségi programgráf súlyozott magasságát vettem végrehajtási időnek.

A szimuláció során a soros program előállításának paramétereit változtattam, és vizsgáltam, hogy ezek miként befolyásolják a párhuzamosítás hatékonyságát. Minden egyes esetet 1000-szer futtattam le, és átlagoltam az eredményeket.

A gyorsítás mértékét a következő képlet adja meg:

*gyorsítás százalék=100*soros futási idő/párhuzamos futási idő*

7.1. Szimuláció eredménye

1. eset

Műveletszám: változó

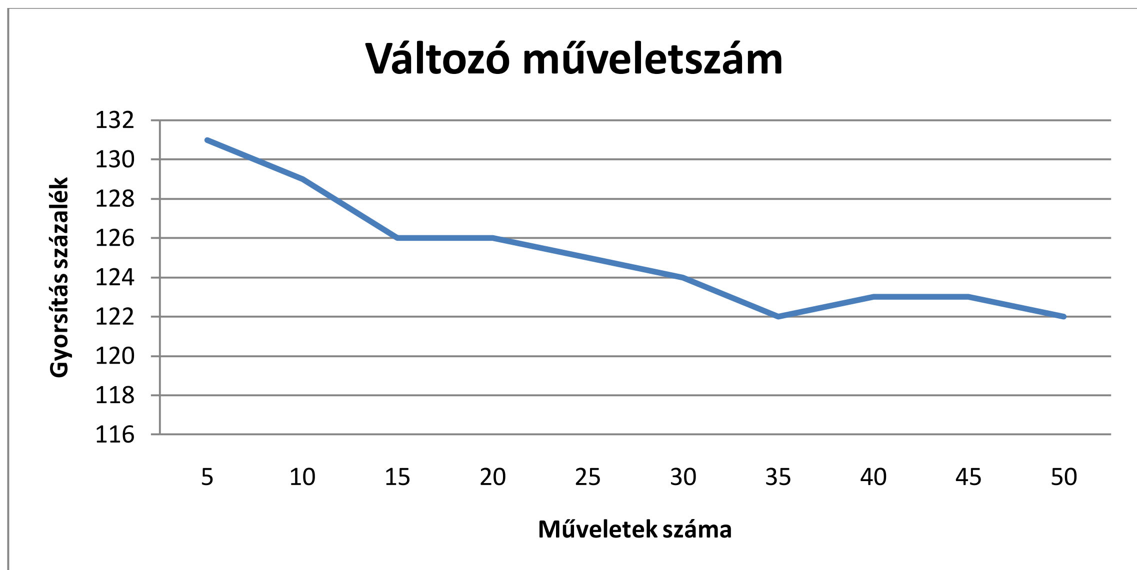
Változószám: 5

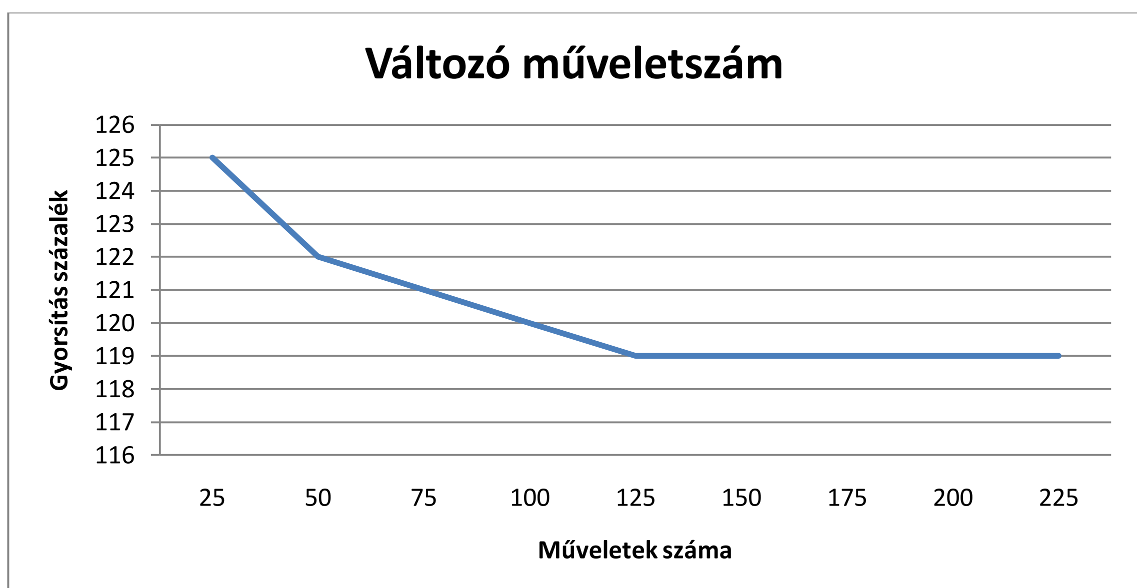
Értékadás esélye: 80%

Elágazás esélye: 10%

Ciklus esélye: 10%

Mélységi mérték: 30%





Ennél a szimulációnál azt láthatjuk, hogy minél több utasítás van egy függvényben, annál kevésbé hatékony a párhuzamosítás. 5 művelet esetén a gyorsítás 131%-os, míg 50 művelet esetén 122%-os a gyorsítás. Ez a gyorsítás romlás egy idő után megáll, 75 feletti műveletszám esetén a gyorsítás 119% körül mozog. Ezt a jelenséget az okozza, hogy sok művelet esetén az elágazások és ciklusok belseje egyre több utasítást tartalmaz. A párhuzamosítás során ezeket külön függvénybe exportáljuk, és a függvény egészére nézzük, hogy nem ütközik-e valamelyik művelete egy másik művelettel. Minél több műveletből áll egy függvény, annál valószínűbb, hogy valamelyik művelete ütközést fog okozni. Egy bizonyos műveletszám után azért áll le a romlás, mert az elágazások és ciklusok belseje már felhasználja az összes változót, és ha a belsejéhez hozzáveszünk még egy műveletet, az a művelet ugyanazokat a változókat fogja használni, amelyeket a ciklus vagy elágazás belsejében eddig már használtunk. Ezért az új művelet már nem fog új ütközést okozni.

2. eset

Műveletszám: 30

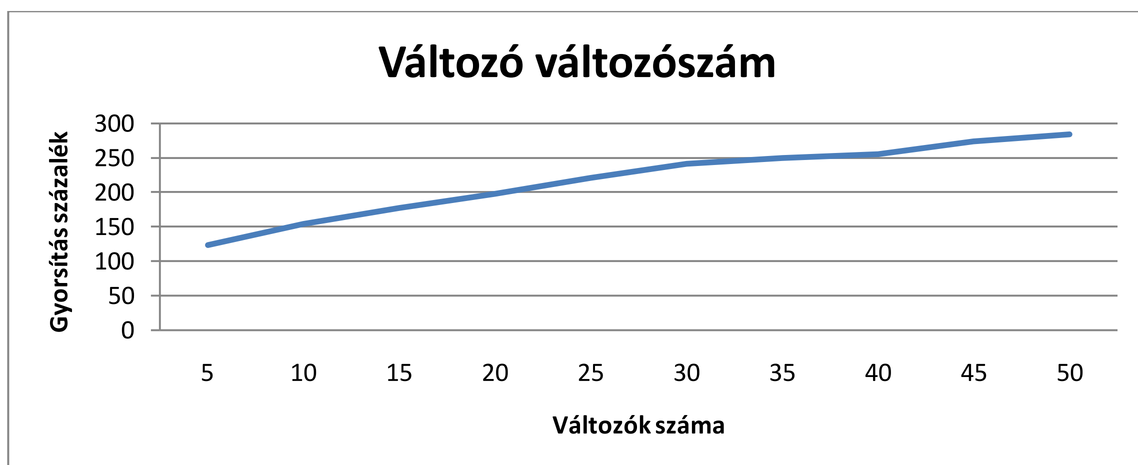
Változószám: változó

Értékadás esélye: 80%

Elágazás esélye: 10%

Ciklus esélye: 10%

Mélységi mérték: 30%



Ez a szimuláció azt mutatja, hogy a változószám nagyon befolyásolja az algoritmus hatékonyságát. 2 változó esetén mindössze 102% a gyorsítás, míg 10 változószám esetén a sebességnövelés már 154%-os. Ez érthető is, mert minél több változó van, annál kisebb az esélye, hogy két művelet ugyanazt a változót használja. A gyakorlati tapasztalatom az, hogy egy függvényben használt változók száma szinte sohasem megy 20 fölé, ezért a 20 felett mért értékek csupán érdekességeknek tekinthetők.

3. eset

Műveletszám: változó

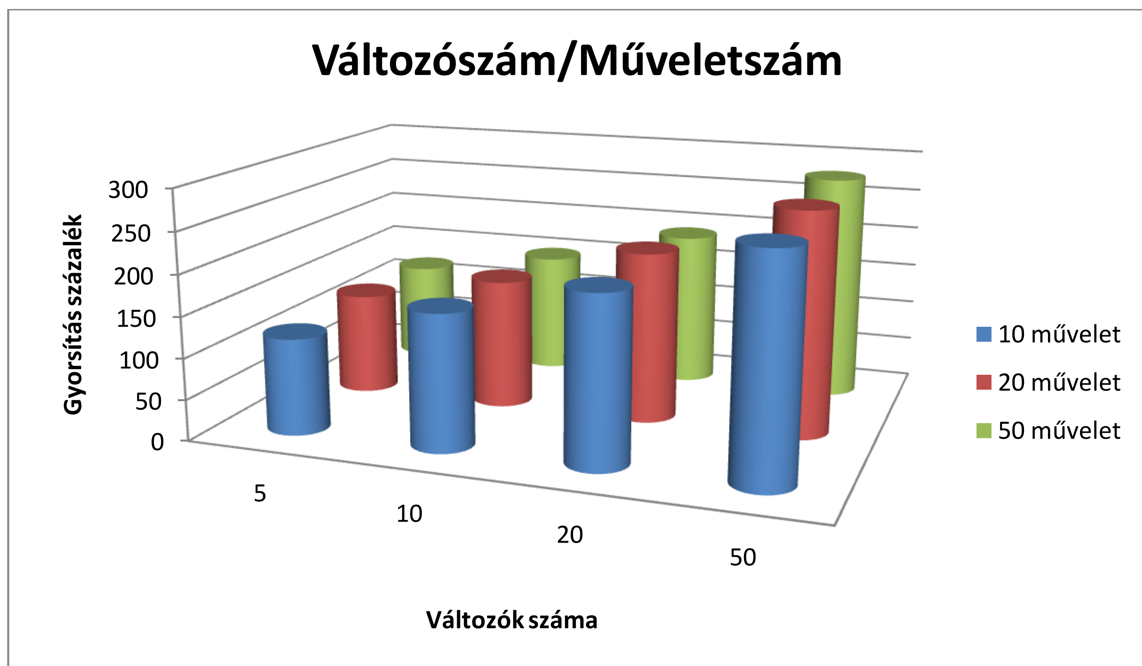
Változószám: változó

Értékadás esélye: 80%

Elágazás esélye: 10%

Ciklus esélye: 10%

Mélységi mérték: 30%



Az első esetben a műveleteknél azt állapítottuk meg, hogy rögzített paraméterek mellett a műveletszám növekedésével csökken a gyorsítás. Ebben a szimulációban a változószámot, és a műveletek számát is változtattuk. Azt kaptuk eredményül, hogy kis változószám mellett valóban csökken a gyorsítás a műveletek számának növelésével, viszont ahogy növeljük a változóink számát, úgy ez megfordul, és minél több műveletből áll a függvény, annál nagyobb gyorsítást kapunk. 50 változó esetén 10 műveletre 269%, 20 műveletre 274%, valamint 50 műveletre 278% gyorsítást kaptunk. Ennek az az oka, hogy nagy változószám mellett a kevés művelet nem tudja az összes változót felhasználni. A párhuzamosítás annál hatékonyabb, minél több változót használ a program. A tapasztalatom az, hogy egy függvényben a műveletek száma általában nagyobb a használt változók számánál, ezért általában a műveletek számának növekedése lassítja a gyorsítást.

4. eset

Műveletszám: 30

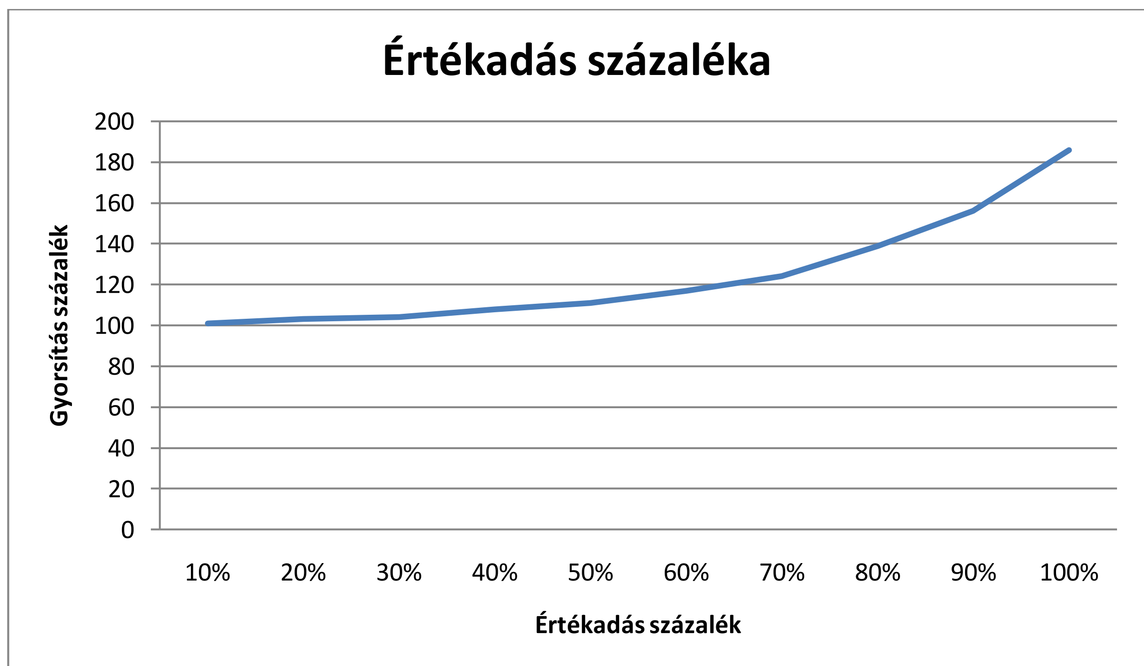
Változószám: 7

Értékadás esélye: változó

Elágazás esélye: (100-értékadás esélye)/2

Ciklus esélye: (100-értékadás esélye)/2

Mélységi mérték: 30%



Ez a szimuláció azt mutatja, hogy minél több az értékadás, annál nagyobb gyorsítást érünk el. 10%-os értékadás esély mellett a gyorsítás 101%-os, míg 100%-os értékadás mellett a sebességnövekedés már 186%-os. Jelentősebb gyorsítás 60-70%-os értékadás után következik be. Ez azért van, mert a párhuzamosító algoritmus az alapján dolgozik, hogy melyik változó mikor kap értéket. Ha nincs értékadás, nem kap értéket semmilyen változó. Minél több az értékadás, annál több helyen lehet megvizsgálni, hogy párhuzamosíthatóak-e a műveletek.

5. eset

Műveletszám: 30

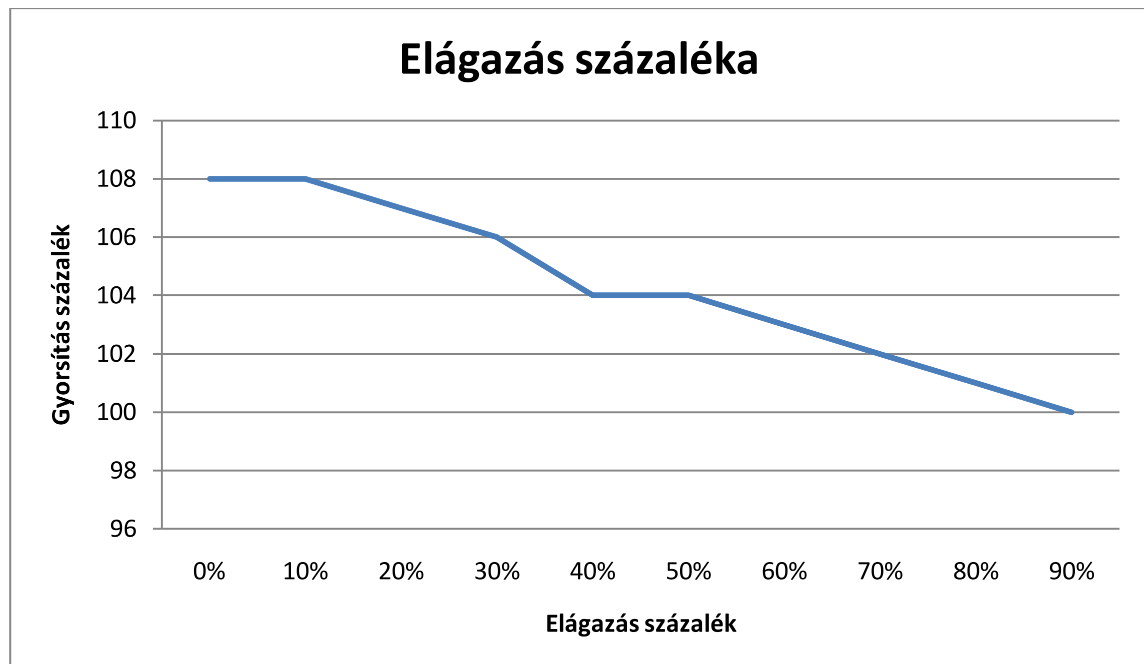
Változószám: 7

Értékadás esélye: (100-elágazás esélye)/2

Elágazás esélye: változó

Ciklus esélye: (100-elágazás esélye)/2

Mélységi mérték: 30%



Ebben a szimulációban azt tapasztaljuk, hogy minél nagyobb az elágazás esélye az algoritmusban, annál kevésbé hatékony programot kapunk. Ennek az az oka, hogy minél több elágazás van, annál kevesebb az értékadások száma.

6. eset

Műveletszám: 30

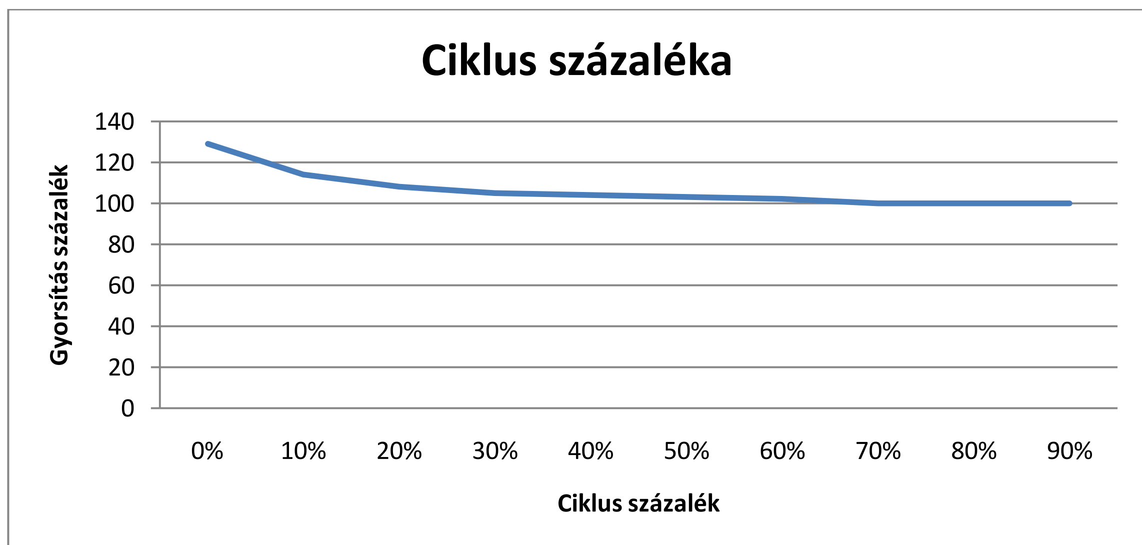
Változószám: 7

Értékadás esélye: (100-ciklus esélye)/2

Elágazás esélye: (100-ciklus esélye)/2

Ciklus esélye: változó

Mélységi mérték: 30%



Ebben a szimulációban azt tapasztaljuk, minél több a ciklus a programban, annál kisebb gyorsítást érünk el. A csökkenést itt is az okozza, hogy az értékadások száma csökken. Azért nagyobb a csökkenés mértéke itt, mint az elágazás vizsgálatának esetében, mert a ciklusfeltétel kiértékelését nem párhuzamosítjuk, és a szimulációban ezt a kiértékelést 1 és 10 közötti véletlen értékszer végezzük el. Az elágazás esetében az elágazás feltételt mindig csak egyszer végeztük el. Észrevehető továbbá, hogy itt 129%os gyorsításról indulunk, míg az elágazást vizsgálatánál a kiindulás csupán 108% volt. Ennek az az oka, hogy az elágazás szimulációjában mért első esetben a ciklusok száma 50% volt. Az értékadás, elágazás és ciklus esetében mért értékek alapján azt mondhatjuk, hogy az algoritmusunk annál hatékonyabb, minél több értékadás van benne. A gyakorlati tapasztalatom az, hogy egy átlagos függvényben az értékadások vannak többségben.

7. eset

Műveletszám: 30

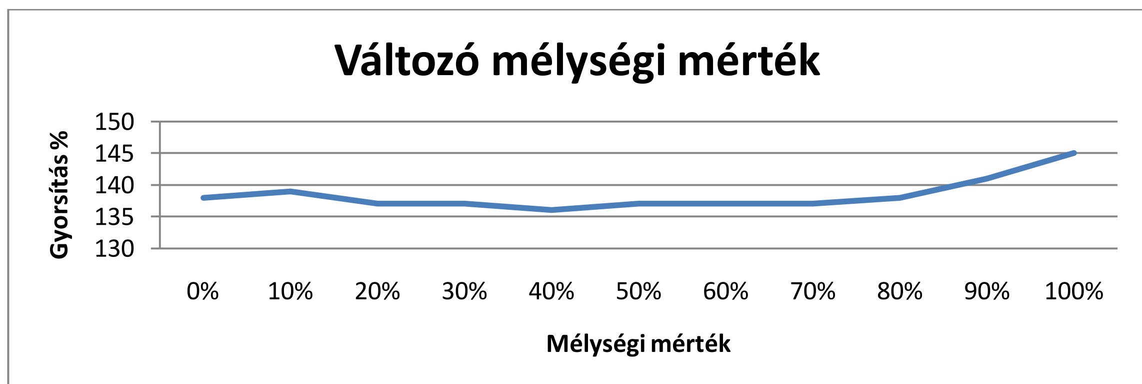
Változószám: 7

Értékadás esélye: 80

Elágazás esélye: 10

Ciklus esélye: 10

Mélységi mérték: változó



A szimuláció alapján azt tapasztalhatjuk, hogy ugyan kis mértékben, de növekszik a gyorsítás, ha egyre mélyebb elágazásokat és ciklusokat képzünk. 80%-os mélységi mértékig ugyan olyan volt a gyorsítás mértéke, majd a 80%-100%-os szakaszon növekedett 7%-ot. Ezt az okozza, hogy nagyon nagy mélységi mérték mellett nincsen egy szinten elágazás és ciklus, mert egymásba vannak ágyazva. Ha egy szinten van elágazás és ciklus, akkor az a párhuzamos gyorsítást korlátozza. Ennek az az oka, hogy az elágazások és ciklusok belsejét függvénybe exportáljuk, ezért egy szinten lévő elágazás és ciklus tekinthető egy szinten lévő két függvényhívásnak. A korlátozást az okozza, hogy ezek egy szinten vannak, mert ha a meghívott függvény bármelyik változóját használja a másik meghívott függvény változója, akkor a két függvényhívás nem futhat egyszerre. Ellentétben, ha egymásba vannak ágyazva a ciklusok és elágazások, akkor egy szinten csak egy exportált függvény van.

Befejeztük a programunk szimulációs elemzését. **Összegezve** azt mondhatjuk, hogy a párhuzamosító módszert leginkább a változók száma, és

az értékadások aránya befolyásolja. Minél több a változó, és minél több az értékadás, annál nagyobb gyorsítást tudunk eredményezni.

Összehasonlítottam a szimulációs eredményeimet Halász Attila szimulációs eredményeivel.[14] Egyik nagy különbség, hogy én összességében tízezres nagyságrendű méréseket végeztem, ő pedig hat példát mutatott be. Halász Attila példáiban 147-375%-os gyorsításokról számolt be, míg a méréseim általában kisebb gyorsítást mutattak. Megnéztem ezt a hat példát, hogy mi okozza ezt a sebességbeli különbséget. Ezekben a példákban nem szerepelnek olyan programok, amelyek a párhuzamosító algoritmus gyengeségeit számításba vennék. Ezekre a programokra az én módszerem is 147-375%-os gyorsítást ad. A szimulációban mindenféle programot számításba veszek, ezért ugyan kisebb gyorsításról számol be, de reálisabb képet ad.

---

[14] Halász Attila munkájában mért gyorsítás eredmények: 375%, 366%, 350%, 175%,147%, 166%. http://compalg.inf.elte.hu/~tony/Oktatas/Diplomamunka-Szakdoli-Nagyprogram/Halasz%20Attila%20-%20Diplomamunka.zip

Összefoglaló

A függőségi programgráf segítségével a soros programunkat szétszedhetjük párhuzamosan végrehajtható egységekre, valamint a gráf programoptimalizálásra is használható.

Első lépésben absztrakt programon mutattuk meg a gráf felépítésének az algoritmusát, majd bebizonyítottuk a gráf által felépített program helyességét.

Második lépésben felépítettünk egy objektum-elvű modellt, amelyben alkalmazható a párhuzamosító algoritmus.

Harmadik lépésben a felépített modellben refaktoráltuk a programot az optimálisabb párhuzamosítás érdekében, majd a modellen elvégeztük a párhuzamosítást.

A függőségi programgráf egy **másik felhasználási módjára** is kitértünk, megmutattuk, hogy a függőségi gráf hogyan alkalmazható programoptimalizálásra.

Összehasonlítást végeztünk Halász Attila módszerével, és megmutattuk, hogy a párhuzamosítás miben más, esetleg több az ő munkájánál.

Tárgyaltuk a készített program **fejlesztői dokumentációját**, majd ismertettük a **szimulációs eredményeket**.

Összegezve megállapíthatjuk, hogy a felépített gráf használható párhuzamosításra, viszont meggondolandó, hogy mennyire érdemes alkalmazni. Ha a hatékonyság növelése a cél, akkor szinte biztos, hogy hatékonyabb megoldást kapunk, ha alapjaiban párhuzamosan gondolkodunk, és több processzorra tervezzük a programot. Akkor érdemes a tárgyalt párhuzamosító módszert használni, ha nincs erőforrásunk alapjaiban újraírni egy meglévő soros programot, de a hatékonyságát szeretnénk növelni, valamint, ha egy új program írásánál nem akarunk párhuzamosan működő módszert kitalálni, hanem hagyományosan sorosan írjuk meg.

Irodalomjegyzék

Az irodalomjegyzékben szereplő aláhúzott szövegrészek a dolgozat elektronikus formátumában linkek.

[1] ***Iványi Antal:*** Párhuzamos algoritmusok, ELTE Eötvös Kiadó, Budapest, 2003, [346], ISBN-963-463-590-3
http://compalg.inf.elte.hu/~tony/Oktatas/Parhuzamos-algoritmusok/Parhuzamos-algoritmusok.pdf
(letöltve:2010.12.18)

[2] ***Iványi Antal:*** Processzorütemezés, ELTE Informatikai Kar, Elektronikus kézirat, Budapest, 2009.
http://compalg.inf.elte.hu/~tony/Oktatas/Utemezesi-algoritmusok/utemezes-2010_marcius_23.pdf
(letöltve: 2010.12.07.)

[3] ***Horváth Zoltán:*** Párhuzamos és elosztott programozás, ELTE Informatikai Kar, Elektronikus kézirat, 2005.
http://www.inf.elte.hu/karunkrol/digitkonyv/Jegyzetek2004/ParhProg.pdf
(letöltve: 2010.12.07.)

[4] ***S. E. Bazhanov, V. P. Kutepov, and D. A. Shestakov:*** Functional Parallel Typified Language and Its Implementation on Clusters, Programming and Computer Software, Vol. 31, No. 5, 2005, pp. 237–269. Translated from Programmirovanie, Vol. 31, No. 5, 2005.
http://www.springerlink.com/content/u22q05g62q251702/fulltext.pdf
(letöltve: 2010. 12. 07.)

[5] ***Andrew S. Tanenbaum - Maarten Van Steen*****:** Elosztott Rendszerek Alapelvek és Paradigmák, Budapest, Panem Kft., 2004, [872], ISBN-978-963-545-387-0

[6] ***Halász Attila Péter:*** Objektumorientált, szekvenciális programok átalakítása párhuzamos környezet számára hatékony implementációval, 2006.
http://compalg.inf.elte.hu/~tony/Oktatas/Diplomamunka-Szakdoli-Nagyprogram/Halasz%20Attila%20-%20Diplomamunka.zip
(letöltve: 2010. 10. 12.)

[7] ***Fóthi Ákos, Horváth Zoltán:*** Bevezetés a programozáshoz, Budapest, ELTE Eötvös Kiadó, 2005, [326], ISBN-9789634638339
http://www.inf.elte.hu/karunkrol/digitkonyv/Jegyzetek2004/BevProg.pdf
(letöltve: 2010.12. 07.)

[8] ***Grady Booch:*** Object Orienteted Analysys And Design, Boston, Addison-Wesley kiadó,1998, ISBN-9780201895513
http://bib.tiera.ru/dvd37/Booch%20G.%20-%20Object-oriented%20analysis%20and%20design%20with%20applications%281993%29%28Second%20Edition%29%28608%29.pdf
(letöltve: 2010. 12. 07.)

[9] ***Láng Csabáné, Gonda János:*** Bevezetés a matematikába II., Budapest, ELTE Eötvös Kiadó, 1995, [240]

[10] Entitás fogalma: http://en.wikipedia.org/wiki/Entity (letöltve: 2010.12.30.)

Csatolmányok

DVD adathordozó, melynek tartalma:

- Diplomamunka docx formátumban
- 3 darab txt kiterjesztésű fájl, amelyek a program által előállított szimulációs eredményeket tartalmazzák
- Szimulációs program