Mohácsi László

Számítástudományi Tanszék

Témavezetők:

Dr. Abaffy József, DSc

Dr. Kovács Erzsébet, CSc

Budapesti Corvinus Egyetem
Gazdaságinformatika Doktori Iskola

Gazdasági alkalmazások párhuzamos architektúrákon

doktori értekezés

Mohácsi László
Budapest, 2014.

Nyilatkozat

Alulírott Mohácsi László doktorjelölt kijelentem, hogy a Budapesti Corvinus Egyetemhez 2014. évben benyújtott Gazdasági alkalmazások párhuzamos architektúrákon című doktori értekezésem önálló szellemi alkotásom. Az értekezést korábban más intézményhez nem nyújtottam be, és azt nem utasították el.

Budapest, 2014. augusztus 25.

Tartalomjegyzék

Ábrák jegyzéke

Táblázatok jegyzéke

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőim, Dr. Abaffy József és dr. Kovács Erzsébet tanácsaiért és emberi támogatásukért. Külön köszönet Dr. Deák Istvánnak, aki nélkül a harmadik fejezet nem születhetett volna meg. Hálával tartozom Csicsman Józsefnek, aki hatalmas szakmai tapasztalatával segítette a mikroszimulációs kutatásokat. A munka során jelentkező rengeteg szakmai kérdésben segítségemre volt még Fekete Ádám, Forgács Attila és Kovács Tibor.

Bevezető

Informatikusként, kutatóként és oktatóként gyakran szembesülök azzal a kérdéssel, hogy lépést tudunk-e tartani a tudományterület gyors változásával. A tapasztalat azt mutatja, hogy bár az informatikát a leggyorsabban fejlődő területek közé szokás sorolni, a technológiai újdonságok mögött álló elvek és paradigmák meglepően időtállónak bizonyulnak. Az SQL adatbázisok hátteréül szolgáló relációs modellt 1970-ben írta le Edgar Frank Ted Codd. A C nyelv is negyven éves, a Java viszonylag fiatal, csak most töltötte be a 18-at. Az objektum-orientált paradigma bár széles körben csak 20 éve terjedt el, az alapelv már ott volt a 70-es években a Smalltalk-ban. A mostanában egyre inkább tért nyerő funkcionális programozás matematikai háttere, a lambda-kalkulus már 1930-ban megszületett, az aktor modell is készen volt már a 70-es években. A technológiai fejlődés legtöbbször gyakorlati problémára ad választ, így a területen dolgozó szakemberek gyorsan adaptálják. Az alapelvekkel és a paradigmákkal ellentétben a nyelvek és a fejlesztőeszközök nagyon gyorsan fejlődnek.

Úgy tapasztaltam, hogy az informatikával foglakozó szakemberek tudományos érdeklődése mostanában a dolgok mennyiségéből fakadó problémák felé fordul, ebből adódóan szervezési jellegű. Nagy mennyiségű adatot kell feldolgozni a döntések meghozatalához, nagy felhasználótömeget kell kiszolgálni, a szimulációkat nagy egyedszámon szükséges futtatni a pontosabb eredményekért. Ezzel szemben az egyes számítóegységek sebességét tekintve a technológiai fejlődés megtorpanni látszik. Az alapkutatások terén sem körvonalazódik olyan eredmény, amely az elkövetkezendő években nagyságrendi növekedést hozhatna az általános célú számítóegységek sebességében. (Az előrelépés útjában fizikai korlátok állnak.) Nagy számításigényű feladatok úgy oldhatók meg hatékonyan általános célú hardveren, ha a feladatokat fel lehet bontani, és több számítóegységen párhuzamosan futtatni.

A probléma nem új keletű – mindig voltak olyan tudományos és számítási feladatok, amelyek meghaladták az ember, vagy az éppen rendelkezésre álló gép kapacitását. Ilyenkor kézenfekvő gondolat a feladatot átszervezni és felbontani. Például az Egyesült Államok hadseregében a lövegröppályák meghatározásához elengedhetetlen

szinusz-táblázatokat egy nőkből álló század kézzel, papíron számolta. (A táblázat értékeit Taylor-sorok alapján összeadásokra és szorzásokra vezették vissza.) Érdekes megjegyezni, hogy szinusz-táblázat már a VI. században is készült Indiában.

Véleményem szerint az informatika fejlődésének vannak minőségi és mennyiségi szakaszai – most egy mennyiségi szakaszba léptünk. A piaci nyomás arra ösztönzi a hardvergyártókat, hogy több számítóegységgel rendelkező architektúrákkal jelenjenek meg a piacon, míg az egyes számítóegységek sebességében nem tapasztalható jelentős előrelépés. A különböző architektúráknál a hardverelemek közti kapcsolat jelentősen eltér, így minden architektúra más-más feladattípusoknál nyújt jó teljesítményt. Az algoritmusokat úgy kell fel- illetve átépíteni, hogy illeszkedjenek a futtató architektúrához.

Az értekezés három, gazdasági számításoknál és szimulációknál is jelentős algoritmus párhuzamos architektúrára történő újszerű alkalmazásával foglalkozik.

A dolgozat első része leíró, elemző jellegű – a különböző általános célú párhuzamos adatfeldolgozásra alkalmas hardverarchitektúrákról és kapcsolódó szoftverfejlesztő eszközökről nyújt összehasonlító áttekintést. Az itt leírtak alapjául szolgálnak számos későbbi részekben meghozott döntésnek. A fejezet kivonatából készült cikk az NJSzT gondozásában megjelenő *GIKOF Journal*-ban került publikálásra. A témában 2013. novemberében a X. Országos Gazdaság-informatikai Konferencián tartottam előadást.

A második rész az ABS lineáris egyenletrendszer-megoldó módszer masszívan párhuzamos architektúrára történő implementációjával és az algoritmus hibaterjedésével foglalkozik. A téma azért aktuális, mert az algoritmus kitűnő stabilitási tulajdonságai 2013-ban kerültek bizonyításra. A problémával mint egy előtanulmánnyal foglalkoztam a masszívan párhuzamos architektúrákra történő algoritmustervezés és implementáció mélyebb megértéséhez.

A harmadik rész alapjául Deák Istvánnal közösen írt *"A parallel implementation of an $O^*(n^4)$ volume algorithm"* című angol nyelvű cikkünk szolgál, mely a *Central European Journal of Operations Research*-be került leadásra. A dolgozatban helyt kaptak azok a magyarázatok is, amelyek a cikkbe terjedelmi okok miatt nem kerülhettek bele. Az elkészült párhuzamos implementáció segítségével az algoritmus viselkedésének tanulmányozására új mélységekben nyílt lehetőség, melyre korábban sebességkorlátok miatt nem volt mód. Az eredményeket 2013. június 13-án adtam elő Balatonőszödön a XXX. Magyar Operációkutatási Konferencián.

A negyedik rész témájául a nyugdíjrendszer működtetésékez szükséges modellezéseknél alkalmazásra kerülő előreszámítások egyikét, a demográfiai előreszámításokat

választottam. A demográfiai előreszámítások kapcsán kétféle megközelítéssel foglalkoztam – a kohorsz-komponens módszerrel és a mikroszimulációs eljárással. A mikroszimulációs megközelítést mutatom be közelebbről, mivel a nyugdíj előszámításokhoz nem elég a makro szintű megközelítés. A modellezésnél igen fontos a nyugdíjasok és a nyugdíjba vonulók száma mellett azok neme, iskolai végzettsége, a nyugdíjazáskor elért jövedelme, stb. Bemutatom az általam épített mikroszimulációs keretrendszert, működésének szemléltetéséhez a születés és halál előrejelzését dolgoztam ki részletesen. A feldolgozandó rekordok nagy száma és a minden rekordon azonos feladatokat végrehajtó algoritmusok miatt programozás-technikai szempontból a mikroszimuláció jól párhuzamosítható.

A számítási eredményeket tartalmazó táblázatok és ábrák függelékekben kaptak helyet.

A dolgozat részét képezi az eredmények alapjául szolgáló általam írt nagyságrendileg 5000 sornyi forráskód, ami nyomtatásban kb. 100 oldalt tenne ki. A kód letölthető a http://web.uni-corvinus.hu/~lmohacs/thesis/ címről. A disszertációt 30 saját szerkesztésű ábra teszi szemléletesebbé.

1. fejezet

Gazdasági számítások párhuzamos architektúrákon

Az utóbbi években az egyes számítóegységek műveleti sebességében nem tapasztalható gyors fejlődés. Az egy processzorra tervezett programok és algoritmusok futási idejének nagyságrendnyi javulása nem is várható a hardvereszközök fejlődésétől, mert ezek nem tudják kihasználni a rendelkezésre álló további számítóegységeket. Nagyságrendi sebességnövekedés csak a megoldandó feladat részfeladatokra történő bontásával érhető el, amelyek megoldása külön számítóegységeken, időben párhuzamosan végezhető. Attól függően, hogy a részfeladatok hogyan kapcsolódnak egymáshoz, más-más párhuzamos hardver architektúra nyújt optimális teljesítményt. A több számítási egységből álló architektúrákra történő szoftverfejlesztés egészen más megközelítést igényel, mint a hagyományos, egyprocesszoros architektúrákra történő algoritmustervezés illetve programírás. Disszertációmnak ez a része a legelterjedtebb párhuzamos architektúrák bemutatása után néhány nagy számításigényű gazdasági problémán keresztül szemlélteti az architektúrák közti különbségeket. A témát a gyakorlati problémamegoldás aspektusából közelítem, a tömeggyártásban elérhető általános célú hardvereket alapul véve. Ez a rész több éves fejlesztői és kutatási tapasztalatot összegez annak eldöntéséhez, hogy egy adott gazdasági számítás elvégzéséhez melyik a legmegfelelőbb párhuzamosítási megközelítés.

1.1. A sebességnövekedés korlátai

A dolgozatban a processzor sebességének fogalmát elvont értelemben használom. A ma elterjedt processzorokra már nem igaz az, hogy négy órajel ütemenként hajtanak végre egy gépi műveletet. Egy művelet végrehajtásához szükséges idő függhet a mű-

velet komplexitásától és a műveletek sorrendjétől is. (lásd: 1.3.1) Az órajel frekvencia fontos katalógusadat, de ugyanúgy nem árul el mindent a processzor számítóteljesítményéről, mint ahogy az autó hengerűrtartalma a jármű gyorsulásáról. Egy 64 bites processzor például 64 biten ábrázolt, 19 jegyű egész számokat is össze tud egy lépésben adni, de erre sok alkalmazásban aránylag ritkán van szükség. A dolgozat második és harmadik fejezetében szereplő alkalmazásokban jelentős sebességnövekedést hoz a 64 bites architektúra. A különböző processzorok számítási teljesítményének mérésére és összehasonlítására többféle teszt és mérőszám létezik, de ezek bemutatása nem a dolgozat célja. A processzorok további sebességnövekedését fizikai jellegű tényezők korlátozzák. Az egyik fő probléma a processzorok működése közben keletkező hő, melyet el kell vezetni az alkatrészről. A túlmelegedés a félvezető meghibásodásához vezetne. A processzor gyakorlatilag a működéséhez szükséges teljes felvett elektromos teljesítményt hő formájában adja át környezetének. A hő formájában disszipálódó energia két részből adódik össze. A különböző áramszivárgások következményeként fellép egy állandó, hő formájában keletkező veszteség, mely nem függ a processzor terhelésétől. A veszteség másik része viszont terhelésfüggő: a tranzisztorok átkapcsolásai során szabadul fel. A felhasznált és hővé alakuló teljesítménynek ez a része attól függ, hogy a működés során hány tranzisztor-átkapcsolás történik. A sebességnövelés egyik kézenfekvő útja a processzor órajelének – és ezen keresztül a tranzisztorok kapcsolási frekvenciájának növelése. Így növelhető az egységnyi idő alatt végrehajtható műveletek száma. A magasabb kapcsolási frekvencián üzemeltetett tranzisztorok megbízható működéséhez meg kell növelni a tranzisztor üzemi feszültségét. A feszültségnöveléssel viszont megnő az egy tranzisztorkapcsolásra jutó disszipált hőmennyiség. Ráadásul az üzemi feszültség és a disszipált hőmennyiség közti összefüggés négyzetes elemet is tartalmaz. Fizikai oldalról közelítve a problémát - energia változatlan processzort feltétezve - ez azt jelenti, hogy egy adott program végrehajtása során felszabaduló hőmennyiség függ az órajel frekvenciától, azaz a futtatáshoz szükséges időtől. (De Vogeleer et al. 2014) Az akkumulátorról működő mobil eszközök esetében különösen fontos a processzorok által felhasznált energia. Éppen ezért a mobil eszközökbe szánt processzorok egy része a terhelés függvényében automatikusan képes szabályozni egyes egységeinek órajel-frekvenciáját és ezzel párhuzamosan az üzemi feszültségét – így próbálva optimalizálni az energiafelhasználást. (A műszaki megoldást a gyártók másmás néven népszerűsítik.) A számítóteljesítmény növelésének másik lehetséges útja az egy lépésben végrehajtható műveletek komplexitásának növelése. Ez egy általános célú processzor esetén bizonyos határon felül már nem hozna jelentős sebességnövekedést, bár a harmadik fejezetben szereplő térfogatszámító algoritmus hasznát tudná

venni egy 128 biten ábrázolt lebegőpontos számokat is kezelő aritmetikai egységnek. [cit] Az adott területegységen felépíthető tranzisztorok számának tekintetében még van tartalék, bár itt is a fizikai határok felé közelít a gyártástechnológia. [1] Például a ma elterjedő Intel Haswell processzorok úgynevezett 22nm-es gyártástechnológiával készülnek, ahol a különböző rétegek közti távolság már csak néhány tíz atom. A megbízható szigetelés létrehozása egyre nagyobb nehézségekbe ütközik. A processzorgyártás során használt félvezető lapkák méretének növelése gazdasági jellegű kockázatot hordoz. A szilícium lapkán, amelyre felépítik az integrált áramkört, előfordulnak kristályhibák. Minél nagyobb a felhasznált szilícium lapka, annál nagyobb a valószínűsége, hogy a területére olyan kristályhiba kerül, ami miatt az alkatrész selejtes lesz.

1.2. Történeti áttekintés

A számítások több számítóegységen történő párhuzamos futtatása nem új találmány. Richard Feynman az atomfegyver kifejlesztését szolgáló Manhattan terv kapcsán már 1944-ben, még a mai értelemben vett számítógépek megjelenése előtt foglalkozott számítások párhuzamosításával (Feynman 2010). A feladat a különböző elrendezésű berobbantó bombák energia-felszabadulásának kiszámítása volt IBM gyártmányú programvezérelt számológépen. Gene Amdahl már 1960-ban felismerte, hogy hiába növeljük a párhuzamosan működő adatfeldolgozó egységek számát, a program egy része - melyet az egymásra épülő eredmények miatt nem lehet párhuzamosítani - gátat szab a sebességnövekedésnek.

A történelemben számtalan párhuzamosan működő számítóegységet tartalmazó célhardver született egy-egy speciális probléma megoldására kihegyezve. Terjedelmi okok miatt csak a legelterjedtebb, általános célú architektúrák kerülnek megemlítésre a dolgozatban.

Michael J. Flynn a számítógép architektúrákat 1966-ban az 1.1. táblázat szerint sorolta be (Flynn 1972). A felosztás a mai napig jól tükrözi a probléma lényegét.

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

1.1. táblázat. A Flynn taxonómia.

[1] Pontosabb információ a gyártástechnológiákról és a kirajzolódó fejlődési pályáról a http://www.itrs.net/reports.html oldalon olvasható.

SISD (Single Instruction, Single Data). A klasszikus Neumann architektúrának felel meg, melyben egyetlen processzor végez műveletet egy időben egy adaton. A programok és algoritmusok nagy része évtizedeken keresztül erre az architektúrára készült.

MIMD (Multiple Instruction, Multiple Data). Több processzor hajt végre egymástól függetlenül programot más-más adathalmazon. Ide tartoznak azok a többprocesszoros gépek, melyekben a processzorok közös memóriát oszthatnak meg. Több processzormag kerülhet egy fizikai tokba is. A szűk keresztmetszetet ebben az esetben a közös memória-hozzáférésből adódó várakozás jelenti. Ugyan csak ebbe a kategóriába tartoznak a független memóriával rendelkező számítógépekből épített közös feladaton dolgozó hálózatok is.

MISD (Multiple Instruction, Single Data). Első olvasásra úgy tűnik, nem sok értelme van egy adaton egy időben több műveletet elvégezni. Gyakorlati alkalmazásai ma nem elterjedtek.

SIMD (Single Instruction, Multiple Data). Ugyanazt a műveletet egyszerre több adaton tudja végrehajtani. Például olyan problémák megoldására alkalmas, amikor egy függvény értékét kell meghatározni sok különböző paraméter mellett. Gyakorlatilag minden paraméter mellett ugyanazt a műveletsort kell végrehajtani.

A gyakorlatban egyre elterjedtebbek a fentiek ötvözeteként felépülő úgynevezett vegyes, vagy más néven heterogén architektúrák.

Érdemes megjegyezni, hogy Neumann János 1945-ben – két évvel a tranzisztor feltalálása előtt – írta le azokat az alapelveket, melyeket ma a tudományos világ "Neumann-elvek"-ként tart számon[2]. (William Bradford Shockley, John Bardeen és Walter Houser Brattain csak 1956-ban kaptak Nobel-díjat a félvezető-kutatásért és a tranzisztorhatás felfedezéséért.) A dolgozatban feldolgozott szakirodalom az egyszerre egy utasítást egy adaton végrehajtó gépeket nevezi „klasszikus” Neumann elvű számítógépnek, mert a Neumann-elvek között szerepel az utasítások szekvenciális végrehajtása. Ez azonban nem jelenti az alapelvek csorbulását a sokprocesszoros gépek esetén.

[2] A Neumann-elvek alapjául szolgáló „The First Draft Report on the EDVAC” címet viselő jelentését Neumann János hivatalosan nem publikálta. (Godfrey/Hendry 1993)

1.3. Párhuzamos megközelítések

A gazdasági számítások széles skálája miatt egyetlen olyan architektúra sem létezik, mely minden felmerülő problémára tökéletes megoldást biztosítana. Az adatfeldolgozás párhuzamosítására - gyorsítására - az alábbi megközelítések a legelterjedtebbek.

1.3.1. Szerelőszalagok

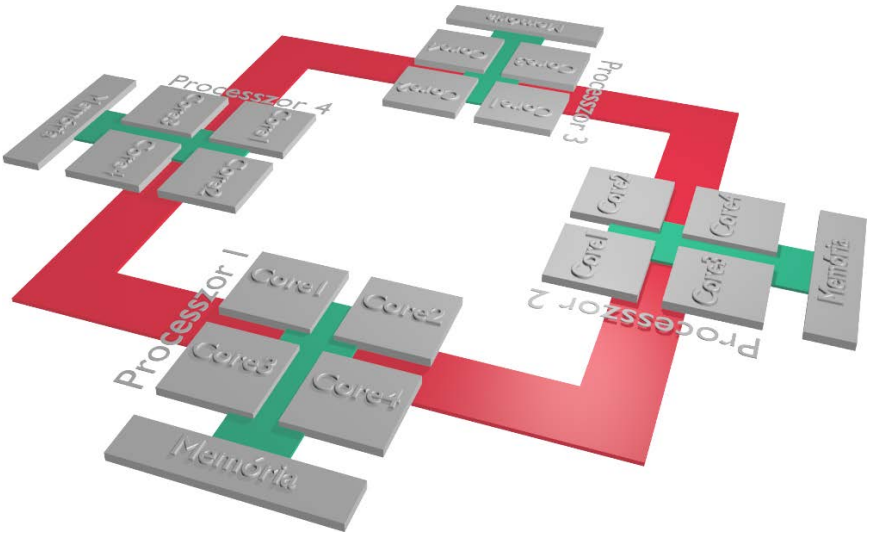
Egy gépi utasítás végrehajtása tipikusan négy órajelütemet vesz igénybe (beolvasás, dekódolás, végrehajtás, visszaírás). A szerelőszalag (pipeline) architektúra egyszerre négy utasítás feldolgozását végzi időben orgonasípszerűen eltolva. A pipeline szintén nem újdonság, már az 1978-ban megjelent 8086-os processzor is 6 byte-tal előre olvasta a memóriát. A probléma az, hogy a feltételes elágazásoknál megtörik a folyamat, bár a Pentiumok óta a processzorok egyre kifinomultabb statisztikai módszerekkel becsülik meg, hogy a feltételes elágazásoknál melyik irányba megy nagyobb valószínűséggel tovább a program futása. A statisztika készítéséhez a processzor számolja, hogy a múltban melyik irányba hányszor ment a program, és ennek alapján ad becslést a nagyobb valószínűséggel bekövetkező ágra.

A pipeline futási sebességre gyakorolt hatását egy egyszerű C# programmal vizsgáltam: egy tömbben tárolt egész számok között számoltam meg a páros, illetve páratlan értékek számát. Ha a tömb rendezett, azaz a páros számok egymás után szerepelnek, a futásidő 20%-al alacsonyabbnak mutatkozik, mintha felváltva szerepelnek a páros és páratlan értékek.

A programozó - ha a fordítója alkalmas rá - az elágazásoknál maga is megjelölheti a nagyobb valószínűséggel bekövetkező ágat. A legegyszerűbb megközelítésben az eredeti, egyprocesszoros architektúrára szervezett kód használható. A fordítóra bízzuk az optimalizálást, amely figyelembe veheti a célprocesszor pipeline-jait, és úgy rendezi az utasításokat, hogy ha lehet, ne törjék meg a pipeline-t. A számtalan optimalizálási lehetőség miatt ma már nem igaz, hogy az ember – ha az ideje nem lenne szűk keresztmetszet – gépi nyelven gyorsabban futó kódot tudna írni, mint a C fordító. A pipeline előnye, hogy változatlan a kód mellett régi programokon is segít, így az újrafejlesztés nem jelent kockázatot. Hátránya, hogy az elérhető sebességnövekedés korlátos, a fejlődés üteme lassú. A pipeline a háttérben végzi a feladatát, a gyakorlatban a fejlesztőnek nem sok dolga van vele.

1.3.2. Többprocesszoros gépek

Egy darab többmagos processzorból álló „klasszikus” architektúra sok feladatra alkalmazható. Minden processzor külön memóriablokkal rendelkezik, de egymás memóriaterületeit is elérhetik - igaz lassabban. A programok több, egymás mellett futó programszálat (thread) indíthatnak, amelyeket az operációs rendszer oszt szét a rendelkezésre álló magok között. A szálak száma meghaladhatja a rendelkezésre álló processzormagok számát, ebben az esetben az egyes szálak futtatása időosztásos rendszerben történik. (Túl sok szál futtatása a szálak közti váltogatás időköltsége miatt nem feltétlenül hatékony.) A szálak egymástól teljesen független utasításokat hajthatnak végre, és használhatnak közös változókat is. A többprocesszoros gépek, illetve a többmagos processzorok a Flynn-féle felosztás szerint a MIMD (multiple instruction, multiple data) kategóriába tartoznak.



1.1. ábra. Többmagos processzorokból álló architektúra.

Többszálúság

A szálak futási sebessége nem determinisztikus, a programozó nem tudhatja, hogy a programban hol, illetve mikor kerül át a vezérlés egy másik szálra. A nehézségek akkor kezdődnek, amikor több szálnak kell hozzáférnie ugyanahhoz a változóhoz. Annak megakadályozására, hogy egy szál olyan változóhoz férjen hozzá, amelyet egy másik szál éppen használ, a változók zárolhatók (lock). Az a szál, mely egy zárolt változóhoz szeretne fordulni, várakozó állapotba kerül, amíg a zárolást kérő szál a zárat fel nem szabadítja. Ebből kialakulhat egy körbetartozáshoz hasonló helyzet, amikor

minden szál egy másikra vár, hogy az felszabadítson egy zárolt változót. A rendszer holtpontra juthat (deadlock), amelyből beavatkozás nélkül nem tud elmozdulni (Goetz/Peierls 2006).

Védelmi elemek beépítésével biztonságosabbá tehető a több szálú program, de ennek az árát teljesítmény oldalon kell megfizetni. Súlyosabb esetben a zárolások túlzott alkalmazása miatt a futásidő meghaladhatja az egyszálú változat futásidejét. Zárolásokat használó többszálú algoritmusok helyességének igazolására nincs használható algoritmus.[3]

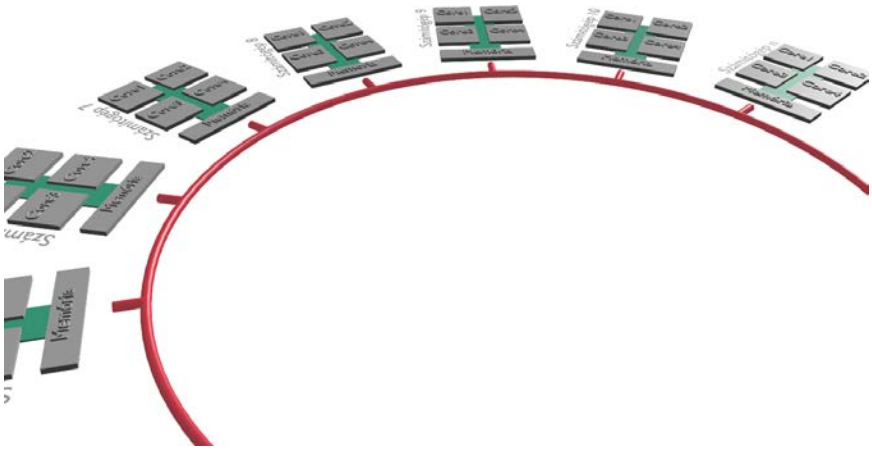
Dijkstra filozófusai

Edsger W Dijkstra egyetemi előadásain a holtpont jelenségét érdekes analógián mutatta be. A példa szerint öt csendes filozófus ül egy kerek asztal körül egy-egy tál spagetti előtt. A tányérok között csak egy-egy villa van. Minden filozófus felveheti a bal illetve a jobb kezénél levő villát, de addig nem kezdhet el enni, amíg mindkét villát meg nem szerezte. A villákat evés után le kell tenni. Azt a villát, amely más kezében van, nem lehet elvenni. Az étkezés holtpontra juthat, ha minden filozófus egy villát tart a kezében, és arra vár, hogy felszabaduljon a másik. Egyik filozófus sem tudja, hogy mi jár a többiek fejében. Olyan algoritmust megfogalmazni, amely biztosan nem jut holtpontra – azaz senki sem hal éhen – nem triviális, bár első ránézésre egyszerűnek tűnik.

1.3.3. Számítási klaszterek

Ha a megoldandó feladat felbontható olyan részfeladatokra, amelyek között nem szükséges gyakori illetve nagy mennyiségű adatcsere, a számítás hálózatba kötött számítógépekből álló klaszteren is végezhető. A klaszter tagjai tipikusan egy teremben vannak, és nagy sebességű hálózaton keresztül kapcsolódnak egymáshoz. (lásd: 1.2. ábra.) Akár az egyetemi géptermi gépek is használhatók klaszterként – a Corvinus Egyetemen is építettünk a hallgatókkal klasztert az egyik 35 gépes terem gépeiből, így összesen 140 processzormag áll rendelkezésünkre. A klaszteren most is folynak kísérletek a demográfiai előrevetítés és a „photon-rendering" képalkotó eljárás párhuzamosítására. A „photon-rendering" egyesével követi a térben elhelyezkedő tárgyakon visszaverődő illetve elnyelődő fotonok útját a fényforrástól tárgylemezig. A módszerrel valósághű kép alkotható, viszont a számításigény hatalmas.

[3] A Microsoft támogat egy kutatást a témában: http://research.microsoft.com/en-us/projects/CHESS/



1.2. ábra. Hálózatba kötött gépekből álló klaszter.

A klaszter kezelésére az egyik legcélszerűbb megoldás az MPI (Message Passing Interface) szabvány alkalmazása. Az MPI megoldja a program több példányban történő futtatását a klaszter tagjain, valamint az üzenetváltást az egyes példányok között. Az MPI klaszter minden gépén ugyanaz a program fut. A program tetszőleges példányszámban indítható. Minden példánynak van egy sorszáma, ez határozza meg a szerepét. Tipikusan az első példány osztja ki a feladatot a többieknek, majd összegyűjti, és összesíti a részeredményeket. A többi példány várja a részfeladatokat az első példánytól, és neki küldi vissza az eredményeket összesítésre.

A klaszterek kezelésének kapcsán mindenképpen meg kell említeni a Scala nyelvet és az aktor modellt, amelyben a programot egymást hívó állapot nélküli függvényekből kell felépíteni. Az Akka aktorjai a függvények futtatását automatikusan osztják szét a hálózat tagjai között a pillanatnyi terhelés függvényében.

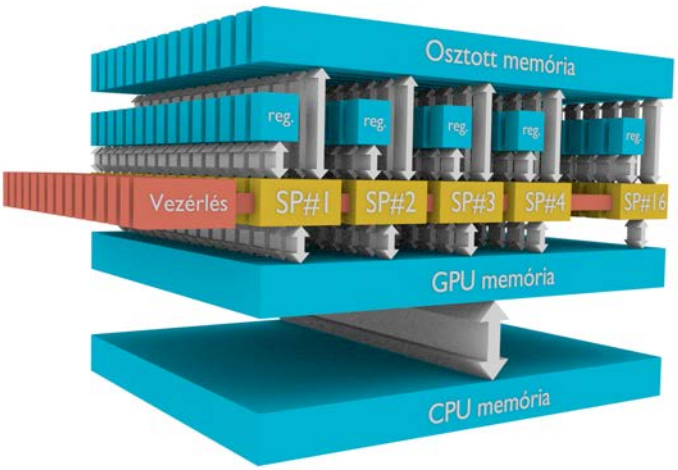
1.3.4. CPU-ból és GPU-ból álló heterogén architektúrák

A kutatók körében mostanában nagyon divatosak a grafikus processzorok (Graphics Processing Unit – GPU). A GPU piacért három nagy gyártó versenyez (nVidia, AMD, Intel) némiképp eltérő architektúrákkal. A kutatók körében talán legnépszerűbb az nVidia CUDA architektúrája. Az nVidia felismerte a grafikától eltérő számításokban rejlő üzleti potenciált, és piacra dobta a kifejezetten általános feladatmegoldásra tervezett Tesla termékcsaládját, amely már nem is rendelkezik videó kimenettel.

Fontos megjegyezni, hogy a hosszabb termékéletciklusban gondolkodó fejlesztők a verseny és a gyors fejlődés miatt gyakran bizalmatlanok. Ezzel szemben a 8086-os processzorral utasítás szinten kompatibilis eszköz 35 éve folyamatosan kapható.

A GPU-k csak olyan problémák megoldásában nyújtanak jó teljesítményt, ahol lépésről lépésre ugyanazt a műveletsorozatot kell elvégezni különböző adatokon. A GPU sematikus felépítését a 1.3. ábra szemlélteti. A grafikus processzorok sok aritmetikai egységgel (SP - Streaming Processor) rendelkeznek. A disszertáció írásának pillanatában csúcskategóriás Tesla K20-ban 2496 aritmetikai egység van. Ez azt jelenti, hogy a grafikus processzor akár több száz vagy ezer aritmetikai műveletet tud párhuzamosan végezni, egy művelet elvégzése viszont tipikusan háromszor annyi időt vesz igénybe, mint egy azonos kategóriájú CPU mag esetében.

A GPU mégsem tekinthető úgy, mint több száz közös tokba épített processzor, amelyek mind külön-külön feladatot végeznek. Tipikusan 16 vagy 32 darab SP rendelkezik egy közös vezérlővel, és alkot egy multiprocesszort (MP). A közös MP-n futó szálak ugyanazokat az utasításokat hajtják végre párhuzamosan, csak más-más adatokkal. A programban természetesen szerepelhetnek ciklusok és feltételes elágazások, de ha egyetlen szál is belefut egy feltétel valamely ágába, ahol további számításokat kell végeznie, az MP többi szála addig várakozik, amíg az ág le nem fut. Ugyanez igaz az eltérő lépésszámú ciklusokra is.



1.3. ábra. A GPU felépítése.

A professzionális grafikus kártyák a CPU-tól független memóriával rendelkeznek, melyet az 1.3. ábrán „GPU memória” felirat jelöl. A GPU a memóriát a CPU-jénál akár ötször szélesebb, 320 bites buszon éri el – azaz egy lépésben 40 byte adatot tud írni vagy olvasni a memóriából. A CPU és a GPU memóriája között a PCI buszon történik az adatmozgatás, amely a számítási időhöz képest jelentős lehet. A GPU önmagában működésképtelen, mindenképp szükség van egy CPU-ra, amely a GPU-t vezérli. A számítás lépései a következők:

1. Az első lépésben a kiinduló adatokat fel kell másolni a CPU memóriából a GPU memóriájába.

2. Ezután kerül sor a számítások elvégzéséhez szükséges kód futtatására a GPU-n. A GPU-n futó kód a „kernel". A kernelek hasonlítanak a függvényekhez: lehetnek argumentumaik, de számításaik eredményét valahol a GPU memóriájában tárolják. A GPU-n a kernel kód tetszőleges számú példányban indítható el - a feldolgozandó adat mennyiségének illetve struktúrájának megfelelően. A kernel kód egy-egy elindított példányát nevezzük szálnak (thread). Minden szál, azaz kernel-példány, egy-egy külön Streaming Processoron (SP) fut. A szálak száma messze meghaladhatja a SP-k számát, a GPU automatikusan gondoskodik a szálak sorbaállításáról és futtatásuk ütemezéséről. A szálak futtatási sorrendje nem determinisztikus. Mivel az összes szálon ugyanaz a kód fut megegyező argumentumokkal, az egyes szálaknak valahogy el kell dönteniük, hogy a GPU memóriába másolt adatok mely részének feldolgozásával foglalkozzanak, illetve az eredményt hol tárolják. Ez a szál sorszáma alapján dönthető el, amelyet a kód le tud kérdezni. Egy tipikus kernel-kód a következő lépéseket végzi el (Sanders/Kandort 2010):

   (a) kiolvassa a szál sorszámát,

   (b) a szál sorszámának megfelelően kiszámolja, hogy a GPU memóriájában mely adatokkal végez majd számításokat,

   (c) elvégzi a számításokat,

   (d) végül az eredményt visszaírja a szál sorszáma alapján meghatározott GPU memóriaterületre.

3. Utolsó lépésként a számítások eredményeit vissza kell másolni a GPU memóriájából a CPU memóriájába további feldolgozás illetve megjelenítés céljából.

Egy algoritmus akkor futtatható jó hatásfokkal a GPU-n, ha egyszerre szálak ezrein lehet ugyanazt a műveletsort végezni. A GPU-val elérhető sebességnövekedést nehéz becsülni. A kísérletek biztató eredményei után, ahogy az algoritmusba egyre több feltételes elágazás kerül, a kezdeti előny elveszhet. A jövőben várhatóan az algoritmusok egyre nagyobb részét tervezik CPU-ból és GPU-ból álló úgynevezett heterogén architektúrákra (Nvidia 2011). A GPU-k a Flynn-féle felosztás szerint a SIMD (Single instruction, multiple data) kategóriába tartoznak.

GPU-t kiaknázó programcsomagok

A GPU adta lehetőségeket több szoftvergyártó is igyekezett beépíteni termékeibe. A hagyományosan megírt Matlab programon nem gyorsít a GPU, viszont a 2010b változat óta rendelkezésre áll néhány olyan beépített függvény, amely kihasználja a grafikus kártyát. Ezek tipikusan mátrix műveletekkel kapcsolatos függvények. Bizonyos műveletek meggyorsítására a Wolfram Mathematica is képes kihasználni a GPU-t. Az adatbázis szerverek fejlesztői is tesznek lépéseket GPU-k irányába a jól párhuzamosítható keresési műveletek felgyorsítására (Bakkum/Skadron 2010). Az Amazon számítási felhő szolgáltatásában (Amazon Web Services) GPU számítókapacitást is lehet bérelni.[4] Ez jól mutatja a trendet, amely szerint a jövőben a szoftverek egyre nagyobb részét tervezik CPU-ból és GPU-ból álló heterogén architektúrára.

1.3.5. Szoftver fordítása hardverbe - FPGA

Az FPGA (Field Programmable Gate Array) nagyszámú logikai blokkot[5] tartalmazó integrált áramkör – felépítését az 1.4. ábra szemlélteti. A blokkok száma százezres nagyságrendű is lehet, ezekben a kapuk kimenetei és bemenetei közötti logikai függvény szoftveresen állítható be. Minden logikai blokknál beállítható, hogy a kimenetei és bemenetei közt milyen logikai függvényt valósítson meg. A logikai blokkok ki- és bemenetei buszrendszeren keresztül kapcsolhatók össze. Az FPGA-ba feltöltött kód bitjei egy-egy lehetséges kapcsolat meglétét vagy hiányát jelentik. FPGA-n gyakorlatilag bármi felépíthető, akár általános célú processzor is. Több gyártó kínál PCI buszra csatlakoztatható kártyára integrált FPGA-t.

Az FPGA-val kapcsolatos munka inkább mérnöki, mint programozói megközelítést igényel: a terv és a prototípus megszületése közti idővel számolni kell.

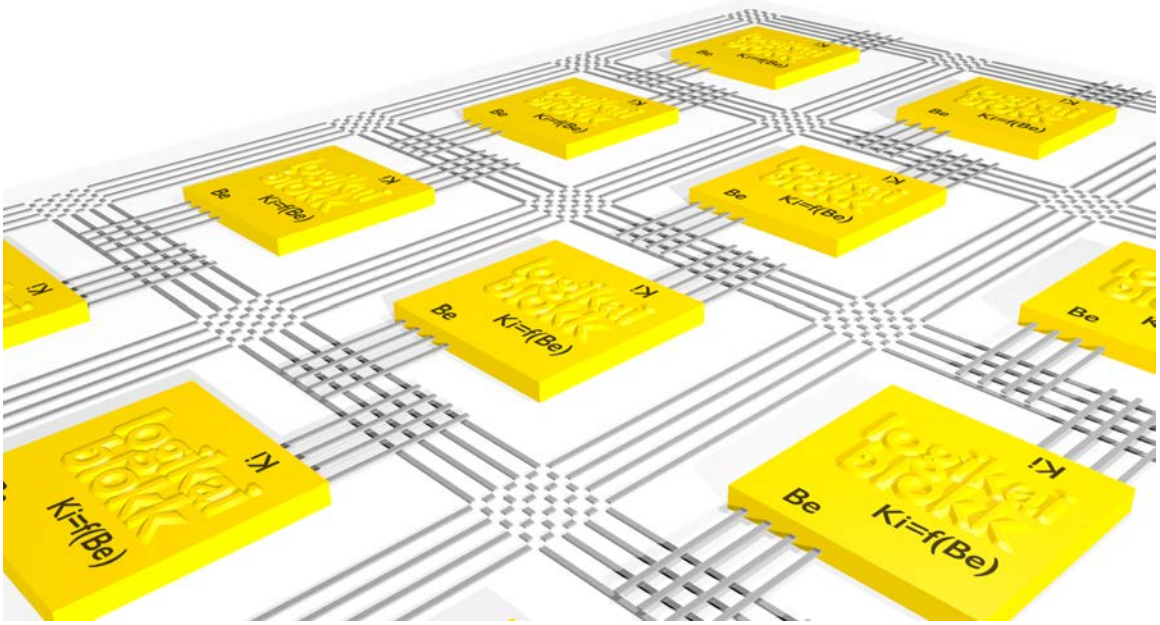
Léteznek olyan eszközök, amelyek a C nyelven megírt függvényeket logikai kapukból álló kapcsolássá alakítják. Az egyes logikai kapuk egyszerre működnek, ezért gyorsabb eredmény érhető el, mint az utasításokat sorban végrehajtó általános célú processzorok esetén. Folynak kutatások FPGA-n felépített neurális hálózatokkal is. Az FPGA-n szimulált neuronok párhuzamosan működhetnek - akárcsak az agy nagyszámú, de aránylag lassú neuronjai.[6]

Az FPGA-k megjelenésével érdekes spirál rajzolódott a technológia fejlődésébe. Az FPGA egy magasabb szinten visszalépés a logikai kapcsolásként felépített bonyolult függvényekhez, amelyek kiváltására született az általános célú Neumann-elvű

[4] http://aws.amazon.com/hpc-applications/

[5] Az elnevezések gyártónként eltérnek.

[6] http://sethdepot.org/papers/src/neural_networks.pdf



1.4. ábra. FPGA.

program-vezérelt számítógép. FPGA-n viszont akár Neumann-elvű processzor is felépíthető. (A gyakorlatban is bevett szokás, az FPGA-n felépített logikai elemeket egy, szintén az FPGA-n felépített kis processzor vezérli.)

1.4. A párhuzamos számítások néhány gazdasági alkalmazása

1.4.1. Optimalizáció

A gazdasági számításoknál felmerülő optimalizációs feladatok általában egy sokváltozós függvény minimum vagy maximumhelyének megkeresését jelentik a változók rögzített intervallumán belül. A megoldást a függvényértékek kiszámolása jelenti a változók különböző értékei mellett, ezek közül kell kiválasztani a legnagyobbat, illetve a legkisebbet. Annak eldöntésére, hogy a változók mely értékei mellett vegyünk mintát a függvényből, több megközelítés létezik. A nyers-erő (brute force grid search) módszer egy szabályos rács pontjai mentén számolja ki a függvényértékeket, de ez túl sok kombinációhoz vezethet, amelyek kiszámítása egy szuperszámítógépen is túl lassú. A keresési tér szűkítésére különféle módszerek léteznek. A genetikus keresési algoritmusok jelentős teret nyertek az elmúlt években. A biológiában észlelt természetes szelekció jól alkalmazható az optimális paramétervektor tenyésztésére, azonban

a módszer nem véd a lokális minimum/maximum problémával szemben. A vezérlő algoritmus paraméter-kombinációkat tenyészt szelekció, kereszteződés és mutáció segítségével. GPU csak akkor használható jó hatásfokkal az egyes függvényértékek kiszámítására, ha nincs sok feltételes elágazás.

A másik megközelítés az optimum meghatározására a sok számítás helyett matematikai oldalról, egyenletrendszerek megoldásán keresztül vezet. A tapasztalataink azt mutatják, hogy a mátrixműveleteken alapuló lineáris egyenletrendszer-megoldó módszerek, mint az ABS (Abaffy/Spedicato 1989) nagyon jól teljesítenek GPU-n. Mátrix-szorzásnál például az eredménymátrix elemei számolhatók párhuzamosan. A mátrix műveletek nagy mennyiségű adatot mozgatnak, amiben a GPU szélesebb adatbusza előnyt jelent.

Az ABS módszer implementációját elkészítettem CUDA architektúrára. A 480 számítási egységet tartalmazó, „középkategóriás" GeForce GTX 570 grafikus kártyán egy 4096 ismeretlenes sűrű együttható-mátrixos lineáris egyenletrendszer megoldása 105 másodpercet vett igénybe. Próbaképp egy 8192 ismeretlenes egyenletrendszert is megoldottunk, ez már 23 percet vett igénybe. A két eredmény nem mérhető össze, mert az utóbbi együttható mátrix már nem fért be a GPU memóriájába, az adatmozgatás jelentősen lerontotta a hatásfokot. A dolgozat következő része az ABS párhuzamos implementációját részletesen bemutatja.

1.4.2. Teljesítmény kiértékelés

Gazdasági modellek és kereskedési stratégiák teljesítményének múltbéli adatokon történő kiértékelésének algoritmusai eredendően jól párhuzamosíthatóak, akár több processzoros gépeken, akár klasztereken. Az egyes szimulációk egymástól függetlenek és egy futtatás az idő mentén is több részfeladatra bontható (Chan 2013).

Egyszerű példa a deviza-keresztárfolyamon (EUR/USD) végzett mozgóátlag kereszteződésen alapuló kereskedési stratégia (MA-crossover strategy). Amennyiben a gyors mozgóátlag értéke nagyobb a lassú mozgóátlagénál, venni akarjuk a párt, ellenkező esetben pedig eladni. (A mozgóátlag sebességét az átlag tagjainak száma határozza meg, a kevesebb elemből álló átlag gyorsabban reagál az új változásokra mint a nagyobb időtávon számított. (Chan 2008)(Pardo 2011))

Az algoritmus teljesítménye a múltbéli adatokon hozott kereskedési döntések és azok hozamai alapján mérhető. A párhuzamos feldolgozás érdekében a múltbéli adatok évekre tagolhatók, amennyiben minden év utolsó napján minden felvett piaci pozíció lezárásra kerül. Erre azért van szükség, hogy az egyes részfeladatok között ne

legyen semmiféle áthatás, azaz egymástól függetlenül futtathatóak legyenek. A probléma így ugyanazon algoritmus futtatása több éves idősoron, párhuzamosan, majd az egyes részeredmények (teljesítmény mutatók, mint például hozam, kitettség, Sharpe ráta, összes felvett pozíciók száma) összegzése. Természetesen a végeredményhez minden részfeladat eredményére szükség van, mégis hasznos az egyes évek teljesítményének riportálása abban a pillanatban, ahogy rendelkezésre állnak. A „Backtest" futtatói dönthetnek ugyanis úgy, hogy a szimulációt leállítják, ha észlelnek egy katasztrofálisan teljesítő évet (pl. 2008), ami szükségtelenné teszi a többi számítás elvégzését, mivel ilyen magas tapasztalt kockázati kitettség mellett nem áll szándékukban további energiát fektetni az adott modellbe.

A backtesting-feladatok általában egy optimalizációs probléma részei, amikor is az adott algoritmus különböző paraméterbeállításai mellett vizsgáljuk a teljesítményt, azt remélve, hogy találunk olyan faktorokat a modellben, melyek jelentősen befolyásolják az eredményt. Az n paraméterből álló paraméter-mátrix egyes dimenziói mentén tetszőleges értéket vehet fel, így kifeszít egy nagyobb keresési teret, amely kombinációinak tesztelése ugyancsak jól párhuzamosítható. Ezek a kombinációk mind egy-egy teljes backtestnek felelnek meg az adott időintervallumon.

1.4.3. Hatásvizsgálat – Stress testing

A Bázel II. és III. rendelkezések hatására napjainkban aktívan zajlik a stress testing különböző pénzintézetekben. A felvetett kérdés az, hogy adott piaci volatilitás változására, egy nem várt katasztrófa bekövetkezésére, vagy bizonyos faktorok hatásai mellett az intézet portfóliói, befektetései milyen potenciális kockázatnak (lehetséges veszteségnek) vannak kitéve. Ennek függvényében határozható meg a minimális tőkefedezet, amelyet az intézetnek állandóan biztosítania kell (Berry 2009). A stress testing általában több faktor mentén zajlik és az egyes faktorokat is finom skála mellett értékelik ki. A két dimenzió mentén számtalan kombináció adódik, amelyek azonban egymástól függetlenül számolhatók, így a különböző párhuzamos architektúrák gyorsan elterjedtek a nagyobb pénzintézetekben.

1.4.4. Nyugdíj mikroszimuláció

A Budapesti Corvinus Egyetemen folyó nyugdíj mikroszimuláció a nyugdíjasok számának alakulását modellezi Monte-Carlo módszerrel. Arra a kérdésre keressük a választ, hogy az elkövetkezendő években hány gyermek, munkaképes állampolgár illetve nyugdíjas él majd az országban. A nyugdíjmodellezésnek nagy jelentősége van a

nyugdíjrendszer finanszírozhatóságának számításánál. A szimuláció a teljes lakosság adataiból indul ki, majd éves körökben születési és halálozási valószínűségi táblákra alapozva egyénenként modellezi a lakosság életpályáját. (A korábbi évek statisztikai adatai alapján összeállított táblázatból kiolvasható, hogy egy férfi vagy egy nő hány éves korában milyen valószínűséggel hal meg. Érdekes, hogy a táblázat a javuló egészségügyi ellátás és a változó életmód miatt évről évre más eloszlást mutat, ezért a múlt adatait a jövőre extrapolálni kell. Hasonló módszerrel összeállítható táblázat az elkövetkező években várható születésszámokról és a munkaképtelenné válás valószínűségeiről is.) Ezek fontos kérdések például a biztosítási díjak kalkulációjánál is. A szimuláció egymás után többször kerül futtatásra, az egyes futtatások eredményei alapján valószínűségi eloszlás rajzolható a várható élettartamról és a nyugdíjasok várható számáról.

Az egyszerű modellben az egyedek között nincs kapcsolat. A bonyolultabb modell családokban gondolkodik, és figyelembe veszi azt is, hogy a házas emberek életkilátásai jobbak, mint az egyedülállóké. A házastársak várható élettartama között is összefüggés mutatkozik. A jelenséget a szakirodalom "összetört szív" szindrómaként említi. Itt az egyedek közt már van kapcsolat, így a szimuláció használ közös memóriát.

A szimuláció futtatására a GPU nem alkalmas, mivel az algoritmusban nagyon sok a feltételes elágazás. A modellben az egyedek illetve a családok sorsa független, így a feladat futtatható a fent említett MPI klaszteren úgy, hogy a klaszter minden tagja a lakosság egy részének sorsát követi.

1.5. Összefoglalás

A legegyszerűbbek az olyan feladatok, mint a nyers erő keresés vagy backtesting, ahol egy függvény értékeit kell kiszámolni sokféle paraméter mentén. Ezekben az esetekben az egyes számítások eredményei nem függenek egymástól, így akár többprocesszoros gépen, akár több gépből álló klaszteren jól párhuzamosíthatók. A GPU-k olcsó alternatívát jelentenek a kutatóknak, de sok vállalat idegenkedik tőlük a technológia gyors fejlődése és az eltérő megoldásokkal versengő nagy gyártók miatt. Csak olyan feladatok megoldásában jelentenek alternatívát, ahol lépésről lépésre ugyanazt a műveletsorozatot kell elvégezni különböző adatokon. Ilyenek például a mátrix műveletek.

Ha az egyes részfeladatok közös változókat használnak, a többszálú megközelítés alkalmazható. Itt a változóíráskor bekövetkező időbeni ütközések jelentik a problémát, mely feloldására nem létezik általános megoldás. Ebből adódóan ritkán fellépő

és nehezen javítható hibák léphetnek fel, amelyek kockázatot jelentenek a fejlesztési idő tervezésénél.

Egyszerre több párhuzamosítási módszer is alkalmazható lehet. A IV. részben bemutatásra kerülő mikroszimulációs keretrendszerem továbbfejlesztése a több gépből álló klaszterek irányába tart, ahol a klaszter tagjain többszálú program fut. Célom az, hogy rövid idő alatt több-százszor futtathassuk a szimulációt, annak érdekében, hogy a futási eredmények eloszlásából lehessen következtetni a bekövetkezési valószínűségre.

A párhuzamosítás nem csupán programozás-technikai szakmunka - elképzelhető, hogy a kívánt eredmény érdekében a bevált algoritmus működési elvét is meg kell változtatni. Az elérhető teljesítménynövekedés és a fejlesztési idő becslése nehéz feladat.

2. fejezet

Lineáris egyenletrendszerek megoldása ABS-módszerrel

2.1. Az ABS algoritmus

Az ABS módszer első verzióját Dr. Abaffy József publikálta az Alkalmazott Matematikai Lapokban (J. 1979). Később csatlakoztak a kutatáshoz Charles G. Broyden és Emilio Spedicato matematikusok (Abaffy/Spedicato/Broyden 1984). A módszer sokváltozós lineáris egyenletrendszerek hatékony megoldását teszi lehetővé. Az algoritmus különlegessége, hogy a részeredmények tárolásához mindössze egy mátrixot használ, így nagyon gazdaságos a memóriakihasználása – ebből adódóan általános célú hardveren is alkalmas nagy méretű feladatok megoldására. A hibaterjedés vizsgálatára elkészítettem az algoritmus implementációját masszívan párhuzamos GPU architektúrára. Ennek segítségével vizsgálom a hibaterjedést nagy méretű, sűrű együtthatós mátrix esetén.

2013-ban került bizonyításra az ABS módszer módosított Huang változatának stabilitása. A módosított Huang módszer jobb stabilitási tulajdonságokkal rendelkezik, mint az eddig legstabilabbnak tartott módosított Gram-Smidth (MGS) módszer (Gáti 2013). Az alábbiakban az eredeti algoritmus és a módosított Huang módszer vázlatos leírása következik, a részletes leírás (Abaffy/Spedicato 1989)-ban olvasható.

	Matlab kód	
	`function [ x ] = absCpu( A, b )`	$Ax = b \quad (A \in \mathbb{R}^{m \times n})$
(a)	`  [m,n] = size(A);`	
	`  x = zeros(n,1);`	$x_0 \in \mathbb{R}^n, x_0 = 0$
	`  errLimit = 1E-10;`	$e = 10^{-10}$
	`  H = eye(n);`	$H_0 = I \quad (H \in \mathbb{R}^{n \times n})$
	`  for i=1:m`	ciklus $i = 1, ..., n$
(b)	`    a = A(i,:)';`	$a_i = A_i^T$
	`    s = H * a;`	$s_i = H_{i-1} a_i$
	`    if s < errLimit`	
	`      t = a' * x - b(i);`	
(c)	`      if t~=0`	ha $a_i x - b_i \neq 0$ ...
(d)	`        break`	ellentmondás
	`      else`	ha $a_i x - b_i = 0$ ...
(e)	`        continue`	lineáris függőség
	`      end`	
	`    end`	
(f)	`    p = H' * a;`	$p_i = H_{i-1}^T a_i$
(g)	`    p = H' * p;`	$p_i = H_{i-1}^T p_i$
(h)	`    x = x - ((a' * x - b(i))` `/(a' * p)) * p;`	$x_i = x_{i-1} - \frac{a_i^T x - b_i}{a_i^T p_i} p_i$
(i)	`    H = H - ((p*p') / (p'*p))` `/ (a' * H * a);`	$H_i = H_{i-1} - \frac{p_i}{p_i^T p_i} \cdot p_i^T$
	`  end`	
	`end`	ciklus vége

2.1. táblázat. A módosított Huang-módszer.

A táblázatban használt jelölések megtalálhatók a letölthető C++ forráskódban is. Értelmezésükhöz az alábbi magyarázat nyújt segítséget:

(a) Itt kerül meghatározásra az együtthatómátrix mérete.

(b) Az a_i vektor az együtthatómátrix i-edik sora.

(c) Ha a $a_i x - b_i \neq 0$ feltétel teljesül, az együtthatómátrix éppen vetített sora vagy lineárisan függ az előzőektől, vagy ellentmondásra vezet. Ennek eldöntésére a következő két pont szolgál:

(d) Ha $t = 0$, akkor a_i ellentmondáshoz vezet, ezért az algoritmus futását fel kell függeszteni.

(e) Ha $t \neq 0$, akkor a_i lineárisan függ az együtthatómátrix már feldolgozott soraitól, így figyelmen kívül hagyható.

(g) A $p_i = H_{i-1}^T p_i$ lépés a Huang módszer szerint újravetíti a p_i vektort. Ennek a lépésnek az elhagyásával az eredeti Huang módszerhez jutunk.

2.2. Tervezési szempontok CUDA arhitektúrára

2.2.1. Szálak szervezése

A grafikus kártyán futtatott függvényeket *kernel*eknek nevezzük. A kerneleket sok példányban futtatjuk – a példányok száma akár ezres vagy milliós is lehet. A futtatott kernelpéldányok a *szál*ak. A szálak úgynevezett *blokk*okban kerülnek futtatásra, a szálblokkok könnyebb kezelhetőség érdekében egy, két vagy három dimenziósak lehetnek.

Egy további szervezési szinten a blokkok rácsba vannak szervezve, mely szintén lehet egy, két vagy három dimenziós. A kernel indításakor megadható, hogy hányszor hány szálat szeretnénk indítani egy blokkon belül, illetve dimenziónként hány blokk legyen a rácsban.

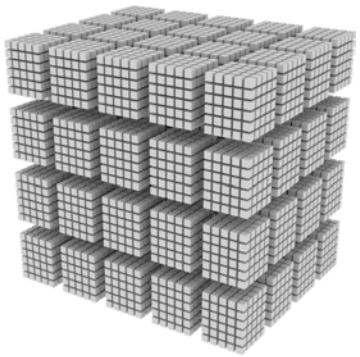
Minden szál a rácsban, illetve a blokkban elfoglalt helye alapján döntheti el, hogy a memóriából mely adatokkal végez műveletet, illetve az eredményt hol tárolja. (A kernelek kaphatnak bemenő argumentumokat, de függvényhez hasonló visszatérési értékük nincs. Az eredményt a GPU memóriában tárolják.)

A többdimenziós megközelítés az algoritmusok tervezőit segíti. Egy véges-elemes műszaki szimulációnál, ahol a testet elemi kockára bontjuk, a blokkokat és a rácsot szervezhetjük három dimenzió mentén. Mátrixműveletek esetén a kétdimenziós elrendezés tűnik célszerűnek.

A szálak blokkokba történő szervezése nem csupán áttekinthetőségi szempontokat szolgál. A közös blokkban lévő szálak biztos, hogy ugyanazon a multiprocesszoron kerülnek futtatásra, és így elérik a multiprocesszorhoz tartozó osztott memóriát. Ettől eltekintve az algoritmusfejlesztőnek nincs ráhatása arra, hogy az egyes szálak milyen sorrendben és melyik számítóegységen kerüljenek futtatásra.

2.2.2. Algoritmustervezési megfontolások

- Az nVidia kártyák által támogatott műveletek halmazát a “compute capabiliy” paraméter mutatja meg. Az 1.3-as *compute capability*-nél magasabb verziójú



2.1. ábra. $4 \times 4 \times 5$ *blokk*ból álló *rács*, blokkonként $6 \times 5 \times 5$ *szál*lal.
Az ábrán minden kis kocka egy szálnak felel meg.

kártyák már támogatják a 8 byte-on ábrázolt dupla-pontosságú lebegőpontos számok használatát. A korábbiak csak a 3D grafikában használt 4 bájtos egyszeres pontosságú float típust ismerik. A dupla-pontosságú aritmetikát használó tudományos alkalmazások futtatásának minimális feltétele a compute capability 1.3 megléte.

- A GPU saját memóriával rendelkezik. Az adatmozgatás PCI buszon keresztül történik, és természetesen időköltsége van. A kommerciális kártyák memóriájának szokásos mérete 1 Gb és 4 Gb közé esik. A CUDA architektúrán futtatott programoknál a közösen használt globális memóriára történő várakozás lehet a szűk keresztmetszet, nem a processzorok száma. Ezen segít a szélesebb adatbusz, mely a csúcskategóriás kártyákon akár 512 bit is lehet. (A fejlesztéshez használt kártya adatbusza 320 bit széles, ami azt jelenti, hogy egy lépésben 40 byte adatot képes mozgatni a memória és a számítóegységek között. A ma elterjedt általános célú CPU-k ezzel szemben csak 64 bites busszal rendelkeznek, mely 8 byte egyidejű mozgatását teszi lehetővé.) Nem közömbös, hogy az adatok hogyan helyezkednek el a memóriában. Ha a számításokhoz szükséges adat szétszórtan helyezkedik el a GPU memóriájában, a széles adatbusz előnyei nem mutatkoznak.

- A közös memóriahasználatból fakadó időveszteség enyhítésére a GPU rendelkezik egy kisebb memóriával konstansok tárolására, amelyet minden szál várakozás nélkül olvashat (Reg). Ezen felül minden multiprocesszorhoz tartozik egy úgynevezett osztott memória (Shared Memory), amelyet csak az adott multiprocesszoron futó szálak érnek el és használnak közösen. Algoritmustervezésnél

mérlegelni kell, hogy érdemes-e a gyakran hivatkozott adatokat a globális memóriaterületről az osztott területre másolni.

- A szálak futtatási sorrendje nem determinisztikus. A tervező annyit tehet, hogy szinkronizációs pontokat helyezhet el a kernel kódjában. Ebben az esetben a GPU elviszi az összes szálat a szinkronizációs pontig, és csak akkor engedi őket tovább, ha már mindegyik elérte a szinkronizációs pontot.

2.2.3. Fejlesztőeszközök

Jelenleg három elterjedt környezet létezik, mellyel GPU-n futó programot lehet készíteni:

CUDA: Az nVidia architektúrája, amely csak nVidia hardverrel használható. 1996-ban jelentették be, amit egy évvel később követett az 1.0-ás változat. Jelenleg a 4.0-ás változat a legfrissebb. CUDA architektúrára többféle nyelven lehet programot fejleszteni. Létezik C, Fortran, Python, C#, Java és Perl fordító is. A CUDA program két részből áll: CPU-n futó kódból valamint néhány GPU-n futtatandó kernelből. A CPU-futtatandó kód és a GPU-n futtatandó kernel ugyanazon a nyelven írható, de a kerneleket a programban külön kell jelölni.

OpenCL: A CUDA-nál sokkal általánosabb célú OpenCL fordító segítségével AMD és az nVidia eszközökre, valamint többmagos CPU architektúrákra is lehet alkalmazásokat fejleszteni. A nyelv a C-hez hasonlít. Szakmai fórumok egyetértenek abban, hogy a nyílt forrású OpenCL még nem annyira kiforrott, mint a CUDA, ugyanakkor a többféle támogatott platformnak köszönhetően nagy jövő jósolható neki.

DirectCompute: A tudományos világban általában a megoldani kívánt feladathoz választják a hardvert. A Microsoft üzletpolitikája ezzel szemben azt kívánja, hogy szoftverei minél több hardveren fussanak. Ennek megfelelően megoldásuk több gyártó hardverével működik. A DirectCompute kevéssé elterjedt. Jelen változata nem támogatja a dupla pontosságú aritmetikát.

2.2.4. A GPU-val szemben felmerülő kritikák

Internetes fórumokon többször találkozunk olyan vélekedéssel, mely szerint a GPU-k nem rendelkeznek úgynevezett ECC (error-correcting code) mechanizmussal a hibák javítására, ezért számítás közben előfordulhatnak bithibák. Ez valóban igaz a régebbi

típusokra. (Haque/Pande 2010) foglalkozik mélyebben a kérdéssel, és megállapítja, hogy a legtöbb esetben ez nem jelent gyakorlati problémát – a helyzet nem sokkal rosszabb, mint az általános célú CPU-k esetében. (A grafikus kártyák eredeti feladatánál egy-egy pixelhiba gyakorlatilag észrevehetetlen.) Az újabb sorozatok, mint a Quadro vagy a Tesla már támogatják az ECC-t.

A hosszabb termékéletciklusban gondolkodó vállalatok gyakran idegenkednek a CUDA használatától, mert attól tartanak, hogy az egymást követő változatok nem lesznek szoftver szinten visszafelé kompatibilisek egymással. A CUDA fordítók a kernel kódját egy köztes kódra fordítják, amelyet a kernel futtatásakor a grafikus kártya meghajtószoftvere fordít le a gépben lévő kártya gépi kódjára. Ez nem változtat azon, hogy a CUDA architektúra nem egy széles körben, több gyártó által is elfogadott ipari szabvány, hanem egyetlen cég megoldása egy adott problémára. Az elmúlt évek tapasztalata óvatosságra inti a döntéshozókat – korábban megingathatatlannak hitt vállalatok is kerültek csőd szélére néhány elhibázott stratégiai döntés okán. Ezzel veszélybe kerülhet az egyetlen gyártóhoz köthető termékek hosszútávú támogatottsága. A kutatók, akik gyakran egy-egy konkrét feladat megoldásához készítenek programot, és nem évtizedes termék-életciklussal számolnak, nyitottabbnak tűnnek a GPU technológiák irányában.

2.3. Az ABS optimalizálása CUDA architektúrára

2.3.1. Memóriahasználat

Egy dupla-pontosságú lebegőpontos szám tárolásához 8 byte memória szükséges.[1] A 2.2. táblázat az ABS algoritmus memóriaigényét mutatja a független változók számának függvényében. (Az eredeti és a módosított Huang módszer között memóriahasználat tekintetében nincs különbség.)

A Huang módszer megvalósításához az alábbi vektoroknak illetve mátrixoknak kell helyet biztosítani a memóriában: $a, s, p, ha \in \mathbb{R}^n$ vektorok és $H \in \mathbb{R}^{n \times n}$ mátrix a részeredmények tárolásához szükséges, $A \in \mathbb{R}^{m \times n}$ mátrix és $b, x \in \mathbb{R}^n$ vektorok a kiinduló adatokat illetve ez eredményt tárolják. Feltéve, hogy az együtthatómátrix négyzetes ($m = n$), az algoritmus futtatásához $2 \cdot n^2 \cdot 8 + 6 \cdot n \cdot 8$ byte memóriára van szükség. Nagyobb együtthatómátrix esetén ez nehézségeket okozhat, hiszen a GPU memóriája véges. A fejlesztés során használt kártya 1.4 Gb memóriával rendelkezik,

[1] A dupla-pontosságú lebegőpontos számok ábrázolásának szabályait az IEEE 754 szabvány írja elő.

n	Memória igény	
4	448	byte
8	1.4	byte
16	4.8	Kb
32	17.5	Kb
64	67.0	Kb
128	262.0	Kb
256	1.0	Mb
512	4.0	Mb
1024	16.0	Mb
2048	64.1	Mb
4096	256.2	Mb
8192	1.0	Gb
16384	4.0	Gb

2.2. táblázat. Memóriahasználat a változók számának függvényében.

de ennek egy részét elfoglalja az operációs rendszer a monitorokon megjelenő kép tárolásához. A vektorok tárolásához használt memória eltörpül A és H mátrixok memóriaigénye mellett. A legtöbb nVidia kártya nem csak a saját memóriájába másolt adatokkal képes számolni, hanem eléri a CPU memóriáját is. A CPU memória elérése nagyságrendekkel lassabb, mint a GPU memóriáé, ezért ezt a szükségmegoldást csak abban az esetben érdemes bevetni, ha a változók nem férnek el a GPU memóriában. A GPU-ra tervezett algoritmus esetén a 8091 változós esetben az együttható mátrix már a CPU memóriájában maradt. Azért az együtthatómátrix marad a lassan elérhető CPU memóriában, mert ennek minden sorához csak egyszer kell hozzáférni a futtatás során, így végeredményben itt a legkisebb a veszteség. (A CPU memória használatából adódó teljesítménycsökkenésről a későbbiekben lesz szó.)

Az ABS Huang alosztályának egyik fontos elméleti tulajdonsága, hogy H_i projekciós mátrixok szimmetrikusak, így tárolásukhoz nem szükséges $8 \cdot n^2$ bájt (n a H_i négyzetes mátrix dimenziója). A szimmetricitás kihasználásával a memóriaigény csökkenthető lenne kismértékű sebességromlás árán.

2.3.2. Mátrix műveletek GPU-n

Két mátrix összeszorzása GPU-n első ránézésre nem tűnik bonyolult feladatnak. Tekintsük az $R = AB$ feladatot példaként, ahol minden mátrix n-ed rendű és négyzetes. A végrehajtandó műveletsor első megközelítésben a következő:

1. A kiinduló adatok a CPU memóriájában foglalnak helyet.

2. Lefoglaljuk az A, B és R mátrixok tárolásához szükséges területet a GPU memóriájában.

3. A és B mátrixokat felmásoljuk a CPU memóriájából a GPU memóriájába.

4. Futtatjuk a mátrixszorzásra írt kernelt $n \cdot n$ szálon, egy blokkban. Minden kernel-példány R mátrix egyetlen elemének kiszámításáért felel. Az, hogy melyik elemet számolja ki, illetve az eredményt hol tárolja, a szálblokkban elfoglalt helyétől függ.

5. Az eredményül kapott R mátrixot lemásoljuk a GPU memóriájából a CPU memóriájába további feldolgozás vagy megjelenítés végett.

6. Ha a GPU memóriájában tárolt mátrixokra már nincs szükség a további számításoknál, felszabadítjuk a lefoglalt GPU memóriát.

Ez a megközelítés nem optimális, hiszen minden szál a globális GPU memóriából olvas, és ide írja az eredményt is. Ebből adódóan a memóriára történő várakozás a szűk keresztmetszet, így számítóegységek kapacitását nem tudjuk kihasználni. Gyorsabb végrehajtás érhető el, ha a mátrixok al-mátrixait az osztott memóriába másoljuk, és a szorzást részenként végezzük el. Az al-mátrixok másolására fordított idő kevesebb, mint a közös memóriahasználatnál a várakozásból adódó veszteség. A témával alaposabban a (Sanders/Kandort 2010) foglalkozik.

A fejlesztők életét megkönnyítendő több nyílt forrású lineáris algebrát támogató kódkönyvtár létezik. A legismertebb talán a C nyelven írott BLAS (Basic Linear Algebra Subprograms)[2]. A BLAS CUDA architektúrára átültetett párhuzamos változata a cuBLAS. A különféle módon tárolt ritka mátrixokat is támogató cuSPARSE könyvtárral együtt szabadon felhasználható[3].

2.3.3. Az adatforgalom csökkentése

Mint az előző pontból is látszik, a lineáris algebra és a mátrixműveletek területe jól körbejárt, mind a CPU, mind a GPU tekintetében. További teljesítménynövekedés úgy érhető el, ha az elemi mátrix és vektorműveleteket összevonjuk annak érdekében, hogy a memória terhelését csökkentsük. A 2.3. táblázat bal oszlopa a módosított Huang módszer lépéseit tartalmazza. A jobb oldali oszlop az egyes lépéseket megvalósító

[2] http://www.netlib.org/blas/
[3] https://developer.nvidia.com/cuda-toolkit

CUDA kernel nevét tartalmazza. Az ABS algoritmus futtatásához készített kernelek tipikusan több elemi műveletet vonnak össze, és felhasználják az előző pontban szereplő optimalizálási lehetőségeket.

	Algoritmus lépései	CUDA kernel név
	$Ax = b \quad (A \in \mathbb{R}^{m \times n})$	
(a)		
	$x_0 \in \mathbb{R}^n, x_0 = 0$	
	$e = 10^{-10}$	
	$H_0 = I \quad (H \in \mathbb{R}^{n \times n})$	`MakeEye`
	Ciklus $i = 1, ..., n$	
(b)	$a_i = A_i^T$	
	$s_i = H_{i-1} a_i$	`MatMulColvec`
		`VecIsLessThen`
		`RowvecMulColvec`
(c)	ha $a_i x - b_i \neq 0$ ...	`VecIsNull`
(d)	ellentmondás	
	ha $a_i x - b_i = 0$ ...	
(e)	lineáris függőség	
(f)	$p_i = H_{i-1}^T a_i$	
(g)	$p_i = H_{i-1}^T p_i$	
(h)	$x_i = x_{i-1} - \frac{a_i^T x - b_i}{a_i^T p_i} p_i$	`VecSubVecMulConst`
(i)	$H_i = H_{i-1} - \frac{p_i}{p_i^T p_i} \cdot p_i^T$	`MatSubMatDivConst`
	ciklus vége	

2.3. táblázat. A módosított Huang módszert megvalósító C függvények.

`MakeEye` Egységmátrixot hoz létre a GPU memóriájában. A bemutatott implementáció oszlopfolytonosan tárolja a mátrixok elemeit a memóriában.

`MatMulColvec` Összeszoroz egy mátrixot egy oszlopvektorral. (A memóriában a sor és az oszlopvektorok ábrázolása közt nincs különbség. Mindkét esetben egyszerűen felsorolásra kerülnek a dupla-pontosságú formában ábrázolt elemek. Az elnevezés csak a könnyebb megértést szolgálja.)

`VecIsLessThen` Megvizsgálja, hogy egy vektor összes elemének abszolút értéke egy határérték alá esik-e. Az algoritmus (b) lépése megvizsgálja, hogy az $a_i x - b_i$ vektor

nullvektor-e. A számábrázolási hibák terjedéséből adódóan ez az érték magasabb dimenziókban nem nulla lesz, hanem egy nagyon kicsi szám.

`VecIsNull` Egy vektorról eldönti, hogy nullvektor-e.

`VecSubVecMulConst` Egy vektort megszoroz egy konstanssal, majd két vektor különbségét képezi.

`MatSubMatDivConst` Egy mátrixot eloszt egy konstanssal, majd a két mátrix különbségét képezi.

2.4. Számítási eredmények

A 2.4. táblázat a számításokhoz használt grafikus kártya paramétereit tartalmazza. Összehasonlításul a jelenleg csúcskategóriás Tesla K20 kártya 2496 aritmetikai egységet tartalmaz.

Típus:	GeForce GTX 570
Globális memória mérete:	1.34 Gb
Osztott memória mérete blokkonként:	49 Kb
Regiszterek mérete blokkonként:	32 Kb
Multiprocesszorok (MP) száma:	15
SP-k száma MP-nként (warp size):	32
Memória busz szélessége:	320 bit
Órajel:	1.56 GHz
Kernel eléri a CPU memóriában tárolt adatokat:	Igen
Egyidejű adatmásolás és kernelfuttatás:	Támogatva

2.4. táblázat. A számításokhoz használt GPU főbb paraméterei.

A hibaterjedés és a futásidő vizsgálatához használt egyenletrendszerek véletlenszám generátorral kerültek előállításra:

A mátrix és x vektor elemei véletlen számok [0,1)-ben, $b = Ax$. Az algoritmus bemenő paraméterként az A mátrixot és a b vektort kapta meg, míg az eredményül kapott x' vektor összevetésre került az eredeti x-el. A számítási eredményeket tartalmazó 2.5. táblázat 100-100 független futtatás eredményeit tartalmazza. Minden futtatásnál az x és x' vektorok elemeiből páronként különbséget képzünk, hibának legnagyobb abszolút értékű különbséget tekintjük. A 100 futtatás alapján átlagolt hiba a táblázat utolsó oszlopába került. Sokismeretlenes egyenletrendszerek együtthatómátrixai meghaladhatják a rendelkezésre álló GPU memóriáját. (lásd: 2.3.1.) Ebben az esetben az együtthatómátrix olvasható a CPU memóriából – de ennek a

kényszermegoldásnak természetesen időköltsége van. A táblázatban a 2-vel jelölt futásidő CPU memóriában tárolt együtthatómátrixszal értendő, míg az 1-gyel jelölt GPU memóriába másolt együtthatómátrixszal készült. Természetesen a másolásnak is van időköltsége. A 8192 ismeretlenes esetben az együtthatómátrix már nem fér el a GPU memóriában, ezért itt futásidő adat nem áll rendelkezésre.

n	Futásidő[1]	Futásidő[2]	Hibaátlag
2	6 ms	8 ms	0
4	8 ms	10 ms	7.81597E-14
8	10 ms	10 ms	2.13163E-14
16	16 ms	14 ms	1.00364E-13
32	24 ms	24 ms	1.82077E-13
64	46 ms	43 ms	7.43805E-12
128	100 ms	95 ms	6.81943E-12
256	245 ms	242 ms	7.41984E-12
512	886 ms	1 262 ms	4.81473E-11
1024	3.2 sec	4.6 sec	1.89602E-11
2048	14.7 sec	20.8 sec	8.06466E-13
4096	105.9 sec	191.4 sec	1.29308E-12
8192	–	23 min	2.35101E-12

2.5. táblázat. Számítási eredmények.

3. fejezet

Egy $O^*(n^4)$ algoritmus párhuzamos architektúrán konvex testek térfogatának kiszámítására

3.1. Konvex testek térfogatszámítása

Lovász László és Santosh Vempala nemrég közölt egy $O^*(n^4)$ műveletigényű algoritmust n dimenziós konvex testek térfogatának meghatározására. Az algoritmus több egymáshoz kapcsolódó, de kis mértékben eltérő szimulációs lépésből áll. Az első számítógépes implementáció lehetővé tette az algoritmus viselkedésének tanulmányozását numerikus szempontból, de a vizsgált dimenziók száma nem haladhatta meg a 10-et, és az eredmények szórása sem volt kielégítő. Ebben a részben az algoritmus egy masszívan párhuzamos GPU architektúrára optimalizált, módosított változata kerül bemutatásra, amely többféle módszert tartalmaz a variancia csökkentésére. A grafikus kártya több száz párhuzamosan működő számítóegységének kihasználásával olyan sebességnövekedést sikerült elérni, amellyel már a gyakorlatban is meghatározható 2 és 20 dimenzió közötti testek térfogata.

Konvex testek térfogatának polinomiális idő alatt történő kiszámítása régóta foglalkoztatja a matematikusokat. A 80-as években bizonyítást nyert, hogy nem létezik a problémára polinomiális idejű megoldás. Ennek hatására a figyelem a statisztikai módszereken alapuló térfogatbecslő eljárások felé fordult. Az áttörést Dyer, Frieze és Kannan (Dyer/Frieze/Kannan 1991) cikksorozata jelentette, amelyben olyan véletlenen alapuló algoritmusokat mutattak be, amelyek polinomiális időben becslik konvex testek térfogatát. A véletlenen alapuló algoritmusokat *randomizált módsze-*

*rek*nek nevezték. A 3.1. táblázat Simonovits (Simonovits 2003) nyomán mutatja be a randomizált algoritmusok történetét – Simonovics maga is társszerző számos, a témát érintő cikkben. A fokozatos fejlesztések eredményeképp Lovász, Simonovits, Kannan és Vempala (Kannan/Lovász/Simonovits 1997) a műveletigényt $O(n^{27})$-ről $O^*(n^4)$-re szorították le. Az eddigi legjobb eredmény Lovász és Vempala (Lovász/ Vempala 2003), (Lovász/Vempala J. of Computer and System Sciences 2006) két cikkében olvasható - a második írás az első javított és magyarázatokkal bővített változata.

A randomizált algoritmusok úgynevezett tagsági orákulumot használnak annak eldöntésére, hogy egy kérdéses pont a test belsejében helyezkedik el, vagy a testen kívül. Az egyes algoritmusok futásideje a térfogat meghatározásához szükséges orákulumhívások száma alapján kerül összehasonlításra. Az $O^*(n)$ kifejezésben n a test dimenzióinak számát jelöli, a $*$ pedig arra utal, hogy műveletigény meghatározásánál a logaritmikus faktorokat figyelmen kívül hagyjuk.

A Lovász-Vempala algoritmus (továbbiakban LV) a K konvex test $vol(K)$ térfogatát polinomiális idő alatt becsüli ϵ_{rel} hibahatáron belül, tetszőleges p_{rel} valószínűséggel.

Az LV algoritmus első számítógépes implementációját Lovász és Deák (Lovász/ Deák 2012) készítették el, amelynek segítségével 2 és 10 dimenzió közötti hiperkockák térfogatát számolták. A teljesítménynövelés érdekében az LV algoritmus néhány ponton módosításra került. Erre a módosított változatra a továbbiakban LVD rövidítéssel hivatkozunk. A variancia csökkentésére tett törekvések ellenére az eredmények pontossága nem érte el a várakozásokat. Számottevő pontosságnövekedés csak az algoritmus lépésszámának nagyságrendi növelésével érhető el, ehhez viszont a gyakorlati alkalmazhatóság érdekében a futásidőt vissza kell szorítani. A grafikus kártyák több száz párhuzamosan működő számítóegysége elméletileg két nagyságrendnyi sebességnövekedést tesz lehetővé, de az elvi teljesítménymaximum megközelítéséhez az algoritmus tervezésénél több kritériumot is figyelembe kell venni. (Lásd: 2.2. fejezet.) A GPU-k sajátosságaira való tekintettel az algoritmust több ponton módosítani kellett. Erre a változatra a továbbiakban PLVDM néven hivatkozunk. A rövidítésbe a „P” a párhuzamos megközelítésre utal, a többi betű az érintett szerzők neveire utal (Lovász, Vempala, Deák, Mohácsi). A bemutatott implementáció nVidia GTX 570 típusú grafikus kártyán került tesztelésre, amely 480 párhuzamosan működő számítóegységével középkategóriás eszköznek számít. PLVDM segítségével lehetőség nyílt az algoritmus több számítási példán történő futtatására, és ennek köszönhetően sajátosságainak mélyebb megismerésére.

A következő fejezetekben az algoritmussal kapcsolatos számítási problémák és megoldási javaslatok kerülnek bemutatásra, különös tekintettel az algoritmus párhuzamosítására, és a variancia csökkentésének érdekében tett módosításokra. Az LVD algoritmus elméleti matematikai hátterének részletes leírása megtalálható (Simonovits 2003) cikkében. Magával az algoritmussal a korábban említett (Lovász/ Vempala 2003), (Lovász/Vempala J. of Computer and System Sciences 2006) cikkek foglalkoznak mélyebben.

Az LV térfogatszámító algoritmus lényegében egy olyan ceruzaszerű test felett vett Monte-Carlo integrálokból álló sorozattal számol, amelynek alapja az a konvex test, amelynek a térfogatát meg kívánjuk határozni. Az eredmény a ceruza térfogata, melynek alapján elfogadás-elvetés technikával meghatározható az eredeti test térfogata. A véletlen-szám generálással és a különböző Monte-Carlo számításokkal a (Deák 1990), (Hammersley/Handscomb 1964) és (Devroye 1986) foglalkozik.

A Monte-Carlo integrálás alapvetően jól párhuzamosítható. Az elért sebességnövekedés lehetővé tette több varianciacsökkentő módszer vizsgálatát a gyakorlatban is, amelyek között az úgynevezett ortonormált becslővektorok módszere bizonyult a leghatékonyabbnak. A két nagyságrendnyi sebességnövekedésnek köszönhetően lehetőség nyílt 20 dimenziós testek térfogatának meghatározása.

A 3.2. fejezetben általános összefoglaló olvasható az 1990-2010 között kifejlesztett térfogatszámoló algoritmusokról és a Lovász–Vempala algoritmus mögött meghúzódó főbb gondolatokról. A 3.3. fejezet az LV algoritmust mélyebben mutatja be a matematikai jelölésrendszerrel együtt. A 3.5. fejezet a PLVDM implementáció felépítését tárgyalja, különös tekintettel a grafikus kártyák felépítéséből adódó megfontolásokra. Az ezt követő fejezet számítási eredményeket közöl, amelyek alapján meghatározásra kerültek az optimális futtatási paraméterek.

Az utolsó fejezetben bemutatásra kerül a térfogatszámítás alkalmazása az optimalizáció néhány területén (lásd (Lovász/Deák 2012), (Fábián 2013), (Romeijn/ Smith 1994), (Zverovich et al. 2012)).

3.2. Térfogatszámító algoritmusok története

Ebben a szakaszban rövid áttekintést adunk a korábbi véletlenen alapuló térfogatszámítási algoritmusokról ((Kannan/Lovász/Simonovits 1997), (Lovász/Vempala J. of Computer and System Sciences 2006)), amelyet a Lovász-Vempala algoritmus vázlatos leírása követ (Lovász/Vempala 2003). A Lovász-Vempala algoritmusnak az a

változata, amely alapján a Deák-féle LVD implementáció készült, a következő fejezetben kerül bemutatásra.

Szerzők	Megjelenés éve	Műveletigény
Dyer-Frieze-Kannan	1989	$O^*(n^{27})$
Lovász-Simonovits	1990	$O^*(n^{16})$
Applegate-Kannan	1990	$O^*(n^{10})$
Lovász	1991	$O^*(n^{10})$
Dyer-Frieze	1991	$O^*(n^8)$
Lovász-Simonovits	1992,93	$O^*(n^7)$
Kannan-Lovász-Simonovits	1997	$O^*(n^5)$
Lovász	1999	LV algoritmus
Kannan-Lovász	1999	
Lovász-Vempala	2002	
A.Kalai-Lovász-Vempala	2003	$O^*(n^4)$
Deák	2012	LVD algoritmus
Deák-Mohácsi	2014	PLVDM algoritmus

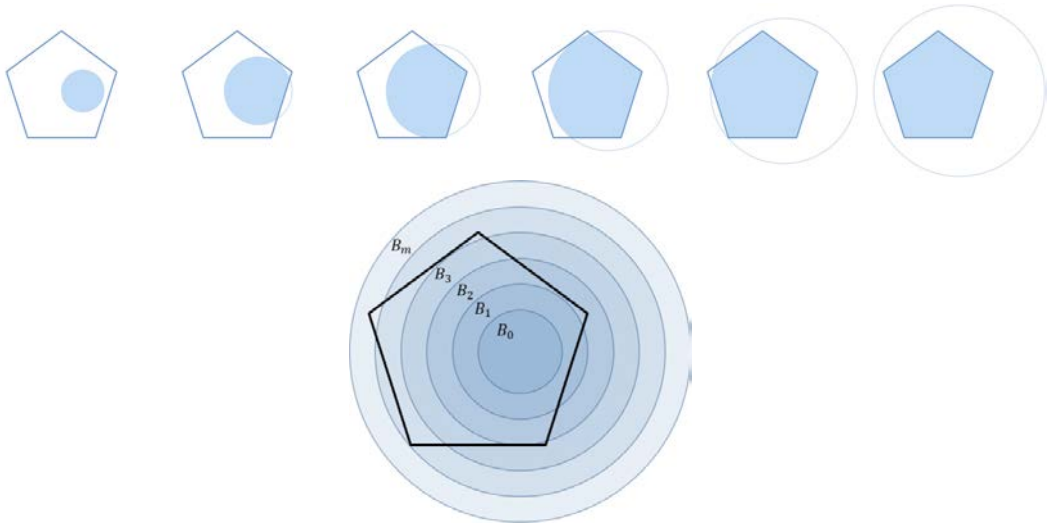
3.1. táblázat. Térfogatszámító algoritmusok története.

A randomizált térfogatszámító algoritmusokra általában jellemző, hogy csak izotropikus pozícióban elhelyezkedő, jól kerekített testek esetén működnek helyesen. Ezért a térfogatszámítás előtt minden konvex testnek át kell mennie egy előfeldolgozáson. Ennek eredményeképp

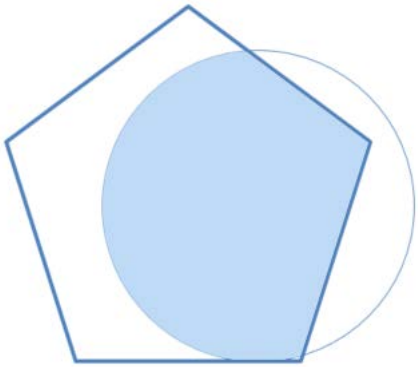
- a test tartalmaz egy B egységgömböt, és köré írható egy $O^*(\sqrt{n})$ sugarú gömb, miközben
- a test közel-izotropikus pozícióba kerül, azaz tömegközéppontja az origóba tolódik, és a második momentuma (megközelítőleg) az egységmátrix lesz.

Affin transzformációk sorozatával mindkét előbb említett feltétel biztosítható. Az izotropikus pozíció $O^*(n^4)$ lépésben érhető el (lásd (Rudelson 1999)). Maga a térfogatszámoló algoritmus ezen a standardizált konvex testen indítható csak el.

A 3.1. táblázatban szereplő algoritmusok alapelve megegyezik. Növekvő térfogatú, egymást tartalmazó testeket hoznak létre: $K_0 \subseteq K_1 \subseteq \cdots \subseteq K_m = K$, ahol $K_{i-1} \subseteq K_i$. Az első K_0 test egy olyan gömb, melyet K teljes egészben magában foglal, és amelynek ismert a térfogata: $K_0 = B_0$. K_m az utolsó konvex test, amelynek a térfogatát meg akarjuk határozni. A B_i gömbök és a $K_i = B_i \cap K$ testek növekvő halmazok. (Lásd 3.1. és 3.2. ábrák.) Elfogadás-elvetés módszerrel meghatározható az egymást követő konvex testek térfogatának $vol(K_i)/vol(K_{i-1})$ aránya. A térfogatarányok kiszámítását minden $i = 1, \ldots, m$-re el kell végezni. A $vol(K_i)/vol(K_{i-1})$



3.1. ábra. A K poliéder és $B_0, B_1, \ldots, B_m$ gömbök sorozata.



3.2. ábra. Gömb és a poliéder metszete $K_2 = K \cap B_2$.

arány meghatározása egy egyszerű Monte-Carlo számítás: egyenletes eloszlású véletlen pontokat generálunk a nagyobb test belsejében, és megszámoljuk, hogy ezek közül hány esik a kisebb test belsejébe. A test végső térfogatát úgy kapjuk, hogy az ismert $vol(K_0)$ térfogatot összeszorozzuk az összes $vol(K_{i+1})/vol(K_i)$ térfogatarány szorzatával:

$$vol(K) = vol(K_0) \prod_{i=0}^{m-1} \frac{vol(K_{i+1})}{vol(K_i)}. \tag{3.2.1}$$

Az ilyen elven működő algoritmusokat *randomizált algoritmus*oknak nevezzük, mivel véletlenen alapuló Monte-Carlo számítások alapján adnak becslést a térfogatra. (Annak ellenére, hogy a konvex testek felbontása determinisztikus.) A Lovász-Vempala

(LV) térfogatszámító algoritmus alapjául is a fenti gondolatmenet szolgál, de van néhány lényeges eltérés.

Abban a tekintetben, hogy a testet K_i testekre kell felbontani, a LV algoritmus alapötlete megegyezik a régivel, de a felbontást a véletlen felől közelíti – a test K_i testekre történő felbontásánál sztochasztikus módszert alkalmaz. A korábbi $K_0 \subseteq K_1 \subseteq \cdots \subseteq K_m = K$ felbontás determinisztikus módon történt, pontosan meghatározott K_i halmazokra, míg a Lovász-Vempala algoritmus esetén az integrálásra használt pontokat csak valamilyen 1-hez közeli valószínőséggel tartja K_i belsejében. A számítás *fázis*oknak nevezett lépésenként halad előre. Minden fázishoz egy R_i arány kerül számításra, mely hozzávetőleg a korábbi $vol(K_i)/vol(K_{i-1})$ aránynak felel meg. Így logkonkáv sűrűségfüggvényű véletlen pontok kerülnek generálásra. Az i-edik fázisban az e^{-x_0/T_i}-vel arányos sűrűségfüggvény a K_i test belsejében tartja a pontokat. (x_0 a véletlen pont nulladik koordinátája, $T_i > 0$ pedig egy i-vel növekvő skaláris paraméter.) Így a generált pontok fázisról fázisra felfúvódó felhőhöz hasonlítanak, amelyek végül majdnem teljesen egyenletes eloszlással töltik ki a test belsejét.

A Lovász-Vempala algoritmus másik meglepő tulajdonsága az, hogy nem közvetlenül a K test térfogatát határozza meg, hanem a problémát egy dimenzióval magasabb dimenziójú térbe transzponálja, ahol az eredeti K testet egy ceruzához hasonló K' testté terjeszti ki. Az algoritmus először a ceruza térfogatát határozza meg, majd ez alapján vezeti vissza az eredeti test $V(K)$ térfogatát.

A randomizált folyamathoz létre kell hozni logkonkáv függvények $f_0 \leq f_1 \leq \cdots \leq f_m$ sorozatát, amely függvények arányosak lesznek az aktuális ponthalmaz sűrűségfüggvényével. Az első (f_0) függvény integrálja (közelítéssel) könnyen meghatározható, az utolsó függvény integrálja pedig maga a keresett ceruza-térfogat. A korábbi algoritmusok a $vol(K_i)/vol(K_{i-1})$ arányt számolták, az LV algoritmus $R_i = \int_{K_{i-1}} f_{i-1} / \int_{K_i} f_i$ integrálok arányát számolja. Az $\int f_{i-1} / \int f_i$ arányokat egy olyan eloszlásból történő mintavételezéssel lehet becsülni, amelynek sűrűségfüggvénye arányos f_i-vel, az így generált pontokon számítjuk és átlagoljuk az $\int f_{i-1} / \int f_i$ értékeket. (A mintavételi pontok hit-and-run Monte-Carlo módszerrel kerülnek generálásra – nagyon hasonlóan a régebben használt módszerekhez.) Ha f_i indikátorfüggvénye lenne K_i-nek (ahol $f_i(\mathbf{x}) = 1$, ha $\mathbf{x} \in K_i$, és $f_i(\mathbf{x}) = 0$, ha $\mathbf{x} \notin K_i$), ugyanoda jutnánk, mint a régebbi determinisztikus testfelbontáson alapuló módszerek. Logkonkáv f_i függvények használatával a becslők varianciája csökkenthető, és ezáltal számítási munka takarítható meg.

A mintavétel ezekből a logkonkáv függvényekből mintavételi pontonként $O^*(n^3)$ orákulum-hívást igényel.

A pontszálak Markov-láncot alkotnak. Ha a Markov-láncot rögzített pontból indítjuk, a véletlen pontok előállítása nagyobb számítási teljesítményt igényel, mint a véletlen pontból történő úgynevezett *melegindítás*.

3.3. Az LVD algoritmus főbb lépései

Az LVD algoritmus vázlatos működése a következő: Tekintsünk egy $K \subset \mathbf{R}^n$ n-dimenziós konvex testet. Az algoritmus K test V térfogatát nem közvetlenül határozza meg, hanem a test dimenzióinak számát eggyel kiterjeszti, és a problémát ebben az $\mathbf{R}^{n+1}$ térben oldja meg. A V térfogat közvetlen meghatározása helyett először az $n' = (n+1)$-dimenziós ceruza-szerű K' test V' test térfogata kerül kiszámításra. Maga a térfogatszámítás többfázisú Monte-Carlo módszerrel történik. A ceruzának a véletlen bolyongásnál van jelentősége: a véletlen pontsorozat a ceruza hegyéből indul, és fázisról fázisra halad előre a ceruza tompa vége felé. Az LVD algoritmus egyetlen pontsorozatot használ, míg a PLVDM implementáció pontsorok ezreivel dolgozik. A pontok előrehaladását (robbanását) szemléltetik a 3.3.7. fejezet 3.5-3.8. ábrái. Egy kétdimenziós négyzeten végzett mintafuttatás fázisai végén a pontok eloszlását a C. melléklet ábrasorozata mutatja be. Utolsó lépésként egyszerű elfogadás-elvetés módszerrel kerül meghatározásra a ceruza V', és az ezt tartalmazó n'-dimenziós hasáb térfogatának aránya, amely alapján V' ismeretében V meghatározható.

3.3.1. Előfeltételek

Az algoritmus feltételezi, hogy K már keresztülment a következő előfeldolgozási lépéseken:

1. Ahhoz, hogy az algoritmus jól működjön, a konvex testet először izotropikus pozícióba kell helyezni. Ehhez a test $\mathbf{b}$ tömegközéppontját az origóba kell tolni. (Ha $\boldsymbol{\xi}$ egy egyenletes eloszlású vektor K-ban, akkor $\mathbf{E}\boldsymbol{\xi} = \mathbf{b} = \mathbf{0}$), valamint második momentumok várható értéke meg kell, hogy egyezzen az egységmátrixszal ($\mathbf{E}(\mathbf{b}-\boldsymbol{\xi})(\mathbf{b}-\boldsymbol{\xi})' = I$). Ezt a testen belül generált egyenletes eloszlású pontokkal és affin transzformációkkal lehet elérni Rudelson leírása szerint (Rudelson 1999).

2. A második teljesítendő előfeltétel szerint K testnek nem elég izotropikus pozícióban elhelyezkednie, de jól-kerekítettnek is kell lennie. Ez azt jelenti, hogy K tartalmaz egy B egységgömböt, valamint K köré is írható egy D sugarú gömb, ahol $D = O(\sqrt{n})$.

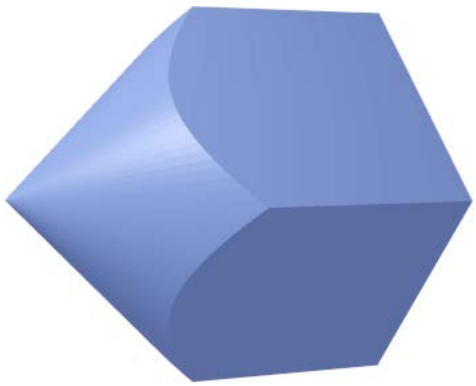
Ezeket a lépéseket mind az LVD, mind a PLVDM implementáció előfeltételként kezeli, gyakorlati megvalósításukra nem térek ki.

3.3.2. Konvex test leírása orákulum segítségével

Egy $K \subset \mathbf{R}^n$ konvex poliéder sokféleképp felírható. Megadható egyenlőtlenségrendszerrel, pontok által kifeszített konvex burokként, vagy határoló hipersíkokkal, stb. Az LVD algoritmus – mint a legtöbb randomizált térfogatszámító algoritmus – feltételezi, hogy a test egy úgynevezett tagsági orákulum segítségével van megadva. A K testhez tartozó tagsági orákulum egy $\mathbf{x}$ pontra IGEN választ ad, ha a pont a testen belül van, és NEM-mel válaszol, ha a pont a testen kívülre esik. A K test tagsági orákulumának ismeretében könnyen előállítható a ceruza-szerű K' test tagsági orákuluma – erről a következő fejezetben lesz szó. A tagsági orákulum fontos szerepet játszik az egész algoritmusban, mert az elméletileg szükséges számításigény az orákulum-hívások számában kerül kifejezésre.

3.3.3. A ceruza előállítása

Az algoritmus egyik kulcslépése a $K \in \mathbf{R}^n$ test kiterjesztése egy további dimenzióval. Ezt a további dimenziót a továbbiakban az $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_n)$ vektor x_0 eleme jelöli. Az x_0 változónak az integrálok meghatározásánál lesz fontos szerepe. Maga a K konvex test a további n koordinátán keresztül érhető el.



3.3. ábra. $n' = 3$ dimenziós ceruza – a ceruza alapja egy $n = 2$ dimenziós négyzet.

A ceruza előállításához vegyünk egy konvex kúpot:

$$C = \left\{ \mathbf{x} \mid \mathbf{x} \in \mathbf{R}^{n+1}, x_0 \geq 0, \sum_{i=0}^{n} x_i^2 \leq 2x_0^2 \right\}.$$

Az eredeti n-dimenziós konvex sokszög a terének egy további dimenzióval történő kiterjesztése után K egy $n' = (n+1)$-dimenziós hasábbá terjeszthető ki, melynek alapja az eredeti K konvex poliéder, magassága pedig $2D$ (D a K köré írható gömb sugara). A K' ceruza a hasáb és a kúp metszeteként adódik:

$$K' = C \cap [(0, 2D) \times K].$$

Mivel K jól kerekített, a ceruza is jól kerekített: tartalmaz egy egységsugarú egységgömböt, és köré írható egy $2D$ sugarú gömb. A ceruza hegyes vége kerül az origóba, a tompa végénél $x_0 = 2D$.

3.3.4. A paraméteres integrál

A térfogatszámítás az integrálás egy különleges esete. Korábbi cikkeiben Kannan (Kannan/Lovász/Simonovits 1997), valamint Lovász és Simonovits (Lovász/Simonovits Random Structures and Algorithms 1993) konvex testek sorozata fölött vett indikátorfüggvényeket integráltak. A Lovász-Vempla algoritmus fontosság szerinti mintavétellel integrál a K' konvex test fölött vett logkonkáv függvények szerint. A logkonkáv függvények alakja a következő:

$$f_i(\mathbf{x}) = e^{-a_i x_0},\ i = 0, 1, \ldots, m,\ \mathbf{x} = (x_0, x_1, \ldots, x_n) \in K' \subset \mathbf{R}^{n+1},$$

ahol $a_i, i = 0, 1, \ldots, m$ konstansok csökkenő sorozatát jelöli. Meg kell jegyezni, hogy az i-edik fázisban használt f_i arányos egy, a K' ceruza fölött értelmezett exponenciális eloszlás sűrűségfüggvényével. A K' testen belüli véletlen pontok ennek a sűrűségfüggvénynek megfelelően kerülnek generálásra. A sűrűség alakja $e^{-a_i x_0} / \int_{K'} e^{-a_i \xi_0} d\boldsymbol{\xi}, i = 0, 1, \ldots, m$, ahol $\boldsymbol{\xi} = (\xi_0, \xi_1, \ldots, \xi_n)$. (Pontosabban a sűrűségfüggvény véletlen metszete kerül csak felhasználásra, melyet egy véletlen szakasz x_0 tengelyen képzett vetülete jelöl ki, lásd: 3.3.7. fejezet.)

Az integrálás az i-edik fázisban a fázisra jellemző f_i függvény szerint történik. A függvényeket meghatározó a_i paramétereknek teljesíteniük kell a $a_0 \geq a_1 \geq \ldots \geq a_m > 0$ feltételt. Az f_i függvény μ_i mértéket generál, az f_i és f_{i+1} csak kis mértékben különböznek egymástól (az egymást követő μ_i és μ_{i+1} mértékek differenciájának L_2 normája aránylag kicsi (Lovász/Vempala J. of Computer and System Sciences 2006)). Az $f_0, f_1, \ldots, f_m$ függvények sorozata köti össze f_0-t f_m-el. f_0 integrálját K' felett könnyű meghatározni, f_m integrálja K' felett pedig a keresett $V' = vol(K')$ térfogat.

(f_m közelítőleg a konvex test indikátor függvénye, mivel ebben a fázisban a_m már közel nulla.)

Bevezetünk egy $a > 0$ paramétertől függő integrált:

$$Z(a) = \int_{K'} e^{-ax_0} d\mathbf{x},$$

ahol x_0 az $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_n)$ vektor első koordinátája. Az alábbiakban a $Z(a)$ integrál viselkedése kerül bemutatásra két esetben.

1. Az első esetben legyen a értéke "nagy". (Annyira, hogy az (3.3.1) egyenlet két oldala majdnem megegyezik.) Ha $a_0 \geq 6n$, az integrál kis hibától eltekintve megegyezik a teljes kúp felett vett integrállal:

$$Z_0 = Z(a_0) = \int_{K'} f_0(\mathbf{x}) d\,\mathbf{x} \leq \int_C e^{-a_0 x_0} d\,\mathbf{x} = n! \pi_n a_0^{-(n+1)}, \tag{3.3.1}$$

   ahol $\pi_n = \frac{2}{n} \frac{\pi^{n/2}}{\Gamma(n/2)}$ az n-dimenziós egység sugarú gömb térfogata. Az a_0 értékének megválasztásakor némiképp eltértünk az eredeti $a_0 = 2n$ javaslattól. Magasabb dimenziókban az egyenlőtlenség két oldala közti különbséget jelentősen csökkenti az $a_0 = 6n$ választás. Ennek megfelelően a következő megoldásnál maradtunk:

$$a_0 = \begin{cases} 6n & , n \leq 6, \\ 2n & , n > 6. \end{cases}$$

   A sorozat többi a_i konstans tagja a_0-ból származtatható a következő összefüggéssel: $a_i = a_0 \left(1 - \frac{1}{\sqrt{n}}\right)^i, i = 1, \ldots, m.$

2. A másik megvizsgálandó eset a kis értéke mellett tapasztalható. Az utolsó fázisban, ahol $a_m \leq \epsilon_{\text{rel}}^2 / D$, az integrál

$$Z_m = Z(a_m) = \int_{K'} f_m(\mathbf{x}) d\mathbf{x},$$

   mely közelítőleg a K' test térfogatával egyezik meg, hiszen az integrandus majdnem konstans. K' indikátorfüggvényének integrálja V'. Így egyetlen integrálás helyett $Z(a_0), Z(a_1), \ldots, Z(a_m)$ integrálok sorozatát kell kiszámolni, amelyek közül az első egyszerűen számolható, az utolsó pedig a test V' térfogatát közelíti. A továbbiakban a $Z_i = Z(a_i)$ jelölést is használjuk.

3.3.5. A ceruza térfogatának meghatározása fázisonként

Az előző pontban bevezetett $Z(a_i)$ integrálokból kifejezhető $Z(a_m)$, mely közelítőleg megegyezik a ceruza térfogatával. $Z(a_m)$ kifejezhető $R_i = Z(a_{i+1})/Z(a_i), i = 0, 1, \ldots, m-1$ arányok szorzataként:

$$Z(a_m) = Z(a_0) \prod_{i=0}^{m-1} \frac{Z(a_{i+1})}{Z(a_i)}. \tag{3.3.2}$$

Ezek az R_i arányok emlékeztetnek a 3.2. fejezetben bemutatott régebbi algoritmusok "determinisztikusan felbontott" testeinek $vol(K_i)/vol(K_{i-1})$ térfogatarányaira. Az i-edik fázisban meghatározzuk az R_i arányt, amely becsülhető egy olyan eloszlásból történő mintavételezéssel, amelynek sűrűségfüggvénye arányos f_i-vel. Az R_i arány meghatározásához a Lovász-Vempala lemma (Lovász/Vempala J. of Computer and System Sciences 2006) jelenti a kulcsot.

Lemma: Legyen $\boldsymbol{\xi} = (\xi_0, \xi_1, \ldots, \xi_n)$ véletlen vektor μ_i mértékkel, és $\eta = e^{(a_i - a_{i+1})\xi_0}$, ekkor $\mathbf{E}(\eta) = \frac{Z(a_{i+1})}{Z(a_i)}$.

Ezek szerint ahhoz, hogy megbecsüljük az R_i arányt, olyan $\mathbf{x}^{(j)} = (x_0^{(j)}, x_1^{(j)}, \ldots, x_n^{(j)})$, $j = 1, \ldots, k$ mintákat kell generálni K'-ben, melyek sűrűsége arányos f_i-vel, majd mintaátlagot kell számolni a következő képlet szerint:

$$W_i = \frac{1}{k} \sum_{j=1}^{k} e^{(a_i - a_{i+1})x_0^{(j)}}, \tag{3.3.3}$$

amely torzítatlan becslése R_i-nek; $R_i = E(W_i)$. Megfelelően nagy k elemszámú minta esetén nagy valószínűséggel a $\left|W_i - \frac{Z(a_{i+1})}{Z(a_i)}\right|$ hiba tetszőlegesen kicsi lesz.

Az integrálásnak ezt a módszerét, amely egymást követő fázisokban generált pontokat vesz alapul, tekinthetjük a szimulált hűtés ellentettjének. A pontok felhőben helyezkednek el, és fázisonként egyre nagyobb sebességre tesznek szert, így terjednek a ceruza tompa vége felé, amíg el nem érik az egyenletes eloszlást a ceruza belsejében. (A szimulált hűtés során a pontok az algoritmus végére megdermednek (Metropolis et al. 1953)). Ezt a viselkedést a C. melléklet ábrái szemléltetik. Ezek alapján a generált pontok legelső koordinátái segítségével kerültek kiszámításra a fenti összegben szereplő $e^{(a_i - a_{i+1})x_0^{(j)}}$ értékek.

3.3.6. A szál kezdeti pontjának meghatározása

Azokat a pontsorozatokat, amelyek felett kiszámoljuk a Z_i integrálokat, pont-szálnak hívjuk. A pont-szálak Markov-láncokat alkotnak. A kérdés az, hogy milyen mód-

szerrel generáljuk a szál első (a számításban Z_0 meghatározását szolgáló) pontját a $n' = (n+1)$-dimenziós K' test belsejében. Ha a szálakat fix pontból indítjuk, $O^*(n^4)$ lépésre van szükség ahhoz, hogy elérjük a megfelelő véletlen eloszlást. (Ez a szám a Markov-lánc keveredési ideje.) Az úgynevezett melegindítás a szálat egy véletlen pontból indítja, és csak $O^*(n^3)$ az időköltsége, ezért emellett a megoldás mellett döntöttünk. A véletlen kezdőpontok, amellett hogy a keveredési időt alacsonyan tartják, biztosítják a szálak függetlenségét.

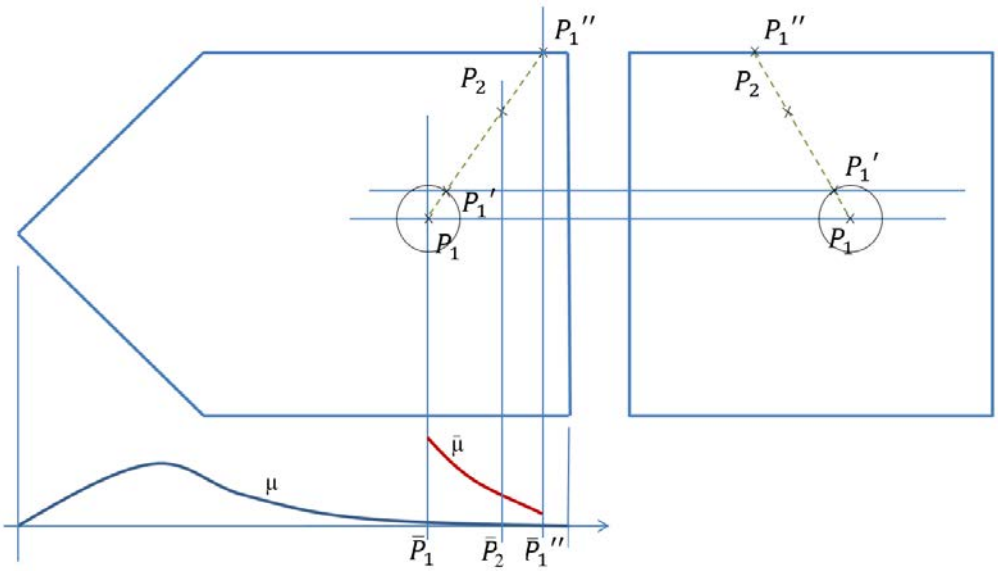
Mivel R_0 becslésére úgyis csak a generált pont első koordinátáját használjuk fel, az egész problémát felfoghatjuk az $e^{(a_0-a_1)x_0}$ függvény egydimenziós integráljaként. Ezeket a kezdeti pontokat egy olyan sűrűségfüggvény szerint generáljuk, amely arányos az $e^{-a_0x_0}$, $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_n) \in K'$ függvénnyel. (Ez egy χ^2 sűrűségfüggvény $2(n+1)$ szabadságfokkal.) Az ezen sűrűségfüggvény által generált valószínűségi mérték μ_0, és a pontok eloszlásfüggvényét $\overline{F}(x)$ jelöli. Annak érdekében, hogy az összes pontot K'-n belül tartsuk, minden pontot a kúpban generálunk, majd eldobjuk azokat, amelyek K'-n kívül esnek. A megmaradt pontok első koordinátájának sűrűségfüggvénye $\overline{\overline{F}}$ lesz. Az integrálási probléma rövid formában a következőképp összegezhető:

$$R_0 = \frac{Z(a_1)}{Z(a_0)} = \int_{K'} e^{(a_0-a_1)x_0} d\mu_0 = \int_0^{2D} e^{(a_0-a_1)x_0} d\overline{\overline{F}}(x_0) = \int_{v=0}^{1} e^{(a_0-a_1)\overline{\overline{F}}^{-1}(v)} dv. \tag{3.3.4}$$

Az integrálás hatékonyságának növelésére R_0 kiszámításához rétegzett mintavételi módszert használtunk, melynek részletes leírása (Lovász/Deák 2012)-ban található. A χ^2 eloszlású kezdeti pontokat a [0,1) intervallum nem egyenlő hosszúságú részekre való felbontásával generáltuk, ahol minden részben külön-külön inverziós módszert használtunk. Ezzel az ötlettel az összes fázis közül R_0 integrálása a leggyorsabb, és egyben a legpontosabb is.

3.3.7. Véletlen pontok generálása a K' ceruzában – az alapmódszer

A pontok generálására szolgáló módszer μ_i mértékkel a R. L. Smith (Smith 1996) és E. Romeijn (Romeijn/Smith 1994) által publikált „hit-and-run" módszeren alapul ($c_i f_i(\mathbf{x})$ sűrűséggel konvex test belsejében, ahol c_i egy megfelelően választott konstans). Később Lovász megmutatta, hogy az ezzel az eljárással generált pontok gyorsan keverednek – tetszőleges kezdeti pontból kiindulva a generált pontok eloszlása viszonylag kevés munkával stacionárius lesz, $O^*(n^3)$ idő alatt, és ez az elérhető leg-



3.4. ábra. Mintavétel az oldal- illetve felülnézetben ábrázolt ceruzában. A bal-alsó sarokban μ_i mérték sűrűsége, mely fölött a diagramon a $[\overline{P}_1, \overline{P}_1'']$ intervallumra csonkolt μ_i látható. A kiinduló pont P_1, melyből P_2-be lépünk tovább.

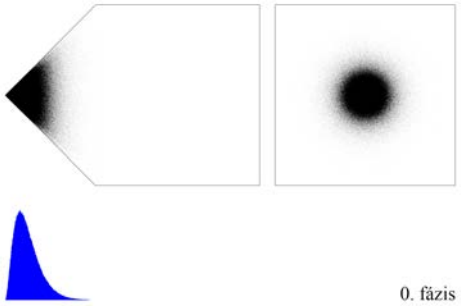
jobb korlát n-ben és D-ben a keveredési időre. A „gyorsan keveredik" megfogalmazás csak elméleti összefüggésben igaz, a keveredés jelenleg elérhető felső korlátja nagyságrendileg 10^{10}. Ez a pontgenerálási technika és az általa n-dimenzióban is elérhető sebesség kulcsfontosságú az algoritmus gyakorlati implementációjában. Elsőként a véletlen pontok generálására szolgáló módszert mutatjuk be, majd ismertetünk néhány továbbgondolt változatot a variancia csökkentésére.

A K' testen belül a pontokat f_i-vel arányos sűrűségben állítjuk elő.

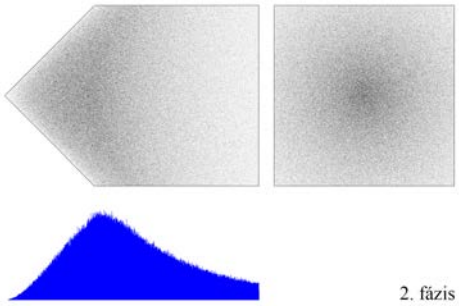
Egy pont-szálon belül az egymást követő pontok szakaszonként kerülnek generálásra. A pont-szál μ_0 mértékű eloszlással generált $\mathbf{x}^0$ pontból indul. Ezt követően minden $\mathbf{x}^1, \mathbf{x}^2, \ldots \mathbf{x}^m$ szakaszra adott a jellemző $\mu_1, \mu_2, \ldots, \mu_m$ mérték, amely meghatározza a pontok eloszlását az adott szakaszban.

Tekintsünk egy f_i-vel arányos sűrűségfüggvényt, amelyhez tartozó mérték μ_i. A véletlen pontok generálása K'-ban μ_i mérték szerint egy módosított „hit-and-run" technikával történik (Lásd: 3.4. ábra):

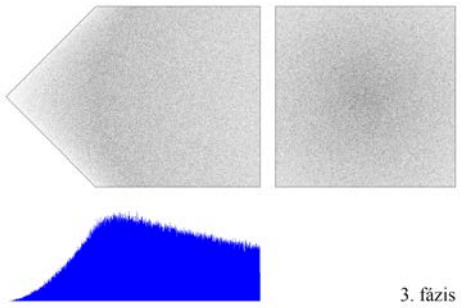
1. A $P_1 = \mathbf{x} \in K'$ pontból kiindulva generálunk egy véletlen irányt. Az irányt egy P_1 középpontú egységgömb felszínén egyenletes eloszlással választott pont jelöli ki (Deák Problems of Control and Information Theory 1979).



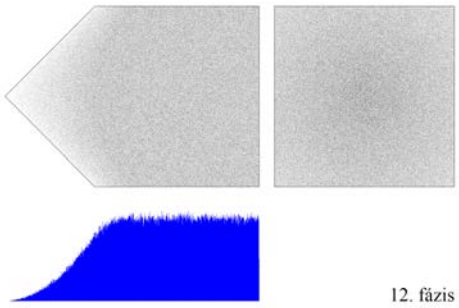
3.5. ábra. Szálak kezdeti pontjai a felül- illetve oldalnézetből ábrázolt ceruzában; a bal-alsó sarokban lévő diagram a pontok empirikus eloszlását mutatja.



3.6. ábra. Mintavételi pontok elhelyezkedése a 2. fázis végén.



3.7. ábra. Mintavételi pontok elhelyezkedése a 3. fázis végén.



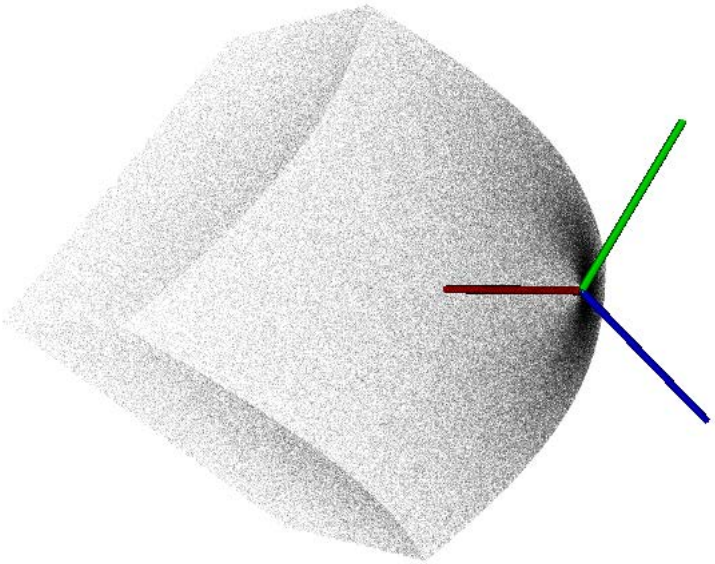
3.8. ábra. Mintavételi pontok elhelyezkedése az utolsó fázis végén. A pontok térbeli eloszlása a ceruzában közel egyenletes.

2. Az $\mathbf{x}$-ből indulva felveszünk egy félegyenest az előbb generált irány mentén, majd meghatározzuk a ceruza felszínének P_1'' metszéspontját a félegyenessel.

3. Meghatározzuk P_1 és P_1'' merőleges vetületét az x_0 tengelyen, melyet $[\overline{P}_1, \overline{P}_1'']$-vel jelölünk.

4. Az $[\overline{P}_1, \overline{P}_1'']$ intervallumra csonkolt, fázisra jellemző μ_i mérték mellett pontot generálunk, melyet visszavetítünk a $[P_1, P_1'']$ szakaszra. A visszavetítéssel kapott pont $P_2 = \mathbf{y}$.

A $P_1, P_1', \overline{P}_1'', \overline{P}_2$ pontok nyomon követhetők az ábrán. A pontsorozatot generáló algoritmus $\mathbf{x}$ bemeneti pontból, $\mathbf{y}$ kimeneti pontot állítja elő. A léptető algoritmus $S(\mathbf{x}, \mathbf{y}, a_i)$ alakú - ismételt alkalmazása révén jön létre a pont-szál. A 3.9. ábra egy kétdimenziós négyzet feletti ceruza felületén keletkezett P_n'' pontok térbeli elhelyezkedését mutatja. A próbafuttatás során nyert pontok kirajzolják a ceruza felszínét.



3.9. ábra. Kétdimenziós négyzet feletti ceruza felületén keletkezett P_n' pontok a térben.

Ha $\mathbf{x}$ μ_i eloszlását követi, akkor $\mathbf{y}$ is μ_i eloszlású lesz, de $\mathbf{x}$ és $\mathbf{y}$ nem lesznek függetlenek. Ahhoz, hogy kvázi-független pontsorozatot kapjunk, $S(\mathbf{x}, \mathbf{y}, a_i)$-t többször kellene meghívni a pontok léptetésére, mielőtt egy pontot felhasználnánk a tényleges integráláshoz. Az LVD algoritmus csak egy pontszálat léptetett. Ezért az LVD nem használt fel minden pontot az integráláshoz, hanem egy fázisonként változó alacsony értéknek megfelelő számú pontot kihagyott minden integrálás után. Az LVD és a PLVDM közti egyik lényeges különbség az, hogy a PLVDM több-ezer pont-szálat léptet. Ebből adódóan nem kell, hogy kihagyjon pontokat az integrálások között, mert a pontok száma már olyan magas, hogy a kihagyott pont helyén (vagy legalábbis közvetlen közelében) hamarosan úgyis keletkezne egy másik pont.

A számítógépes kísérletek alátámasztották ezt a feltevést: több pont-szál használata hozzájárult a generált pontok függetlenségének biztosításához.

Igaz, más okból, de bizonyos pontok kihagyása a PLVDM algoritmus esetén sem elkerülhető. Minden fázis elején szükség van egy valahány lépésből álló várakozásra, amely alatt a pontok elérik a fázisra jellemző μ_i stacionárius eloszlást. Ennek a szakasznak a hossza a *keveredési idő*, amely alatt a pontokat csak továbbléptetjük, de az integrál számításához nem használjuk fel. A keveredési idő meghatározásával a következő pont foglalkozik részletesen.

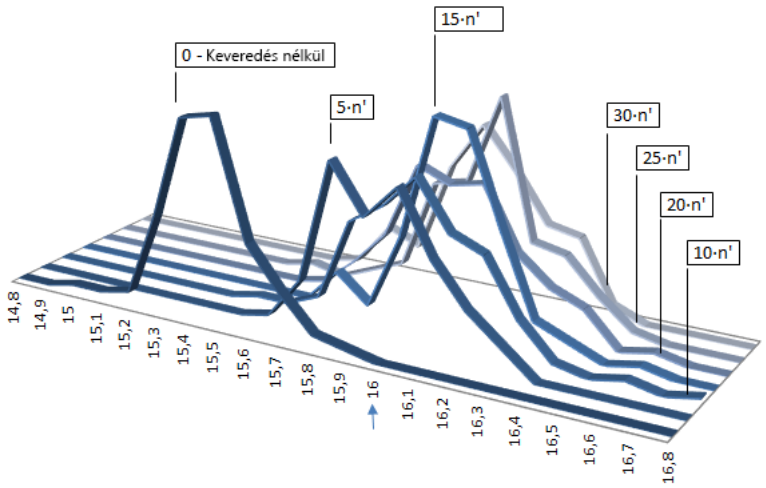
3.3.8. A Markov-lánc keveredési ideje

Az előző számítógépes megvalósítás (Lovász/Deák 2012). cikk állítása szerint adaptív szabály alapján fázisonként csökkentette $d_{i,I}$ keveredési konstans értékét. Ennek megfelelően minden fázisra külön keveredési idő került meghatározásra - a szükséges keveredési idők fázisról fázisra csökkentek. A nagyságrendek érzékeltetéséhez érdemes megjegyezni, hogy $n' = 3$ dimenzió esetén $d_{i,I} = 25 - 13$ közti értékek adódtak - a legnagyobb az első fázisban, a legkisebb az utolsóban. $n' = 5$ dimenzió esetén már $d_{i,I} = 273 - 35$, $n' = 9$ dimenzióban pedig $d_{i,I} = 133 - 22$. Természetesen $d_{i,I}$ meghatározásánál két, egymásnak ellentmondó szempontot kell figyelembe venni. A 10^{10} nagyságrendű elméleti felső korlát közelében nagyon jó a keveredés, ugyanakkor az algoritmus a gyakorlatban is elfogadható futásidejének érdekében a keveredési időt le kell szorítani.

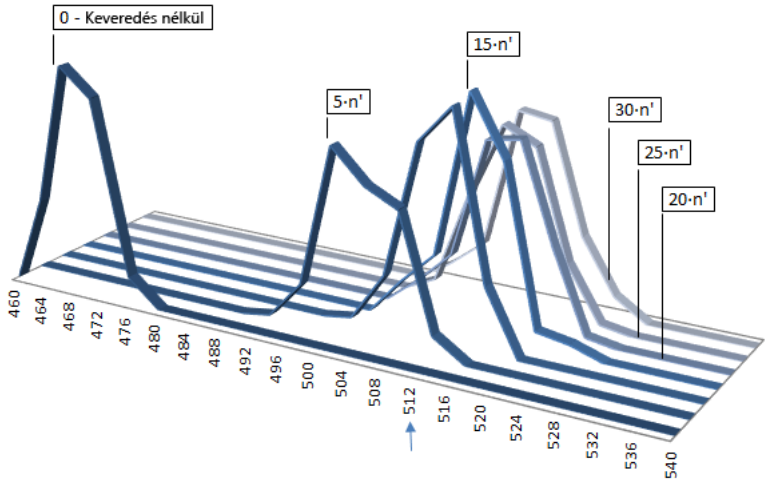
A párhuzamos implementáció futtatásához minden fázisra $d_{i,I} = 15n'$ többé-kevésbé kielégítő értéket választottuk. A döntés kísérleti futtatások eredményei alapján született. Annak érdekében, hogy találjunk egy többé-kevésbé megfelelő értéket, két egység oldalhosszúságú hiperkockákon végeztünk futtatásokat olyan keveredési idők mellett, mint $d_{i,I} = 0, 5n', 10n', \ldots$. Minden paraméterérték mellett 100 független futtatást végeztünk, majd ezek eredményei alapján empirikus sűrűségfüggvényeket rajzoltunk fel. Az $n = 4$-dimenziós kísérletsorozat eredményeit az alábbi 3.10 és 3.11 ábrák mutatják, a hiperkocka térfogatának pontos értéke ebben az esetben 16. Annak érdekében, hogy a különböző paraméterek melletti futásidők megegyezzenek, az egyes fázisok lépésszámán nem változtattunk, viszont minden fázis elején a paraméternek megfelelő számú lépést kihagytunk az integrálszámításból. A diagramról leolvasható, hogy a keveredési idő nélkül, illetve túl alacsony keveredési idővel futtatott kísérletek várható értéke a test tényleges térfogata alatt marad. Túl magas paraméterérték mellett a várható érték nem romlik ugyan, de az eredmények szórása megnő. Ez nem meglepő, hiszen minél nagyobb a keveredési idő, annál kevesebb pont alapján kerül kiszámításra az integrál. A tapasztalatok azt mutatják, hogy $d_{i,I} = 15n'$ jó választás, az ennél nagyobb értékek csak pazarolják a számítókapacitást. Az $n' = 5$-dimenziós futtatássorozat hasonló eredményt hozott, ebben az esetben a hiperkocka pontos térfogata 512. Az $n = 3, 8, 15$ dimenziós testeken végzett futtatások hasonló eredményt hoztak.

A $15n'$ paraméter érték egybevág a korábbi kísérletek tapasztalataival, annak ellenére, hogy azok más módszerrel jutottak hasonló eredményre.

Magasabb dimenziók esetén, mint az $n' = 20$ dimenziós futtatás, a keveredési időt sajnos $d_{i,I} = 30n', 60n'$ értékre kellett emelni, hogy elfogadható eredményt kapjunk.



3.10. ábra. $n' = 5$ dimenziós feladat: különböző keveredési idők mellett kapott eredmények eloszlása.
A vizsgált keveredési-idő tartomány a keveredési idő elhagyásától $30n'$-ig terjed. Az $n = 4$ kockához tartozó helyes érték 16, melyet az ábrán nyíl jelöl.



3.11. ábra. $n' = 10$ dimenziós feladat: különböző keveredési idők mellett kapott eredmények eloszlása.
A vizsgált keveredési idő tartomány a keveredési idő elhagyásától $30n'$-ig terjed. Az $n = 10$ kockához tartozó helyes érték 512, melyet az ábrán nyíl jelöl.

3.4. Mintavételezés egyszerű és dupla pontos módszerrel

Lépésenként egy mintavételi pont - a durva becslő

A generált pontok alapján a Lovász-lemma szerint W_i mintaátlagot számolunk, amelyet R_i becslésére használunk. $R_1, \ldots, R_{n-1}$ becslése az alábbi átlag kiszámításával történik:

$$\Theta_1 = \frac{1}{s}\sum_{j=1}^{s} \exp\{(a_i - a_{i-1})x_0^j\}, \tag{3.4.1}$$

ahol x_0^j a pont-szál j-edik $\mathbf{x}^j \in K'$ pontjának nulladik koordinátáját jelöli, mely az $f_i(\mathbf{x})$ függvénnyel arányos sűrűséggel kerül mintavételezésre. Ez a megközelítés az úgynevezett *durva becslő*, amelyet Θ_1 jelöl.

Lépésenként két mintavételi pont - a dupla-pontos becslő

Mivel az $\exp\{(a_i - a_{i-1})x_0^j\}$ K'-ben konvex, különböző varianciacsökkentő eljárások vethetők be. A PLVDM implementáció két megoldást tartalmaz a variancia csökkentésére: az első az úgynevezett dupla-pontos módszer, a másik az ortonormált irányvektorok módszere (lásd: következő szakasz). A két módszer együttesen is alkalmazható.

A fent bemutatott *durva* becslési technika kézenfekvő kiterjesztése az úgynevezett *dupla-pontos becslő*. A dupla-pontos módszernél a félegyenest tükrözzük a pontra, és az így kapott félegyenesre is számolunk becslést. (Ez gyakorlatilag azt jelenti, hogy nemcsak a félegyenest rajzoljuk meg a ceruzában, hanem a két félegyenesből álló teljes egyenest, majd mindkét félegyenesre elvégezzük a becslést.) A dupla-pontos becslő formális leírása a következő:

$$\Theta_2 = \frac{1}{2s}\sum_{j=1}^{s} [\exp\{(a_i - a_{i-1})e_{1,j}\} + \exp\{(a_i - a_{i-1})e_{2,j}\}] \tag{3.4.2}$$

Természetesen mindkét becslő torzítatlan becslést ad az integrálok kiszámítására.

3.4.1. Variancia-csökkentő módosítás – ortonormált vektorok

A többdimenziós normális eloszlás eloszlásfüggvényének kiszámítására több speciális integrálási technikát fejlesztettek ki. Ezt a technikát ortonormalizált becsléseknek hívták (Deák IIE Transactions (Operations Engineering) 2002), (Gassmann/Deák/ Szántai 2002) – vagy más cikkekben ezt irány-menti integrálásnak is nevezték, amit sikeresen lehetett szóráscsökkentésre használni még 1000 dimenzióban is (Deák Central European Journal of Operations Research 2011).

Az LVDM implementáció tartalmaz néhány másik módszert a variancia csökkentésére. Az $\mathbf{x}$ pontból kiinduló egyetlen véletlen irányvektor helyett tekintsük az irányvektorok $U = \{\mathbf{u}^\ell\}_{\ell=0}^n$ halmazát, ahol az $\mathbf{u}^\ell$ vektorok egységnyi hosszúak és ortogonálisak egymásra. Az előző fejezetben bemutatott dupla-pontos becslési módszert az

összes irányra alkalmazva $2(n+1)$ mintához jutunk, melyeket $e^{\ell}_{1,j}, e^{\ell}_{2,j}$-nek jelölünk. Az *ortonormált becslő*, mely U vektorhalmaz s realizációja alapján ad becslést R_i értékére, a következőképp írható fel:

$$\Theta_3 = \frac{1}{2(n+1)s} \sum_{j=1}^{s} \sum_{\ell=0}^{n} \exp\{(a_i - a_{i-1})e^{\ell}_{1,j}\} + \exp\{(a_i - a_{i-1})e^{\ell}_{2,j}\} \tag{3.4.3}$$

Ezt az ortonormált becslőt O1 jelöléssel láttuk el a táblázatokban és a programkódban is. (Az O az ortonormáltságra utal, az 1-es szám pedig arra, hogy közvetlenül az ortonormált vektorrendszer elemeit használjuk a becsléshez.) Az O1 becslő $2(n+1)$ pontot illetve függvényértéket generál, mely pontok egyenletesen szóródnak az egység-hipergömb felszínén. A módszer hozzájárul a variancia csökkentéséhez. Ortonormált pontrendszer használata hipergömb feletti integrálása hasonlít ahhoz az esethez, amikor egyenlő távolságú pontrendszert (vagy ponthálót) használunk szakasz vagy háromszög feletti integráláshoz.

Egy másik, kissé kifinomultabb módszer a fentiek szerint generált ortogonális vektorrendszer összes lehetséges párjának összegét használja a becsléshez, ami $n(n-1)$ vektort jelent, amelyek *"még egyenletesebben"* fedik le a gömb felszínét.

3.4.2. Utolsó lépés: K konvex test V térfogatának meghatározása

Az algoritmus utolsó lépése a test keresett V térfogatának meghatározása a ceruza V' térfogata alapján. (Ez a befejező lépés azután következik, miután már meghatároztuk V' értékét.) Ehhez először szükség van a ceruza és a $(0, 2D) \times K$ hasáb térfogatarányára. A térfogatarány meghatározásához egyenletes eloszlású pontokra van szükség a ceruzát magában foglaló hasábon belül.

A PLVDM algoritmus esetében ez a gyakorlatban úgy valósul meg, hogy az utolsó fázis után folytatjuk a pontszálakat, és f_m-el arányos sűrűséggel további pontokat generálunk. Ezen pontok utolsó n koordinátája már (majdnem) egyenletes eloszlást mutat K'-ben. Ezután a pontok nulladik koordinátáit lecseréljük egy $(0, 2D)$-ben egyenletes eloszlással generált értékre. Az így kapott, a hasáb belsejében egyenletes eloszlású pontok alapján egyszerű elfogadás-elvetés módszerrel meghatározzuk a ceruza és a hasáb r térfogatarányát. A hasáb térfogata $2DV$. Mivel $r = V'/2DV$, a következő összefüggés alapján adódik

$$V = \frac{V'}{2rD},$$

mely a konvex test keresett térfogata.

3.4.3. Hibabecslés

Ebben a szakaszban elsőként az LVD algoritmusban használt lépésről lépésre történő hibabecslésről lesz szó, amelyet a PLVDM algoritmusnál használt megoldás követ.

A hibabecslésre azért van szükség, hogy meg tudjuk határozni az algoritmus különböző részeinél szükséges minta elemszámát. A hibaelemzés két részből áll. Elsőként az R_i arányok szorzatának hibájának becslésére kerül sor, a második rész az r arány értékelésével foglalkozik. A számítások során szerzett gyakorlati tapasztalatok azt mutatják, hogy a második rész hibája sokkal kisebb, mint az első részé, így csak ezt fejtjük ki részletesen.

Az R_i becslése $W_i = R_i + \varepsilon_i$, $i = 0, 1, \ldots, m-1$, ahol a becslés torzítatlan (az ε_i véletlen hibára $\mathbf{E}(\varepsilon_i) = 0$), és $\sigma_i^2 = \mathbf{D}^2(\varepsilon_i) = \mathbf{D}^2(W_i)$. Jelölje az $R_0 R_1 \cdots R_{m-1}$ szorzatot Z, becslését pedig Z_m, a $\sim$ jelölés a közelítő egyenlőségre utal.

A Z szorzat becsléséhez tartozó $\varepsilon_{V'}$ hiba a következő képletből számítható:

$$Z_m = Z(a_0)\Pi_{i=1}^{m} W_i = Z(a_0)\Pi_{i=0}^{m-1}(R_i + \varepsilon_i),$$

amiből a hiba első rendű közelítéssel kerül meghatározásra:

$$\varepsilon_{V'} = Z_m - Z \sim Z(a_0) Z \sum_{i=0}^{m-1} \frac{\varepsilon_i}{R_i}.$$

A végső hiba durva közelítésére a fenti szorzat szórásának háromszorosát vesszük, mely a $V^{'}$ térfogat meghatározásánál felső hibakorlátként tekinthető:

$$\delta_{V'} = 3Z(a_0) Z \sqrt{\sum_{i=0}^{m-1} \frac{\mathbf{D}^2(W_i)}{W_i^2}}. \tag{3.4.4}$$

Ez a teljes hiba a dimenziószám növekedésével együtt nő (a többi paraméter változatlan értéke mellett). Ezenfelül konkrét konvex test térfogatának számításakor a legnagyobb hiba az első fázisban tapasztalható, mely gyorsan csökken az egymást követő további fázisok során:

$$\frac{\mathbf{D}^2(W_i)}{W_i^2} \geq \frac{\mathbf{D}^2(W_{i+1})}{W_{i+1}^2}, i = 1, 2, \ldots, m.$$

A PLVDM implementációban elhagyhattuk a fentiekben leírt hosszadalmas számításokat. A csökkenő futásidő lehetővé tette, hogy az algoritmust többször (20-100) futtassuk, és a szórást a kapott eredmények alapján határozzunk meg. Ennek a megközelítésnek viszont az a hibája, hogy minden egyes fázisban ugyanannyi lépést teszünk, ami hatásfokcsökkenéssel jár.

3.5. Megvalósítás és számítási eredmények

3.5.1. Az algoritmus leírása

A PLVDM térfogatszámító algoritmus implementációja a következő részekből épül fel:

1. Próbatest előállítása.

2. Kiinduló pontok generálása a ceruza hegyében a „melegindításhoz". Ebben a lépésben készül el minden pont-szál első pontja, valamint itt kerülnek inicializálásra a pontszálakhoz tartozó Mersenne-Twister véletlenszám generátorok.

3. Kalibráció: annak meghatározása, hogy hány fázisban történjen az integrálás. Az egyes fázisokban számított integrálok térfogatarányok, amelyek a fázisok előrehaladásával egyhez tartanak: $\lim_{i\to\infty} \int_{K_{i-1}} f_{i-1} / \int_{K_i} f_i = 1$. Ebben a lépésben alacsony szál- és lépésszámmal egy próbaintegrálás sorozat történik, amely alapján meghatározható, hogy az egymást követő integrálok aránya hol csökken egy paraméterként megadható határérték alá. Így dől el, hogy a tényleges számítás hány fázisban történjen.

4. Táblázatok előkészítése a részeredmények tárolására. A futtatás több szálon, az előző lépésben meghatározott fázisszámmal történik. Miután pont-szál minden fázis végén visszaad egy integrál értéket, amelyet táblázatban kell tárolni a végső összesítésig.

5. Első integrálok kiszámítása és rögzítése a részeredmény táblában. Az első integrálok meghatározása a többiektől eltérő módszerrel történik, így ezt a lépést külön függvény valósítja meg.

6. További integrálok meghatározása fázisonként.

(a) n-dimenziós gömb felületén egyenletes eloszlású pont generálása, amely irányvektorként kerül felhasználásra.

(b) Félegyenes és a ceruza-szerű K' test metszéspontjának meghatározása.

(c) Véletlen értékek generálása exponenciális eloszlás alapján.

(d) Véletlen szám generálása csonkolt exponenciális sűrűséggel a pont továbblépteté séhez.

(e) Becslés a (4.1.1) képlet alapján.

7. K' ceruza V' térfogatának meghatározása a részeredmény táblázat alapján.

8. A ceruza és a hasáb térfogatarányának meghatározása elfogadás-elvetés módszerrel.

9. A hasáb térfogata alapján K konvex $vol(K)$ térfogatának meghatározása.

Egy futtatás során az (a)-(e) lépések 10^{10} nagyságrendben kerülnek végrehajtásra. A B melléklet különböző dimenziókban, oszloponként eltérő paraméterek mellett indított 100 független futtatás eredményeit tartalmazza a kísérletek szemléltetésére. A számíráshoz használt módszer oszlopról oszlopra változik. Az oszlopok fejlécében szereplő rövidítések a számítási módszerre utalnak, amelyek leírása a 3.5.3 fejezetben található. A próbatest, amelyre a számításokat elvégeztük, minden esetben két egység oldalhosszúságú hiperkocka volt.

Az eredmények összesítése és kiértékelése a későbbi alfejezetekben olvasható. További táblázatok és leírások, valamint az algoritmus forráskódja letölthető a

$$http://web.uni-corvinus.hu/lmohacs/plvdm/$$

címről. A honlapon található táblázatok sokkal részletesebbek, több dimenzióban különböző paraméterek mellett történt futtatássorozatok eredményei letölthetők. Itt csak az eredmények kivonata kerül bemutatásra.

3.5.2. Az orákulum

Lovász László eredeti cikkében az orákulum nagyon egyszerű volt (Lovász/Simonovits 1992). Numerikus példaként hiperkockákat használt, melyek középpontja az origóba esett, és határsíkjai párhuzamosak voltak a koordinátatengelyek által kifeszített hipersíkokkal. Ebből adódóan az orákulumnak az $|x_i| \leq 1$, $i = 1, \ldots, n$ egyenlőtlenségek teljesülését

kellett ellenőriznie annak eldöntésére, hogy egy pont a testen belül helyezkedik el, vagy sem.

Az eredeti algoritmus a félegyenes metszéspontját a ceruza felszínével intervallumfelezéses módszerrel határozta meg egy előre beállított hibaküszöbön belül. A test átmérője ismert, így a félegyenes mentén felezett lépéshosszokkal lépkedve a hibahatártól függő maximális lépésszámmal megtalálható a metszéspont.

Ez a megközelítés GPU architektúrán nem optimális. A GPU akkor teljesít jól, ha minden szálon pontosan ugyanazokat a műveleteket kell végrehajtani, csak más-más adatokon. Az eltérő lépésszámú ciklusok és a feltételes elágazások jelentősen lerontják a hatásfokot. Ennek megfelelően az orákulum helyett a masszívan párhuzamos architektúrára tervezett algoritmusváltozat más megközelítést használ.

A konvex test a testet határoló hipersíkok halmazaként, illetve félterek metszeteként is megadható. A félterek metszete egyenlőtlenség-rendszerként is felírható, illetve az egyes hipersíkok megadhatók egyenletekként. A mi megközelítésünkben a metszéspont meghatározása a következő módon történik:

- Feltételezhetjük, hogy P_n a test belsejében van.
- P_n pontból kiindulva P'_n ponton át félegyenst állítunk (lásd: 3.4. ábra).
- Egyesével kiszámoljuk a hipersíkok és félegyenes metszéspontját, majd meghatározzuk a P_n pont és a P''_n metszéspont távolságát.
- Lesznek olyan hipersíkok, amelyeket nem metsz a félegyenes, ebben az esetben a hipersíkhoz tartozó távolságot végtelennek tekintjük.
- A keresett metszéspont a legkisebb távolsághoz tartozó metszéspont lesz.

A dimenziószám növelés után ceruzává kiterjesztett test esetében a metszéspont meghatározásánál a hipersíkokon felül a félegyenes és a kúp metszéspontját is figyelembe kell venni.

Az algoritmust n-dimenziós hiperkockán vizsgáltuk, de természetesen tetszőleges hipersíkokkal határolt test vizsgálható. A kocka csúcsai $\mathbf{v} = (\delta_1, \delta_2, \ldots, \delta_n)$ $\delta_i = 1$ illetve $\delta_i = -1$ minden lehetséges kombinációjára. Az implementáció tartalmaz egy függvényt az n-dimenziós hiperkocka illetve az $n+1$-dimenziós ceruza előállítására.

A ceruza és a hasáb térfogatarányának meghatározása elfogadás-elvetés módszerrel történik, így az orákulumot is el kell készíteni a testhez. Mivel itt minden pont vizsgálatakor csak egyetlen orákulumhívás történik, az orákulumhívások eltérő száma nem rontja le a hatásfokot. Az orákulum nem hagyható ki teljes mértékben az

	$n=2$	$n=5$	$n=8$	
$n'=n+1$	3	6	9	
k_{thr}	25	25	25	
V'	8.268	101.7	1052.5	
$\varepsilon_{V'}$	0.43E-1	0.39E+1	0.22E+2	
r	0.729	0.718	0.715	
számított$V$	4.011	31.67	260.0	
pontos V	4.000	32.00	256.0	
ε_V	0.66E-1	0.55E+1	0.31E+2	
futásidő (sec)	807	1901	7551	

3.2. táblázat. Eredmények három, kockából készült ceruzához P_3, P_6, P_9 (Forrás: (Lovász/Deák 2012)).

algoritmusból. A kezdeti pontok generálásánál is szükség van rá: a testen kívülre eső pontokat el kell dobni.

A dupla-pontosságú aritmetika számábrázolási hibája miatt néhány ritka esetben a metszéspont a testen kívülre kerül, ami tönkreteszi a futási eredményt. Ezeket az eseteket kiszűri a program, és a sérült pontokat eldobja. A jelenség főként magasabb dimenziókban ($n > 15$) fordul elő, ezen belül is leggyakrabban az első fázisban, ahol a pontok még a ceruza csúcsában sűrűsödnek. Annak, hogy számábrázolási hiba folytán a második fázis után kerüljön testen kívülre egy pont, a tapasztalatok szerint nagyon kicsi az esélye. 10^{10} nagyságrendű pontléptetésnél néhány hibás pont eldobásának a végeredmény szempontjából nincs jelentősége.

A Deák-féle első számítógépes implementáció eredményeit összehasonlításul a 3.2. táblázat tartalmazza.

3.5.3. A PLVDM algoritmusban és a táblázatokban használt jelölések

A K konvex test, amelynek térfogatát meg akarjuk határozni n dimenziós, a K' ceruza dimenzióinak száma pedig $n' = n + 1$. A programban és a táblázatokban, ha külön nincs jelölve, dimenzió alatt n'-t értjük, mivel az algoritmus idejének legnagyobb részében a ceruzával dolgozik. Az eredeti dimenziószám csak a futtatás legvégén, az eredmények összesítésénél kerül elő újból.

Itt k_i az egy szálon, fázisonként végzett lépések számát jelöli. A kísérletsorozatban k_i értéke minden fázisban megegyezik, de a magasabb fázisokban lehetne az értékét akár tizedére vagy századára csökkenteni. Ez egy lehetséges fejlesztési irány.

N_p a pontszálak számát jelöli, amíg M a GPU-n egy időben futtatható szálak számát. Az kísérletekhez használt nVidia GTX 570 típusú grafikus kártya $M = 480$ utasításfeldolgozó egységgel rendelkezik, azaz egy időben ennyi szálat tud párhuzamosan futtatni. N_p természetesen lehet nagyobb, mint M, de optimális kihasználtság akkor érhető el, ha $N_G = \ell M$, ahol ℓ egy egész szám.

Az alábbiakban a különböző integrálási pont generáló módszerek jelölése kerül felsorolásra. A jelölések mellett az eredeti publikációban és a programban is használt angol nyelvű elnevezések is szerepelnek. Mint az a korábbi leírásokból is kiderült, a pontsorozat egy új elemének generálása több lépésben történik:

1. Véletlen irányvektor generálása és metszéspont számítása.

2. Új pont generálása az eredeti pont és a metszéspont között.

Irányvektorok generálására két új módszert is bevezet a PLVDM algoritmus:

- C Durva módszer (Crude method): Az eredeti megközelítés, a durva módszer csak egy irányvektort generál minden lépésben.

- O1 Ortogonális módszer (Orthogonal method): n' vektorból álló ortogonális vektorrendszer kerül generálásra Gram-Smith módszerrel, ahol n' a ceruza dimenziószáma. A pont-szál továbbléptetéséhez csak a vektorrendszer első eleme kerül felhasználásra, a többi vektorra az integrálszámításhoz van csak szükség. Ugyanez igaz a következő módszerre is.

- O2 Ez a módszer is egy n' elemű ortogonális vektorrendszer előállításával kezdődik, de itt nem magukat az ortogonális vektorokat használjuk, hanem azok összes lehetséges páronként képzett összegét. Ennek eredményeképp $n'(n'-1)$ irányvektor születik, amelyeket mind fel lehet használni az integrálszámításhoz.

A generáló eljárás másik része a pontgeneráló eljárás. Ha adott a félegyenes, két pontgeneráló módszer van:

- S Egypontos mintavételezés (Single point). A félegyenes mentén csak egyetlen pont kerül generálásra az integrálszámításhoz.

- D Kétpontos mintavételezés (Double point). A félegyenes és annak ellentettje is felhasználásra kerül az integrálszámításhoz. (Ez a módszer két pontot generál egy egyenesen az integrálszámításhoz.)

A háromféle iránygeneráló és a kétféle pontgeneráló módszer tetszőleges kombinációban használható, így hatféle módszert kellett kipróbálni a gyakorlatban.

A paraméterek hatása a következős példán követhető: egy $n' = 10$ dimenziós test térfogatának meghatározására az O2 D módszert használjuk. A szálak száma $N_p = 6400$, a szálankénti lépésszám $k_i = 6000$ fázisonként. Az O2 módszerrel 10 dimenzióban $10 \times 9/2$ irányt generálunk, melyen a dupla pontos (D) módszerrel irányonként kép pontot választunk. A számítás 32 fázison keresztül tart, ami azt jelenti, hogy összesen $10 \times 9 \times\ 6400 \times 6000 \times 32$, nagyságrendileg 10^{10} pontot generáltunk a teljes számítás alatt. A keveredési idő $15 \times n' = 150$, így minden fázis elején a pont-szál első 150 pontja eldobásra kerül, azaz nem számít bele az integrálba, csak a továbblépéshez kell.

3.5.4. Számítási eredmények

A 3.3. és a 3.4. táblázatok 100 futtatás eredményeit összegzi 5 illetve 9 dimenziós hiperkockán. A 3.5. táblázatban adatai 10 futtatás alapján készültek 19 dimenziós kockán.

Számítási módszer	C S	C D	O1 S	O1 D	O2 S	O2 D
Szálak száma N_p	6400	6400	6400	6400	6400	6400
lépések száma k_i	6000	6000	3000	3000	600	600
100 futtatás eredményének átlaga	31.98	31.98	31.98	31.98	31.98	31.98
Eredmények szórása	0.05	0.02	0.02	0.03	0.04	0.03
Egy futtatáshoz szükséges idő (sec)	74	180	159	179	90	90
Hatásfok	1.0	2.11	2.17	2.11	1.83	1.83

3.3. táblázat. 100 futtatás eredményének összesítése 5 dimenziós kockán. A pontos térfogat 32.

A táblázatok első sora tartalmazza az irányvektor– illetve pontgeneráláshoz használt módszer megnevezését az előző fejezetek alapján. A második sorba került a

Számítási módszer	C S	C D	O1 S	O1 D	O2 S	O2 D
Szálak száma N_p	6400	6400	6400	6400	6400	6400
lépések száma k_i	6000	6000	6000	6000	600	600
100 futtatás eredményének átlaga	511.56	511.78	513.10	512.88	512.37	512.30
Eredmények szórása	1.59	1.40	1.49	1.25	1.58	1.33
Egy futtatáshoz szüks. idő (sec)	214	230	1750	1889	617	675
Hatásfok	1.00	0.18	0.14	0.18	0.35	0.45

3.4. táblázat. 100 futtatás eredményének összesítése 9 dimenziós kockán. A pontos térfogat 512.

Számítási módszer	C S
Szálak száma N_p	6240
lépések száma k_i	6000
10 futtatás eredményének átlaga	500359
Eredmények szórása	19085
Egy futtatáshoz szükséges idő (sec)	989
Hatásfok	1.0

3.5. táblázat. 10 futtatás eredményének összesítése $n = 19$ dimenziós kockán. A pontos térfogat$1024 \times 512 = 524288$.

pontszálak száma, melyet a fázisonkénti lépésszám követ. Az átlag és a szórás 100 illetve 10 azonos paraméterek mellett végrehajtott futtatás eredményéből került számításra. Az egyes futtatások futási ideje csak nagyon kis mértékben tér el egymástól, ezért csak a futásidők átlaga került feltüntetésre a táblázatokban.

Példaként a 3.3. táblázatban szereplő C D oszlopa szerint egy futtatás szórása 0.05. Így a 100 futtatás eredményének átlaga közelítőleg 0.015 hibával terhelt, mely a Csebisev tétel szerint nagy valószínűséggel az „Eredmények szórása" tized részének háromszorosaként adódik. Az utolsó (O2 D) oszlopban szereplő 15.99-es érték hibája 95%-os valószínűséggel 0.003.

A különböző szimulációs módszerek összehasonlítására bevezetett mérőszám a *hatékonyság* (Deák 1990), (Deák Central European Journal of Operations Research 2011), (Hammersley/Handscomb 1964). Tegyük fel, hogy a viszonyítási alapként választott C S módszer futtatásához t_1 idő szükséges, az ismételt futtatások eredményeinek szórása σ_1. A másik módszer futásideje t_2 az eredmények σ_2 szórása mellett. A második módszer elsőhöz mért hatékonysága a következő összefüggéssel adható meg:

$$\text{Hatásfok} = \frac{t_1\sigma_1^2}{t_2\sigma_2^2} \tag{3.5.1}$$

Viszonyítási alapként minden táblázatban a C S módszer szerepel. (Egy irányvektoron egy pont.) Kísérleteinkben a különböző módszerek hatékonyságára általában 1 és 10 közé eső értékek jöttek ki, de találkozunk 1 alatti értékkel is. Ez messze elmaradt a várakozástól. A jelenség hátterében feltehetőleg az áll, hogy az algoritmus legnagyobb számításigényű lépése a véletlen szám generálás, a varianciacsökkentő eljárások pedig az egy integrál előállításához szükséges véletlen számok tekintetében nem érnek el csökkenést. A tapasztalatok elemzése további vizsgálatokat igényel.

3.6. Következtetések

A párhuzamos PLVDM és az egy processzorra tervezett LVD algoritmus összehasonlításánál a futásidő mellett az eredmények hibáját is figyelembe kell venni. A szekvenciális LVD algoritmusnak 1900 másodpercre volt szüksége ahhoz, hogy kiszámolja az ötdimenziós esetre a 31.67 ± 5.5 értéket. Az $n = 9$ dimenziós példánál a 260 ± 31 eredményre 7551 másodpercet kellett várni (a 3.2. táblázat alapján). Fontos megjegyezni, hogy az algoritmus által adott hibabecslés egy nagyságrenddel felülbecsüli az empirikus hibát, amely az első esetben 1-nek, a másodikban 3-nak adódott, ami nagyvonalú közelítéssel a szórás háromszorosának felel meg.

Az összehasonlításhoz használt 4-dimenziós példa értékei a 3.3. táblázat „C D" oszlopából származnak. A számításhoz szükséges idő ± 0.15 hibával 60 másodperc volt, míg a 9 dimenziós esetre az eredmény ± 4.20 hiba mellett 230 másodperc alatt született meg. (Lásd: 3.4. táblázat „C D" oszlopa.)

A (3.11) egyenlet szerint a párhuzamos implementáció hatásfoka az előző, egy processzorra tervezett változathoz képest a következőképp alakul:

$$\frac{1900 \times 1.0^2}{60 \times 0.15^2} \sim 1400 \quad \text{és} \quad \frac{7551 \times 12^2}{230 \times 4.2^2} \sim 260 \tag{3.6.1}$$

Az első eredmény némiképp torz, mert a párhuzamos futtatás eggyel alacsonyabb dimenzió mellett történt. Az LVD változat esetén nem állt rendelkezésre megfelelő mennyiségű futtatási eredmény a szórás meghatározásához. Emellett a táblázatban szereplő értékek a legjobban teljesítő esetekből származnak. Ha a párhuzamos számítások eredményeiből a legrosszabbul teljesítőket vesszük, akkor a megfelelő sebességnövekedések körülbelül egy-ötödnek adódtak, ami egy tizede az előbbi becsléseknek, szám szerint 140-280 és 26-50.

A fentiek alapján megállapítható, hogy a párhuzamos változat a szekvenciálishoz képest két nagyságrendnyi sebességnövekedést ért el - ami körülbelül megegyezik azzal az értékkel, amit egy 480 számítóegységgel rendelkező grafikus kártyától vártunk. Ez nagy vonalakban azt jelenti, hogy százszoros sebességnövekedést sikerült elérni.

A sebességnövekedés folytán lehetőség nyílt arra, hogy kísérletezzünk különböző irányvektor és pontgenerálási módszerekkel. A rétegzéses módszer az első integrálok meghatározásánál és néhány ellentétes mintavételi megoldás a Monte-Carlo számításoknál még mindig elengedhetetlen ahhoz, hogy elfogadható időn belül jussunk eredményhez, de ezek hatása korántsem akkora, mint azt az előző cikk feltételezte (Lovász/Deák 2012).

A megnövelt futási sebesség lehetővé tette a hit-and-run Markov-lánc optimális keveredési idejének empirikus meghatározását, amely az eredeti várakozásoknál nagyobbnak adódott, de sokkal kisebb, mint az elvi felső határ ($\approx 10^{10}$).

A kísérleti futtatások során a dimenziószám növelésével egyre több pont-szál esetén adódott végtelennek az integrál. $n' = 20$ dimenzióban már a pontszálak 90%-a végtelen értékkel tért vissza. A nemvárt jelenség hátterében szintén a duplapontosságú aritmetika számábrázolási hibája áll. Minél magasabb a dimenziószám, annál nagyobb a valószínűsége annak, hogy az x_0 tengelyre merőleges irányvektort generáljunk. (Matematikailag pontosan merőleges irányvektor generálására végtelen kicsi az esély, de a számábrázolási hiba miatt a jelenség aránylag gyakran bekövetkezik.) A 3.3.7. fejezetben tárgyalt pont-visszavetítési algoritmus x_0-ra merőleges irányvektor esetén nullával való osztáshoz vezet, ebből adódik a végtelen integrál. A jelenség korrigálására az algoritmust úgy módosítottuk, hogy azokat az irányvektorokat, melyre végtelen W_i érték jön ki, nem vesszük figyelembe az összeg meghatározásánál. A probléma megoldását a nagyobb számábrázolási pontosság jelentené, amit viszont a GPU nem támogat. Két dupla pontosságú szám összekapcsolásából szervezett változótípus pedig nagyon lerontaná a hatásfokot. (Az nVidia kártyák *compute capability 2.0* előtti sorozatai nem támogatták a dupla pontosságú számábrázolást sem, ezeknél két egyszeres pontosságú *(float)* típusú változó használatával lehetett próbálkozni.) Ezért úgy tűnik, hogy a GPU-ra tervezett implementáció alkalmazhatóságának felső határa $n' = 20$ dimenzió. A magasabb dimenziókban történő futtatáshoz a párhuzamos implementációt számítógép-klaszterre kéne alkalmazni, mely a jövőben tervezett fejlesztések iránya.

A jövőben érdemes kísérletezni más sokdimenziós testekkel is, mint a szimplex, melynek szintén rendelkezésre áll a generátor algoritmusa és a térfogata tetszőleges dimenzióban.

4. fejezet

A nyugdíj-előreszámítás támogatása mikroszimulációs eljárással

4.1. Demográfiai előreszámítások

A magyarországi nyugdíjak előrejelzése egyike a legfontosabb társadalompolitikai kérdéseknek. Az 1997. évi LXXXI. törvénnyel bevezetett – és azóta néhány részletében többször módosított – jelenlegi nyugdíjrendszer fenntarthatósága, igazságossága, áttekinthetősége az elkövetkező évtizedek egyik legfontosabb társadalmi, gazdasági kérdése. Világosan kell látni a demográfiai folyamatokat, ismerni kell a várható élettartamot, hogy eldönthessük, hány embernek és milyen hosszan nyújt szolgáltatást a nyugdíjrendszer (E. 2010). A megoldásokhoz nélkülözhetetlen a nyugdíjrendszerrel összefüggésben álló számos hatásvizsgálat.

A nyugdíjszámításhoz legfontosabb információ az, hogy egy adott évben hány nyugdíjas van, azoknak milyen a nyugdíjeloszlásuk, illetve az adott évben hányan lesznek nyugdíjasok, és hányan hagyják el a nyugdíjat – hányan halnak meg. Az úgynevezett halálozási mutató, a várható élettartam és az egyes koronkénti eloszlás fog választ adni a nyugdíjban érdekelt személyek számára.

A demográfiai előreszámításokhoz kétféle megközelítéssel foglalkoztam dolgozatomban. Röviden bemutatom a kohorsz-komponens módszert, de elsősorban a mikroszimulációs megközelítést mutatom be közelebbről – a születés és halál előrejelzését dolgoztam ki részletesen – mivel a nyugdíj előszámításokhoz nem elég a makro szintű megközelítés. Igen fontos a nyugdíjasoknál és a nyugdíjba vonulóknál azok neme, iskolai végzettsége, a nyugdíjazáskor elért jövedelme, stb.

A mikroszimulációs módszertan, mely – mint nevéből is következik – nem csoportokkal számol, hanem az egyének sorsát egyesével követi. Az egyéneket tetszőleges számú tulajdonsággal jellemezhetjük, valamint az egyének közti családi kapcsolat is modellezhető. A jövedelem továbbvezetés család szinten történik, hiszen a költekezési és megtakarítási szokásokat a család összjövedelme és az eltartottak száma döntően befolyásolja.

A feldolgozandó rekordok nagy száma, és a minden rekordon azonos feladatokat végrehajtó algoritmusok miatt programozástechnikai szempontból a mikroszimuláció jól párhuzamosítható. A rendelkezésre álló számítóegységek számától függően oszthatjuk részekre az adatállományunkat, és a részeken egyszerre végeztethetjük el a számítási feladatokat.

Feltételezésem az volt, hogy megfelelő programozástechnikai eszközök segítségével építhető olyan közgazdászok által is könnyen használható mikroszimulációs keretrendszer, mely személyi számítógépen futtatva is néhány percen belül eredményre jut – megadva a lehetőséget az elemző közgazdászoknak a modellezésre. A disszertáció negyedik része a feltételezés igazolására készített szoftveres megoldásomat mutatja be. Ugyancsak bemutatásra kerülnek a fejlesztés során figyelembe vett követelmények, és a döntési helyzetekben alkalmazott megfontolások.

Mikroszimulációs szolgáltató rendszert már korábban is fejlesztettek (J. 2001; J./C. 2003) egyetemi környezetben. Munkám során követtem az ott kialakított metódusokat, viszont szállító-független eszközt fejlesztettem ki, mely használja a párhuzamos programozás lehetőségeit – hatékonnyá téve a modellek futtatását.

A kidolgozott keretrendszerrel az elemző közgazdász önállóan végezheti el becslési feladatait, leírhatja a feldolgozandó adatállományait, karbantarthatja a becslésekhez szükséges paramétertáblázatokat, vezérelheti a mikroszimulációs futtatásokat, illetve futtathatja az eredmények értékelésére szolgáló előredefiniált lekérdezéseket. A keretrendszer segítségével létrejöhet az a munkamegosztás, ahol a közgazdászok definiálhatják a futtatandó algoritmusokat (jövedelem továbbvezetés, intézményes továbbírás a nyugdíjba menetelkor, stb.), és informatikusok kifejlesztik az algoritmusokat megvalósító számítógépes alkalmazást, melyet ismét a végfelhasználó tesztel, illetve futtatja a különböző változatokat a döntések előkészítésnek érdekében.

Célom egy olyan mikroszimulációs keretrendszer létrehozása volt, mely a párhuzamos programozási technikák alkalmazásán keresztül személyi számítógépen is képes sok adattal dolgozó mikroszimulációk tervezésére és futtatására. A keretrendszer működésének bemutatására a nyugdíjrendszer modellezéséhez szükséges előreszámítások egyikét, a népesség előreszámításokat választottam. Ezek a tényezők döntően be-

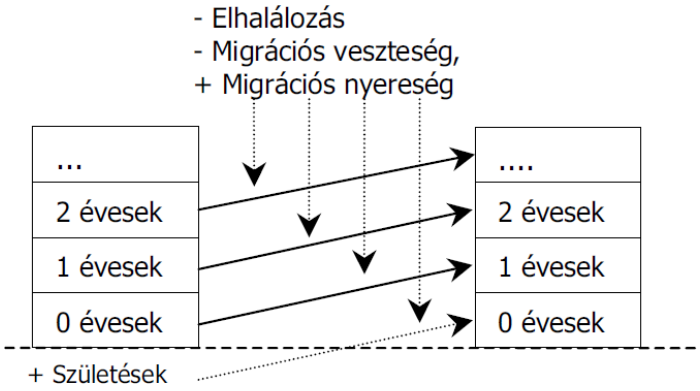
folyásolják – egyebek mellett – a mindenkori potenciális járulékfizetők, a majdani nyugdíjvárományosok számát és a munkaerő kínálati oldalt. A kialakított keretrendszer a nyugdíjrendszerekhez kapcsolódó más tényezők számításaira is alkalmazható. Felhasználható már kialakított modellek továbbfejlesztéséhez, illetve meglévők folyamatos karbantartására, javítására is.

4.2. Szimulációs megközelítések

4.2.1. A kohorsz-komponens módszer

A nemzetközi gyakorlatban a népesség előreszámítás kohorsz-komponens módszerrel készül – ezt követi a magyar gyakorlat is. Az eljárás reprodukálja az utánpótlási folyamatokat, pontosan modellezi a születéseket, halálozásokat, vándorlásokat, az idő előrehaladását a népesség életkorában.

Hablicsek László 2007-ben állított össze Magyarországra demográfiai előszámítást a Nyugdíj és Időskor Kerekasztal megbízásából (L. 2007), ez 2012-ben került frissítésre. (Az előszámítás hat tématerületre terjedt ki: népesség-előreszámítás, iskolázottsági előreszámítás, munkaerőkínálati előreszámítás, családi állapot szerinti előreszámítás, roma résznépesség előrebecslése, valamint a fogyatékkal élő résznépesség előrebecslése.) A népesség-előreszámítás alapjául a kohorsz-komponens, más néven alkotóelem-módszer szolgált, mely hosszú múltra tekint vissza a népességkutatásban – alapjait Pascal K. Whelpton fektette le 1928-ban. A kohorsz-komponens módszer a populációt csoportokba sorolja, a csoportokat tagjai számával jellemzi. Demográfiai számításoknál a népességet nem és életkor mentén lehet csoportokra osztani. A módszer egyes csoportok létszámát éves lépésekben, statisztikai adatok alapján összeállított táblázatok alapján módosítja. Például a halálozási statisztikák alapján összeállított aránytáblázat segítségéve kerül meghatározásra, hogy a 65 éves férfiak csoportjából hányan jutnak el a 66 éves férfiak csoportjába. Hasonló módszerrel kerül kiszámításra a különböző korú nők szülési arányai alapján az újszülöttek száma. Ahogy az elnevezés is utal rá, a módszer nem tesz különbséget csoportokba tartozó egyedek között. Ebből adódik előnye, a kis számításigény. Viszont olyan további jellemzők figyelembe vétele, mint a családi állapot, ugrásszerűen megnöveli a csoportok számát, és a számítás bonyolultságát. (A módszer vázaltos bemutatására a következő fejezetben kerül sor.)



4.1. ábra. A kohorsz-komponens módszer logikája (T./I. 2007).

Jelölje a teljes lakosságot t naptári évben $\bar{P}^t$ vektor, melynek $\bar{P}^t_x$ eleme az x éves népesség létszáma az év elején. A teljes népesség a férfiak és nők létszámából adódik: $\bar{P}^t = \bar{P}^{t,f} + \bar{P}^{t,n}$, ahol az f illetve az n jelzi, hogy nőkről vagy férfiakról van szó.

A népesség alakulásának előrebecslésénél számolni kell a születésekkel, a halálozással illetve a ki- és bevándorlással:

- Az $\bar{m}^{t,B}$ vektor az élveszülési arányszámokat tartalmazza, $\bar{m}^{t,B}_x$ eleme azt mutatja meg, hogy t évben az x éves nők hány százaléka szül gyermeket. Ez alapján az élveszülések száma t évben a $B^t = \sum \bar{m}^{t,B}_x \cdot \bar{P}^{t,n}$ összefüggéssel becsülhető. Kb. 105-106 fiúszülés esik 100 lányra, így a születések nemek szerint bonthatók.
- A $\bar{Q}^t$ vektor az elhalálozási valószínűségeket tartalmazza.
- A $\bar{VE}^t$ vektor a vándorlási egyenlegeket írja le a be- illetve kivándoroltak számának különbségeként.

A következő év eleji népesség az életkorokra előállítható a halálozási valószínűség, a vándorlási egyenlegek és az év eleji népesség ismeretében:

$$\bar{P}^{t+1}_{x+1} = \left(\bar{P}^t_x - \frac{1}{2}\bar{VE}^t_x\right)\left(1 - \bar{Q}^t_x\right),\ x \neq 0$$

$$\bar{P}^{t+1}_0 = \left(B^t - \frac{1}{2}\bar{VE}^t_0\right)\left(1 - \bar{Q}^t_0\right)$$

Q, m és VE vektorok változásaira az idő függvényében többféle hipotézis állítható, ezek különböző kombinációi mellett kell elvégezni az előreszámítást.

A módszer logikája a 4.1. ábrán követhető. A fenti példa a lakosságot nemek és életkorok szerint osztja csoportokba. Ahogy az elnevezés is utal rá, a kohorsz-komponens módszer nem tesz különbséget csoportokba tartozó egyedek között. (A

kohorsz [lat. cohors, 'zászlóalj'] a római császári hadseregben a légió kisebb egysége.) A vizsgálat csak annak becslésére terjed ki, hogy egyik csoportból hányan kerülnek át egy másik csoportba – például hány 30 éves férfi éli meg a 31 éves kort, illetve a csoport létszámát a ki- és bevándorlás hogyan befolyásolja. A módszer a csoport egyedeit egyéb jellemzőik alapján már nem különbözteti meg. Természetesen a csoportok száma növelhető: a (T./I. 2007) tanulmány, mely az erdélyi magyar népesség előreszámítását mutatja be, Erdély megyéire illetve 42 régiójára is közöl számításokat. (A megyék közti belső vándorlás modellezéséről a tanulmány adatok hiányában lemondott.)

A kohorsz-komponens módszer hatalmas előnye, hogy számításigénye kicsi, egyszerű táblázatkezelő program segítségével elvégezhető az előrevetítés. Ha az egyedeket nemük, életkoruk és esetleg tágabb értelemben vett lakóhelyükön felül egyéb tulajdonságokkal szeretnénk jellemezni, a szükséges csoportok számának ugrásszerű növekedéséből adódóan a modell nehezen kezelhetővé és áttekinthetetlenné válna. Éppen ezért az iskolai végzettségek vagy a családi állapotok aránymódszerrel kerülnek felvetítésre a korfára.

Ha az előrevetítést az egyén életpályája felől közelítjük a csoportok közti mozgások helyett, a mikroszimulációs módszertan alkalmazható. A mikroszimulációban már nem a csoportokat jellemezzük tagjainak számával, hanem az egyedet soroljuk csoportokba, illetve jellemezzük más mutató típusú tulajdonságokkal. A mikroszimuláció számításigénye – a kohorsz-komponens módszerrel ellentétben – hatalmas.

4.2.2. A mikroszimulációs módszertan és gyakorlati megvalósítása

A mikroszimulációs eljárások (Orkutt/Greenberger 1961) a feldolgozott adatállomány minden egyes rekordjára – jelen példában minden egyénre – végrehajtják az idő múlását szimuláló algoritmusokat. Matematikai háttere miatt (O'Donoghue 2001) a teljes sokaságra szabad az eljárásokat végrehajtani, így egy társadalompolitikai előrejelzés esetén Magyarországon 10 milliós népességre kell az algoritmusokat futtatni. Az igazi hatásvizsgálatokhoz a hosszú távú előrejelzések a fontosak, így esetünkben a 10 milliós lakosságra 30-50 éves távlatokra szeretnének a felhasználók becsléseket készíteni, amihez igen nagy számítókapacitás szükséges.

A mikroszimulációs elmélet első számítógépes megvalósításait a 70-es években dolgozták ki társadalompolitikai intézkedések hatásvizsgálatára Észak-Amerikában, Ausztráliában és a skandináv országokban. A módszertan Európában a 80-as években

terjedt el, majd szerves része lett az államigazgatási munkában a törvények hatásvizsgálatának (Klevmarken/Olovsson 1996; Frederik 2001; Immerwoll 2001; Morris 2001).

A módszertan Magyarországon a 80-as évek második felében jelent meg – elsősorban a KSH-ban (r M. 1987; J. 1987). Segítségével a dotációk megszüntetésének hatásait modellezték, illetve az SZJA törvények hatásait vizsgálták. A 90-es években folytatódott a munka a KSH-ban, majd később a Pénzügyminisztérium Adófőosztályán is.

Egyedi számítástechnikai megoldások alakultak ki, melyek csak a konkrét feladatok megoldására voltak alkalmasak, így a módszertan szélesebb körben nem terjedt el. Egyetemi kereteken belül fejlesztette ki a Csicsman József vezette kutatócsoport az elsősorban SAS alapokra épülő mikroszimulációs szolgáltató rendszert. Az ő szakmai támogatásával fejlesztettem ki a továbbiakban ismertetett megoldást, melyhez nem szükséges az igen drága SAS szoftver használata.

A mikroszimulációs módszertan lényege, hogy egy jól ismert statisztikai sokaság adatait az idő függvényében továbbírjuk a számítógép segítségével. A vizsgált objektumok adatainak továbbírásához a valószínűségszámítási eszközök, a törvényekben lévő szabályok, illetve tapasztalati tények alapján létrejött algoritmusok használhatóak leggyakrabban. A „mikro" szó azt jelenti, hogy ez a szimulációs eljárás mikroszintű, azaz a szimuláció során alkalmazott összes utasítást a sokaságot alkotó egyedek szintjén kell végrehajtanunk. Statisztikai szempontból fontos jellemzője, hogy alkalmazásával olyan becsült adatokhoz jutunk, amelyeket csak újabb adatfelvétellel lehetne produkálni.

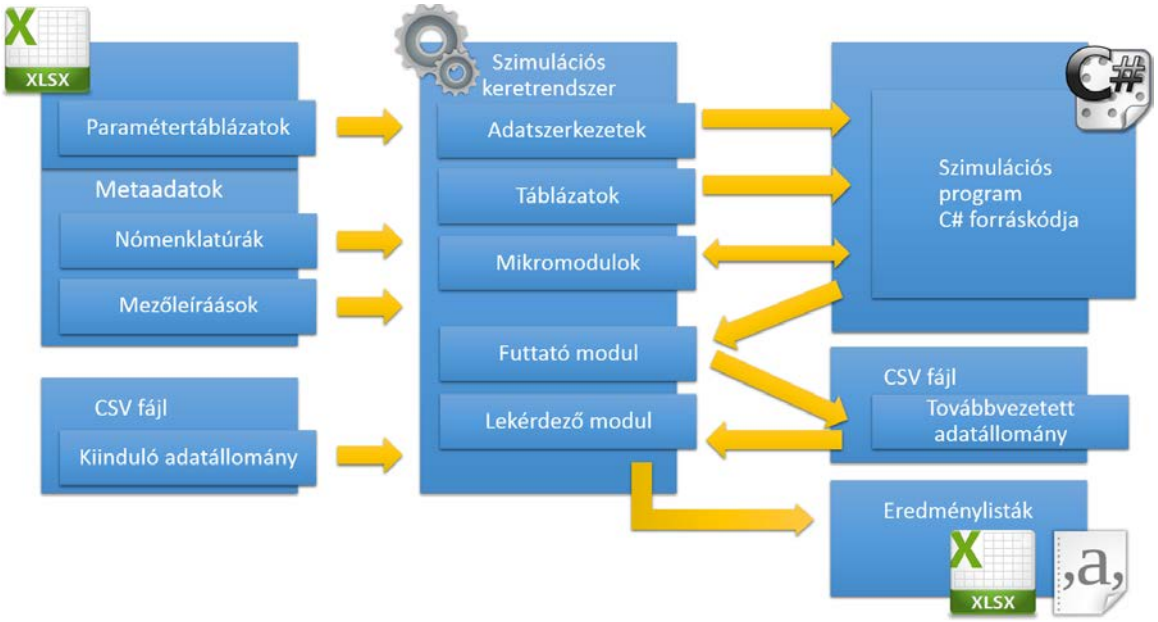
Napjainkban a pénzügyi-gazdasági élet szereplőinek is nagy hangsúlyt kell fektetniük az intézkedéseik, döntéseik előkészítésére, ha versenyképességüket és piaci pozíciójukat meg szeretnék őrizni. Új termék bevezetése vagy már meglévő kondíciók változtatása esetén fontos tudniuk ügyfeleik reakcióit. A vizsgálandó bemeneti és kimeneti paraméterek nagy száma, valamint a probléma mérete azonban nem teszi lehetővé, hogy manuálisan, emberi erővel végezzük el a döntések meghozatalához szükséges kiértékeléseket. Ilyenkor léphetnek színre azok a számítástechnikai eszközök, amelyek nagymértékben segíthetik a döntéshozók munkáját.

Az óriási adatbázisok és számítási kapacitások lehetővé teszik, hogy a magas szintű gazdasági vagy politikai döntéshozásban, például pénzintézeti döntések meghozatala, szociális, társadalombiztosítási, nyugdíjrendszert érintő reformlépések bevezetése előtt több lehetőséget, javaslatot elemezve azt válasszák, amelyek a leginkább sikeresek lehetnek megvalósításuk során. A mikroszimuláció segítségével még a törvény-

beiktatás előtt képet lehet alkotni egy–egy kormányzati döntés társadalmi, gazdasági következményeiről (Molnar/Sinka 2007).

A nyugdíjszámításokhoz szükséges legalább 50 éves előrejelzésekhez kidolgozott születés és halálozási szimulációs modulok igen számításigényesek, ezért a párhuzamos programozási technikát használtam. Ezzel a megoldással a felhasználó közgazdász elfogadható futásidejű megoldást kap. Annak érdekében, hogy a mikroszimulációs módszertannal továbbvezetett adatállományon kialakítható legyen a karrier, a nyugdíjbavonulás és a meglévő nyugdíjak továbbvezetése, keretrendszert hoztam létre. A keretrendszer megoldja a bonyolult adatkezelési eljárásokat és a párhuzamos futtatáshoz szükséges adatkezelési feladatokat. Így a végfelhasználó koncentrálhat saját szakterületének algoritmusaira – módja van az úgynevezett mikromodulok és a hozzájuk kapcsolódó paramétertáblák kialakítására és módosítására. A végfelhasználó és az informatikai megoldások között az úgynevezett metaadatok teremtik meg a kapcsolatot.

A mikroszimuláció gyakorlati alkalmazásának végrehajtási menetét a 4.2. IPO (Input, Process, Output) ábra szemlélteti.



4.2. ábra. A keretrendszer IPO diagramja.

- Az eljárás bemenő adata a továbbírandó kiinduló adatállomány. Ahhoz, hogy a keretrendszer tetszőleges adatállományt feldolgozhasson, meg kell adni a feldolgozandó adatállomány mezőinek leírását, az úgynevezett metaadatokat. Excel

táblázatban definiálhatjuk az értékadatokat (a KSH konvencióit követve) a mutatókat és a kategóriaváltozókat, a nomenklatúrákat.
A mikroszimulációs mintafeladat a magyarországi személyek sorsát követi végig. A személyek korábbi adatgyűjtésből olyan tulajdonságokkal rendelkeznek, mint születési év, életkor, nem, iskolai végzettség, jövedelem, stb. A személyeket leíró adatok típusa kétféle lehet: nómenklatúra vagy mutató.
A mutatók értékváltozók, szám típusú tulajdonságok, mint például a születési dátum vagy a jövedelem. Értelmezhető felettük az összes aritmetikai művelet.
A nómenklatúrák kategóriák vagy osztályozó változók, melyeknek általában ismertek a lehetséges értékei, ezek a nómenklatúra elemek.
Nómenklatúra például Magyarország megyéinek vagy régióinak listája, de nómenklatúraként értelmezhető a legmagasabb iskolai végzettség, vagy a nem is.
A nómenklatúrák elemeinek sorrendje a kulcsok értéke alapján értelmezhető, akárcsak a statisztikában használatos ordinális változóké, de az aritmetikai műveletek nem értelmezhetők felettük.
A megadott mutatók és nomenklatúrák, mint elemi adatmezők kombinációjával írjuk le a feldolgozandó adatállományok szerkezetét, a rekordleírásokat.
A mikroszimulációs eljárások során paramétertáblázatok segítségével végezzük becsléseinket, esetünkben a szülési és a halálozási valószínűségek megadásával hangolhatjuk eljárásunkat. A paramétertáblázatok dimenzióit a nomenklatúrák határozzák meg, ezért kötelező a paramétertáblázatokban használt nomenklatúrák elemeinek meghatározása is.

- A végrehajtás, a Process során első lépésben generáljuk a végrehajtandó *mikroszimulációs program* C# forráskódját. A mikroszimulációs program legfontosabb funkciója, hogy egyszer (és csak egyszer) beolvassa az input adatállomány adatait, futtatja rajta a felhasználó által definiált mikromodulokat és létrehozza az eredmény adatállományát. A kódgenerálás természetesen párhuzamosan futó kódot generál. A generált kód tartalmazza a több processzormagon történő futtatáshoz szükséges kódrészeket, illetve az adatállományt szétválasztja annyi részre, amennyi processzormag rendelkezésre áll a végrehajtásra.
  A mikromodulok az egyes demográfiai, gazdasági események realizációi. A felhasználónak módja van ellenőrizni az előzetesen megírt modulokat, illetve lehetősége van azok módosítására, korrigálására. (A keretrendszer jelen verziójában csak C# kódok írhatók és módosíthatók. A jövőben – ha az szükséges – kialakítható felhasználóbarát modultervezési lehetőség is). A generált mikroszimulációs programot a *Futtató modul* hajtja végre, mely létrehozza az eredmény

adatállományt azonos szerkezetben, mint amilyen a kiinduló volt. A futtatás legfontosabb paraméterét, amely meghatározza, hogy milyen hosszú idősorra legyen végrehajtva a szimuláció, a felhasználó állíthatja be.
Mind a kiinduló állományt, mind az eredmény állományt előredefiniált *Lekérdezés*sel ismerhetjük meg, ennek felhívását a Process funkcióban aktivizálhatjuk. Nagyon fontosak a szimulációs eljárás működését dokumentáló kontroll táblák is.

- A keretrendszer eredményei a korábban említett végrehajtható programkódok, a szimuláció eredményét tartalmazó adatállomány, illetve az eredményeket leíró Excel és nyomtatási fájlok.

4.3. A keretrendszer alkalmazása a születés és a halál események 50 éves továbbvezetésére

4.3.1. A kiinduló állomány

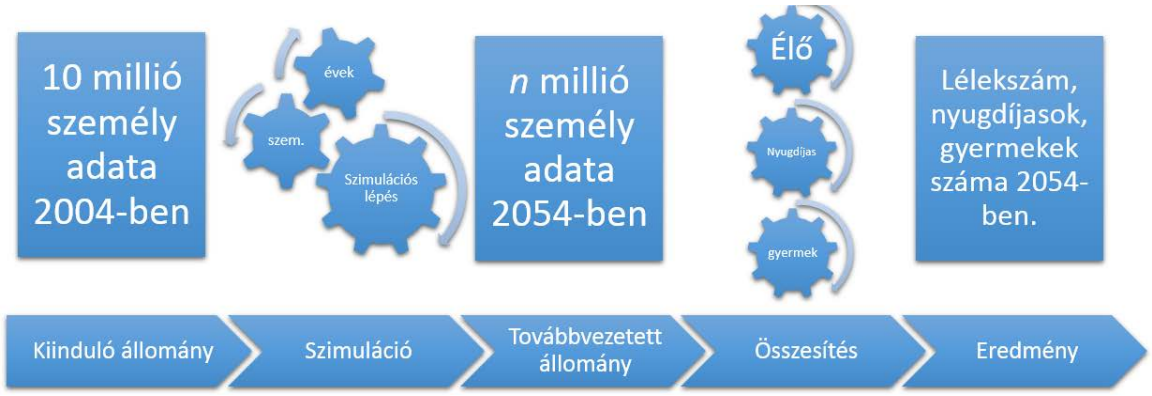
A nyugdíjszámítást segítő mikroszimuláció indításához szükség van a teljes népesség releváns adatait tartalmazó *kiinduló állomány*ra. Erre a célra a 2004-es KSH Háztartási Költségvetés felvétel (HKF) és a 2005-ös Mikrocenzushoz tartozó Jövedelem felvétel Statistical Matching eljárással összekapcsolt úgynevezett Kutató adatállományt (Csicsman/László 2012) használom, ebben 25 ezer személy adatai szerepelnek.

A személyrekordokon szereplő egyedi súlyok azt határozzák meg, hogy az adott személy hány főt reprezentál a teljes magyarországi sokaságból. Mivel a súlyok szórása igen nagy, a szimulációs eljárás során nem használhatjuk a súlyozott állományt. A súlyozás feloldását úgy tudjuk megoldani, hogy a rekordokat, azonos tartalommal megsokszorozzuk súlyszámuk szerint.

A rekordokon 279 változót mért a KSH, ezek többek között tartalmazzák a vizsgálatban szereplő személyek születési évét, nemét, családi állapotát, legmagasabb iskolai végzettségét, és más, a személyre vonatkozó – elsősorban gazdasági adatokat.

4.3.2. Az állomány továbbvezetése

A keretrendszer a *futtató* lépésében először felépíti a teljes népességet alkotó személyek listáját a kiinduló állomány alapján, majd éves lépésekben vezeti tovább az állományt. Minden évben minden egyedre végre kell hajtani a szimulációs lépést, mely meghatározza, hogy a vizsgált személy életben marad-e illetve születik-e gyermeke.



4.3. ábra. Az adat-továbbvezetés lépései.

A kutatóállományban több száz adatmező van, a memóriafelhasználás optimalizálása miatt természetesen nem a teljes adatrekordokkal dolgozunk, hanem annak csak a demográfiai továbbvezetés szempontjából érintett adatmezőit használjuk.

4.3.3. Mikromodulok

Személyeken végrehajtott *szimulációs lépés* egymást követő mikromodulokból áll. A mikromodulok felelősek a személy életében különböző valószínűséggel bekövetkező események kezeléséért, melyek megváltoztatják a személy tulajdonságait. Jelenleg a nyugdíj előrejelzésekhez a következő mikromodulokat dolgoztam ki részletesen:

- A *halálozás mikromodul* felelős annak eldöntésért, hogy a szimulációs lépésben vizsgált személy meghal-e az adott évben. A halálozási valószínűségeket KSH adatokra épülő paramétertábla tartalmazza, amely nemenkénti és korcsoportonkénti bontásban tartalmaz egy éven belüli halálozási valószínűségeket. A mikromodul egy $[0, 1)$ intervallumban generált egyenletes eloszlású véletlen szám alapján határoz a személy sorsa felől. Ha a generált véletlen szám kisebb, mint a vizsgált személy tulajdonságai alapján a paramétertáblából kiolvasott halálozási valószínűség, a személy *aktív* tulajdonsága hamis értéket vesz fel.

- A *születés mikromodul* szintén KSH adatok alapján készített paramétertáblára épít, melyből a családi állapot, az ötéves korcsoport és a lakhely régiója alapján olvasható ki annak a valószínűsége, hogy egy nőnek az adott évben gyermeke születik. (Természetesen csak nők szülhetnek.) A gyermek születésének valószínűsége erősen összefügg a családi állapottal, mely lehet *hajadon*, *házas*, *özvegy* illetve *elvált*. A születési valószínűség paramétertábla nem tartalmaz adatokat

15 év alatti illetve 50 év feletti nőkre. Az újszülött neme szintén véletlen szám generátorral kerül meghatározásra.

- A nyugdíj-továbbvezetéshez szükséges karrier, jövedelem-továbbvezetés, nyugdíjazás és nyugdíj-továbbvezetés mikromoduljaival dolgozatom nem foglalkozik, ezek felépítésén közgazdász kollégáim dolgoznak.

A következő pontokban bemutatott szimulációs keretrendszer alkalmas olyan további mikromodulok futtatására is, mint a jövedelem-továbbvezetés, vagy a munkaképesség változás, a nyugdíjassá válás, illetve a meglévő nyugdíjak továbbvezetése. A további mikromodulok megépítéséhez szükséges a témában szakértő közgazdászok támogatása.

4.3.4. Mikroszimulációs kertrendszerrel szemben támasztott követelmények

A mikroszimulációs keretrendszerrel szemben támasztott követelményeket az alábbi felsorolásban fogalmaztam meg:

- A keretrendszerben meg kell tudni határozni, hogy a továbbírandó adat – esetünkben a személy – milyen tulajdonságokkal rendelkezzen. Tetszőlegesen meg kell tudni adni az input állomány szerkezetét, esetünkben a személy rekord adatleírását.

- A személyek tulajdonságai lehetnek nómenklatúra illetve mutató típusúak. A szimulációban szereplő nómenklatúrák és azok elemeinek kezelését is meg kell oldani.

- A szimulációs feladatokat időszakonként (évenként) kell végrehajtani elemi szinten, esetünkben személyenként.

- A mutató-, nómenklatúra- és rekordleírások együttesét metainformációknak nevezzük. A keretrendszernek kezelnie kell a metaadatokat.

- A kiinduló adatállomány alapján fel kell tudni építeni a személyekből álló, össznépességet reprezentáló adatállományt. A kiinduló adatállomány tartalmazhatja a teljes népesség adatait vagy lehet egy reprezentatív minta. Reprezentatív minta esetén az állomány minden sora alapján a hozzá rendelt súlynak megfelelő számú új személy-rekordot kell létrehozni.

- A nagy számításigényre való tekintettel a szimulációs lépéseket a többmagos processzorok számítókapacitását kihasználva párhuzamosan kell végrehajtani.
- A mikroszimulációs eljárások végrehajtása egymástól független személyekre történik, ezért a kiinduló állomány tetszőleges részekre osztásával megvalósítható a párhuzamos processzorokon való végrehajtás. A keretrendszer automatikusan végezze a kiinduló adatállomány annyi részre való bontását, ahány processzoron történik a végrehajtás, illetve automatikusan összesítse a különböző processzorokon létrehozott eredményeket.
- Egy szimulációs lépésben különböző gazdasági és társadalmi változásokat modellező mikromodulok legyenek futtathatók kötött, vagy véletlen sorrendben.
- A mikromodulok döntéseiket becslési algoritmusok segítségével hozzák meg, melyeket a becsléshez tartozó többdimenziós paramétertáblák vezérelnek. A keretrendszernek alkalmasnak kell lennie a paramétertáblák kezelésére.
- Kezelni kell új rekordok keletkezését, esetünkben a személyek születését, akiket hozzá kell adni az eredmény adatállományához. Az új rekord szerkezete természetesen azonos a többiekével, így az újszülött tulajdonságait be kell tudni állítani a születési mikromodulban. A lakóhelyet például a gyermek örökli a szülőtől.
- A szimulációs lépések futtatása után az eredményeket ki kell tudni értékelni. Előredefiniált lekérdezéseket kell futtatni a kiinduló és az eredmény állományokon, illetve a paramétertáblázatok szerinti sorsolásokat, a születéseket és a halálozásokat úgynevezett kontrolltáblákon kell követni. A kontroll táblák segítségével ellenőrizhetjük, hogy az elvárt valószínűségek megjelennek-e a kontrolltáblák esemény/eset mutatóiban.
- A teljes szimulációnak általános célú személyi számítógépen percek alatt le kell futnia.

4.4. Mikroszimulációs keretrendszer kialakítása

Ebben a fejezetben ismertetem a mikroszimulációs keretrendszer, mint számítástechnikai szolgáltatás főbb funkcionalitásait, a megvalósítás során megoldott problémákat, a kialakított megoldást és az ahhoz tartozó programtechnikai megközelítéseket.

4.4.1. A mikroszimulációs keretrendszer részei

A metainformációs adatok kezelése

A szimulációs lépés elkészítéséhez szükség van a személyt leíró adatszerkezetre. A személy különböző tulajdonságokkal rendelkezik. A személyek tulajdonságainak nagy részét a kiinduló adatállomány tartalmazza, de lehetnek olyan tulajdonságok is, amelyek nem szerepelnek a kiinduló állományban. A tárolt és a képzett adatok mindegyikének leírását tartalmazza a metaalrendszer.

A kiinduló adatállomány kezelése

A kiinduló adatállomány alapján egyenként végig kell olvasni a személy-rekordokat, és az adatokat be kell tölteni a memóriába. Ezeken a memóriába töltött adatokon kell majd végrehajtani a mikromodulokat éves lépésenként.

A teljes népesség adatait tartalmazó kiinduló adatállomány vesszővel tagolt szövegfájl (CSV) formájában áll rendelkezésre. A fájl első sora tartalmazza az oszlopok elnevezését.

A kiinduló állomány lehet súlyozott, ami azt jelenti, hogy egy rekord több személyt reprezentál az össznépességben. A személy rekordon lévő súly adja meg, hogy az adott rekord hány valós személyt reprezentál a teljes népességből. Ennek megfelelően lehetőséget kell biztosítani arra, hogy a kiinduló állomány egy rekordja alapján a megfelelő számú személy szerepeljen a memóriában.

Kiinduló CSV állomány kezelése méreténél fogva nehézkes. A nagyméretű szövegfájlt a legtöbb szövegszerkesztő program meg sem tudja nyitni – mérete több száz megabyte is lehet. Mint később látni fogjuk, az egyedek kiinduló adatainak tárolására mégis a vesszővel tagolt szövegfájl tűnik az egyik legcélravezetőbb megoldásnak.

Paramétertáblák

A mikroszimulációs lépésben a személy sorsát paramétertáblák alapján hozott döntések határozzák meg. A paramétertáblák az azt kifeszítő nómenklatúrák értékeinek megfelelően rendelnek valószínűségeket. A nyugdíj szimulációban a halálozási valószínűségeket olyan paramétertábla tartalmazza, amely a személyhez kapcsolódó nem, régió és életkor tulajdonságok kombinációihoz rendeli az egy éven belüli elhalálozás valószínűségét. A paramétertáblák egyes elemei lehetnek üresek, például a szülési paramétertábla nem tartalmaz adatokat gyerekekhez és idősekhez.

A mikromodulok építése

A különböző társadalmi, gazdasági események továbbvezetését ún. mikromodulokban célszerű realizálni. Más és más tudás szükséges például a demográfiai események, esetünkben a születés és a halálozás becsléséhez, mint a jövedelmek és a nyugdíjadatok továbbvezetéséhez. Az egyes mikromodulokat becslési függvények és elemi programkódok realizálják. A keretrendszerrel dolgozó felhasználóknak látniuk kell az egyes mikromodulokat, melyeket szükség szerint módosíthatnak. Ugyancsak a mikromodulokban kell kitölteni az ún. kontrolltáblákat, ezek segítségével ellenőrizhető az egyes algoritmusok helyes futása. A becslési algoritmus paraméterként kapja meg az éppen feldolgozott személy adatait és ezekhez az adatokhoz tartozó paramétertábla értéket. Például a halálozási esemény az adott nemű/korú személyre akkor következik be, ha a futtatáskor a $[0, 1)$ intervallumon generált véletlen szám kisebb, mint az adott nemhez és korhoz tartozó halálozási valószínűség.

Az egyes mikromodulok programkódja a gyakorlati tapasztalatok szerint sokkal egyszerűbb elemekből áll, mint a futtatását lehetővé tevő környezet kódja.

A futtató modul

A futtató modul felelős a szimulációs lépés, azaz a mikromodulok az összes személyen történő végrehajtásáért. Mivel a modell szerint a személyek között nincs kapcsolat, a szimulációs lépések – kihasználva a többmagos számítógépek kapacitását – párhuzamosan is végrehajthatók. A futtatás paraméterei a kiinduló és az azonos szerkezetű eredményeket tartalmazó adatállomány neve, illetve az a szám, mely megmutatja, hogy hány évre kívánjuk a mikroszimulációs eljárást végrehajtani.

Adatok összesítése

A mikroszimulációs eljárás során a párhuzamos processzálás miatt szétbontott eredményeket egy közös eredmény állományba kell összesíteni. Annak érdekében, hogy a rendszer felhasználói megismerhessék az eljárások eredményeit, mind a kiinduló, mind az eredmény állományokon lekérdezéseket lehet futtatni.

A lekérdezések

A keretrendszerben előredefiniált, vagy ad-hoc lekérdezésével ismerhetjük meg a kiinduló és az eredmény állományokat. Legtöbbször csak olyan a kérdésekre keressük a választ, hogy hány élő személy lesz 50 év múlva, vagy hány nyugdíjaskorú lesz 50 év múlva. Elképzelhető azonban olyan eset, amikor minden év után szeretnénk

összesítést készíteni, hogy a népesség vizsgált jellemzőjének alakulását folyamatában követhessük – ezért kell a lekérdezési funkciót önállóan kezelni.

4.4.2. Megvalósítást előkészítő döntések

A keretrendszer gyakorlati megvalósításánál több, látszólag egymásnak ellentmondó szempontot kell figyelembe venni. Az egyik legfontosabb szempont a mikroszimulációs eljárás futtatásához szükséges idő leszorítása.

Felhasználhatóság szempontjából az egyik véglet a kifejezetten az adott szimulációs feladat megoldásra fejlesztett célprogram, a másik végletet az általános célú szimulációs keretrendszer jelenti. Az általános célú rendszereknél a szerteágazó lehetőségek miatt a teljesítményoldalon is fizetni kell.

Általános célú keretrendszer például az Új Calculus Bt. MicroSim rendszere, melynek metainformációs rendszerében tetszőleges adatok írhatók le és a becslési algoritmusokhoz kapcsolódó paramétertáblák építhetők. Ezeket a rendszer PostgreSQL adatbázisban tárolja. A továbbvezetendő adatállomány tekintetében a MicroSim az IBM SAS rendszerére épít. A MicroSim nagy előnye, hogy a mikromodulok szerkesztésére grafikus felületet tartalmaz, mely alapján SAS kódot generál.

Az adatbázis-kezelő rendszerek sok funkciót kínálnak készen a fejlesztőknek, de ezért cserébe teljesítmény oldalon áldozatokat kell hozni. A relációs adatbázis-kezelők legtöbb előnye, mint például az adatok biztonságos és konzisztens tárolása, a bonyolult adatszerkezetek optimalizált kezelése, vagy a jogosultságkezelés a mikroszimulációs alkalmazásoknál nem használható ki. Ma már a továbbvezetendő adatállomány mérete sem indokolja külön adatbázis-kezelő rendszer használatát.

A fenti megfontolások alapján olyan *mikroszimulációs keretrendszer*t készítettem, mely képes a *mikroszimulációs program* forráskódját generálni, majd lefordítani és futtatni. A generált szimulációs program külső adatbázis-kezelő helyett a memóriában kezeli az adatokat, amivel jelentős sebességnövekedés érhető el.

A szimulációs program kódja a nómenklatúrák, paramétertáblák és a szimulációban részt vevő egyedek tulajdonságainak ismeretében automatizáltan legenerálható és futtatható. Az automatikusan generált kód, amellett, hogy időt takarít meg, véd az emberi hibáktól is. Ebbe a környezetbe ágyazhatók be például a születésért, halálozásért felelős mikromodulok kódrészletei.

A programozási nyelv illetve a fejlesztőkörnyezet tekintetében több szempontot kell figyelembe venni. Ezek közül a legfontosabb a futási sebesség, a jól olvasható kód, valamint a mikromodulok értelmezéséhez és kezelhetőségéhez a meredek tanulá-

si görbe. A C# programozási nyelv mellett erős érvként szólt az IntelliSense névre keresztelt automatikus kódkiegészítő, mely a forráskód szerkesztése közben folyamatosan elemzi a kód szövegét, és javaslatokat tesz a felhasználónak a szintaktikailag és szemantikailag helyes kódrészletek beszúrására. Esetünkben ez a mikromodulok szerkesztését és a nómenklatúrákkal való munkát könnyíti meg. A C illetve a C++ nyelvek jobb teljesítményt eredményeznének, de ezeknek a nyelveknek a tanulási görbéje sokkal laposabb, illetve felépítéséből adódóan a nyelv kitettebb a fejlesztői hibáknak.

További szempont, hogy a C# nyelvhez is rendelkezésre állnak azok az MPI (Message Passing Interface) könyvtárak, melyek segítségével több számítógépből álló klaszteren, osztott rendszerben is végezhetők számítások. A szimulációs keretrendszer tekintetében ez az egyik tervezett fejlesztési irány.

A mikroszimulációs környezet felépítése elsősorban szoftvertervezői kompetenciát és a választott programozási nyelv mély ismeretét igényli. A környezet kialakításánál a nagy számításigény miatt számtalan teljesítményoptimalizálással kapcsolatos kérdést is figyelembe kell venni. Ezzel szemben a mikromodulokat leíró programkód mindössze néhány egyszerű lépést tartalmaz, megírása illetve olvasása nem igényel nagy szoftverfejlesztői gyakorlatot. A szimulációs lépés megtervezése elsősorban gazdasági és statisztikai jellegű kérdéseket vet fel.

Az értekezés következő pontjai a szimulációt megvalósító szimulációs program kódját generáló és futtató szimulációs keretrendszert mutatják be.

4.4.3. Szoftvertervezési és megvalósíthatósági megfontolások

Elnevezési konvenciók

A modern szoftverfejlesztő eszközök már gépelés közben is folyamatosan elemzik a kódot, és egy kódelem első néhány karakterének leütése után javaslatot tesznek a befejezésre. Az automatikus kódkiegészítésnek köszönhetően megváltoztak a változóelnevezési szokások: a könnyen legépelhető néhány karakterből álló rövidítések helyét átvették a hosszú, több szóból beszédes változónevek, melyek nagyon megkönnyítik a kód értelmezését. Több szóból álló változónevek betűközzel történő tagolását a C# nem engedi meg, ezért a teve púpjairól elnevezett CamelCase írásmód terjedt el: e szerint minden egyes szó kezdőbetűjét nagybetűvel írják. (Az első betű lehet kisbetű is.)

Nómenklatúrák beolvasása

Az .xlsx kiterjesztésű Open XML formátumú állományok szabványosak, olvasásukhoz illetve írásukhoz nincs feltétlenül szükség telepített Excelre, vagy más táblázatkezelőre. A keretrendszer a Microsoft Open XML Format SDK 2.5 csomagot használja az Excel állományok feldolgozására. A megoldás előnye, hogy nem szükséges hozzá telepített Excel, és így a különböző Excel verziók közti különbségek sem okozhatnak fennakadást.

```
public enum Régió {
        KözépDunántúl=2,
        NyugatDunántúl=3,
        DélDunántúl=4,
        ÉszakMagyarország=5,
        ÉszakAlföld=6,
        Budapest=8,
        PestMegye=9,
        DélAlföld=7 }
```

Listing 4.1. Nómenklatúra megadása enumerációként.

Paramétertáblák kezelése

A paramétertáblák tárolását tömbökkel oldottam meg. A paramétertáblák dimenziója tetszőleges lehet, a paramétertáblák egy n-dimenziós teret feszítenek ki. Az Excel táblában a paramétertáblák elemeit mindig koordinátáival kell megadni. A következő fejezetben lévő 4.7. ábra egy halálozási valószínűségeket leíró paramétertáblázatot tartalmaz, nem és életkor dimenziók mentén. Előfordulhat, hogy a tömb tartalmaz üres elemeket. Ebben az esetben az üres elemek *null* értékkel kerülnek feltöltésre. A tömb indexei minden dimenzió mentén 0-val kezdődnek, és folyamatosan futnak egy előre megadott maximum értékig. Ezért az optimális memória-kihasználás érdekében a paramétertábla adatait minden dimenzió mentén érdemes az origóba tolni. A példában a nemekhez tartozó értékek 1-gyel kezdődnek, a régiók számozása pedig 2-vel kezdődik. Ebből adódóan a nemek indexeit 1-gyel, a régiókét 2-vel kell negatív irányba eltolni. Az eltolásokról, illetve kereséskor a visszatolásról a varázsló gondoskodik, a szimuláció tervezőjének ezzel nem kell foglalkoznia.

Az indexek ismeretében a tömbből nagyon gyorsan ki lehet keresni egy értéket – néhány szorzás és összeadás művelet után meghatározható az elem helye a memóriában. Ezért cserébe a hézagosan feltöltött tömbök memória-kihasználása nem optimális.

Egyedek tulajdonságait tartalmazó kiinduló adatállományok

A teljes népesség adatait tartalmazó kiinduló adatállomány kezelése méreténél fogva nehézkes. Mint később látni fogjuk, az egyedek kiinduló adatainak tárolására a vesszővel tagolt szövegfájl tűnik az egyik legcélravezetőbb megoldásnak.

Személyek listájának tárolása a memóriában

Első ránézésre kézenfekvőnek tűnik tömbben tárolni az egyedeket. A tömbök tartalmát nagyon hatékonyan lehet lemezre írni, illetve vissza lehet olvasni. Közelebbről megvizsgálva a problémát a tömbök használata már nem tűnik jó választásnak. A megfelelő elemszámú tömb létrehozásához összefüggő szabad memóriára van szükség. A mi esetünkben, $\backsim 10^7$ egyed esetén tulajdonságonként $\backsim 40$ Mb tárigényt jelent, azonban a több száz Mb összefüggő memóriaterület nem feltétlenül áll rendelkezésre. További problémát jelent, hogy a tömbök elemszámát előre meg kell határozni. A születések miatt az elemszám a szimuláció során nő. Az `Array.Resize<T>` metódus lehetőséget biztosít a tömbök elemszámának utólagos megváltoztatására, de a futásidő nagy tömbök esetén elfogadhatatlanul hosszú. Minden méretváltoztatásnál lefoglalásra kerül a megváltoztatott méretű tömbnek megfelelő memória, és a régi tömb tartalma átmásolásra kerül.

A `List<>` szerkezet jó választásnak tűnik, de többszálú futtatás esetén lehetnek vele problémák. Listák esetén nem feltétel, hogy a szükséges memória összefüggő blokkban álljon rendelkezésre. A$\backsim 10^7$ objektum esetén a lemezre történő szerializáció és visszaolvasás nagyon lassan futott. A szövegfájl soronkénti feldolgozása egy nagyságrenddel jobb teljesítményt nyújtott.

Ha a listát szerializálva próbáljuk háttértárra írni, ugyancsak szükség van az állomány méretének megfelelő összefüggő területre a memóriában.

Szimulációs lépések végrehajtása az egyedeken

A szimuláció jelen esetben éves körökben zajlik – minden éves körben az összes egyeden végre kell hajtani a szimulációs lépést, azaz az egymást követő mikromodulokat. Az évek léptetését és az egyes éveken belül a szimulációs lépések végrehajtását az egyedeken két egymásba ágyazott ciklus vezérli a `Run()` metódusban. A belső ciklusban az adat-párhuzamos feldolgozást `parallel.for` ciklus végzi.

Többszálú futtatás

A rendelkezésre álló processzormagok kihasználásához célszerű a szimulációs lépéseket több szálon futtatni. Az egyes egyedeken egymástól függetlenül végrehajthatók a szimulációs lépések, ezért elméletben több szál csak új egyed születésekor próbálhat meg közös memóriaterülethez hozzáférni. A `ConcurrentBag<T>` osztály használata a `List<T>` helyett megoldja a problémát.

Véletlen számok generálása többszálú programban

A .NET környezet `Random` osztálya nem támogatja a több szálról történő elérést – az angol terminológia szerint nem *threadsafe*. Ha több szál próbál a `Random` osztály ugyanazon példányával véletlen számot generáltatni, az eredmény hibaüzenet nélkül 0 lesz. A probléma megoldására többféle megközelítést próbáltam ki:

1. Az osztály új példányának létrehozása minden szimulációs lépésben nem jöhet számításba, mivel az így generált számok között erős kapcsolat mutatkozna.

2. Lehet készíteni egy statikus véletlenszám generátort, mely zárolja a többi szál hozzáférését, amíg a generálás folyamatban van. Ebben a megközelítésben egyszerre csak egy véletlen szál generálása lehet folyamatban, a többi szálnak, ha véletlen számot szeretne, várakoznia kell, amíg a generátor felszabadul. A zárolást alkalmazó véletlenszám generátor kódját a 4.2. lista mutatja. Ez a módszer nem vált be, a futási idő több lett, mint az egyszálú változat esetén. A szimuláció során a véletlen számok előállításának a legnagyobb az időköltsége: a szálak idejük nagy részét a véletlenszám generátorra való várakozással töltik. Ezzel a módszerrel a kétmagos processzoron végzett párhuzamos futtatás közel 50% futásidő növekedést hozott, a további processzormagok várhatóan nem járulnának hozzá a teljesítmény növekedéséhez.

```
public static class RandomGen1
{
    private static Random _inst = new Random();

    public static Double NextDouble()
    {
        lock (_inst) return _inst.NextDouble();
    }
}
```

Listing 4.2. Szál-biztos véletlenszám generátor zárolással.

3. A másik megoldás a `[ThreadStatic]` attribútum használata, melynek segítségével a `Random` osztályból minden szálhoz külön példány hozható létre. (4.3. lista.) Így kiküszöbölhető a közös véletlenszám generátorra történő várakozás.

```
public static class RandomGen2
{
    private static Random _global = new Random();
    [ThreadStatic]
    private static Random _local;

    public static double NextDouble()
    {
        Random inst = _local;
        if (inst == null)
        {
            int seed;
            lock (_global) seed = _global.Next();
            _local = inst = new Random(seed);
        }
        return inst.NextDouble();
    }
}
```

Listing 4.3. Szál-statikus véletlenszám generátor.

4. A `parallel.for` egyik túlterhelése (overload) lehetőséget ad arra, hogy minden futó szál számára inicializáljunk külön-külön példányt egy osztályból. Ezzel a megoldással is külön véletetlenszám generátora lesz minden szálnak.

```
Parallel.For(0, egyedLista.Count,
    new ParallelOptions(),
    () => { return new Random(); },
    (j, loopState, random) =>
    {
        SzimulációsLépés(egyedLista[j], i, ref random);
        return random;
    },
    _ => { });
```

Listing 4.4. Véletlen szám generátor Parallel.For ciklussal.

Referenciaként használt egyszálú megoldás	61,8 sec	100%
Közösen használt, zárolt generátor.	114,4 sec	60%
[ThreadStatic] attribútummal ellátott generátor:	41.7 sec	165%
parallel.for szálanként külön generátorral.	39.1 sec	176%

4.1. táblázat. Párhuzamos véletlenszám generátorok futásideje.

A fenti megoldások futásideje alapján a mikroszimulációs programban a `parallel.for` használata mellett döntöttem.

4.5. A szimuláció futtatása

Ebben a fejezetben a megvalósított mikroszimulációs keretrendszer működése kerül bemutatásra.

A keretrendszer indítása után az első képernyőn három adatot kell beállítani:

1. Ki kell választani a metaadatokat, paramétertáblákat és nomenklatúrákat tartalmazó Excel állományt.

2. Ki kell választani a kiinduló adatokat tartalmazó CSV formátumú input állományt.

3. Meg kell adni azt a mappát, ahova a generált szimulációs program kerül. A keretrendszer egy már meglévő C# projekt jelölt pontjain helyezi el a generált kódrészleteket, még nagyobb flexibilitást biztosítva a felhasználónak.

4.5.1. Nómenklatúrák és paramétertáblák felépítése

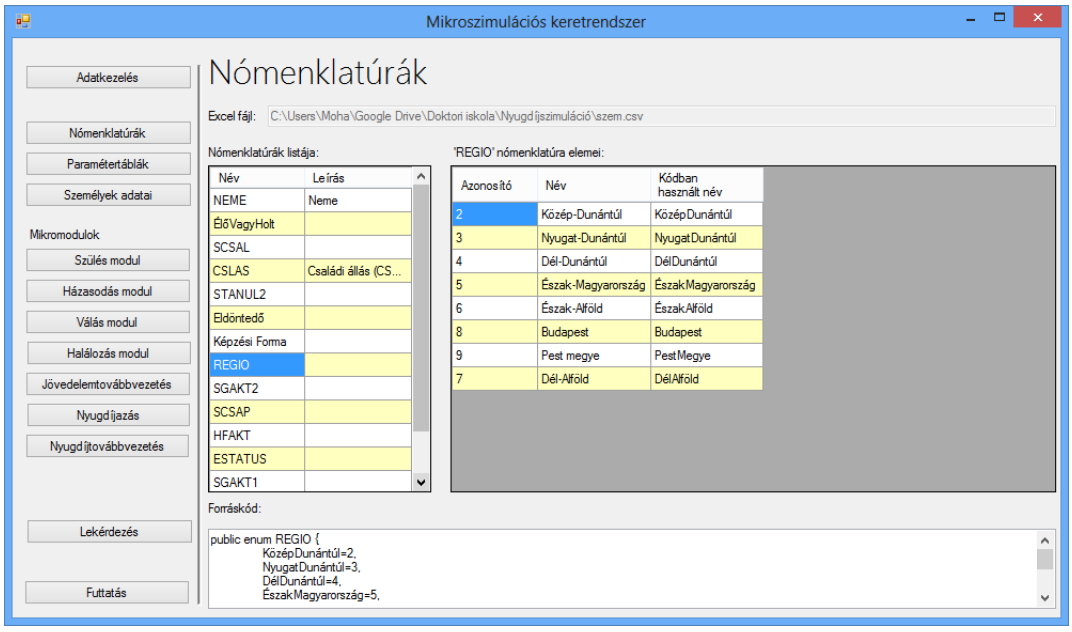
A kiinduló adatállomány és a paramétertáblák több oszlopa kódokat tartalmaz, így értelmezéséhez ismerni kell a kódokhoz tartozó jelentéseket.

Úgy gondoltam, hogy mikroszimuláció futtatásához szükséges nómenklatúrák és paramétertáblák összeállításához a legmegfelelőbb eszköz az Excel – vagy más táblázatkezelő program. A keretrendszer ezért célszerűen Excel tábla munkalapjairól gyűjti ki a nómenklatúrákat és a paramétertáblákat. Az Excel tábla felépítésére vonatkozó megkötés annyi, hogy a nómenklatúrák elemeit tartalmazó listák feletti cellának "Lista:[listanév]" formátumúnak kell lennie, míg a paramétertáblák kezdetét "Tábla:[táblanév]" formátumú cella jelöli. A nómenklatúrákat illetve paramétertáblákat a munkafüzeten úgy kell elhelyezni, hogy legalább egy üres cellasor illetve –oszlop válassza el egymástól őket. Egyéb megkötés nincs, a munkafüzet összes munkalapja használható.

	A	B	C	D	E
1	**Lista:NEME**			**Lista:REGIO**	
2	Nő	2		Közép-Dunántúl	2
3	Férfi	1		Nyugat-Dunántúl	3
4				Dél-Dunántúl	4
5	**Lista:CSLAS**			Észak-Magyarország	5
6	Elvált	4		Észak-Alföld	6
7	Özvegy	3		Budapest	8
8	Házas	2		Pest megye	9
9	Hajadon/nőtlen	1		Dél-Alföld	7

4.4. ábra. Nómenklatúrák megadása Excel táblázatban.

A 4.4. ábrán szereplő példa néhány nómenklatúrát tartalmaz. Ha a nómenklatúra elemei mellett azonosító szám nem kerül megadásra, a keretrendszer automatikusan beszámozza az elemeket. A nómenklatúra elemei tetszőleges sorrendben követhetik egymást, nem szükséges az azonosító értéke szerint sorba rendezni őket. A 4.5. ábra azokat a nómenklatúrákat mutatja, melyeket a keretrendszer kigyűjtött az Excel táblából.



4.5. ábra. Nómenklatúrák a keretrendszerben.

4.5.2. Metaadatok kezelése

A metaadatok kezelésére ugyancsak az Excel tábla tűnt a legcélszerűbbnek. (4.6. ábra.) A metaadatok a „Meta:” tartalmú cella alapján, szótárszerűen kerülnek felsorolásra. Adatszótár létrehozása nem kötelező, hiányos adatszótár esetén sem kapunk hibaüzenetet.

	A	B	C	D	E	F
1	Meta:					
2	MEGYE	Navajonmiaz				
3	TERUL	Terület				
4	SZLOK	Címazonosító - számlálókörzet				
5	CIMSSZ	Címazonosító - lakássorszám				
6	HSOR	Háztartás sorszáma a lakásban				
7	CSLAS	Családi állás (CSLAS)				
8	JC	Jogcím				
9	NEME	Neme				
10	CSPOT	Családi állapota:				
11	SZEV	Születésének ideje év				
12	HO	Születésének ideje hónap				
13	APOLG1	1.1 Melyik ország állampolgára?				

4.6. ábra. Metaadatok megadása Excelben.

4.5.3. Nómenklatúrák ellenőrzése

Feleslegesnek tartottam egy nómenklatúrákat szerkesztő modul beépítését a keretrendszerbe, hiszen ezek az adatok eredendően táblázat formájában állnak rendelkezésre. A programozástechnikában egy fontos alapelv szerint az „igazság egy helyen van”, azaz ha az adatok csak egy helyen kerülnek tárolásra, a különböző változatok közti eltérésből adódó hibák kiküszöbölhetők. Annak ellenőrzése, hogy a nómenklatúrák megfelelően kerültek-e felolvasásra az Excel táblából, fontos lépés. A 4.5 ábra tartalmazza az Excel táblában talált nómenklatúrák listáját – ahol rendelkezésre állnak metaadatok, ott leírással kiegészítve.

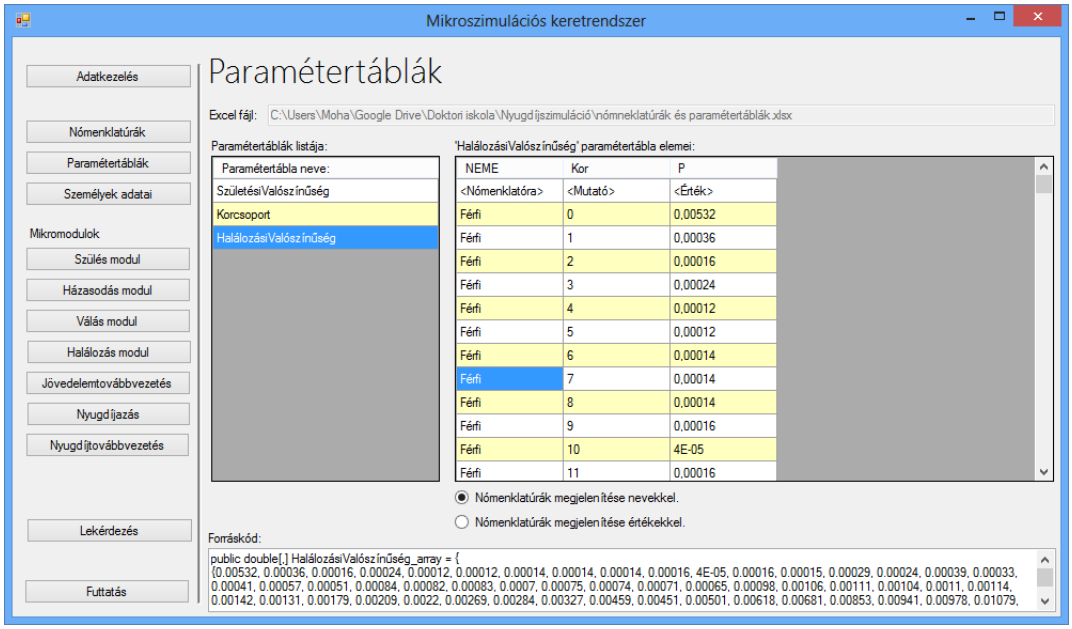
4.5.4. Paramétertáblák kezelése

A paramétertáblák és a nómenklatúrák közti kapcsolatot a tábla oszlopainak elnevezése teremti meg. Ha egy táblában szereplő oszlopnév megegyezik valamelyik nómenklatúra nevével, akkor azt a táblaoszlopot a varázsló nómenklatúra-típusúnak értelmezi, és beolvasáskor ellenőrzi, hogy csak olyan értékek szerepelnek-e benne, amelyek érvényesek az adott nómenklatúrában. Ha a tábla oszlopneve nem egyezik meg egyetlen nómenklatúra nevével sem, akkor mutatóként kerül feldolgozásra, és tetszőleges egész értéket felvehet.

	A	B	C
1	Tábla:Halálozási Valószínűség		
2	NEME	Kor	P
3	2	0	0,494%
4	2	1	0,038%
5	2	2	0,017%
6	2	3	0,021%
7	2	4	0,009%
8	2	5	0,011%
9	2	6	0,011%
10	2	7	0,013%
11	2	8	0,008%
12	2	9	0,013%
13	2	10	0,009%
14	2	11	0,013%
15	2	12	0,016%
16	2	13	0,017%
17	2	14	0,016%
18	2	15	0,013%
19	2	16	0,038%

4.7. ábra. Paramétertáblák megadása Excel táblázatban.

A 4.7. ábrán látható a halálozási valószínűségeket tartalmazó Excel tábla részlet. A szimulációs keretrendszerben a 4.8. ábra szerint ellenőrizhető, hogy rendelkezésre állnak-e, illetve megfelelően vannak-e elnevezve a nómenklatúrák.



4.8. ábra. Paramétertáblák a keretrendszerben.

4.5.5. Személyek adatai és a kiinduló állomány

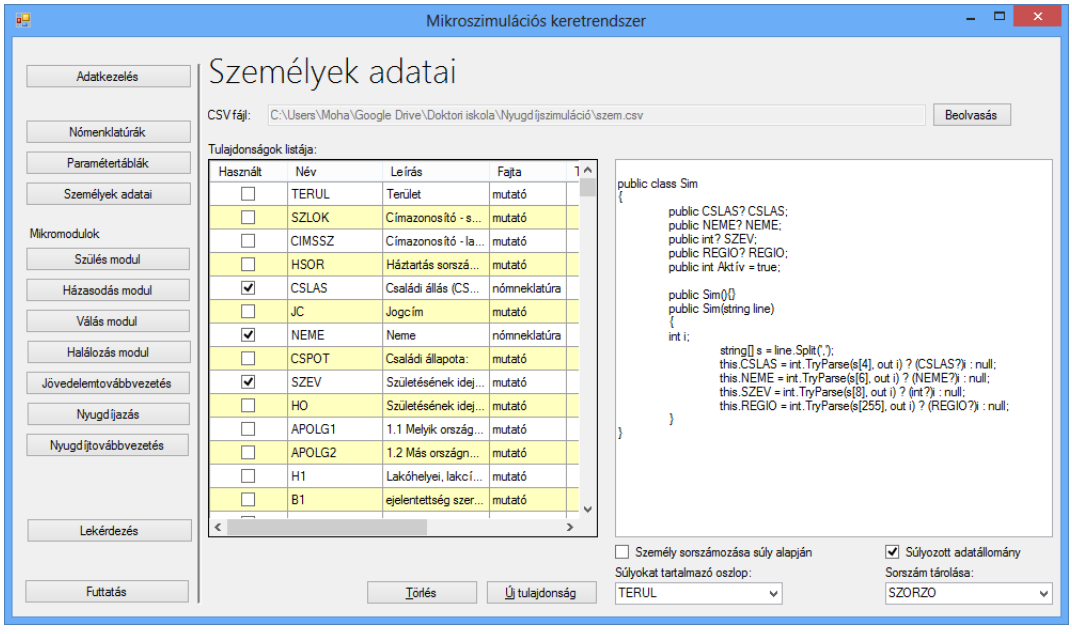
A teljes népesség adatait tartalmazó kiinduló állomány méretéből adódóan Excelben már nem kezelhető. A szimulációs keretrendszer az egyedek tulajdonságait vesszővel tagolt szövegfájlokból olvassa ki. A szövegfájl első sora az oszlopok elnevezését tartalmazza, minden további sor egy-egy személy tulajdonságait írja le. Az oszlopok elne-

vezésének itt is van jelentősége: ha az oszlopnév megegyezik valamelyik nómenklatúra nevével, akkor az oszlop nómenklatúra típusúként lesz kezelve, egyébként mutatóként kerül beolvasásra. Nómenklatúra típusú oszlopoknál a nómenklatúrák elemeit azonosítószámukkal kell kódolni. Az egyes adatsorokban természetesen előfordulhatnak üres értékek is.

```
TERUL,SZLOK,CIMSSZ,HSOR,CSLAS,JC,NEME,CSPOT,SZEV,HO,APOLG1,APOLG2,H1,B1,TERUL2,H2,B2,TE
02112,0007,037,1,1,1,1,2,1950,10,01,00,1,1,0,,,,,,,,1,1975,6,1,,,,,,,,,0,,,2,1975,10,
02112,0007,037,1,1,1,2,2,1957,10,01,00,1,1,0,,,,,,,,1,1975,6,1,,,,,,,,,0,,,2,1975,10,
02112,0008,096,1,2,1,1,1,1960,7,01,00,1,1,0,,,,,,,,0,,,,,,,,,,,,1,1998,5,1,1998,4,,,,
02112,0008,096,1,2,1,2,1,1972,1,01,00,1,1,0,,,,,,,,0,,,,,,,,,,,,1,1998,5,1,1998,4,,,,
02112,0008,096,1,4,1,2,1,1998,4,01,00,1,1,0,,,,,,,,0,,,,,,,,,,,,0,,,0,,,,,,,,02,1,,8
02112,0008,096,1,5,1,2,4,1929,4,01,00,1,1,0,,,,,,,,1,1952,9,0,1995,6,1,,,,,,,0,,,2,194
02112,0014,019,1,1,1,1,2,1944,1,01,00,1,1,0,,,,,,,,1,1969,8,1,,,,,,,,,0,,,2,1971,1,19
```

4.9. ábra. Részlet a személyek adatait leíró CSV állományból.

A tulajdonságok listája a CSV fájlban tárolt kiinduló adatállomány alapján kerül felépítésre az 4.9. ábra szerint. A keretrendszerben (4.10. ábra) a *Leírás* oszlop az Excel táblában felsorolt metaadat szótár alapján nyújt információt az egyes tulajdonságokról.



4.10. ábra. Egyedek adatainak megadása.

Az ablak jobb oldalán látható a személyeket leíró *Személy* osztály C# nyelvű kódja ellenőrzés céljából. Ezt teljes egészében a keretrendszer generálja. Az osztály azokkal a tulajdonságokkal rendelkezik, melyeket a keretrendszerben a lehetséges tulajdonságok közül kijelöltünk. Az osztály két konstruktorral rendelkezik. Az egyik

konstruktor üres, ezt akkor használjuk, ha a mikroszimuláció során új személy születik. A másik konstruktorra akkor van szükség, amikor a kiinduló adatállomány alapján felépítjük a személyek listáját. Ez a konstruktor paraméterként egy stringekből álló tömböt vesz át, ennek alapján állítja be az osztály tulajdonságait. A tömb irreleváns elemeit egyszerűen figyelmen kívül hagyja.

Minden tulajdonságnál megadhatjuk, hogy az adott tulajdonság tartalmazhate *null*, azaz üres értéket. A kiinduló adatállomány több oszlopa jellegéből adódóan nem feltétlenül van kitöltve. Azoknak a tulajdonságoknak a kitöltése azonban, melyek alapján paramétertáblában akarunk keresni, kötelező.

Mint ahogy az korábban említésre került, a kiinduló adatállomány lehet súlyozott. Az ablak jobb alsó sarkában lehetőség van a súlyokat tartalmazó oszlop kiválasztására. Ugyanitt lehetőség nyílik a kiinduló állomány azonos sora alapján létrehozott egyedek sorszámozására, illetve az egyed tulajdonságai között a sorszám tárolására.

Az „Új tulajdonság” gomb segítségével olyan tulajdonsággal is bővíthető az egyed osztály, mely nem szerepel a kiinduló állományban. Az itt felvett tulajdonságok is lehetnek mutató vagy nómneklatúra típusúak, azaz lehetnek elemi változók vagy enumeráció típusúak.

4.5.6. Mikromodulok szerkesztése

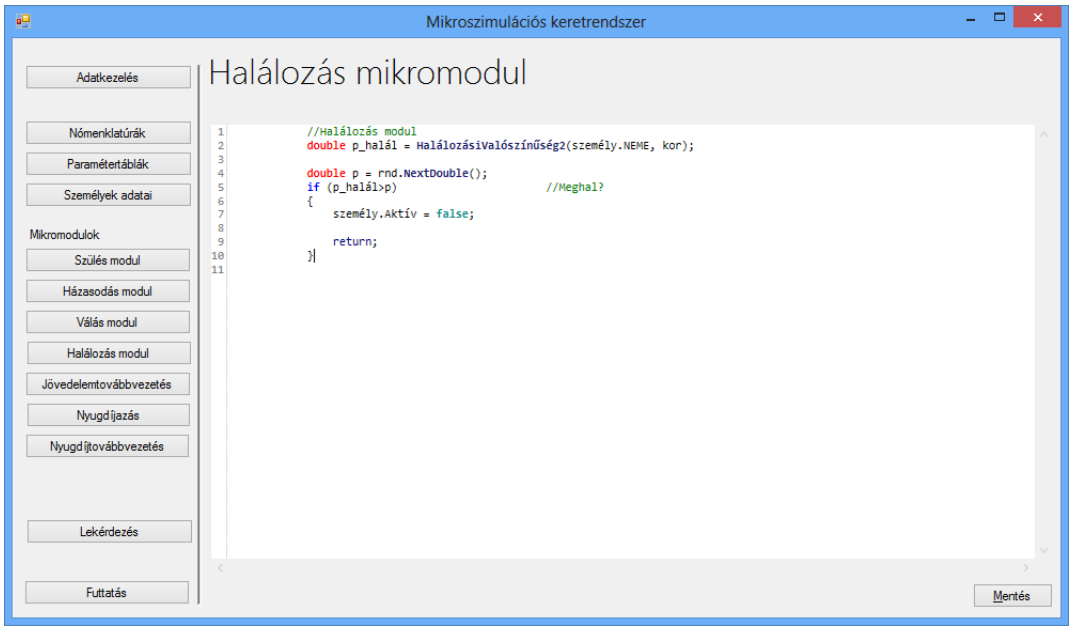
Ebben a lépésben nyílik lehetőség a mikromodulok programkódjának szerkesztésére. A mikromodulok a 4.11. ábrán látható sorrendben kerülnek elhelyezésre a generált mikroszimulációs programban.

4.5.7. Fordítás és futtatás

A mikroszimuláció futtatását két lépés előzi meg: el kell helyezni a keretrendszer által generált kódrészleteket és paramétereket a mikroszimulációs program forráskódjában, majd a forráskódot le kell fordítani futtatható állománnyá. Ez után kerülhet csak sor a tényleges futtatásra, majd az eredmények megjelenítésére.

Kódgenerálás: Ebben a lépésben kerülnek elhelyezésre a keretrendszer által generált kódrészletek a futtatandó szimulációs program forráskódjában. Az egyes kódrészletek a forráskódban meghatározott helyekre kerülnek beszúrásra a 4.2. táblázat szerint.

Fordítás: A .NET keretrendszer, mely a mikoroszimulációs keretrendszer illetve az általa generált mikorszimulációs program futtatásához szükséges, a Windows



4.11. ábra. Mikromodul szerkesztése a keretrendszerben.

telepítésekor kerül a számítógépre. (Windows 7 operációs rendszerrel a .NET 3.5-ös, míg Windows 8-al a 4.5-ös verziója kerül telepítésre.) Linux alapú illetve Macintosh környezetben a Mono projekt biztosít eszközöket a fordításhoz illetve futtatáshoz. A .NET keretrendszer könyvtárában található fordítóprogram, így a generált szimulációs program fordításához nem szükséges, hogy a Visual Studio, vagy más fejlesztőeszköz telepítve legyen.

Futtatás: A futtatás eredménye sikeres futtatás után a vágólapra kerül, így átadható más programnak további feldolgozásra.

Kódrészlet funkciója	Beszúrás helye	Megjegyzés
Nómenklatúrák alapján generált enumerációk:	`#region nomenclatures` `#endregion nomenclatures`	A régióba.
Paramétertáblák által generált tömbök és keresőfüggvények:	`#region paramtables` `#endregion paramtables`	A régióba.
A személyt leíró osztály:	`#region SimulationEntity` `#endregion SimulationEntity`	A régióba.
Születés mikromodul:	`#region SzületésMikromodul` `#endregion SzületésMikromodul`	A régióba.
Halálozás mikromodul:	`#region HalálozásMikromodul` `#endregion HalálozásMikromodul`	A régióba.
Súlyozott állományt jelölő logikai változó:	`bool UseWeights = true;`	Értékként.
Súlyt tartalmazó oszlop sorszáma:	`int WeightColumn = 256;`	Értékként.
Metaadatok:	Megjegyzésként a kódba.	

4.2. táblázat. Változó kódrészletek jelölése a szimulációs programban.

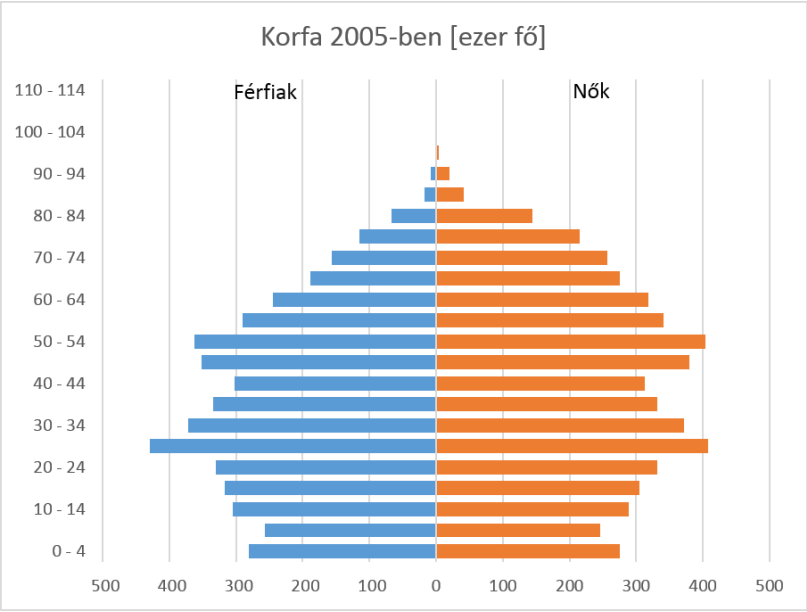
4.6. Futási eredmények

A szimulációs program helyességének vizsgálatára születtek a szimulációs kontrolltáblák. A 4.3. táblázat példaként a halálozásokkal kapcsolatos kontrolltábla egy részletét tartalmazza. Kiolvasható belőle, hogy egy szimulációs lépésben hány adott nemű/korú személy élt, ennek megfelelően hányszor generáltunk véletlen eseményt a halálozás eseményének eldöntéséhez, és ebből hányszor következett be a haláleset. Az elhunytak számával csökken az adott nemű és korú lakosság, így nem mindenki léphet következő életévébe. A kontrolltáblák azt ellenőrzik, hogy sikerültek-e az eseménysorsolások, azaz ha a kontrolltáblák egyes celláiban szereplő esetek számát elosztjuk az azonos kategóriába tartozó emberek számával, visszakapjuk a szülési illetve halálozási valószínűségeket.

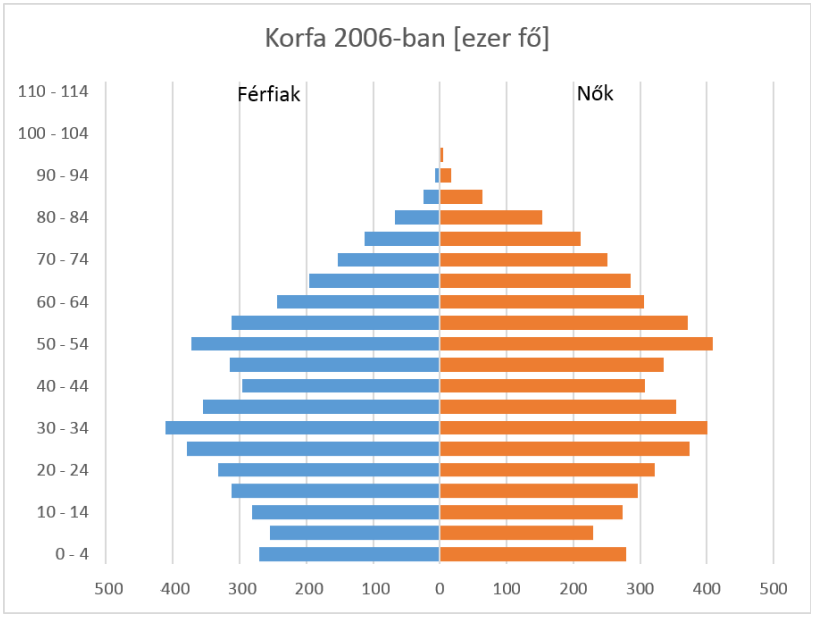
Követve a demográfiai szakirodalomban megszokott korfa táblákat, elkészítettem a szimuláció eredményeként létrejövő adatállományokból is a korfákat. Az első években ezek teljesen megegyeznek az ismert, a KSH által közzétett korfákkal, míg az 50 évvel előrevezetettek esetében – elsősorban a paramétertáblák bizonytalansága miatt, eltér az eredmény a megszokottól.

Nem	Életkor	Halálozási valószínűség paramétertábla alapján	2005				2006				2054			
			Kísérletek száma	Ebből a bekövetkezett esetek száma	Bekövetkezés aránya a kísérlethez képest	Eltérés (különbség)	Kísérletek száma	Ebből a bekövetkezett esetek száma	Bekövetkezés aránya a kísérlethez képest	Eltérés (különbség)	Kísérletek száma	Ebből a bekövetkezett esetek száma	Bekövetkezés aránya a kísérlethez képest	Eltérés (különbség)
Férfi	0	0,532%	56210	279	0,496%	0,036%	47779	251	0,525%	0,007%	33458	199	0,595%	-0,063%
Férfi	1	0,036%	46386	20	0,043%	-0,007%	55931	13	0,023%	0,013%	42361	14	0,033%	0,003%
Férfi	2	0,016%	45542	10	0,022%	-0,006%	46366	5	0,011%	0,005%	42139	4	0,009%	0,007%
Férfi	3	0,024%	46549	20	0,043%	-0,019%	45532	10	0,022%	0,002%	43010	15	0,035%	-0,011%
Férfi	4	0,012%	50168	6	0,012%	0,000%	46529	4	0,009%	0,003%	46588	3	0,006%	0,006%
Férfi	5	0,012%	55185	4	0,007%	0,005%	50162	10	0,020%	-0,008%	50996	9	0,018%	-0,006%
Férfi	6	0,014%	45844	5	0,011%	0,003%	55181	4	0,007%	0,007%	42278	2	0,005%	0,009%
Férfi	7	0,014%	46487	6	0,013%	0,001%	45839	6	0,013%	0,001%	43444	6	0,014%	0,000%
Férfi	8	0,014%	56622	8	0,014%	0,000%	46481	7	0,015%	-0,001%	52184	7	0,013%	0,001%
Férfi	9	0,016%	54596	8	0,015%	0,001%	56614	7	0,012%	0,004%	50741	11	0,022%	-0,006%
Férfi	10	0,004%	53179	2	0,004%	0,000%	54588	3	0,005%	-0,001%	48879	2	0,004%	0,000%
Férfi	11	0,016%	55917	6	0,011%	0,005%	53177	4	0,008%	0,008%	55802	11	0,020%	-0,004%
Férfi	12	0,015%	54841	5	0,009%	0,006%	55911	6	0,011%	0,004%	54713	13	0,024%	-0,009%
Férfi	13	0,029%	62837	18	0,029%	0,000%	54836	16	0,029%	0,000%	62660	17	0,027%	0,002%
Férfi	14	0,024%	68571	19	0,028%	-0,004%	62819	13	0,021%	0,003%	68430	13	0,019%	0,005%
Férfi	15	0,039%	62185	16	0,026%	0,013%	68552	31	0,045%	-0,006%	62012	25	0,040%	-0,001%
Férfi	16	0,033%	66932	17	0,025%	0,008%	62169	23	0,037%	-0,004%	66761	26	0,039%	-0,006%
Férfi	17	0,041%	54328	28	0,052%	-0,011%	66915	29	0,043%	-0,002%	54213	27	0,050%	-0,009%
Férfi	18	0,057%	60795	33	0,054%	0,003%	54300	28	0,052%	0,005%	60642	31	0,051%	0,006%
Férfi	19	0,051%	65853	25	0,038%	0,013%	60762	29	0,048%	0,003%	65701	31	0,047%	0,004%
Férfi	20	0,084%	68467	54	0,079%	0,005%	65828	70	0,106%	-0,022%	68332	45	0,066%	0,018%

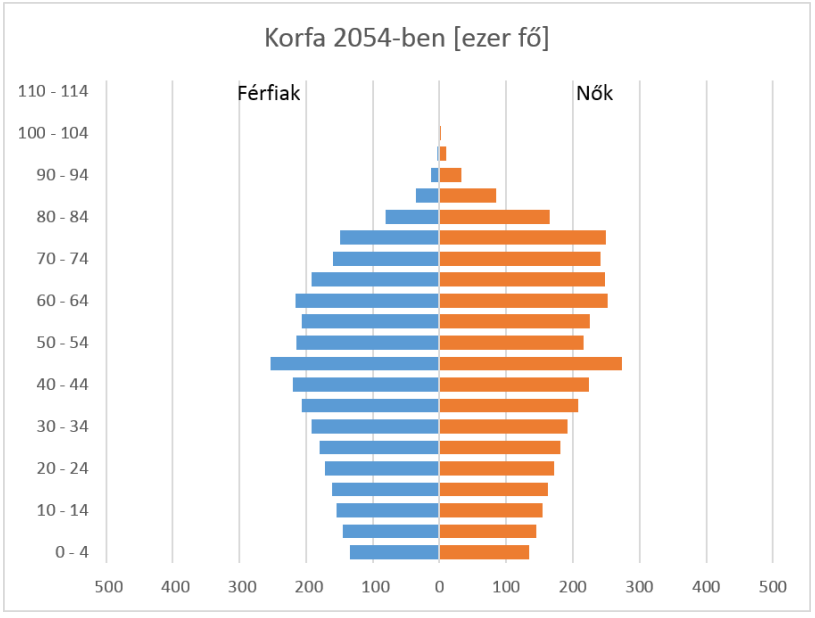
4.3. táblázat. Szimulációs kontrolltábla részlete.



4.12. ábra. Korfa a kiinduló állomány alapján, 2005-ben.



4.13. ábra. Korfa a továbbvezetett állomány alapján, 2006-ra.



4.14. ábra. Korfa a továbbvezetett állomány alapján, 2054-re.

5. fejezet

Összefoglalás

5.1. Főbb eredmények összefoglalása

5.1.1. Az ABS algoritmussal kapcsolatban megfogalmazott tézisek

T1: A módosított Huang módszer nagy dimenzióban is megjelenő stabilitása alátámasztja Gáti Attila dolgozatában állított stabilitást.

T2: A módosított Huang algoritmus stabil, a generált pi vektorok ortogonálisak, amennyiben a H1=I mátrixszal indítunk. Alacsony memóriaigénye miatt különösen alkalmas GPU-n történő futtatásra.

T3: Parlett és Kahan bebizonyította, hogy az általuk a CGS klasszikus Gram-Scmitt módszerre kidolgozott "twice is enough" eljárás lényegesen pontosítja a lineáris egyenletrendszer megoldását. Ez az eljárás alkalmazható a módosított Huang módszerre is, amely szintén ortogonális vektorokat gyárt, és az általam alkalmazott ABS projekciós mátrixával való újravetítés ezt a funkciót látja el. A megoldás javítása nagy dimenzióban (8000 dimenzióban) telt mátrixokra is működik. Ezt igazoltuk a teszt feladatokon.

5.1.2. A Lovász-Vempala algoritmussal kapcsolatban megfogalmazott tézis

T4: Az LVD algoritmus csak egy pontszálat léptetett, ezért pontok függetlenségének biztosításához nem használ fel minden pontot az integráláshoz, hanem egy fázisonként változó alacsony értéknek megfelelő számú pontot minden integrálás

után kihagy. A disszertációban bemutatott PLVDM több-ezer pont-szálat léptet. Ebből adódóan nem kell kihagyni pontokat az integrálások között, mert a pontok száma már olyan magas, hogy a kihagyott pont helyén (vagy legalábbis közvetlen közelében) hamarosan úgyis keletkezne egy másik pont. A számítógépes kísérletek alátámasztották azt a feltevést, hogy a több pont-szál használata hozzájárul a generált pontok függetlenségének biztosításához. Ezáltal a Lovász-Vempala algoritmus PLVDM implementációja nem csupán a számítóegységek száma mentén ér el hatékonyságnövekedést. A PLVDM algoritmus által bevezetett varianciacsökkentő eljárások bár javítják az algoritmus hatékonyságát, nem váltották be a hozzájuk fűzött reményeket.

5.1.3. A mikroszimulációs keretrendszerrel kapcsolatban megfogalmazott tézisek

T5: A hagyományos mikroszimulációs szolgáltató rendszerek futásának többórás várakozási idejét a mai felhasználó nem tolerálja, ezért fel kellett gyorsítani a jól ismert algoritmusokat. A párhuzamos processzálási technikával sikerült a futásteljesítményt a végfelhasználók által elfogadható szintre hozni.

T6: Továbbra is cél, hogy a mikroszimulációs keretrendszerek jól paraméterezhetők legyenek, és különböző célú feladatokra legyenek alkalmazhatók. A kidolgozott rendszer paraméterezhető, mind a feldolgozott adatállomány szerkezetét illetően, mind a becslési algoritmusok és paramétereik, mind a mikromodul építés és annak futtatási sorrendjének meghatározása szempontjából.

5.2. Tapasztalatok összegzése

A gazdasági életben hozott döntéseket támogató számítások között sok a nagy számításigényű probléma. A különböző véletlen számokon alapuló Monte Carlo szimulációk, az összefüggő valószínűségek modellezései is nagy számításigényű feladatok, ahol az eredmény pontosságát a végrehajtott szimulációs lépések száma döntően befolyásolja. A gyakorlatban egy számítás elvégzésére rendelkezésre álló idő általában korlátozó tényező.

A piacon elérhető számítóegységek számítókapacitása a fizikai korlátokat közelíti, ebből adódóan az időegység alatt elvégzett műveletek terén nagyságrendi előrelépés csak több számítóegységből álló párhuzamos architektúra alkalmazásával érhető el. A párhuzamos programfuttatásra alkalmas, több számítóegységet tartalmazó eszközök

lassan megtalálhatók minden számítógépben és mobiltelefonban. A feladat olyan algoritmusok tervezése, amelyek hatékonyan tudják kihasználni a rendelkezésre álló további számítóegységeket.

Az első lépés a feladathoz legjobban illeszkedő architektúra megtalálása. Sok probléma esetén egyedi fejlesztésű célprocesszor nyújtaná a legjobb teljesítményt, de a költségek és az időkorlátok miatt általában be kell érni valamely általános célú párhuzamos architektúra alkalmazásával. (Az FPGA-k hibrid megoldásként lehetőséget adnak célprocesszorok kialakítására programozottan összekapcsolt logikai kapukból, de a fejlesztéshez szükséges időbeni és anyagi ráfordítások miatt az FPGA alapú megoldások aránylag ritkák.) A különböző architektúrák más-más feladattípusok esetén nyújtanak jó teljesítményt. Általános célú párhuzamos architektúrából többféle áll rendelkezésre a piacon. A dolgozatban szereplő algoritmusok tapasztalatai segítenek a választásban. Az első fejezet áttekintést ad arról, hogy mely általános célú architektúra milyen feladattípusnál hozhat jó teljesítményt. A párhuzamos architektúrákra történő algoritmustervezés és szoftverfejlesztés egészen más megközelítést igényel, mint az egyszálas megközelítés. A dolgozat írása során is beigazolódott, hogy a meglévő algoritmusok implementációja párhuzamos architektúrán nem csupán szoftverfejlesztési szakmunka, hanem komoly tervezést igénylő matematikai és mérnöki feladat, mely sok intuitív ötletet is igényel. Nem létezik olyan automatizálható módszertan, mellyel egy eredetileg egy processzorra tervezett algoritmus hatékonyan átültethető lenne több párhuzamos rendszerre.

A gazdasági élethez kapcsolódóan sok feladat vezethető vissza lineáris egyenletrendszerek megoldására. A sokismeretlenes egyenletrendszerek megoldása komoly számítókapacitást igényel, valamint számolni kell a gépi számábrázolásból és az algoritmusok instabilitásából adódó hibákkal is. A lineáris egyenletrendszerek megoldása elsősorban mátrix és vektorműveleteket igényel. A mátrix műveletek sajátossága, hogy nagyon sok adatot mozgatnak meg, melyen egyszerű összeadás és szorzás műveleteket kell csak végrehajtani. A GPU architektúrák széles adatbuszuk miatt különösen alkalmasak mátrixművletek felgyorsítására. Az ABS algoritmus módosított Huang változata az adatmozgatások mintázata és a kedvező memóriaigény miatt különösen jól illeszkedik a CUDA architektúrához. Az adatmozgatások optimalizálása után egy 4096 ismeretlenes egyenletrendszer megoldását GeForce GTX 570-es kártyán 100 másodperc környékére sikerült leszorítani, miközben az algoritmus kiváló stabilitási tulajdonságai empirikusan is igazolást nyertek.

Nagyon nehéz jó becslést adni arra, hogy egy adott probléma megoldásánál mely konkrét párhuzamos architektúra ténylegesen milyen teljesítményt nyújt a gyakor-

latban. Ez különösen igaz a CUDA architektúrára, ahol a párhuzamosan működő aritmetikai egységek csoportonként közös vezérlőre vannak kötve. Ebből adódóan a közös vezérlőn párhuzamosan futó szálaknak feltételes elágazások és eltérő lépésszámú ciklusok esetén be kell várniuk egymást. A harmadik fejezet írása közben szerzett gyakorlati tapasztalat is azt mutatja, hogy az algoritmus fejlesztésének előrehaladtával egyre több eset kerül azonosításra és kezelésre, és emiatt rohamosan nő a feltételes elágazások száma a programban. Ez azzal járhat, hogy a kísérleti projekt alapján becsült, ígéretes futási sebességtől a tényleges futási sebesség fokozatosan elmarad. Ilyenkor az algoritmus működésén kell módosítani ahhoz, hogy egy adott architektúrán jó teljesítményt nyújtson. Jó példa erre dolgozat harmadik fejezetében szereplő Lovász-Vempala térfogatszámító algoritmus, ahol többek között a sokdimenziós testek kezelésén kellett változtatni ahhoz, hogy az algoritmus illeszkedjen a CUDA architektúrához. Párhuzamos programozási technikák alkalmazásával az algoritmus módosított változatán (PLVDM) keresztül először vált vizsgálhatóvá a Lovász-Vempala algoritmus viselkedése magasabb dimenziókban, és a gyakorlatban is alkalmazható implementáció született.

A fejlesztéshez szükséges emberi erőforrást különösen a közösen használt memóriaterületeket zároló, bonyolult, többszálú algoritmusok esetén nehéz becsülni. A zárak feloldására váró szálak között például körbetartozáshoz hasonló helyzet alakulhat ki, melyből a program nem tud elmozdulni. A szálak futási sebessége nem determinisztikus, éppen ezért bizonyos hibák csak ritkán, bizonyos időbeni együttállásoknál jelentkeznek, ezért nagyon nehéz őket tetten érni és reprodukálni. Ez bizonytalan működéshez, illetve időben elhúzódó fejlesztésekhez vezethet. Viszont gondos tervezéssel és implementációval nagy számításigényű problémák is elfogadható időn belül oldhatók meg mindenki számára elérhető általános célú hardveren.

Ilyen például Magyarország népességének mikroszimulációval történő továbbvezetése. A mikroszimulációs módszer egyesével követi a népesség minden egyedének sorsát tipikusan éves szimulációs lépésekben. A negyedik fejezetben bemutatott mikroszimulációs keretrendszer egyfajta „program a programban”. Az egyed sorsáról – elhalálozás, szülés, házasodás, válás, stb. a szimulációs lépésekben történik döntés paraméter-táblák alapján. A szimulációs lépés programkódja, az un. mikromodul aránylag egyszerű, a nem informatikus végzettségű elemző közgazdász számára is könnyen értelmezhető, módosítható. A szimulációs lépés kódjának változtatásával, illetve a születési és halálozási valószínűségeket tartalmazó paramétertáblák alakításával sokféle szcenárió és jövőalternatíva kipróbálható. A mikroszimulációs keretrendszer, mely a népszámlálási adatokból kiindulva a teljes népesség adatait képes

beolvasni és ezeken a szimulációs lépéseket évenként végrehajtani összetettebb, párhuzamos programozási technikákra épül. A keretrendszer tervezése és felépítése szoftvertechnológiai jellegű feladat volt.

Több architektúrához és szoftverfejlesztő környezethez rendelkezésre állnak olyan monitorozó eszközök, amelyek segítségével meg lehet határozni a program futását korlátozó szűkös erőforrást.

A párhuzamos algoritmusok tervezőjének jól kell ismernie azt az architektúrát, melyre az algoritmust tervezi; az algoritmustervezési és a szoftverfejlesztői munka összefonódik.

5.3. További fejlesztési tervek és irányok

- A PLVDM algoritmussal tervezett további kísérletezéshez szimplexeket szeretnék próbatestnek használni, mivel ezek generátor algoritmusa és térfogata is ismert tetszőleges dimenzióban.
- A 20 dimenziós korlát átlépéséhez több számítógépből álló klaszterre tervezem átírni az algoritmust, amivel a futásidő közel lineárisan skálázhatóvá válik.
- A szimulációs keretrendszer alkalmassá tehető arra, hogy MPI (Message Passing Interface) segítségével több számítógépből álló klaszteren végezze a szimulációt. Ezzel a megoldással akár az egyetemi géptermek tanórán kívüli kapacitása is felhasználható tudományos célokra. A megvalósíthatósággal kapcsolatos kísérleteket – az egyetemi informatikai biztonsági házirend figyelembevételével – elvégeztem.

Irodalomjegyzék

Abaffy, J./Spedicato, E./Broyden, C.G (1984): A Class of Direct Methods for Linear Equations. Numerische Mathematik, 45, 361–376

Abaffy, Jozsef/Spedicato, Emilio (1989): ABS projection algorithms: mathematical techniques for linear and nonlinear equations.

Bakkum, P./Skadron, K. (2010): In Accelerating SQL Database Operations on a GPU with CUDA., Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units

Berry, R. (2009): Stress Testing Value-at-Risk. Investment Analytics and Consulting No. 6

Chan, E. (2008): Quantitative Trading: How to Build Your Own Algorithmic Trading Business. John Wiley & Sons

Chan, E. (2013): Algorithmic Trading: Winning Strategies and Their Rationale. John Wiley & Sons

Csicsman, J./László, A. (2012): Microsimulation Service System. Hungarian Electronic Journal of Sciences

De Vogeleer, Karel et al. (2014): The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In Parallel Processing and Applied Mathematics Springer, 793–803

Deák, I. (1979): Comparison of methods for generating uniformly distributed random points in and on a hyperspere. Problems of Control and Information Theory, No. 8, 105–113

Deák, I. (1990): Random number generators and simulation, in: Mathematical Methods of Operations Research (series editor A. Prékopa). Budapest: Akadémiai Kiadó

Deák, I. (2002): Probabilities of simple n-dimensional sets in case of normal distribution. IIE Transactions (Operations Engineering), No. 34, 1–18

Deák, I. (2011): Efficiency of Monte Carlo computations in very high dimensional spaces. Central European Journal of Operations Research, 19, 177–189

Devroye, L. (1986): Non-Uniform Random Variate Generation. Springer Verlag, 843

Dyer, M./Frieze, M./Kannan, R. (1991): A random polynomial-time algorithm for approximating the volume of convex bodies. Journal of the Association for Computing Machinery, 38, 1–017

E., Kovács (2010): A nyugdíjreform demográfiai korlátai. Hitelintézeti Szemle, 2, 128–149

Fábián, C. I. (2013): Computational aspects of risk-averse optimization in two-stage stochastic models. Stochastic Programming E-Print Series No. 3

Feynman, R. (2010): Richard Feynman. Quantum Algebra and Symmetry,, 680

Flynn, Michael (1972): Some computer organizations and their effectiveness. Computers, IEEE Transactions on, 100 No. 9, 948–960

Frederik, H. (2001): The Danish Micro Simulation Tax model., V. Pénzinformatikai Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2001. Október 15-16.

Gassmann, H./Deák, I./Szántai, T. (2002): Computing multivariate normal probabilities. J. Computational and Graphical Statistics, 11, 920–949

Gáti, A. (2013): Automatic roundoff error analysis of numerical algorithms. Ph. D thesis, Apllied Informatics Doctoral School, Óbuda University

Godfrey, M. D./Hendry, D. F. (1993): The Computer As Von Neumann Planned It. IEEE Ann. Hist. Comput. 15 No. 1, 11–21 ⟨URL: `http://dx.doi.org/10.1109/85.194088`⟩, ISSN 1058–6180

Goetz, B./Peierls, T. (2006): Java Concurrency in Practice. Addison-Wesley

Hammersley, J.M./Handscomb, D.C. (1964): Monte Carlo methods. London: Methuen

Haque, Imran S/Pande, Vijay S (2010): Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing. IEEE, 691–696

Immerwoll, H. (2001): The Impact of Inflation on Income Taxes and Social Insurance Contributions., V. Pénzinformatikai Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2001. Október 15-16.

J., Abaffy (1979): A lineáris egyenletrendszerek általános megoldásának egy direkt módszerosztálya. Alkalmazott Matematikai Lapok, 5, 223–240

J., Csicsman (1987): A mikroszimulációs rendszer számítástechnikai hátterének kialakítása., (KSH) A Háztartási Mikroszimulációs Rendszer munkálatai, Ts-3/8/8 tanulmánysorozat, 1. kötet

J., Csicsman (2001): A BME PIKK Mikroszimulációs projektjének célkituzései és meggondolásai., (V. Pénzinformatikai Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2001. Október 15-16.)

J., Csicsman/C., Fényes (2003): A Mikroszimulációs Szolgáltató Rendszer fejlesztése. Alma Mater sorozat - Üzlet, folyamat, monitoring

Kannan, R./Lovász, L./Simonovits, M. (1997): Random walks and an $O^*(n^5)$ volume algorithm for convex bodies. Random Structures and Algorithms, 11, 1–50

Klevmarken, N. A./Olovsson, P. (1996): Direct and behavioral effects of income tax changes: simulations with the Swedish model MICROHUS. Amsterdam: Elsevier Science Publishers

L., Hablicsek (2007): Társadalmi-demográfiai eloreszámítások a nyugdíjrendszer átalakításának modellezéséhez., jelentés a Nyugdíj és Időskor Kerekasztal számára

Lovász, L./Deák, I. (2012): Computational results of an $O^*(n^4)$ volume algorithm. European Journal of Operational Research, 216, 152–161

Lovász, L./Simonovits, M. (1992): On the randomized complexity of volume and diameter. In 33rd IEEE Annual Symp. on Foundations of Comp. Sci., 482–491

Lovász, L./Simonovits, M. (1993): Random walks in a convex body and an improved volume algorithm. Random Structures and Algorithms, No. 4, 359–412

Lovász, L./Vempala, S. (2003): Simulated annealing in convex bodies and an $O^*(n^4)$ volume algorithm. In Proc. of FOCS., 650–659

Lovász, L./Vempala, S. (2006): Simulated annealing in convex bodies and an $O^*(n^4)$ volume algorithm. J. of Computer and System Sciences, 72, 392–417

M., Zafí r (1987): A háztartási mikroszimulćió. Koncepció, rendszerleírás., (KSH) A Háztartási Mikroszimulációs Rendszer munkálatai, Ts-3/8/8 tanulmánysorozat, 1. kötet

Metropolis, N. et al. (1953): Equation of state calculations by fast computing machines. J. Chem. Phys. 21, 1087–1092

Molnar, Istvan/Sinka, Imre (2007): Toward Agent-Based Microsimulation–Another Approach. In Modelling & Simulation, 2007. AMS'07. First Asia International Conference on Modelling and Simulation. IEEE, 403–408

Morris, D. (2001): A Micro-Simulation Modelling System for HM Treasury., V. Pénzinformatikai Konferencia, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2001. Október 15-16."

Nvidia, CUDA (2011): NVIDIA CUDA C Programming Guide. electronic document

O'Donoghue, Cathal (2001): Dynamic Microsimulation: A Methodological Survey. Brazilian Electronic Journal of Economics 4 No. 2

Orkutt, G./Greenberger, M. (1961): Microanalysis of socioeconomic systems. New York: Harper and Brothers

Pardo, Robert (2011): The Evaluation and Optimization of Trading Strategies. John Wiley & Sons

Romeijn, E./Smith, R.L. (1994): Simulated annealing for constrained global optimization. J. of Global Optimization, 5, 101–126

Rudelson, M. (1999): Random vectors in the isotropic position. J. Func. Anal. 164, 60–72

Sanders, J./Kandort, E. (2010): CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional , 312 pages

Simonovits, M. (2003): How to compute the volume in high dimensions. Math. Programming Ser. B. 97, 337–374

Smith, R.L. (1996): The hit and run sampler: a globally reaching Markov chain sampler for generating arbitrary multivariate distribution. In Proc. 28th Conference on Winter Simulation., 260–264

T., Kiss/I., Csata (2007): A magyar népesség elöreszámításának lehetoségei Erdélyben. Demográfia No. 7

Zverovich, V. et al. (2012): A computational study of a solver system for processing two-stage stochastic LPs with enhanced Benders decomposition. Mathematical Programming Computation, 4, 211–238

Publikációk jegyzéke

Csetényi, A./ Mohácsi L./ Várallyai L. (2007): Szoftverfejlesztés. HEFOP, Debrecen, ISBN : 978-963-9732-56-8 p. 157

Mohácsi L./Rétallér O. (2013): A mechanical approach of multivariate density function approximation. Proceedings of the International Conference on Modeling and Applied Simulation, ISBN 978-88-97999-17-1, pp. 179-184.

Forgács A. / Mohácsi L. (2014): Gazdasági számítások párhuzamos architektúrákon. GIKOF Journal, HU ISSN 1588-9130, pp. 6-14.

Mohácsi L. / Deák I. (2014): A parallel implementation of an O*(n4) volume algorithm. Central European Journal of Operations Research, DOI :10.1007/s10100-014-0354-7.

Függelékek

A. függelék

Grafikus kártya paraméterei

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:          2
Minor revision number:          0
Name:                           GeForce GTX 570
Total global memory:            1341849600
Total shared memory per block:  49152
Total registers per block:      32768
Warp size:                      32
Maximum memory pitch:           2147483647
Maximum threads per block:      1024
Maximum dimension 0 of block:   1024
Maximum dimension 1 of block:   1024
Maximum dimension 2 of block:   64
Maximum dimension 0 of grid:    65535
Maximum dimension 1 of grid:    65535
Maximum dimension 2 of grid:    65535
Clock rate:                     1560000
Total constant memory:          65536
Texture alignment:              512
Concurrent copy and execution:  Yes
Number of multiprocessors:      15
Kernel execution timeout:       No
Can map host memory:            Yes
```

B. függelék

Térfogatszámító algoritmus eredménye

```
* Generating a cube in 3 dimensions:

  1.000000 0.000000 0.000000 < 2.828427
  0.000000 1.000000 0.000000 < 1.000000
  0.000000 0.000000 1.000000 < 1.000000
  0.000000 1.000000 0.000000 > -1.000000
  0.000000 0.000000 1.000000 > -1.000000

* Random seed generator kernel started for 64000 threads.. Done.
* Initial point generator kernel started..Done. 5000 Kbytes
have been allocated
in GPU memory
* Calculating first integrals..

  Determining number of intervals:
  #     ti_uppper                chiValue
  1     0.50000000000000000000   0.222838
  2     0.25000000000000000000   0.326700
  3     0.12500000000000000000   0.416323
  4     0.06250000000000000000   0.499012
  5     0.03125000000000000000   0.577464
  6     0.01562500000000000000   0.653014
  7     0.00781250000000000000   0.726435
  8     0.00390625000000000000   0.798216
  9     0.00195312500000000000   0.868691
  10    0.00097656250000000000   0.938099
  11    0.00048828125000000000   1.006614
```

```
12      0.00024414062500000000   1.074374
13      0.00012207031250000000   1.141481
14      0.00006103515625000000   1.208022
15      0.00003051757812500000   1.274065
16      0.00001525878906250000   1.339665
17      0.00000762939453125000   1.404870
18      0.00000381469726562500   1.469720
19      0.00000190734863281250   1.534248
20      0.00000095367431640625   1.598484
21      0.00000047683715820313   1.662454
22      0.00000023841857910156   1.726178
23      0.00000011920928955078   1.789677
24      0.00000005960464477539   1.852967
25      0.00000002980232238770   1.916064
26      0.00000001490116119385   1.978982
27      0.00000000745058059692   2.041732
28      0.00000000372529029846   2.104325
29      0.00000000186264514923   2.166772
30      0.00000000093132257462   2.229080
31      0.00000000046566128731   2.291259
32      0.00000000023283064365   2.353316
33      0.00000000011641532183   2.415257
34      0.00000000005820766091   2.477089
35      0.00000000002910383046   2.538818
36      0.00000000001455191523   2.600449
37      0.00000000000727595761   2.661986
38      0.00000000000363797881   2.723434
39      0.00000000000181898940   2.784798
40      0.00000000000090949470   2.846081
Number of sufficient intervalls: 39

Preliminary calculations for the first integral:
#       StDev                    Avg
1       0.031724056458968        1.779821932519904
2       0.028604168994740        2.554282482750348
3       0.027841427941043        2.874832990631441
4       0.026602422233796        2.977975703852009
5       0.024962888111412        2.946765738697611
6       0.023081028311838        2.830498698344823
7       0.021087566087186        2.662050138147862
8       0.019078455772862        2.464202365834726
9       0.017120439995976        2.252875133489078
10      0.015257655461085        2.039103262275927
```

11	0.013517298443494	1.830373907745884
12	0.017147639928198	1.577670660650528
13	0.018478991255939	1.352922700588939
14	0.020546541703267	1.114632991450738
15	0.020311504472320	0.91623832708357 9
16	0.019606931489839	0.734067624894141
17	0.018082287696976	0.585388905071997
18	0.016166364144928	0.467892377604038
19	0.014276912927884	0.375738042950740
20	0.012452423555982	0.287454590739676
21	0.010720968562539	0.229328007974209
22	0.009157297284841	0.178635167390807
23	0.007781538431742	0.141913854945337
24	0.006513177530908	0.110357569320605
25	0.005459493928263	0.087600033455740
26	0.004563227032494	0.070130965110430
27	0.003812034416890	0.055764940072505
28	0.003177918926780	0.045006130915944
29	0.002661901715485	0.036618863354028
30	0.002203975507944	0.028704823744927
31	0.001852865304625	0.023637325778554
32	0.001513986942400	0.018091479335970
33	0.001272245490138	0.015036022497898
34	0.001054809101041	0.012035328813974
35	0.000870865390545	0.009582418163905
36	0.000720553230783	0.007680678833992
37	0.000595265670018	0.006179083798499
38	0.000498076048352	0.005136515768462
39	0.000410180243656	0.004057214405851

Determining sample sizes:

interval#	Sample size
1	70656
2	63680
3	61984
4	59232
5	55584
6	51392
7	46944
8	42464
9	38112
10	33952
11	30080

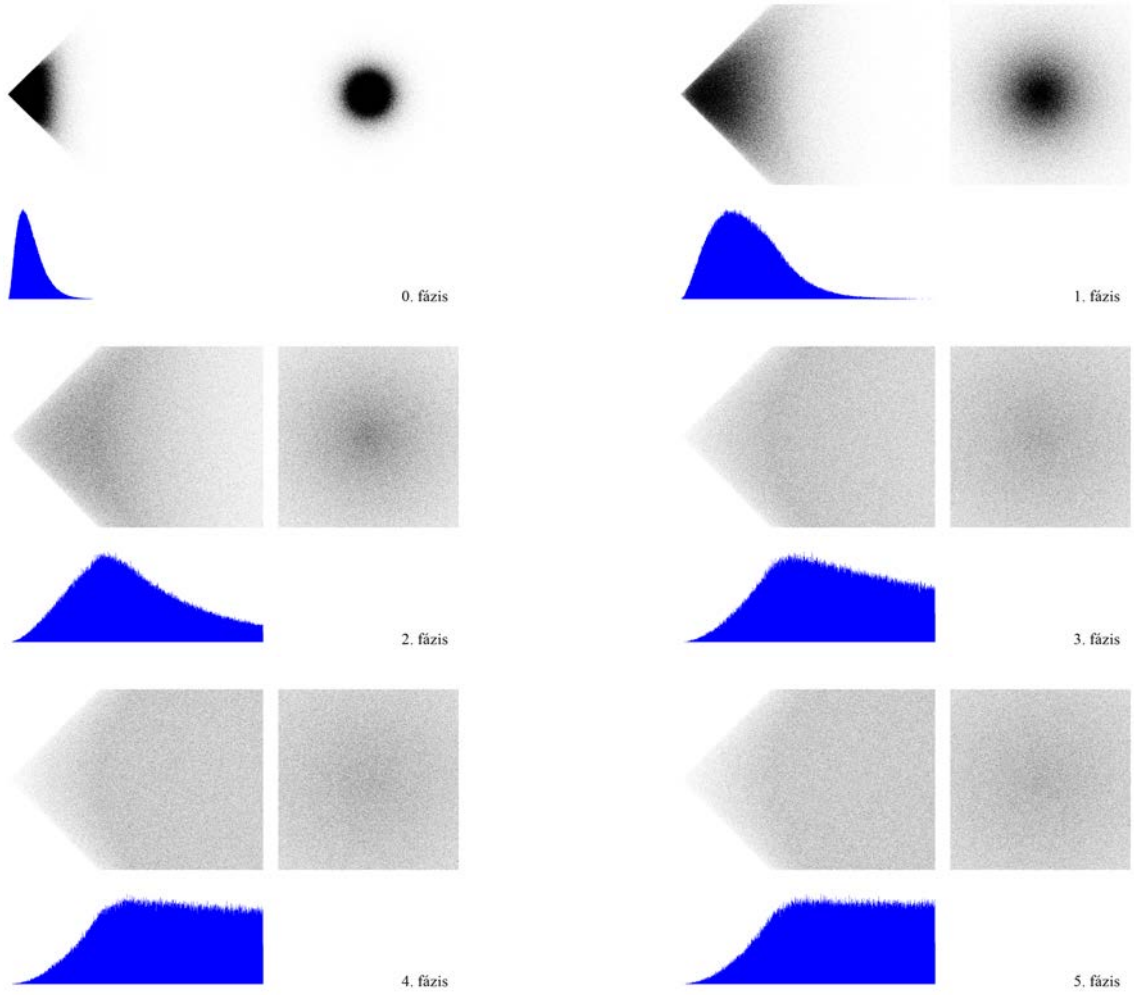
```
12              38176
13              41152
14              45760
15              45216
16              43648
17              40256
18              36000
19              31776
20              27712
21              23872
22              20384
23              17312
24              14496
25              12128
26              10144
27              8480
28              7072
29              5920
30              4896
31              4096
32              3360
33              2816
34              2336
35              1920
36              1600
37              1312
38              1088
39              896
Total error of sample first integrals: 0.470787

Calculating the first integral:
interval#       StDev           Avg
1               0.002719        1.788930
2               0.002595        2.562396
3               0.002560        2.883165
4               0.002502        2.984746
5               0.002423        2.952505
6               0.002332        2.835384
7               0.002230        2.666021
8               0.002121        2.467470
9               0.002010        2.255793
10              0.001899        2.042381
11              0.001791        1.834043
```

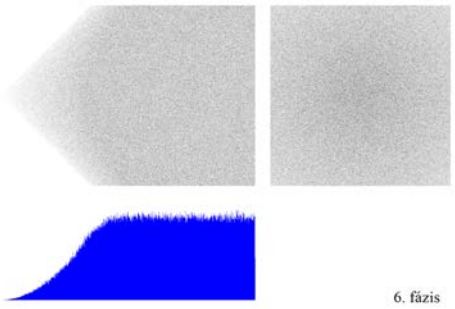
```
12              0.001780        1.599586
13              0.002143        1.341067
14              0.002209        1.103344
15              0.002221        0.896751
16              0.002148        0.720391
17              0.002055        0.575512
18              0.001949        0.456396
19              0.001827        0.362893
20              0.001700        0.287686
21              0.001577        0.229043
22              0.001458        0.182448
23              0.001341        0.145187
24              0.001242        0.116911
25              0.001142        0.093015
26              0.001051        0.074859
27              0.000968        0.060747
28              0.000890        0.049442
29              0.000814        0.039763
30              0.000742        0.031359
31              0.000679        0.025420
32              0.000618        0.020001
33              0.000560        0.016069
34              0.000506        0.012480
35              0.000463        0.010170
36              0.000420        0.008135
37              0.000379        0.006417
38              0.000349        0.005385
39              0.000316        0.004254

First integral          :35.747564
Error of first integral :0.058727

* Calculating integrals for 2+1 dimensions on 64000 threads
at 600 iteration steps:

#       Ai              gammai          Average
1       3.514719        2.485281        0.755579
2       1.029437        1.000000        1.368416
3       0.301515        1.000000        1.658409
4       0.088312        1.000000        1.741010
5       0.025866        1.000000        1.765811
6       0.007576        1.000000        1.777405
7       0.002219        1.000000        1.775515
```

```
  8      0.000650          1.000000          1.779031
  9      0.000190          1.000000          1.772430
  10     0.000056          1.000000          1.779993
  11     0.000016          1.000000          1.776664
  12     0.000005          1.000000          1.775950

* Running Acceptance-rejection test.. Done.

* Dumping main chart of integral results:

  Phase Z0 average        Reuslt average    stDev             Relative error
  0      0.000000          35.747564         5.872746e-002     0.013539
  1      0.755579          12.566572         1.135111e-001     0.074444
  2      1.368416          3.006880          2.504641e-002     0.068649
  3      1.658409          1.436559          9.074782e-003     0.052062
  4      1.741010          1.115751          4.300330e-003     0.031765
  5      1.765811          1.032906          2.237495e-003     0.017853
  6      1.777405          1.009555          1.194898e-003     0.009755
  7      1.775515          1.002791          6.441803e-004     0.005294
  8      1.779031          1.000817          3.480744e-004     0.002866
  9      1.772430          1.000239          1.889809e-004     0.001557
  10     1.779993          1.000070          1.017630e-004     0.000839
  11     1.776664          1.000021          5.551667e-005     0.000458
  12     1.775950          1.000006          2.867964e-005     0.000236

  Acceptance-rejection test results:

  Sum of accepted points:          19189537
  Count of tested points:          26308203
  Ratio of accepted points:        0.729413
  Error estimation of acceptance: 8.661533E-005

* Calculating results

  za0                                  3.636103E-003
  Estimated volume of the pencil:      8.241527
  Error of pencil volume estimation: 0.362627
  Epsilon V prime:                     2.301999

  RESULT :      3.994748
  TOTAL ERROR: 0.236494 (=0.234358+0.002136)

  Time elapsed:24.508000 ms
```

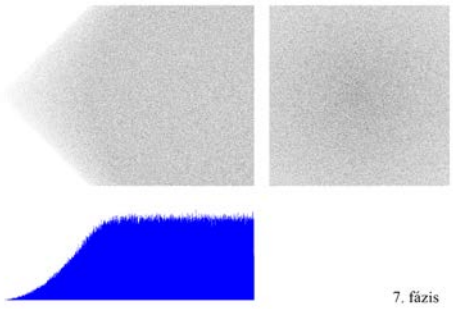
C. függelék

Mintavételi pontok terjedése fázisonként

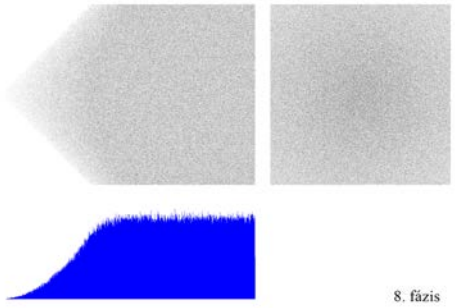




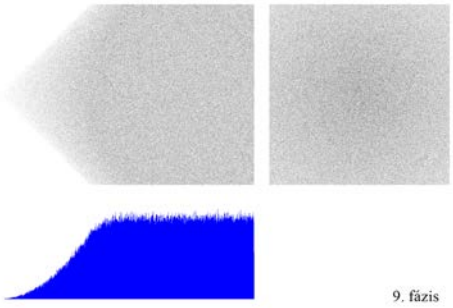
6. fázis



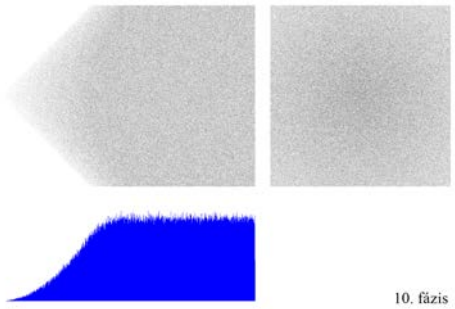
7. fázis



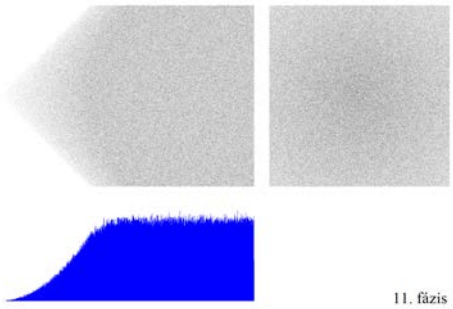
8. fázis



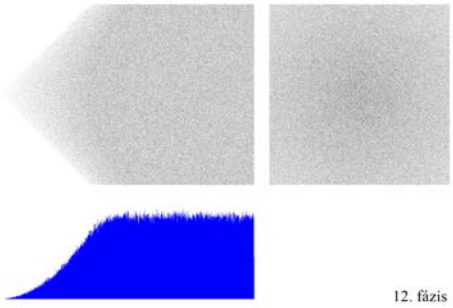
9. fázis



10. fázis



11. fázis



12. fázis

D. függelék

Térfogatszámítási eredmények táblázatokban

5D	C S		C D		O1 S	
Szálak száma(N_p)	512	6400	512	6400	512	6400
Lépésszám fázisonként	8000	6000	3200	6000	2600	6000
Eredmények átlaga	15,98	16,00	15,96	15,98	16,00	15,99
σ	0,17	0,07	0,13	0,05	0,20	0,02
$t\sigma^2$	0,89	0,24	0,61	0,16	1,67	0,11
Futásidő [mp]	32,16	52,93	36,35	60,20	41,81	180,49
Hatásfok	1,00	0,27	0,69	0,18	1,87	0,12
1. futás	15,91	16,03	16,04	16,00	16,00	15,93
2. futás	15,88	16,04	15,97	15,92	16,02	15,97
3. futás	15,99	16,10	16,01	15,98	15,96	16,00
4. futás	15,98	16,02	15,91	15,98	15,96	15,97
5. futás	15,79	16,03	15,94	15,95	15,97	16,02
6. futás	16,05	15,92	15,86	15,95	15,82	15,97
7. futás	15,90	15,97	15,99	16,05	15,94	15,98
8. futás	16,00	15,91	16,15	15,93	15,91	15,97
9. futás	15,99	15,96	15,83	15,99	15,91	15,98
10. futás	16,15	16,00	15,95	16,07	16,01	15,97
11. futás	16,13	15,92	16,17	15,96	16,19	16,00
12. futás	15,94	15,99	15,82	15,94	16,09	16,02
13. futás	15,69	16,06	16,12	15,90	16,13	15,93
14. futás	15,95	16,00	15,91	15,95	15,89	15,95
15. futás	15,86	15,99	16,15	15,97	15,91	16,00
16. futás	16,00	16,02	16,01	16,00	15,90	16,00
17. futás	16,42	15,99	15,84	15,95	15,99	15,95
18. futás	15,86	15,81	15,74	15,92	15,90	16,05
19. futás	15,91	15,93	16,01	15,99	15,97	15,97
20. futás	15,87	15,95	15,99	15,96	16,07	15,96
⋮	⋮	⋮	⋮	⋮	⋮	⋮
100. futás	15,76	16,02	15,89	15,84	16,04	16,01

Table D.1. Futási eredmények $n' = 5$ dimenzióra - 1. rész.

5D	O1 D		O2 S		O2 D	
Szálak száma(N_p)	512	6400	512	6400	512	6400
Lépésszám fázisonként	600	6000	800	6000	800	6000
Average of results	15,99	15,99	16,02	15,99	16,00	15,99
σ	0,11	0,02	0,13	0,04	0,12	0,01
$t\sigma^2$	0,43	0,11	0,45	0,69	0,47	0,11
Futásidő [mp]	38,84	205,93	28,43	497,20	32,31	576,83
Hatásfok	0,48	0,12	0,51	0,77	0,53	0,13
1. futás	16,05	16,03	16,03	15,96	15,95	15,99
2. futás	15,96	15,99	16,00	16,00	15,98	15,99
3. futás	15,96	15,99	16,17	15,97	15,92	15,99
4. futás	15,86	16,03	16,10	15,97	15,83	16,00
5. futás	15,99	15,97	16,10	15,98	15,84	16,00
6. futás	15,81	16,00	16,01	16,00	16,09	15,98
7. futás	16,09	15,99	16,19	15,98	16,03	16,00
8. futás	15,84	15,99	16,06	15,98	16,09	15,99
9. futás	16,25	16,02	15,95	15,99	15,93	15,98
10. futás	15,95	16,01	16,23	15,99	15,92	15,96
11. futás	16,12	16,01	16,11	15,98	16,01	15,97
12. futás	15,94	15,99	15,99	16,00	15,85	16,00
13. futás	15,96	15,97	15,82	16,01	15,87	16,00
14. futás	16,03	15,98	16,15	15,97	15,84	15,99
15. futás	15,94	15,99	16,02	16,00	15,98	15,96
16. futás	16,04	16,01	16,12	15,99	16,17	15,97
17. futás	16,10	15,97	16,04	15,97	16,00	16,01
18. futás	16,01	15,97	16,16	15,98	15,91	15,99
19. futás	15,82	16,04	16,16	15,96	16,29	15,99
20. futás	15,95	16,00	16,05	16,00	15,79	15,97
⋮	⋮	⋮	⋮	⋮	⋮	⋮
100. futás	16,20	15,98	16,00	15,97	16,22	15,98

Table D.2. Futási eredmények $n' = 5$ dimenzióra - 2. rész.

5D	C S		C D		O1 S		O1 D		O2 S		O2 D	
Szálak száma(N_p)	512	6400	512	6400	512	6400	512	6400	512	6400	512	6400
Lépésszám fázisonként	8000	6000	3200	6000	2600	6000	600	6000	800	6000	800	6000
Average of results	15,98	16,00	15,96	15,98	16,00	15,99	15,99	15,99	16,02	15,99	16,00	15,99
σ	0,17	0,07	0,13	0,05	0,20	0,02	0,11	0,02	0,13	0,04	0,12	0,01
$t\sigma^2$	0,89	0,24	0,61	0,16	1,67	0,11	0,43	0,11	0,45	0,69	0,47	0,11
Futásidő [mp]	32,16	52,93	36,35	60,20	41,81	180,49	38,84	205,93	28,43	497,20	32,31	576,83
Hatásfok	1,00	0,27	0,69	0,18	1,87	0,12	0,48	0,12	0,51	0,77	0,53	0,13
1. futás	15,91	16,03	16,04	16,00	16,00	15,93	16,05	16,03	16,03	15,96	15,95	15,99
2. futás	15,88	16,04	15,97	15,92	16,02	15,97	15,96	15,99	16,00	16,00	15,98	15,99
3. futás	15,99	16,10	16,01	15,98	15,96	16,00	15,96	15,99	16,17	15,97	15,92	15,99
4. futás	15,98	16,02	15,91	15,98	15,96	15,97	15,86	16,03	16,10	15,97	15,83	16,00
5. futás	15,79	16,03	15,94	15,95	15,97	16,02	15,99	15,97	16,10	15,98	15,84	16,00
6. futás	16,05	15,92	15,86	15,95	15,82	15,97	15,81	16,00	16,01	16,00	16,09	15,98
7. futás	15,90	15,97	15,99	16,05	15,94	15,98	16,09	15,99	16,19	15,98	16,03	16,00
8. futás	16,00	15,91	16,15	15,93	15,91	15,97	15,84	15,99	16,06	15,98	16,09	15,99
9. futás	15,99	15,96	15,83	15,99	15,91	15,98	16,25	16,02	15,95	15,99	15,93	15,98
10. futás	16,15	16,00	15,95	16,07	16,01	15,97	15,95	16,01	16,23	15,99	15,92	15,96
11. futás	16,13	15,92	16,17	15,96	16,19	16,00	16,12	16,01	16,11	15,98	16,01	15,97
12. futás	15,94	15,99	15,82	15,94	16,09	16,02	15,94	15,99	15,99	16,00	15,85	16,00
13. futás	15,69	16,06	16,12	15,90	16,13	15,93	15,96	15,97	15,82	16,01	15,87	16,00
14. futás	15,95	16,00	15,91	15,95	15,89	15,95	16,03	15,98	16,15	15,97	15,84	15,99
15. futás	15,86	15,99	16,15	15,97	15,91	16,00	15,94	15,99	16,02	16,00	15,98	15,96
16. futás	16,00	16,02	16,01	16,00	15,90	16,00	16,04	16,01	16,12	15,99	16,17	15,97
17. futás	16,42	15,99	15,84	15,95	15,99	15,95	16,10	15,97	16,04	15,97	16,00	16,01
18. futás	15,86	15,81	15,74	15,92	15,90	16,05	16,01	15,97	16,16	15,98	15,91	15,99
19. futás	15,91	15,93	16,01	15,99	15,97	15,97	15,82	16,04	16,16	15,96	16,29	15,99
20. futás	15,87	15,95	15,99	15,96	16,07	15,96	15,95	16,00	16,05	16,00	15,79	15,97
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
100. futás	15,76	16,02	15,89	15,84	16,04	16,01	16,20	15,98	16,00	15,97	16,22	15,98

Table D.3. Futási eredmények $n' = 5$ dimenzióra.

10D	C S			C D			O1 S		
N_p	512	6400	6400	512	6400	6400	512	6400	6400
lépésszám	2600	600	6000	2600	600	6000	0,00	600	6000
átlag	510,65	510,44	511,56	513,33	508,98	511,78	511,73	511,61	513,10
σ	8,16	5,76	1,59	7,99	5,02	1,40	4,73	1,41	1,49
$t\sigma^2$	2538,83	817,50	539,46	2598,28	657,02	452,16	1188,44	288,36	3906,38
futásidő [mp]	38,15	24,65	214,68	40,67	26,09	230,40	53,14	144,55	1750,57
hatásfok	1,00	0,32	0,21	1,02	0,26	0,18	0,47	0,11	1,54
1. futás	520,68	514,65	507,83	509,18	504,92	511,38	513,35	511,57	516,13
2. futás	509,51	507,59	509,71	520,57	510,34	511,77	516,15	510,54	513,56
3. futás	524,65	520,31	510,76	514,15	513,13	511,09	514,87	510,02	513,33
4. futás	505,36	515,61	511,48	504,30	518,20	510,52	512,76	511,42	513,35
5. futás	513,90	504,97	510,41	511,93	512,46	510,17	505,82	509,82	*
6. futás	512,19	514,60	512,61	511,78	508,44	509,10	515,42	511,14	515,64
7. futás	505,35	514,86	510,07	511,05	517,97	509,98	496,06	510,60	513,02
8. futás	504,75	501,88	513,41	510,38	507,92	511,41	508,75	511,87	513,58
9. futás	506,79	501,23	507,30	517,13	503,84	514,74	512,31	512,45	514,29
10. futás	509,63	514,23	513,09	513,47	510,26	510,73	507,36	511,06	513,52
11. futás	512,63	511,25	512,53	511,01	500,94	512,09	514,20	512,05	512,77
12. futás	502,53	499,86	514,96	506,25	512,13	511,77	513,75	511,76	513,02
13. futás	521,61	499,54	511,64	514,89	509,45	511,39	509,72	511,46	513,13
14. futás	506,27	510,76	510,51	502,73	507,87	512,29	504,03	514,18	511,59
15. futás	500,17	505,57	511,87	519,11	510,55	510,94	514,07	512,18	*
16. futás	509,01	501,42	512,35	503,15	512,60	512,19	514,80	509,79	511,70
17. futás	507,27	505,73	511,62	537,40	509,92	510,66	522,34	513,31	511,56
18. futás	507,96	504,33	509,34	504,02	516,06	512,89	515,70	512,24	515,10
19. futás	520,11	513,22	513,73	523,73	506,21	514,56	511,67	510,96	511,83
20. futás	513,27	507,41	511,00	507,39	503,99	509,05	505,96	513,28	514,17
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
100. futás	523,97	497,47	509,69	526,26	507,78	513,17	500,34	512,13	513,35

Table D.4. Futási eredmények $n' = 10$ dimenzióra - 1. rész.

10D			O1 D			O2 S		O2 D	
N_p	6400	6400	512	6400	6400	6400	512	6400	512
lépésszám	600	6000	600	600	6000	600	200	600	200
átlag	511,61	513,10	511,45	511,54	512,88	512,37	510,22	512,30	511,30
σ	1,41	1,49	4,61	1,39	1,25	1,58	8,92	1,33	7,46
to^2	288,36	3906,38	1212,69	302,67	2953,35	1548,29	2371,00	1190,84	1791,22
futásidő [mp]	144,55	1750,57	57,13	155,76	1889,09	617,94	29,80	675,08	32,20
hatásfok	0,11	1,54	0,48	0,12	1,16	0,61	0,93	0,47	0,71
1. futás	511,57	516,13	501,59	511,72	511,72	512,74	508,07	511,94	514,72
2. futás	510,54	513,56	504,55	510,39	513,56	512,94	498,12	511,04	500,56
3. futás	510,02	513,33	514,22	510,61	512,44	510,75	499,90	514,33	505,23
4. futás	511,42	513,35	516,81	512,24	513,68	*	521,73	511,87	504,59
5. futás	509,82	*	506,25	511,53	515,09	511,50	517,22	510,96	504,13
6. futás	511,14	515,64	513,35	512,63	512,96	511,53	519,96	511,67	517,73
7. futás	510,60	513,02	510,44	510,06	512,62	522,21	489,03	511,84	504,92
8. futás	511,87	513,58	513,66	509,68	512,31	511,80	513,15	511,24	504,39
9. futás	512,45	514,29	509,91	511,31	511,64	511,53	508,78	513,82	508,95
10. futás	511,06	513,52	515,06	513,53	512,75	512,57	517,31	515,21	508,67
11. futás	512,05	512,77	511,22	511,15	513,46	515,63	515,66	*	512,47
12. futás	511,76	513,02	500,01	510,67	511,47	511,46	530,83	511,41	522,17
13. futás	511,46	513,13	513,24	512,91	515,50	511,67	500,85	*	511,93
14. futás	514,18	511,59	513,07	512,97	*	511,87	510,07	511,01	515,59
15. futás	512,18	*	517,08	511,08	512,99	512,26	487,49	512,20	522,95
16. futás	509,79	511,70	509,65	510,20	512,54	513,95	518,97	*	508,65
17. futás	513,31	511,56	513,46	513,32	513,52	510,80	514,75	512,95	508,33
18. futás	512,24	515,10	509,71	515,26	513,12	512,01	508,16	512,47	520,00
19. futás	510,96	511,83	508,29	511,59	511,47	515,27	511,09	512,33	526,56
20. futás	513,28	514,17	519,31	511,07	512,71	512,33	501,52	513,68	502,52
:	:	:	:	:	:	:	:	:	:
100. futás	512,13	513,35	516,59	511,90	511,58	511,09	516,20	512,50	510,28

Table D.5. Futási eredmények $n' = 10$ dimenzióra - 2. rész.

15D	C S
Szálak száma(N_p)	6240
Lépésszám fázisonként	6000
Eredmények átlaga	15180,03
σ	147,62
$t\sigma^2$	11625971,47
Futásidő [mp]	533,53
1. futás	15293,40
2. futás	15272,62
3. futás	15025,56
4. futás	15136,72
5. futás	15126,95
6. futás	15355,55
7. futás	15259,76
8. futás	15402,56
9. futás	15258,73
10. futás	15115,35
11. futás	14956,16
12. futás	15359,93
13. futás	15110,61
14. futás	15493,48
15. futás	15164,78
16. futás	15012,46
17. futás	15291,61
18. futás	15120,89
19. futás	15444,97
20. futás	14998,03
⋮	
100. futás	15163,31

20D	C S
Szálak száma(N_p)	6240
Lépésszám fázisonként	6000
Eredmények átlaga	500359,65
σ	19085,92
$t\sigma^2$	3,60E+11
Futásidő [mp]	989,26
1. futás	491746,89
2. futás	505558,54
3. futás	495117,97
4. futás	505646,28
5. futás	543504,03
6. futás	468754,76
7. futás	478030,26
8. futás	499981,29
9. futás	510897,53
10. futás	504358,98

Table D.6. Futási eredmények $n' = 15$ és $n' = 20$ dimenziókra.