Szekvenciális és párhuzamos algoritmusok

Sequential and Concurrent Algorithms

Dr. KALLÓS Gábor

Széchenyi István Egyetem, Győr
Számítástechnika Tanszék

Abstract

In this paper we present a relatively new research field: the theory of algorithms for concurrent machines. We analyze how classical algorithms and programming methods can be transformed into the concurrent environment, and how the totally new ideas can be applied. Among some interesting exercises which are included, the motivated reader can study further following the references.

Összefoglalás

A cikkben egy érdekes és viszonylag új kutatási területtel, a párhuzamos gépekre kifejlesztett algoritmusokkal foglalkozunk, elsősorban elméleti eredmények bemutatásával.

Először áttekintjük három különböző fejlett programozási módszer (rekurzió, dinamikus programozás, mohó algoritmusok) párhuzamosítási lehetőségeit. Ezután néhány klasszikus matematikai probléma megoldásának felgyorsítását mutatjuk be párhuzamos környezetben, majd a cikk végén röviden foglalkozunk a teljesen új elveken alapuló párhuzamos algoritmusok megalkotásához használható ötletekkel.

Az anyag önálló gyakorlását feladatok segítik. A további tanuláshoz, kutatáshoz a cikkben angol és magyar nyelvű szakirodalmi hivatkozásokat helyeztünk el, itt a bemutatott problémák egy része részletesebben is áttekinthető, ill. további érdekes anyagrészek és feladatok is találhatók.

A szerző e-mailben szívesen válaszol a felmerülő kérdésekre.

Az algoritmusok hatékonyságának mérése

Az algoritmus olyan pontosan definiált számítási eljárás, amely egy meghatározott feladatot elvégezve bemeneti (input) adatokból kimeneti (output) értékeket állít elő. Egy algoritmust a feldolgozandó elemek függvényében a lépésszámmal (és így végeredményben a futási idővel) jellemezhetünk. Ez a költségfüggvény elméletileg minden algoritmus esetében meghatározható.

Az algoritmusok programnyelvtől független megadásához és elemzéséhez elméleti gépmodelleket használunk, amelyek egy processzort és írható-olvasható memóriát tartalmaznak. A processzor lépésenként (szekvenciálisan) végrehajtja az algoritmust, eközben használhatja a memóriát. Az algoritmust a gép számára valamely általános algoritmus-leíró nyelvben adjuk meg. Ilyen modellek például a Turing-gép és a RAM gép (random-access machine).

A Turing-gép a gépi számítások legrégibb és legismertebb modellje. Turing angol matematikus még 1936-ban vezette be. Egy korlátos központi részből és egy végtelen tárból áll. Előnye, hogy bármely számítás elvégezhető rajta, amelyet akár előtte, akár utána bármilyen más gépmodellen el tudtak végezni. Lényeges hátránya ugyanakkor, hogy a távoli memóriablokkokat csak szekvenciálisan lehet elérni. Mivel erősen eltér a valós gépektől, így főleg elméleti vizsgálatokra alkalmas.

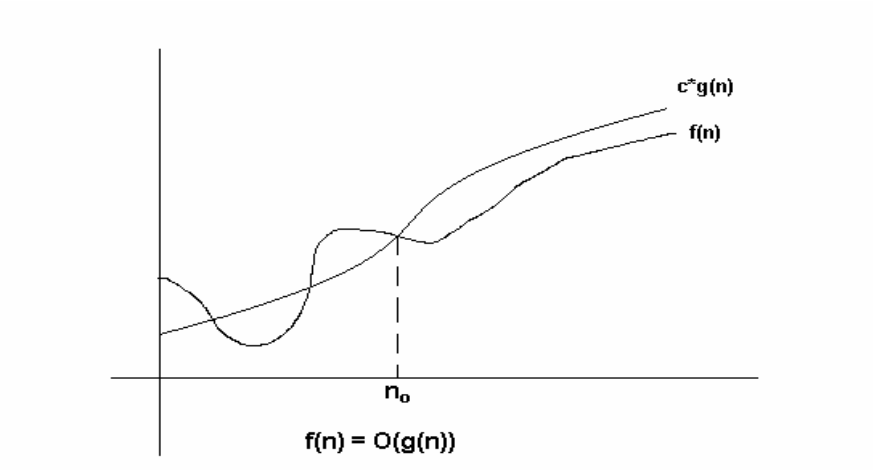
A RAM gép a memória bármely részét egy lépésben el tudja érni, ezért inkább tekinthető a valódi számítógépek leegyszerűsített modelljének, mint a Turing-gép. Ezen gép memóriája szintén korlátlan, és minden memóriarekeszben tetszőlegesen nagy egész szám tárolható.

Ezeknél a modelleknél a műveletek költségét a szükséges memória-hozzáférések száma határozza meg.

Def.: Azt mondjuk, hogy az f(n) függvény eleme az Ordó(g(n)) halmaznak, ha van olyan c konstans és n_0 küszöbérték úgy, hogy valamely $n>n_0$-ra mindig

$$0 \le f(n) \le c \cdot g(n).$$

Sok esetben az „f(n) eleme Ordó(g(n))” helyett az „f(n) = Ordó(g(n))” vagy az „f(n) Ordó(g(n))”-es írásmódot is használjuk. Rövidítés: O(g(n)).



Az Ordó(g(n)) f egy felső korlátja, amely lehet éles, illetve nem éles. A tovább már nem finomítható korlátot éles korlátnak nevezzük.

Például n^2 = Ordó(n^2) éles korlát, n^2 = Ordó(n^3) nem éles felső korlát. A továbbiakban az Ordó jelölést külön említés nélkül éles korlátként használjuk.

Megj. Ez a megkötés egyszerűsítéshez vezet, ugyanis így nem tudjuk megkülönböztetni az éles és a nem éles korlátot. Pontosabb mérést tesz lehetővé a „Théta” jelölés bevezetése (ld. [Co]).

Sok klasszikus probléma, illetve algoritmus esetében már régóta ismert, hogy a megoldások költségfüggvénye milyen Ordó(g(n)) függvényosztályba tartozik. Ugyanazon probléma esetén azokat a megoldásukat tartjuk hatékonynak, amelyek szintén ilyen költségűek.

F: Vizsgáljuk meg, hogy milyen éles korlátot tudunk adni a következő klasszikus algoritmusok műveletigényére (n elemű tömböket használunk). Foglalkozzunk külön a legrosszabb, a legjobb és az átlagos esettel:

Szekvenciális keresés, logaritmikus keresés; buborékos rendezés, kiválasztásos rendezés, beszúrásos rendezés, gyorsrendezés. (3)

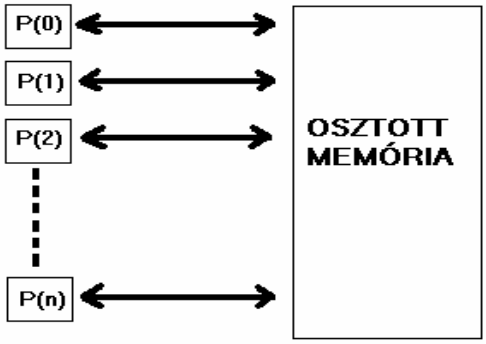
Megj. A feladatok utáni számok az adott feladat nehézségi szintjét adják meg.

Egy párhuzamos gépmodell

Bár már az 1940-es években készült első számítógépek is rendelkeztek bizonyos párhuzamosítási lehetőséggel (pipeline technika), az igazi többprocesszoros számítógépek csak 1-2 évtizede kezdtek elterjedni. Ezzel megnőtt az érdeklődés a párhuzamos algoritmusok iránt is, amelyekben egyidejűleg több művelet is futhat. Ma már sok olyan probléma megoldására is léteznek párhuzamos algoritmusok, amelyeket azelőtt közönséges, szekvenciális algoritmussal oldottunk meg.

A párhuzamos algoritmusok megadására és jellemzésére használt elméleti modell a párhuzamos véletlen hozzáférésű gép (PRAM gép, paralell random-access machine).

A PRAM modellben a processzorok az osztott memóriához csatlakoznak. Az osztott memóriában bármely szót minden processzor egységnyi idő alatt ér el. A PRAM gép processzorai egymással párhuzamosan dolgozni képes RAM gépeknek tekinthetők.



A PRAM modellben az algoritmus hatékonyságát a párhuzamos memória-hozzáférések számával mérjük. Ez a közönséges RAM modellnél használt azon feltételezés közvetlen általánosítása, miszerint a memória-hozzáférések száma (aszimptotikusan) jó közelítést ad a futási időre.

Megj. A valóságban a processzorok számának növekedésével nő az elérésük ideje is.

Egy párhuzamos algoritmus futási ideje az algoritmust végrehajtó processzorok számától is függ. Ezért amikor egy PRAM algoritmust vizsgálunk, a feladat bonyolultsága mellett a processzorok számát is figyelembe kell vennünk, ellentétben a szekvenciális algoritmusokkal. Egy algoritmus futási ideje és az általa használt processzorok száma között általában szoros kapcsolat áll fenn, amennyiben az egyik paraméter csak a másik kárára javítható.

PRAM gépekre írt egyes algoritmusokkal a cikk végén még részletesen foglalkozunk.

Rendezések és rekurzió

A klasszikus rendezések – buborékos, kiválasztásos, beszúrásos – Ordó(n^2)-es algoritmusok, hiszen a megoldáshoz két egymásba ágyazott ciklust használunk fel. Az Ordó-ban „elbújtatott” konstansok eredményezhetnek ugyan jelentős sebesség-különbséget egyes algoritmusok (például a hagyományos és a javított buborékos rendezés) között, de a különbség csak *konstansszoros*.

A gyorsrendezés

Az Ordó(n^2)-es algoritmusoknál ismerünk hatékonyabb rendező eljárásokat is. A legjobb általánosan is használható módszer a gyorsrendezés, amelyet C. A. Hoare mutatott be 1961-ben. Az algoritmus rekurzív, a teljes sorozat rendezését visszavezeti két félsorozat, majd négy negyedsorozat, …, rendezésére. A rendező eljárás meghív egy felosztó rutint, amely egy ún. „könyökelem” kiválasztása után átcserélgeti az eredeti tömb elemeit úgy, hogy a tömb felső részében ne maradjon a könyökelemnél kisebb, alsó részében pedig nála nagyobb elem.

Az algoritmus pszeudokódja:

```
gyorsrendezés(A, p, r)
      ha p < r akkor
            q := feloszt(A, p, r)
            gyorsrendezés(A, p, q)
            gyorsrendezés(A, q+1, r)
      elágazás vége
eljárás vége

feloszt(A, p, r)
      x := A[p]
      i := p - 1
      j := r + 1
      ciklus amíg i < j              # while ciklus, bennmaradási feltétel
            ciklus                   # repeat-until ciklus
                  j := j - 1
            amíg A[j] <= x           # kilépési feltétel
            ciklus
                  i := i + 1
```

```
            amíg A[i] >= x
            ha i < j akkor csere(A[i], A[j])
        ciklus vége
        return(j)
eljárás vége
```

A gyorsrendezés hatékonyságát alapvetően az határozza meg, hogy a felosztó eljárás hogyan vágja szét az „A” tömböt kisebb részekre, illetve, hogy milyen stratégia szerint választjuk ki a „q” könyökelemet.

A legszerencsésebb esetben a felezés után a részsorozatok elemszáma közel megegyezik, és ez a tulajdonság a rekurzió során végig megmarad.

Jelöljük „T(n)”-nel „n” elem rendezésének költségét. Ekkor

$$T(n) = 2 \cdot T(n/2) + \text{Ordó}(n),$$

ahol Ordó(n) a szétválogatás költsége (feloszt eljárás), „T(n/2)” pedig az „n/2” elem rendezésének költsége. A képletet kifejtve

$$T(n) = 2 \cdot T(n/2) + O(n) = 4 \cdot T(n/4) + 2 \cdot O(n) = \ldots = n \cdot T(1) + \lg n \cdot O(n) = O(n \cdot \lg n).$$

Megj. A képlet kiszámolható rekurzívan kifejtve illetve az ún. Mester tétel (ld. [Co]) alkalmazásával. Ugyanez a tétel használható a cikkben található többi rekurzív képlet kiszámítására is. A technikai részletekkel nem foglalkozunk.

A legrosszabb felosztás esetén az egyik félsorozat elemszáma mindig 1. Ekkor

$$T(n) = T(n - 1) + O(n) = T(n - 2) + 2 \cdot O(n) = \ldots = n \cdot O(n) = O(n^2).$$

A gyorsrendezés gyakorlati alkalmazhatóságát erősen korlátozná, ha egy véletlen számtömb esetén sokszor kapnánk a legrosszabb esetet. Szerencsére – mint a következőkben láthatjuk – ez nem így van.

A gyorsrendezés átlagos futási ideje sokkal közelebb áll a legjobb, mint a legrosszabb futási időhöz. Vegyük például azt az esetet, amikor a felosztás mindig 9:1 arányban történik. Ekkor

$$T(n) = T(9n/10) + T(n/10) + O(n) = \ldots = O(n \cdot \lg n)$$

Ugyanezt kapnánk minden állandó arányú felosztáskor is, hiszen a rekurziós fa mélysége ekkor Ordó(lgn), és a költség minden szinten Ordó(n). A futási idő tehát Ordó(n·lgn) bármilyen állandó arányú felosztáskor. Ezeket a felosztásokat ezért kiegyensúlyozott felosztásoknak is nevezzük. A gyakorlati tapasztalatok azt mutatják, hogy általános tömb esetében a rossz felosztások viszonylag ritkán fordulnak elő, általában valamilyen kiegyensúlyozott felosztást kapunk. További igazolás nélkül megemlítjük, hogy a gyorsrendezés futási teljesítménye „n” elem összes permutációit tekintve csak néhány esetben éri el vagy közelíti meg a legrosszabb esetet (részletes igazolás: [Co]).

Tovább javítható a rendezés várható hatékonysága a könyökelem véletlen kiválasztásával, illetve az elemek véletlen átrendezésével az algoritmus alkalmazása előtt.

Összefoglalva a fenti elemzést:

A gyorsrendezés futási ideje átlagos esetben (gyakorlatilag szinte mindig) Ordó(n·lgn).

F: Elemezzük, hogyan viselkedik a fenti algoritmus teljesen rendezett és fordítva rendezett tömb esetén. (2)

Rekurzív algoritmusok párhuzamosítása

A klasszikus rendező algoritmusok nem párhuzamosíthatók kézenfekvően, a gyorsrendezés esetén azonban több processzor használatával rövidebb futási időt érhetünk el az egymástól független lépések egyidejű végrehajtásával. Két processzorral rendezhetjük például az első rekurziós lépés után kapott két félsorozatot, majd négy processzorral a második lépés utáni négy negyedsorozatot, és így tovább. Végeredményben, ha van „n” illetve „n/2” darab processzorunk, akkor a teljes időigény csak

$$O(n) + O(n/2) + O(n/4) + \ldots + O(1) = O(n),$$

figyelemben tartva azt, hogy a teljes műveletigény továbbra is T(n) = O(n·lgn). Természetesen ez az eredmény elméleti jellegű, mert nem foglalkoztunk azzal, hogy mily módon lehet a részfeladatokat az egyes pro-

cesszoroknak kiosztani, és az eredményt tőlük begyűjteni. Ez a gyakorlatban a teljes végrehajtás lelassulását eredményezi.

Hasonló párhuzamosítási lehetőség adódik bármely olyan rekurzív algoritmusnál, ahol egymástól független részfeladatokat kell végrehajtani.

Dinamikus programozás

A dinamikus programozás a rekurzióhoz hasonlóan az eredeti feladatot részproblémákra osztással oldja meg, de a részproblémák most nem (teljesen) függetlenek egymástól, sőt egyes részproblémák megoldása ismétlődően előfordulhat. Emiatt a már megoldott részfeladatok eredményét táblázatban tároljuk, és amennyiben újból szükség van rájuk, visszakeressük a megoldást. Erről a táblázatos megadásról nevezték el magát a módszert is. A dinamikus programozást optimalizálási problémák megoldásához használjuk, ilyenkor a feladatra található egy vagy több optimális megoldás, és nekünk egyet elő kell állítani, illetve az értékét meg kell határozni. Az előállítás lépései a következők:

1. Jellemezzük az optimális megoldás szerkezetét.
2. Rekurzívan definiáljuk az optimális megoldás értékét.
3. Kiszámítjuk az értéket alulról felfelé.
4. Megszerkesztünk egy optimális megoldást (amennyiben ez is szükséges).

A rekurzióhoz hasonlóan a dinamikus programozással megoldható feladatok esetében is gyorsulást eredményezhet a párhuzamos megoldás. Nagyon lényeges azonban a processzorok közötti jó kommunikáció, hiszen szükséges, hogy a már kiszámított részeredmények a többi processzor számára is hozzáférhetőek legyenek.

Mohó algoritmusok

Hasonlóan a dinamikus programozáshoz, a mohó stratégiát is optimalizálási problémák megoldásához használjuk – van több optimális megoldás, egyet előállítunk. A mohó stratégia kevesebb lépéssel dolgozik, mint a dinamikus programozás. Mindig az adott lépésben optimálisnak látszót választja , feltételezve, hogy a lokális optimum globális optimumhoz vezet. Mivel ez a feltételezés sok problémára nem teljesül, ezért a mohó stratégia nem alkalmazható minden esetben.

Azok a problémák oldhatóak meg a mohó stratégiával, amelyekre teljesül a következő két feltétel:

a) mohó választási tulajdonság (lokális optimumot választunk, az aktuális választás függhet a korábbi, de nem függhet a későbbi választásoktól);
b) optimális részproblémák tulajdonság (optimális megoldás építhető a részproblémák optimális megoldásából).

Klasszikus mohó stratégiával megoldható feladat például a pénzkifizetési probléma „normál” pénzrendszer esetén, illetve a Huffman-kód előállítása karaktersorozat optimális kódolására.

A mohó stratégia szekvenciális jellege miatt nem párhuzamosítható, ezért nem foglalkozunk vele részletesebben.

Megj: A rekurzió, a dinamikus programozás és a mohó algoritmusok részletes bemutatása (szekvenciális környezetben) megtalálható a [Co] könyvben. Ugyanitt kidolgozott példákat is találhatunk, feladatokkal együtt.

Párhuzamos technikák, párhuzamos algoritmusok

A párhuzamos számítások lehetősége az algoritmusok szervezésében sokkal nagyobb szabadságot biztosít, mint a szekvenciális megközelítés. Ezt az alábbi hasonlattal szemléltethetjük:

- a szekvenciális algoritmus egy dimenziós konstrukció, euklideszi térben;
- a párhuzamos algoritmus több (változó) dimenziós konstrukció, változó térben.

Megj. A második pont arra utal, hogy a párhuzamos program lehetséges végrehajtásait egy bonyolult nagy fával szemléltethetjük, amelynek egyes ágait a szinkronizáció miatt levágjuk, és a struktúra annál bonyolultabb, minél több processzor áll a rendelkezésünkre. Ráadásul a környezet alapvetően nemdeterminisztikus (részletesen ld. [Bu]).

Jelenleg még főleg szekvenciális gépeken dolgozunk, ezért a párhuzamos algoritmusok főként a szekvenciális algoritmusokban amúgy is meglevő párhuzamosítási lehetőségeket használják. A párhuzamosítás alapvetően négyféle módon képzelhető el:

a) triviális párhuzamosítási lehetőségek (nem feltétlenül triviális klasszikus algoritmusokban, pl. Gauss elimináció);
b) klasszikus algoritmusok alkalmas részeinek vagy egészének párhuzamos végrehajtással való felgyorsítása (pl. mátrix-vektor szorzás, aritmetikai kifejezések párhuzamos kiértékelése);
c) új elemek beépítése klasszikus algoritmusokba egyes részfeladatok párhuzamos megoldására (pl. aritmetikai kifejezések párhuzamos kiértékelése);
d) teljesen új elveken alapuló párhuzamos algoritmusok kifejlesztése.

Megj. Teljesen új elveken alapuló párhuzamos algoritmusokat egyszerűbb esetektől eltekintve nehéz kifejleszteni, és a közeljövőben nem várható, hogy ilyenek tömegesen megjelennek. De előfordulhat, hogy néhány évtized múlva ez lesz a számítástechnika fő fejlődési területe (további részletek: [Mo]).

Az előző alfejezetekben megnéztük, hogy milyen párhuzamosítási lehetőségek vannak a rekurzív illetve a dinamikus programozással megoldható algoritmusokban. Most néhány fontos matematikai probléma megoldásának párhuzamos felgyorsítását tekintjük át, a fenti a)-c) pontok módszerei szerint. A cikk végén bemutatunk néhány d) ponthoz tartozó egyszerűbb példát is.

A Gauss-elimináció

A Gauss-elimináció a lineáris egyenletrendszerek megoldására használt klasszikus módszerek egyike. Általános esetben kezdetben adott „n" darab ismeretlen és „m" darab egyenlet a következő elrendezésben:

$$\begin{aligned} a_{11}\cdot x_1 + a_{12}\cdot x_2 + \ldots + a_{1n}\cdot x_n &= b_1 \\ a_{21}\cdot x_1 + a_{22}\cdot x_2 + \ldots + a_{2n}\cdot x_n &= b_2 \\ a_{31}\cdot x_1 + a_{32}\cdot x_2 + \ldots + a_{3n}\cdot x_n &= b_3 \\ \vdots \qquad\quad \vdots \qquad\quad & \vdots \\ a_{m1}\cdot x_1 + a_{m2}\cdot x_2 + \ldots + a_{mn}\cdot x_n &= b_m \end{aligned}$$

A megoldás során az ismeretlenek fokozatos eliminálásával a teljes együtthatómátrixból felső háromszög mátrixot hozunk létre, amelyből aztán a megoldások leolvashatók, illetve visszahelyettesítéssel meghatározhatók. Az elimináció során megengedett műveletek a következők:

- valamely sort egy nem nulla konstanssal megszorozhatunk;
- valamely sorból kivonhatjuk egy másik sor konstansszorosát;
- (ha szükséges) átjelölhetjük a még nem eliminált ismeretleneket.

1. lépés: a_{i1}-et „lenullázzuk" $i < 1$-re, azaz az i-edik sorból kivonjuk az első sor a_{i1}/a_{11}-szeresét.

Ezzel az első sort követő minden sorban az első oszlop lenullázódik. A következő lépés során a második sort követő minden sor második oszlopa is lenullázódik, stb. Az elimináció végén „jó esetben" az első sorban „n" tag összege fog szerepelni, a másodikban „n-1", …, az utolsóban pedig csak egy ismeretlen lesz. A „jó eset" akkor fordul elő, ha a rendszeren belül ugyanannyi független egyenlet található, mint ahány ismeretlen. Egyéb esetben lehet, hogy nem kapunk megoldást (ellentmondásos egyenletrendszer), illetve végtelen megoldás is lehet (egy vagy több ismeretlen értéke szabadon megválasztható). Ezen esetek pontos meghatározásával most nem foglalkozunk, ez bármely egyetemi/főiskolai lineáris algebrai kurzus anyagában szerepel.

A lépésszám és a műveletigény négyzetes mátrixra hagyományosan Ordó(n^3). Ha van n darab processzorunk, és ki tudjuk köztük osztani a részfeladatokat úgy, hogy párhuzamosan dolgozzanak (például 1-1 sort egymástól függetlenül számoljanak végig), akkor a lépésszám Ordó(n^2) lesz (ugyanakkor a műveletigény továbbra is Ordó(n^3) marad). Tovább gyorsítható az eljárás n^2 processzorral, ekkor minden mátrixelemhez rendelünk egy processzort, és az azzal dolgozik. Így a lépésszám Ordó(n) lesz. Megjegyzendő, hogy a gyakorlatban a processzorok közötti kommunikációra fordított idő a fenti elméleti határokhoz képest jelentősen lelassítaná a feladat megoldását.

F: Gyorsítható-e a végrehajtási idő újabb processzorok rendszerbe állításával? (1,5)

Mo: A probléma részeredményei tovább már nem függetleníthetők egymástól, tehát ha ennél több processzort használunk, akkor már nem érhető el sebességnövekedés.

Megj. Néhány fontos gyakorlati problémával nem foglalkoztunk, amelyek szintén komoly lassulást okozhatnak a megoldás során. Ilyen például az eliminációra sajátosan jellemező ún. együttható-növekedési probléma: az egymás utáni lépésekben egyre több jegyű számokkal (gyakran egyre hosszabb törtekkel) kell

dolgozni. Ennek részbeni kivédésére javasolhatnánk a közelítő számításokat, de ez sem oldja meg a gondot: eleve elveszítjük a pontos megoldás lehetőségét, ráadásul a közelítések hibái a lépések során összegződnek, így rossz esetben nem is kapunk megoldást a számolás végén! Csak ügyes egyszerűsítések alkalmazásával háríthatók el a fenti problémák általános esetben (részletek: [Ge]).

Polinom-faktorizáció

A polinomok szorzattá alakítása általános esetben bonyolult probléma.

Példa: Bontsuk fel $A(x, y) = y^2 + 2xy - 3y + x^2 - 3x - 4$-et

Erre a feladatra első közelítésben nem látszik használható megoldási módszer.

Az alfejezetben feltételezzük, hogy az olvasó tisztában van a többváltozós polinomokkal kapcsolatos fontosabb fogalmakkal. A szakszerű megalapozáshoz a következő ismeretek, definíciók szükségesek (csak felsorolásszerűen bemutatva):

Z: egész számok gyűrűje (+-ra és *-ra);
Q: racionális számok teste (+-ra és *-ra);
Z[x]: polinomgyűrű;
Z[x, y]: értelmezhető mint Z[x][y] vagy Z[y][x] polinomgyűrű;
Z[x, y, z] hasonlóan, rekurzív módon értelmezhető;
$Z_p[x] = Z[x]$ mod p test, ha p prím.
A fenti tartományokban van egyértelmű faktorizáció és egyértelműen létezik a legnagyobb közös osztó is.

A következő ötletet használjuk:

a) Az eredeti problémát redukáljuk egy sokkal egyszerűbben kezelhető tartományba, és ott oldjuk meg. Ez az egyszerűsítés lehet például a következő:

$$Z[x, y, z, ..., w] \rightarrow Z[x] \rightarrow Z_p[x].$$

b) Ezután megoldjuk a problémát $Z_p[x]$-ben, ahol egy polinom már viszonylag egyszerűen szorzattá alakítható.

c) Utolsó lépésben a kapott megoldást visszaképezzük az eredeti tartományba:

$$Z_p[x] \rightarrow Z[x] \rightarrow Z[x, y, z, ..., w]$$

Példa folyt. $A(x, y)$ Z[x]-ben $x^2 - 3x - 4$ lesz, $Z_3[x]$-ben pedig $x^2 – 1$.
Ez a polinom már egyszerűen szorzattá alakítható. A felbontások:

$Z_3[x]$-ben $(x + 1)(x - 1)$;

Z[x]-ben pedig $(x + 1)(x - 4)$.

Ez a részeredmény már csak abban tér el az eredeti felbontástól, hogy az y-os tagok hiányoznak. Ez visszaképezéssel megkapható. A teljes megoldás Z[x, y]-ban:

$$A(x, y) = (x + 1 + y)(x - 4 + y).$$

A megoldandó részfeladatok elemzése

a) Egyszerűsítés. Ez könnyen végrehajtható rész, adott esetben párhuzamosítható is.

b) A probléma megoldása $Z_p[x]$-ben vagy Z[x]-ben.

Bár a polinom szorzattá alakítása nyilván sokkal egyszerűbb probléma valamely redukált tartományban, intuitív módon (mint a fenti egyszerű példában) általában még most sem lehet célhoz érni. A szorzattá alakító algoritmusok túlnyomó többsége a legnagyobb közös osztó (lnko) meghatározásán alapul. Egy lehetséges ötlet a következő:

lnko(p, p') az eredeti polinom egy valódi osztója, ahol p' a derivált polinom.

Példa: Legyen $p = A \cdot B^2 \cdot C^3$, ahol A, B és C tovább már nem bontható faktorok.
ekkor $p' = A' \cdot B^2 \cdot C^3 + A \cdot 2B \cdot B' \cdot C^3 + A \cdot B^2 \cdot 3C^2 \cdot C'$

(de például $Z_3[x]$-ben csak $p' = A' \cdot B^2 \cdot C^3 + A \cdot 2B \cdot B' \cdot C^3$, ez egyszerűbb)
$lnko(p, p') = B \cdot C^2$, ez egy valódi osztó

F: Mikor nem használható ez az ötlet? (2)

Más felbontó eljárások is ismeretesek. Bonyolult (magas fokú) polinom esetén célszerű lehet (egyszerre) több eljárást is kipróbálni, hogy hamarabb célhoz érjünk. Mivel lényegében az összes felbontó eljárás használ lnko számításokat, ezért szükségünk van ezek hatékony kivitelezésére. Erre az euklideszi algoritmust használjuk, amely $Z_p[x]$-ben és $Z[x]$-ben egyaránt végrehajtható.

Az euklideszi algoritmus a maradékos osztás ismételt alkalmazásával határozza meg lnko(a, b)-t.

A maradékos osztás adott a, b elemekre a következő módon hajtható végre:

$a = b \cdot q + r$, ahol $|r| < |b|$ vagy $r = 0$ (1) (polinomokra a |...| norma a polinom foka)

áll: $lnko(a, b) = lnko(b, r)$ (ennek igazolása nyilvánvaló)

Ezután választhatjuk: új a := b; új b := r; majd újra végrehajtjuk az (1) lépést. Az utolsó nem 0 maradék az eredeti számok lnko-ja.

c) A megoldás visszaképezése az eredeti többváltozós polinomgyűrűbe.

Megjegyezzük, hogy ez a legnehezebb lépés, itt csak a legfontosabb ötleteket tekintjük át.

Nyilvánvaló, hogy egy $Z_p[x]$ vagy $Z[x]$-beli kép sokkal kevesebb információt hordoz, mint ami az eredeti felbontás előállításához kell, ezért további adatokra van szükségünk a feladat megoldásához.

Két megközelítés használható:

(i) sok $Z_p[x]$-beli képet állítunk elő (több különböző p-re), vagy több $Z[x]$-belit;

(ii) csak egy képet használunk, de van még valami plusz információnk.

(i) eset

1. Példa: Tudjuk a Z[x, y]-beli u-ról hogy

$u(x, 0) = - 9x – 21$,

$u(x, 1) = - 3x + 20$,

$u(x, 2) = 5x – 36$.

Ez alapján polinom interpolációval

$u(x, y) = (- 9x - 21) + (6x + 41)(y - 0) + x(y - 0)(y - 1) = xy^2 + 5xy + 41y - 9x - 21$

2. Példa: Tudjuk valamely u számról, hogy

$u = 2 \pmod 5$ és $u = 3 \pmod 7$.

Mennyi u valójában?

Mo: nyilván

$5x + 2$ alakú (lehet: 2, 7, 12, 17, 22, 27, 32, ...) és

$7x + 3$ alakú (lehet: 3, 10, 17, 24, 31, 38, 45, 52, ...)

egyszerre, így lehet 17, 52, ..., de mod 35 egyértelmű a megoldás.

A példák alapján láthatjuk, és általánosan is igazolható, hogy

$Z[x, y] \rightarrow Z[x]$ invertálható polinom interpolációval (y foka + 1 kép kell);

$Z[x] \rightarrow Z_p[x]$ invertálható kínai maradék algoritmussal.

Gondot jelent azonban, hogy összességében nagyon sok kép kell (végeredményben exponenciális), és nem jól párhuzamosítható a folyamat.

(ii) eset

A szükséges plusz információ lehet például a következő:

$F(v, u) = a(x, y, w, ..., z) - v \cdot u$; (2) (két faktorra vágtuk a-t)

és

$u(x, y, w, ..., z) = u_0(x)$; (3) ($Z_p$ [x]-ben)

$v(x, y, w, ..., z) = v_0(x)$. (4) ($Z_p$ [x]-ben)

Ezután $u_0(x)$-et és $v_0(x)$-et felemeljük fokozatosan

Z[x]-be, Z[x, y]-ba, Z[x, y, w]-be, ..., Z[x, y, w, ..., z]-be

úgy, hogy (2)-(4) érvényes maradjon.

A faktorok felemelése egymástól lényegében független (egy szinten belül), így a módszer jól párhuzamosítható. Még többet nyerhetünk egy teljes felbontással:

$F(u_1, u_2, ..., u_n) = a(x, y, w, ..., z) - u_1 \cdot u_2 \cdot ... \cdot u_n$ (5)

– ahol (2)-(4) megfelelői érvényesek – majd ennek a felemelésével. Az (i) és az (ii) esetekben használható részletes algoritmusok (pontos elméleti igazolással együtt) megtalálhatók a [Ge] könyvben.

F: Írjunk olyan programot, amely szorzattá alakít általános többváltozós polinomot (reális fokkorláttal). Ajánlott irodalom: [Ge].

Összefoglalva, a polinomok szorzattá alakítása bonyolult, komoly számításigényű probléma. Mivel ezen számítások jelentős része párhuzamosan is végrehajtható, többprocesszoros gépen a végrehajtás jelentős felgyorsulását érhetjük el.

PRAM algoritmusok

Ebben az alfejezetben néhány egyszerűbb párhuzamos algoritmust mutatunk be, amelyeket PRAM gépekre dolgoztak ki. Az alapvető számítástechnikai algoritmusok közül sok tömbökön, listákon, fákon és gráfokon dolgozó szekvenciális algoritmus felgyorsítható a PRAM modell segítségével.

A PRAM modellben a lehetséges algoritmus-típusok a következő módon csoportosíthatók:
(i) EREW: kizárásos olvasás és írás (exclusive read and exc. write)
(ii) CREW: egyidejű olvasás és kizárásos írás (concurrent read and exc. write)
(iii) ERCW: kizárásos olvasás és egyidejű írás (exclusive read and conc. write)
(iv) CRCW: egyidejű olvasás és írás (concurrent read and conc. write)
változat: P-CRCW ill. P-ERCW (prioritásos modell).

A prioritásos modellben egyidejű írás esetén valamilyen stratégia szerint ki kell választani a „győztest” (vagy győzteseket), aki(k)nek az akarata érvényesül. Erre a következő módszereket használhatjuk:

- a legkisebb sorszámú dönt;
- a közös akarat érvényesül (csak akkor van eredmény, ha ugyanazt írják);
- véletlenszerűen választunk;
- adott képlet alapján döntünk (pl. összeadjuk az eredményeket vagy azok átlagát, maximumát vesszük).

A modellek közül az EREW és CRCW a leggyakoribb.

Példa: Határozzuk meg n darab processzorral, hogy két n hosszú 0-1 sorozat közül melyik a nagyobb!

P-CRCW modellben elég O(1) lépés:

A processzorok egyidejűleg sorszámot írnak, hogy melyik sorozat a nagyobb, a legkisebb helyen levő eltérés dönt.

F: EREW modellben elég O(log n) lépés.

A példában láthattuk, hogy a modellek közötti időkülönbség nem „túl nagy”. Ez általánosan is igazolható:

Áll. Ha egy probléma megoldható P-CRCW modellben p processzorral t idő alatt, akkor megoldható EREW modellben $O(p^2)$ processzorral $O(t \cdot \log p^2)$ idő alatt.

Határozzuk meg egy lista elemeinek a lista végétől való távolságát!

Nem párhuzamos megoldás esetén O(n) lépés szükséges, minden elem a következőtől megkapja a partner távolságadatát, a sajátja ennél eggyel nagyobb lesz.

A párhuzamos megoldásnál n darab processzort használunk, minden elemhez pontosan egyet rendelünk. Így EREW modellben egy Ordó(log n)-es megoldást állíthatunk elő (részletek és további példák: [Co]).

Irodalom

[Bu] ALAN BURNS, GEOFF DAVIES, *Concurrent Programming,* Addison-Wesley Publishing Company, 1994.

[Co] THOMAS H. CORMEN, CHARLES E. LEISERSON, RONALD L. RIVEST, *Algoritmusok,* Műszaki Könyvkiadó, Budapest, 1998.

[Ge] KEITH O. GEDDES, STEPHEN R. CZAPOR, GEORGE LABAHN, *Algorithms for Computer Algebra,* Kluwer Academic Publishers, 1991.

[Mo] JAGDISH MODI, *Paralell Algorithms,* Oxford University Press, 1986.