Párhuzamos algoritmusok nemlineáris peremérték-feladatok megoldására

A Budapesti Műszaki és Gazdaságtudományi Egyetem Irányítástechnika és Informatika Tanszékén a PhD fokozat követelményeinek részleges teljesítéseként benyújtott disszertáció[1].

Szeberényi Imre

Tudományos vezető: Dr. Domokos Gábor
Tanszékvezető: Dr. Arató Péter

Budapest, 2003.

[1] A bírálatok és a védésről készült jegyzőkönyv a későbbiekben a BME Villamosmérnöki és Informatikai Kar Dékáni Hivatalában elérhető.

Summary

This thesis presents new algorithms for the solution of nonlinear boundary value problems associated with ordinary differential equations. Such problems arise most commonly in technical applications, e.g. problems in mechanics and control. The basis of the algorithm is provided by the method suggested by Domokos and Gaspar, introducing the Global Representation Space (GRS), the points of which are in mutually one-to-one correspondence with candidate trajectories for the solution of the boundary value problem. The latter is reduced via the GRS description to a finite dimensional algebraic system the solution of which consists – typically – of one dimensional manifolds, i.e. a set of curves. The Simplex Method decomposes the GRS into simplices and inside them it interpolates linearly the functions of the algebraic system. This algorithm is capable of finding global solution structures of nonlinear problems, however, its application requires massive hardware capacities.

For practical applications, the computations had to be accelerated by several orders of magnitude. This is the main topic of the thesis. The realistic choice was the parallelization of the algorithm. To achieve this goal, the algorithm had to be studied systematically and current (as well as future) hardware and software options had to be considered carefully. The analysis of the algorithm lead to the parallelization with respect to the domains of the GRS. I have chosen an appropriate master-slave model, which can be easily adjusted to the rapidly developing Grid systems.

In the first part of my thesis I briefly introduce the Simplex Algorithm and the most important development tools for distributed parallel systems and Grid systems. As a result of my research the Parallel Simplex Algorithm has been created along with other, more advanced versions such as the Parallel Continuation Algorithm (PCA) and the Parallel Hybrid Algorithm (PHA) introduced in chapter 3, combining continuation and scanning.

The software package which provides a unified framework for the above algorithms has been applied to several mechanics problems, resulting computations in $2 \leq n \leq 9$ dimensional Global Representation Spaces. Some of these examples are discussed in chapter 4. The algorithms have been tested both in homogeneous and heterogeneous environments as well as on Grid systems. The efficiency of the parallelization can be illustrated by the fact that even on 120 processors we have measured linear speedup.

Tartalmi összefoglaló

A dolgozat a mechanikában, és más műszaki tudományban gyakran felbukkanó, közönséges differenciálegyenletekkel leírható nemlineáris peremérték-feladatok megoldására mutat be új algoritmusokat. Az algoritmusok alapját Domokos és Gáspár által javasolt módszer képezi, amely bevezeti a megoldásként szóba jöhető trajektóriák megfelelő pontjait tartalmazó Globális Reprezentációs Teret (GRT). A véges dimenziós GRT segítségével megfogalmazott peremérték-feladat átfogalmazható egy nemlineáris algebrai egyenletrendszerré. Az egyenletrendszer megoldása – tipikusan – egyparaméteres, vagyis egy görbe-halmaz. A Szimplex Algoritmus – nevéhez híven – szimplexekre bontja a GRT-t és a szimplexeken belül lineárisan interpolálja az egyenletrendszert alkotó függvényeket. Ez az algoritmus alkalmas a feladatok globális vizsgálatára, de alkalmazása igen jelentős számítási kapacitást igényel.

Az algoritmus gyakorlati alkalmazása érdekében nagyságrendekkel fel kellett gyorsítani a számításokat. Reális alternatívaként az algoritmus párhuzamosítása kínálkozott. Ez a disszertáció központi témája. A párhuzamosítás érdekében át kellett tekinteni egyrészt magát az algoritmust, másrészt tekintettel kellett lenni a jelenlegi és jövőbeni számítástechnikai ill. technológiai lehetőségekre. Az algoritmus elemzése arra vezetett, hogy a GRT tartományaira vonatkozó párhuzamosítás a legcélszerűbb megoldás. Ehhez illeszkedően mester-szolga modell szerinti párhuzamosítást választottam, ami jól illeszthető a napjainkban gyorsan fejlődő Grid rendszerekhez is.

Dolgozatom első részében röviden bemutatom a Szimplex Algoritmust és az elosztott párhuzamos rendszerek, valamint a Grid rendszerek legfontosabb fejlesztő eszközeit. Munkám eredményeként létrejött a Párhuzamos Szimplex Algoritmus (PSA), valamint annak több továbbfejlesztett változata, például a Párhuzamos Útkövető Algoritmus (PUA) vagy az útkövetést és letapogatást dinamikusan kombináló Párhuzamos Hibrid Algoritmus (PHA), melyet a dolgozat 3. fejezete mutat be részletesen.

Az algoritmusokat egységes keretbe foglaló alkalmazást eddig több különböző mechanikai feladat vizsgálatára használtuk fel, melyek $2 \leq n \leq 9$ dimenziós problémákhoz vezettek. Ezek közül mutat be néhányat röviden a dolgozat 4. fejezete. Az algoritmusokat homogén, illetve heterogén párhuzamos számítási környezetben, valamint Grid környezetben is kipróbáltuk. A párhuzamosítás hatékonyságára jellemző, hogy 120 processzorral is lineáris sebességnövekedést tapasztaltunk.

Tartalomjegyzék

1. Bevezetés **1**

1.1. Párhuzamos számítási rendszerek rövid áttekintése 2

1.1.1. Vektorporcesszoros szuperszámítógépek 3

1.1.2. Masszívan párhuzamos rendszerek (MPP) 3

1.1.3. Elosztott rendszerek 4

1.1.4. Metaszámítógépek és Grid rendszerek 5

1.1.5. Párhuzamos rendszerek fejlődési iránya 6

1.2. Alapelvek, eszközök 7

1.2.1. Párhuzamos számítógép modellje 7

1.2.2. Teljesítménymérés 8

1.2.3. Programozási nyelvek ill. fejlesztő környezetek 9

1.2.4. Futtató ill. ütemező környezetek 13

1.3. Peremérték feladatok általános bemutatása, jelentősége 15

1.4. Globális módszerek lényege, célkitűzése 17

1.5. A szimplex módszer ismertetése 19

2. A szimplex letapogató algoritmus párhuzamos implementációja **24**

2.1. Párhuzamos algoritmus alkalmazásának lehetősége 24

2.2. A PSA környezete és programstruktúrája 36

2.3. A szolgaprogram részletes működése 38

2.3.1. A szolgaprogram felépítése 38

2.3.2. A szolgaprogram algoritmusa 42

3. A szimplex algoritmus általánosításai **45**

3.1. Véletlenszerű szimplex letapogatás (VSA) 48

3.2. A fokozatos sűrítési algoritmus (FSA) 51

3.3. Letapogatással nyert adathalmazok pontosítása útkövetéssel 53

3.3.1. Szimplex útkövetés algoritmusa (SUA) 54

3.3.2. Párhuzamos útkövetés algoritmusa (PUA) 55

3.4. Párhuzamos hibrid algoritmus (PHA) 57

3.5. PSA szakadásos függvényekre 59

3.6. Többparaméteres PF megoldása rekurzív programozással 60
3.7. Teljesítménynövelési módszerek összehasonlítása 62

4. Alkalmazási modell **65**
4.1. A probléma megoldásának folyamata 65
4.2. Számítást segítő keretrendszer, segédprogramok 67
4.2.1. Katalógusstruktúra 67
4.2.2. Matematikai modell 68
4.2.3. Függvénymodul 68
4.2.4. Fordítás, szerkesztés 70
4.2.5. Konfigurációs állomány 71
4.2.6. Futtatás 73
4.2.7. Eredmények 73
4.2.8. Eredmények pontosítása, fokozatos sűrítés 74
4.2.9. Eredmények előfeldolgozása, szűrés 76
4.3. Az eredmények interaktív megjelenítése 76
4.3.1. Gnuplot alkalmazása 77
4.3.2. Gloves alkalmazása 77
4.3.3. IBM DX rendszer alkalmazása 78
4.4. Kutatói együttműködés támogatása 81

5. A programrendszer gyakorlati alkalmazásai **82**
5.1. Gátolt érintkezési feladatok 82
5.2. Folyadékhidak egyensúlya 86
5.3. Mikroszálak konfigurációi 89
5.4. DNS molekulák mechanikai modellje 89
5.5. Növényi indák mechanikai modellezése 91

Tézisek összefoglalása, további kutatási célok **94**

Köszönetnyilvánítás **96**

Irodalomjegyzék **97**

Függelék **101**

A. Programrészletek **101**
A.1. canti.c függvénymodul 101
A.2. canti (starup) shell script 105
A.3. dxconv shell script 110
A.4. Makefile.aimk (architektúra független make) 113

1. fejezet

Bevezetés

A számítógépek kapacitása, ezen belül is az elosztott párhuzamos rendszerek kapacitásának rohamos növekedése lehetővé teszi, hogy az alkalmazott tudományok terén olyan számítási módszereket vezessünk be, melyek nem a hagyományos módszerek számítógéppel történő megvalósításai (általánosításai), hanem merőben új gondolatokra támaszkodva jöttek létre. Jelen munka egy ilyen módszert, és annak párhuzamos implementációját mutatja be. A dolgozatban bemutatott ún. *Szimplex Módszer* a nemlineáris mechanika területén kifejlesztett eljárás, melynek – ellentétben más gépi módszerekkel – nincs „kézi" változata, viszont olyan kérdések megválaszolására alkalmas, melyek kívül esnek más módszerek lehetőségein. Az algoritmus számítási igénye nagy, s a vizsgált feladat nagyságával (dimenziószámával) exponenciálisan nő, ugyanakkor jól párhuzamosítható.

A számítási kapacitás növelésének egyik lehetséges útja a párhuzamos feldolgozás, mely módszer azonban nem alkalmazható (hatékonyan) minden feladatra. Könnyen belátható például, hogy egy mátrix-szorzás egyes műveletei párhuzamosan, egymás eredményétől függetlenül is elvégezhetők. Más a helyzet egy rekurzívan definiált sorozat tagjainak kiszámításával; a sorozat egyes elemeit csak egymás után, az előző eredmény figyelembe vételével lehet meghatározni, ezért párhuzamos feldolgozással a megoldás menete nem gyorsítható[1]. A dolgozatban bemutatott algoritmusok igen jó hatásfokkal párhuzamosíthatók, amit a megvalósított program használata során szerzett tapasztalataink is alátámasztottak.

Az itt bemutatott munka többéves kutató és fejlesztő munka eredménye. A program legelső változata 1994-ben készült el, melyet azóta folyamatosan használunk és fejlesztünk. Ez a korai változat azóta újabb algoritmusokkal és segédprogramokkal bővült, melyek lehetővé tették számos, eltérő jellegű mechanikai probléma vizsgálatát.

A dolgozat első fejezete röviden rendszerezi a párhuzamos számítógépeket, bemutatja azokat az alapelveket, melyek működésük megértéséhez, programozásukhoz szükségesek. Ezt követően bemutatja a mechanikai hátteret és magát a módszert (1.5. alfejezet), amelyhez ezt megelőzően nem készült párhuzamos implementáció. A 2. és a 3. fejezet mutatja be önálló eredményeimet, melyeket a fejezetek elején tézisként emeltem ki. Ezek az algoritmus párhuzamosításához, a program-rendszer megvalósításához és az 1.5. fejezetben ismertetett algoritmus általánosításához kapcsolódnak. A 4. fejezet rövid áttekintést nyújt a megvalósított rendszer használatáról, felhasználói felületéről és a kutatási munkát segítő segédprogramokról. Egy konkrét példán keresztül bemutatja, hogy milyen feladatokat kell elvégeznie egy, a programot használni akaró, nem informatikus képzettségű kutatónak. Az 5. fejezet néhány alkalmazási példát villant fel. Ezek bizonyítják a módszer és az imp-

[1] A sorozat egy újabb elemének számítása közben lehetnek ugyan párhuzamosan elvégezhető lépések is, melyeknek párhuzamosításától várható sebességnövekedés, de ez feltehetően elenyésző.

lementáció jelentőségét a különböző peremérték feladatként megfogalmazható problémák globális vizsgálatában.

1.1. Párhuzamos számítási rendszerek rövid áttekintése

A párhuzamos feldolgozás, párhuzamos gépek és programok készítésének igénye lényegében a kezdetektől foglalkoztatja a számítástechnikai fejlesztőket. Egy anekdota szerint még az elektroncsövek időszakában Daniel Slotnick – aki 1967-ben megépítette az első szuperszámítógépet, az ILLIAC IV-et – párhuzamos gép építésének ötletével fordult Neumann Jánoshoz, aki kis gondolkodás után azt válaszolta, hogy az ötlet megvalósítása túl sok elektroncsőbe kerülne [2]. Ezzel Neumann azokra a nehézségekre utalt, ami az akkori technológiát jellemezte, bár a párhuzamosítás ötlete – mint tudjuk – őt is foglalkoztatta. Azt, hogy az akkori elektroncsöves technológia valóban alkalmatlan volt a párhuzamos gép megvalósítására, a történelem igazolta, hiszen a párhuzamos gépek tömeges megjelenéséig még közel 40 évet kellett várni.

A tranzisztorok, majd az integrált áramkörök megjelenésével a korai gépeknél lényegesen nagyobb teljesítményű és megbízhatóbb gépek készítésére nyílt lehetőség. A számítógépek felhasználási területének szélesedése azonban tovább fokozta a teljesítmény növelésének igényét. A tudományos számítások, különféle mérnöki alkalmazások, szimulációs és modellezési feladatok és az általános célú számítástechnikai alkalmazások egyre nagyobb és nagyobb számítási teljesítményt igényeltek.

Az eszközök és a technológia fejlesztése mellett egyes kutatók a párhuzamosításban látták a számítási teljesítmény növelésének egyik reális útját. Így már a 60-as évek elején több olyan kutatási projekt indult, mely valamilyen módon a párhuzamos számítási rendszerekhez kapcsolódott. Ezek elsősorban a szorosan csatolt rendszerek kifejlesztésére irányultak. A fejlesztések eredményeként kezdetben néhányszor tíz, később több száz, esetenként több ezer processzort is tartalmazó szuperszámítógépeket terveztek, igen magas tervezési és előállítási költségek mellett. A költségek vonatkozásában a technológia alapvetően meghatározó volt, mivel kiemelkedő teljesítmények eléréséhez a mindenkori csúcstechnológiát kellett alkalmazni. A technológiai fejlődés korai szakaszára pedig különösen érvényes az ún. négyzetes törvény, mely szerint a 10-szeres számítási teljesítménynövekedés 100-szoros költségnövekedéssel érhető el. Ez szerencsére ma már nincs így.

A 80-as évek végére úgy tűnt, hogy a processzorok, memóriák és azok összekapcsolása körüli technológiai nehézségek megszűntek [2]. Ennek eredményeként a csúcstechnológia és a tömeggyártásban alkalmazott technológia távolsága csökkent, és egyre több gyártó jelent meg párhuzamos architektúrájú számítógéppel a piacon. Kezdetben a szuperszámítógépek nagy része ún. vektorprocesszorokat tartalmazott, de a 90-es évek elejére ezeket kiszorították a párhuzamos skalár ill. szuperskalár processzorok. Ugyanakkor az integráció növekedésével a legtöbb processzorban megjelentek azok a technikák, melyek korábban a párhuzamos rendszereket jellemezték, mint pl. az utasítás szintű párhuzamosítás, ill. a pipeline technika.

A párhuzamos architektúrákat többféle szempont[2] szerint (technológia, funkcionalitás stb.) osztályozhatjuk. Felépítési és elsősorban történelmi szempontokból a következő jellemző szuperszámítógép típusokat ill. kiemelkedő teljesítményű számítási rendszereket különböztethetünk meg:

- Vektorprocesszoros szuperszámítógépek

[2] Az egyik leggyakrabban alkalmazott osztályozás az ún. Flynn-féle osztályozás (7. oldal), melynek módosított változata a Hwang-Bring-féle felosztás. Egy másik osztályozás, a Hadlers-féle rendszer számhármasokkal ill. számhatosokkal próbálja leírni az architektúra jellemzőit. Sima [45] felosztása ugyanakkor kiterjed a párhuzamosítás szintjeire is.

- Masszívan párhuzamos rendszerek (Massively Parallel Processors)
  A processzorok és a memória kapcsolatát tekintve a következő kialakítások lehetségesek:
  - Üzenetküldéses elosztott memóriás rendszerek (Message-passing Distributed Memory)
  - Szimmetrikus multiprocesszoros rendszerek (Symmetric Multiprocessors)
  - Elosztott közös memóriás rendszerek (Distributed Shared Memory)
- Elosztott számítási rendszerek (Distributed Computing)
  Az együttműködő berendezések egységessége alapján a következő csoportosítás lehetséges:
  - Homogén rendszerek
  - Heterogén rendszerek
- Metaszámítógépek és Grid rendszerek (Metacomputers)

1.1.1. Vektorporcesszoros szuperszámítógépek

Olyan speciális hardvert tartalmazó számítógépek, melyek egy (esetleg több) dimenziós tömbök adatain gyorsan képesek azonos műveletet végezni (pl. két 16 elemű vektort egyetlen utasítással összeadni). Rendszerint ez azt jelenti, hogy az aritmetikai egység a vektor méretének megfelelően többszörözött áramkörökből áll, melyek a műveletet párhuzamosan hajtják végre. Másik szokásos megvalósítás a pipeline elven működő műveletvégző egység. Újabban a vektorprocesszoros és parallel gépek megkülönböztetésére az *uni-processor* és *multi-processor* elnevezéseket használják. A legtöbb numerikus módszer jól illeszkedik ehhez az architektúrához, azonban a nagy számú többszörözött áramkör miatt a kihasználtság rossz. Az ilyen rendszerek nem vagy nem elég finoman skálázhatók[3].

A 90-es években több olyan gyártó volt a piacon, aki vektorprocesszoros gépet kínált. Mára ezek száma jelentősen csökkent. A gyártók inkább masszívan párhuzamos architektúrát, vagy valamilyen hibrid megoldást követnek.

1.1.2. Masszívan párhuzamos rendszerek (MPP)

A masszívan párhuzamos rendszerek speciálisan épített, nagy teljesítményű számítógépek több száz vagy akár több ezer processzorral, melyek gyors kommunikációs csatornákon vagy közös memórián keresztül kommunikálnak egymással. Jellemzőjük a nagy teljesítmény, nagy memória és nagy tárkapacitás. Rendszerint szimulációs ill. modellezési feladatokhoz alkalmaznak ilyen gépeket. Meglehetősen drágák, és üzemeltetésük is költséges.

Üzenetküldéses elosztott memória (MDM)

A hagyományos értelemben vett MPP-k processzorai saját memóriával rendelkeznek. Az adatok ill. a szinkronizációs események üzenetek formájában jutnak el az egyik processzortól a másikhoz. Az üzenetek továbbítására nagy sebességű kommunikációs csatornát ill. csatornákat alkalmaznak. A kommunikációs hálózat kialakításától, fizikai paramétereitől és topológiájától (pl. háló, gyűrű, kocka, hiperkocka, crossbar stb.) függően több száz vagy akár ezer processzorig bővíthető architektúra. Általában jól skálázhatók. Programozásuk speciális fejlesztőeszközöket ill. könyvtárakat igényel.

[3] A teljesítőképesség szabályozhatóságát nevezzük skálázhatóságnak, azaz azt a lehetőséget, hogy a rendszer erőforrásai a feladat nagyságához illeszkedően bővíthetőek-e vagy átcsoportosíthatóak-e.

Szimmetrikus multiprocesszorok (SMP)

Az előzőekben bemutatott elosztott memóriás rendszerek processzorközi kommunikációja korlátozott sávszélességű és jelentős késleltetéseket is tartalmazhat. Ezt a hátrányt küszöböli ki ez az architektúra azzal, hogy a processzorok egy közös memóriához kapcsolódnak. Így a processzorközi kommunikáció ezen keresztül, közel a memória írás-olvasás sebességével zajlik. A közös elérésű memória azonban nagy számú processzor esetén szűk keresztmetszetté válhat. Így ezekre a rendszerekre a kisebb processzorszám és a korlátozottabb skálázhatóság a jellemző. A közös elérésű memória miatt a programozásuk nem igényel különleges eszközöket.

A korábbi rendszereknél kialakult processzorszám 16 körüli volt, de az újabb technológiák felhasználásával 64 vagy ettől nagyobb SMP rendszerek is épülnek. A mai SMP rendszerekre jellemző, hogy a processzoroknak saját, többszintű cache rendszerük és ezen kívül még önálló memóriájuk is van. Ezért a közös elérésű memória gyakran elosztottan jelenik meg a rendszerben. Az adatok elérését pedig speciális, a koherenciát is biztosító hardver támogatja.

Elosztott közös memória (DSM)

A fenti két architektúra kombinációjaként a fejlesztők egy olyan hibrid megoldást dolgoztak ki, amely ötvözi az MDM és az SMP rendszerek előnyeit. A processzorokat kisebb egységekbe, ún. node-okba kapcsolják. Ezek közös memórián keresztül kommunikálhatnak egymással. A node-okat pedig nagysebességű kommunikációs csatornán keresztül kötik össze. Így a különböző node-ok processzorai ezen keresztül, üzenetküldéssel érhetik el a másik node közös memóriáját.

1.1.3. Elosztott rendszerek

Az elosztott számítási rendszerek általános célú számítógépek nagysebességű hálózaton történő összekapcsolásával jönnek létre. A gépek önálló operációs rendszerrel, memóriával, háttértárral rendelkeznek, amelyek egy nagyobb feladat megoldásán dolgoznak együtt. Jellemzőjük, hogy jól skálázhatók, és lényegesen olcsóbbak, mint a hagyományos szuperszámítógépek, mivel általános célú berendezésekből épülnek fel. Ennek ellenére megfelelő hardver és szoftver támogatással hibatűrő rendszerek is kialakíthatók segítségükkel [11]. Kétségtelen hátrányuk, hogy a gépek közötti kommunikációs csatorna sávszélessége viszonylag alacsony, késletetése pedig nagy. Ez alapvetően meghatározza felhasználási területüket is. A gépek közötti információcsere üzenetküldéssel történik.

Az ilyen rendszerek különösen jól alkalmazhatók azokban az esetekben, amikor ugyanazt az alkalmazást kell többször, más-más adathalmazzal futtatni, és az egyes futtatások között nincs adat- ill. sorrendiséget előíró kapcsolat. Így az egyes futtatások egymás után vagy akár egyszerre is elindíthatók annyi példányban, ahány szabad gépünk van. Természetesen alkalmazási szinten ez nem jelent valódi párhuzamos programot. Valódi párhuzamos alkalmazásnak azokat nevezzük, amikor a részfeladatokat megvalósító programrészek vagy programok egy közös feladaton dolgoznak párhuzamosan együtt. A teljes feladat megvalósítása a párhuzamos programrészek együttműködését és különböző mértékű kommunikációját igényli.

Valódi párhuzamos programokhoz is hatékonyan alkalmazhatók a lazán csatolt elosztott rendszerek az ún. *master-worker* [23] számítási modell felhasználásával. Az ilyen alkalmazások programjai egy közös feladat megoldásán dolgoznak együtt, tehát nem függetlenek, de egymással nem kommunikálnak, csak a mesterprogrammal, amely kiosztja a feladatokat, és begyűjti az eredményeket. Természetesen nem minden feladat osztható ilyen módon részfeladatokra, de számos területen (pl.

képfeldolgozás, egyes szimulációk, a dolgozatban bemutatott PSA, stb.) hatékonyan alkalmazható ez a módszer.

Homogén elosztott rendszerek

Az azonos hardver-szoftver jellemzőkkel rendelkező gépekből felépített rendszereket homogén farmoknak vagy homogén klaszterekneknek nevezzük. A gépek rendszerint közös állományrendszerhez kapcsolódnak, így az üzenetküldési mechanizmuson felül ezen keresztül is képesek adatot cserélni egymással. Ebben az esetben az egyes gépeken végrehajtandó programokat nem kell a végrehajtás előtt minden gépre külön eljuttatni. Fő jellemzőjük a jó skálázhatóság és az alacsony beruházási költségek.

Heterogén elosztott rendszerek

A hagyományos szuperszámítógépek azonos teljesítményű és típusú processzorokból épülnek fel. Az elosztott rendszerekben a számítógépek gyakran nemcsak sebességükben, erőforrásaikban különböznek, hanem architektúrájukban is. Az elosztott heterogén rendszerekben ezért számos olyan problémával kell számolni, amelyek homogén architektúra esetén nem vetődnek fel:

- eltérő utasításrendszer,
- eltérő adatábrázolási forma,
- eltérő teljesítmény,
- eltérő terheltség,

Ezzel szemben viszont számos előnyt is jelent az ilyen elosztott rendszerek használata:

- Könnyen és gazdaságosan skálázható rendszer alakítható ki.
- Egyszerű és költségkímélő módon felújítható a rendszer a legújabb számítási eszközökkel. A felújítás akár folyamatos is lehet, hiszen nem kell egyszerre minden berendezést lecserélni.
- Könnyen és főként kis költségekkel hibatűrő [4] alkalmazások is kialakíthatók.
- A rendszer létrehozásánál a különböző architektúrájú meglevő berendezések is felhasználhatók.

1.1.4. Metaszámítógépek és Grid rendszerek

Az egymástól lényegesen különböző szuperszámítógépek ill. számítógép klaszterek nagy sebességű távolsági[5] hálózattal történő összekötésével létrehozott számítási rendszert nevezzük *metaszámítógépnek*. Az ilyen rendszer elemei egymástól akár több ezer km-re is lehetnek egymástól, ugyanakkor úgy használhatók, mintha egyetlen számítógép alkotórészei lennének.

A metaszámítási rendszerek általánosításaként jött létre az ún. *Grid* [24] rendszer fogalma, melyben már tetszőleges számítógépek és erőforrások vannak egyetlen nagy rendszerré összekapcsolva. Ez

[4] Olyan alkalmazások, amelyek bizonyos számú hardver meghibásodás mellett is képesek folytatni a feladatot anélkül, hogy beavatkozást igényelnének.

[5] Az alapgondolat lokális hálózatokon született meg, de a hálózati technológiák fejlődésének köszönhetően szinte minden esetben távolsági hálózatról beszélünk.

a rendszer az elektromos hálózathoz hasonló regionális, nemzeti vagy nemzetközi méretű univerzális számítási, tárolási és feldolgozási teljesítményt képes nyújtani a hálózatba bekapcsolódó felhasználók számára. A *grid* (hálózat) elnevezés éppen az elektromos hálózattal való analógiát kívánja hangsúlyozni. Ezek a rendszerek valójában még csak kísérleti ill. kutatási fázisban vannak. Kiterjedésüktől egyesek azt várják, hogy az információ feldolgozása terén olyan szerepet fognak betölteni, amilyen szerepet a Web játszik az információ elérése terén.

A kutatások legfőbb célja azoknak az eszközöknek és protokolloknak a kidolgozása, amelyek nagy kiterjedésű, igen heterogén, folyamatosan változó elosztott rendszerek biztonságos, ugyanakkor hatékony kezelését teszik lehetővé. A megoldások tekintetében jelenleg több irányzat van kialakulóban, de ezek is folyamatosan változnak. A különböző irányvonalak egységes törekvése a szabványos alkalmazási és felhasználói rétegek kialakítása.

1.1.5. Párhuzamos rendszerek fejlődési iránya

A 80-as évek végén, 90-es évek elején a párhuzamos rendszerek, ezen belül is a masszívan párhuzamos rendszerek jelentős áttörését prognosztizálták. Sokan hatalmas, sok száz vagy ezer processzort is meghaladó szuperszámítógépek elterjedésére számítottak. Mindamellett, hogy jelenleg is épülnek ilyen gépek, jelentőségük, elterjedtségük kisebb a vártnál. Ennek egyik oka az, hogy a párhuzamos architektúrájú gépek fejlesztésével egyidőben a processzoron belüli feldolgozás párhuzamosítása (RISC technológia) terén is jelentős előrelépés történt, így az általános célú processzorok teljesítménye is lényegesen megnövekedett. A másik abban keresendő, hogy a gyártási technológiák fejlődése lehetővé tette a jó minőségű, megbízható alkatrészek tömeges és olcsó előállítását. Így a tömegméretekben előállított általános célú számítógépek és a speciálisan tervezett, kis darabszámban előállított hagyományos szuperszámítógépek ára még jobban eltávolodott egymástól.

Ugyanakkor a párhuzamosítás, mint a sebességnövelés egyik eszköze, egyre szélesebb körben jelenik meg a számítástechnika alkalmazásának minden területén. Itt nem csupán a processzoron belüli párhuzamos végrehajtásra kell gondolni, hanem az elosztott, párhuzamos feldolgozást megvalósító adatbázis-szerverekre, kommunikációs berendezésekre is.

A számítási teljesítmény fokozásának igénye a számítógépek megjelenése óta folyamatos. Ezt az igényt részben a számítástechnika alkalmazási területének kiszélesedése, részben az új, pontosabb számítási módszerek alkalmazásának lehetősége erősíti. Az üzleti szférában az adatbázis jellegű alkalmazások, míg a tudományos és kutatási területeken, valamint a mérnöki munkában a szimulációs és numerikus feladatok igényelnek mind gyorsabb és gyorsabb számítógépeket. A megnövekedett teljesítmény egyre több olyan számítási módszer, algoritmus alkalmazását teszi lehetővé, amelyek a korábbi gépeken nem voltak tesztelhetőek, illetve reális méretű feladatra nem voltak alkalmazhatóak.

Az esetek jelentős részében párhuzamos algoritmusokkal, párhuzamos feldolgozással, alkalmazás- és utasítás szintű párhuzamosítással a számítógépek teljesítménye jelentősen növelhető. Ezért az utóbbi években egyre több párhuzamos elven működő alkalmazás készül, mind a nagy számítási igényű tudományos felhasználás, mind az általános jellegű alkalmazások terén. Ez a növekedés várhatóan tovább fog tartani az elkövetkező időben.

Mára a párhuzamos rendszerekkel kapcsolatos kutatások és fejlesztések az elosztott számítási rendszerek irányába tolódtak el az általános célú számítógépek növekvő teljesítménye és csökkenő ára, valamint a hálózati technológiák gyors fejlődése miatt. A hálózati eszközök és technológiák fejlődésével az általános célú munkaállomásokból, személyi számítógépekből nagy méretű, akár több országot is magába foglaló, igen teljesítőképes elosztott rendszerek hozhatók létre, melyek segítsé-

		DATA	
		SINGLE	**MULTIPLE**
INSTRUCTIONS	SINGLE	**S**ingle **I**nstruction **S**ingle **D**ata **SISD** (serial machines)	**S**ingle **I**nstruction **M**ultiple **D**ata **SIMD** (vector proc.)
	MULTIPLE	**M**ultiple **I**nstruction **S**ingle **D**ata **MISD** (pipeline)	**M**ultiple **I**nstruction **M**ultiple **D**ata **MIMD** (multiproc)

1.1. táblázat. A párhuzamos architektúrák Flynn-féle csoportosítása

gével egyes alkalmazások a szuperszámítógépek teljesítményének megfelelő vagy azt meghaladó teljesítménnyel futtathatók. Véleményem szerint a hálózati technológiák várható gyors fejlődése miatt az elosztott rendszerek egyes felhasználási területeken rövidesen ki fogják szorítani a hagyományos szuperszámítógépeket.

1.2. Alapelvek, eszközök

Az 1.1. pontban a párhuzamos számítási rendszereket felépítésük alapján osztályoztuk. Ebben az alfejezetben a működésüket leíró modelleket, a modellekhez kapcsolódó fontosabb programozási eszközöket mutatjuk be.

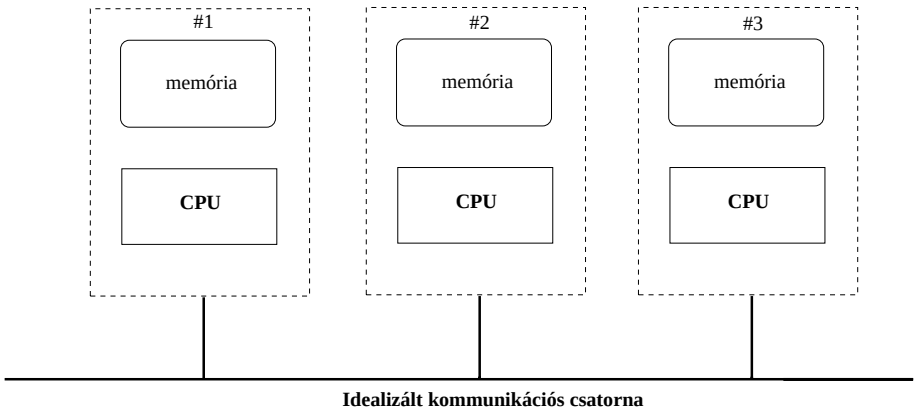
1.2.1. Párhuzamos számítógép modellje

A Neumann-modell a hagyományos számítógépek architektúrális és működési modelljét egyaránt meghatározza. Ez az egyszerű modell, amely szerint a központi egység a hozzá kapcsolódó memóriában tárolt programot hajtja végre, lehetővé tette, hogy a programok és algoritmusok készítése független legyen a számítógép tényleges architektúrájától.

A párhuzamos számítási környezetek közel fél évszázados fejlődése során számos modellt dolgoztak ki a párhuzamos rendszerek működésének leírására. Ezek közül a két legjelentősebb a Flynn-féle [22] felosztás és az ún. *multicomputer* modell [23].

Flynn az adatok és a rajtuk végrehajtott műveletek száma szerint a tényleges architektúrától elvonatkoztatva 4 csoportba sorolta a számítógépeket (1.1. táblázat). Ezen felosztás egyszerűsége ellenére igen jelentős, nemcsak azért, mert az elmúlt évtizedek párhuzamos architektúrái a fenti kategóriák valamelyikébe estek, hanem azért is, mivel alapvető kapcsolatot teremt a Neumann-architektúrával, hiszen a párhuzamos architektúrákat annak kiterjesztésének tekinti.

Szintén a Neumann-modell általánosításaként jön létre a multicomputer modell. Ez nem más, mint számos olyan Neumann-gép, amelyek egy idealizált kommunikációs csatorna felhasználásával vannak összekapcsolva (1.1. ábra). Minden gép saját programot hajt végre, miközben a saját memóriájában levő adatokon műveleteket végez. A gépek egymásnak üzeneteket küldenek, melyek



1.1. ábra. Az idealizált párhuzamos számítógép

segítségével koordinációs feladatokat és adatátviteli feladatokat is el tudnak látni, azaz egy üzenet vonatkozhat egy távoli memória írására vagy olvasására is. Az idealizált kommunikációs csatorna adatátviteli képessége független a csatorna forgalmától és a gépek közötti távolságtól, csupán az üzenet hosszától függ. A helyi memória elérése azonban gyorsabb, mint a távoli memóriáé.

Programozási modell

Mind a masszívan párhuzamos (1.1.2.), mind az elosztott rendszerek (1.1.3.) esetén a párhuzamosan futó programrészek között valamilyen adatcserére és megfelelő szinkronizációs eszközök használatára van szükség. A több évtizedes fejlődés során a párhuzamos programrészek közötti adatcserére programozási szemszögből alapvetően két kommunikációs módszer ill. séma alakult ki:

- közös memóriás módszer (shared memory)
- üzenet-átadásos módszer (message passing)

Magától értetődik, hogy a közös memóriás módszert a szorosan csatolt, masszívan párhuzamos rendszerekben fejlesztették ki, míg az üzenet-átadás elve az elosztott memóriás rendszerekben született meg. Működési elvként ill. modellként tekintve azonban egyik elv sem kötődik szorosan az adott architektúrához. Így találkozhatunk lazán csatolt rendszerekben is közös memória elvre épülő modellel és fordítva[6]. Az üzenet-átadás elvének szélesebb körű implementálhatósága miatt ma a legelterjedtebb módszernek az üzenet-átadásos módszert tekinthetjük, ami lazán csatolt rendszerek esetén is hatékonyan megvalósítható.

1.2.2. Teljesítménymérés

A párhuzamos programozással elért teljesítménynövekedés méréséhez számos fogalmat és mérőszámot alkalmaznak, melyek közül számunkra a legfontosabbak a sebességnövekedés, hatékonyság, és redundancia. Ezek definíciója az alábbi:

[6]Nem a megvalósítást, hanem a programozási modellt vizsgáljuk.

$S_N = \frac{T_S}{T_N}$ - N processzorral elért sebességnövekedés (Speed Up)
$E_N = \frac{S_N}{N}$ - N processzorral elért hatékonyság (Efficiency)
$r = \frac{C_P}{C_S}$ - párhuzamos program redundanciája (Redundancy)

ahol:
N - processzorok száma
T_S - futási idő soros végrehajtás esetén
T_N - futási idő N processzor esetén
C_P - párhuzamos program műveleteinek száma
C_S - soros program műveleteinek száma

Ideális esetben a sebességnövekedés a processzorszámmal lenne egyenlő, de ez természetesen még elméletileg sem érhető el. A ténylegesen elérhető sebességnövekedés számtalan paramétertől függ, melyek közül a legfontosabbak a következők:

- a feladat párhuzamosítható és nem párhuzamosítható részének aránya,
- a párhuzamos programrészek kommunikációjából adódó többletműveletek száma,
- a kommunikációs csatorna sávszélessége,
- a kommunikációs csatorna késleltetése,
- a kommunikációs modell milyensége.

Fontos és érdekes összefüggésre mutat rá az ún. Amdahl szabály [3]. Amdahl egy igen durva becslést ad az elérhető sebességnövekedés felső határára:

Amdahl féle felső határ: $S_N^A = \frac{1}{s+\frac{1-s}{N}} < \frac{1}{s}$

ahol:
S_N^A - N processzorral elérhető sebességnövekedés felső határa
s - a nem párhuzamosítható rész aránya ($0 \leq s \leq 1$)
N - processzorok száma

Ez abból az egyszerű tényből vezethető le, hogy egyetlen program sem párhuzamosítható teljes egészében (pl. az input/output tipikusan soros feladat). Ha eltekintünk minden más hatékonyságot rontó tényezőtől, és csak a feladat párhuzamosíthatóságának mértékét vizsgáljuk, akkor a fenti összefüggés egyszerűen levezethető. Nagy N esetén a nevező második tagja elhagyható, így juthatunk a $S_N^A < \frac{1}{s}$ felső becsléshez, ami minden kommunikációs és egyéb hatást figyelmen kívül hagy, de igen fontos elméleti határ.

A kommunikációs csatorna paramétereinek és a kommunikációs módszer tulajdonságainak ismeretében további, pontosabb becslésekhez juthatunk, melyek jól alkalmazhatók a konkrét feladat és környezet ismeretében. Példaként tételezzünk fel egy olyan master-worker alkalmazást, amelyben minden worker azonos ideig kommunikál a mesterrel, amely csak sorban tudja kiszolgálni a kommunikációs kéréseket. Könnyen belátható, hogy a processzorszám növelésével a mesterprogram fogadóképessége korlátozó tényezőként fog jelentkezni, amiből a konkrét kommunikációs modellre meghatározható az elérhető sebességnövekedés felső határa.

1.2.3. Programozási nyelvek ill. fejlesztő környezetek

A párhuzamos programok tervezéséhez, fejlesztéséhez számos nyelv, programozási és futtató környezet készült. Ezek közül a legelterjedtebben alkalmazottak az ún. imperatív nyelvek párhuzamos

kiegészítései, valamint a hozzájuk kapcsolódó futtató környezetek ill. ütemezők. A következőkben bemutatok néhány ilyen nyelvi kiegészítést és eszközt. Ezek egy része elosztott rendszerekhez kifejlesztett eszköz, de természetesen masszívan párhuzamos környezetben is alkalmazhatók.

Linda

A Linda [30, 8, 9] egy olyan párhuzamos programozási modell, amely a közös memória elvére épül. A processzek közötti kommunikáció egy olyan közös területen zajlik, amely minden processz számára egyformán elérhető. Ez az ún. *Tuple Space*, melyben típussal összerendelt adatok, ill. adathalmazok helyezkednek el. A processzek adatokat helyezhetnek, ill. adatokat olvashatnak be a közös területről. Ehhez összesen 6 igen egyszerű utasítás áll a rendelkezésükre:

out kimásol egy adathalmazt a processz területéről a közös területre

in, inp egy illeszkedő adathalmazt behoz a közös területről úgy, hogy az ott megszűnik. (Az **inp** nem blokkolódik, ha az adat nem létezik.)

rd, rdp egy illeszkedő adathalmazt bemásol a közös területről. (Az **rdp** nem blokkolódik, ha az adat nem létezik.)

eval az **out** művelethez hasonló, de a paraméterként megadott függvényt külön processzként hajtja végre, majd az eredményt a közös területre helyezi. (Ez az egyetlen módja, hogy új processzt hozzunk létre.)

Ezt az egyszerű és szellemes modellt a Yale Egyetemen dolgozták ki 1989-ben. A modell előnye az üzenet-átadásos módszerrel szemben, hogy a processzek közötti kommunikációhoz a processzeknek nem kell egymás azonosítóit ismerni, így azok jobban elrejthetők egymás elől. Ez lényegesen leegyszerűsítheti egy hibatűrő rendszer kialakítását, amelyben a processzeket hiba esetén újra kell indítani, vagy esetleg át kell mozgatni egy másik gépre.

Az alkalmazás oldali egyszerűség azonban számos megvalósítási kérdést vet fel a kommunikációs területet, a Tuple Space-t kezelő programmodul szempontjából, ami elsősorban a hatékonyság miatt kritikus. Valódi közös memória esetén a megvalósítás egyszerű, de lazán csatolt rendszerek esetén többnyire egy központi szerver és természetesen üzenet-átadásos technika alkalmazását igényli.

A Yale Egyetemen kifejlesztett és megvalósított rendszer mellett érdemes megemlíteni a New York Egyetemen készült PLinda (Persistent Linda) [7] változatot, amely az eredeti koncepció hibatűrő tulajdonságokkal történt kiegészítése. Az ötlet és a megvalósítás az egyre növekvő méretű lazán csatolt rendszerek szempontjából igen figyelemre méltó. A kiegészítés legfontosabb eleme a *xstart* és *xcommit* primitív, amellyel a processzek által kezdeményezett Tuple Space műveletek tranzakcióvá szervezhetők. A rendszer automatikusan biztosítja az adatbázisoknál megszokott rollback funkciót, azaz az újraindított rendszer ill. processz automatikusan egy, a hibát megelőző konzisztens állapotba kerül, ahonnan folytatható a feldolgozás.

Express

Az Express [40] programcsomag az elosztott memória elvére épül. Megvalósítását tekintve egy programkönyvtár, amely kb. 160 C-ből vagy Fortranból hívható rutint tartalmaz. Ezek az üzenet-átadásos modellt felhasználva különböző feladatokat látnak el a rendszerben. Így pl:

- programindítás, leállítás,

- logikai kommunikációs topológia kialakítása,
- programok közötti kommunikáció,
- programok közötti szinkronizáció,
- file műveletek,
- grafikai műveletek,
- teljesítmény analízis, monitorozás,
- debug funkciók.

A rendszer az 1.1.3. alpontban már bemutatott master-worker és az ún. Cubix programozási modellt támogatja. Ez utóbbi lényegében a Flynn SPMD modelljének felel meg, azaz minden processzor ugyanazt a programot hajtja végre azzal a módosítással, hogy a processzorok közötti kommunikációs topológia speciális.

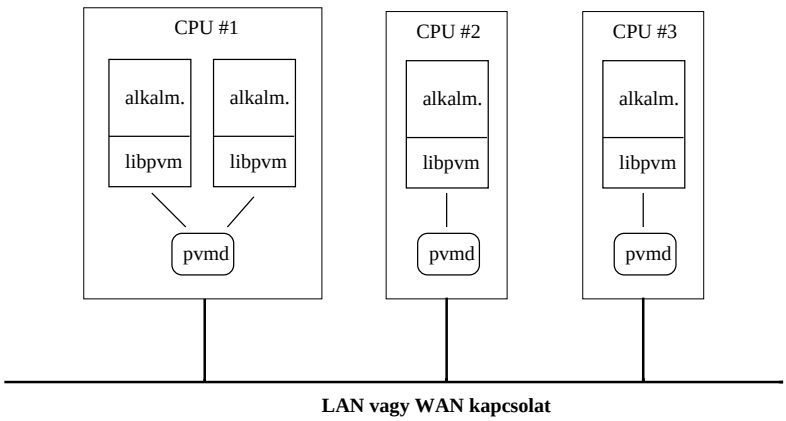
Az Express rendszer speciális fejlesztőeszközöket tartalmaz a programok párhuzamosításához. A fejlesztők alapfilozófiája ugyanis az, hogy a soros programverzió elkészítésével kell kezdeni egy alkalmazás kifejlesztését, majd azok megfelelő részeit kell párhuzamosítani. Az említett eszközök ezt a fejlesztési ciklust támogatják megfelelő grafikus felülettel ellátva.

PVM

A PVM (Parallel Virtual Machine) egy olyan üzenet-átadásos elven működő párhuzamos programozást lehetővé tevő rendszer, amellyel a hálózatba kapcsolt számítógépek egyetlen virtuális gépként kezelhetők. Fejlesztése 1989-ben kezdődött az Oak Ridge National Laboratory-ban [29]. Mára a párhuzamos programok egyik fontos programozási eszközévé vált. Elterjedését nagyban elősegítette, hogy forrás kódja publikus és gyakorlatilag minden UNIX változaton lefordítható. A legújabb változat NT rendszereken is futtatható. Emellett több multiprocesszoros rendszer gyártója biztosítja a saját gépére optimalizált PVM változatot is. Fontos jellemzője, hogy támogatja a heterogén környezetet, és lehetőséget ad hibatűrő alkalmazások kialakítására is. Felépítését tekintve két fő komponense van (1.2. ábra):

- Az egyes számítógépeken futó daemon processzek (*pvmd*), amelyek a gépek közötti kommunikációt koordinálják. A pvmd végzi a gép-gép kommunikációt, és szükség esetén biztosítja az adatok ábrázolásbeli különbségét eltüntető konverziókat.
- Az alkalmazás kódjához hozzászerkesztett könyvtári rutinok (*libpvm*), melyek részben a lokális daemon processzekkel tartják a kapcsolatot, másrészt ezek segítségével alakítható ki a párhuzamos programrészek közötti adatcsere és szinkronizáció.

A daemonok által logikailag egyetlen géppé összefogott gépeken a processzkezelő könyvtári rutinok segítségével egyszerűen kezelhetők a virtuális gép processzei, és dinamikusan átalakítható maga a virtuális gép is.



1.2. ábra. A PVM felépítése

MPI

Az MPI (Message Passing Interface) egy szabvány [46], amelyet 1992-ben kezdtek kialakítani részben a már bemutatott PVM rendszer fejlesztői. A fő cél egy olyan szabványos felület kidolgozása volt, amelyet a gyártók széles körben elfogadnak és alkalmaznak az üzenet-átadásos elven működő többprocesszoros számítógépekben.

A szabvány kialakításáért felelős MPI Forum számos, a párhuzamos programozásban hasznos kommunikációs módszert dolgozott ki, melyek a PVM rendszerben nem voltak alkalmazhatók, ugyanakkor szakítani próbált a PVM virtuális gép szemléletével. Így az MPI-1 szabvány nem definiál processzkezelő eljárásokat, ami miatt az MPI alkalmazások többnyire csak statikusan kiosztott együttműködő taszkok lehetnek, ami megnehezíti a hibatűrő alkalmazások fejlesztését. Ezért egy külön MPI szabványt dolgoztak ki, amely támogatja a hibatűrő alkalmazások írását. Azonban ez nem kompatibilis - ahogyan az MPI számos más változata sem - a korábbi MPI szabvánnyal.

Megítélésem szerint az MPI alapvetően gyártói érdekeket szolgál. Így nem támogatja a heterogén környezet kialakítását sem. Ez és a dinamikus processzkezelés hiánya különösen nagy hátrány az egyre növekvő számú Grid alkalmazások szempontjából. Az MPI szabvány alkalmazásától azt remélték, hogy a széles körben elfogadott felület segítségével számos hordozható párhuzamos alkalmazás készül. Ezen várakozások azonban csak részben teljesültek a fent említett hiányosságok miatt. Erre utal az is, hogy egyre újabb és újabb MPI változatok ill. kiegészítések jelennek meg.

Érdekes és ígéretes fejlődési ágnak tűnik a közelmúltban kifejlesztett LAM (Local Area Multicomputer) [53], amely MPI-1 kompatibilis fejlesztési környezetet nyújt, ugyanakkor különböző kiegészítésekkel megpróbálja az MPI problémáit kiküszöbölni. A PVM rendszernél látott daemon szerepét itt egy nano-kernel veszi át, melynek segítségével számos, az MPI-2 által támogatott processzkezelési feladat is megoldható. A LAM támogatja a heterogén rendszereket is, így jól felhasználható a Grid rendszerekben is.

A LAM rendszer nemcsak futtató, hanem fejlesztési környezetet is biztosít azzal, hogy a rendszerben indított processzek és azok által generált üzenetek teljes mértékben megfigyelhetők és felügyelhetők.

1.2.4. Futtató ill. ütemező környezetek

Futtató környezet ill. ütemező tekintetében a Condor, az SGE és a Globus különösen fontos rendszer, mivel alapvetően elosztott heterogén környezetben működnek. A Condor rendszert elsősorban nagy számítás- ill. erőforrás igényű feladatok megoldásához fejlesztették ki, azonban jól alkalmazható elosztott párhuzamos programok futtatására is.

Condor

A wisconsini egyetemen kifejlesztett Condor [6, 24, 52] rendszer alapvetően a hálózatba kapcsolt munkaállomások szabad erőforrásainak összegyűjtését és ezzel egy rugalmas, nagy feldolgozási képességű (High-Throughput Computing, HTC) rendszer kialakítását teszi lehetővé. A Condor rendszer lényegében egy elosztott, heterogén környezetben alkalmazható feladat-kezelő. Segítségével egyszerűen elérhető, hogy egy adott feladat a hálózatba kapcsolt számítógépek közül mindig azon, az egyébként szabad gépen fusson, amely a leginkább megfelel az alkalmazásnak. Igen fontos tulajdonsága a Condor-nak, hogy a megkezdett, de még be nem fejezett feladatot képes az egyik gépről a másikra átvinni, és ott folytatni. Erre akkor lehet szükség, ha az adott gép valamiért már nem használható tovább, pl. a tulajdonosa akarja azt használni.

A rendszer működésének egyik fontos eleme az ún. *ClassAds* mechanizmus. Ennek lényege, hogy a szabad erőforrásokkal rendelkező gépek „hirdetik" szabad erőforrásaikat, az erőforrást használni akaró programok pedig bejelentik igényeiket a rendszer megfelelő programjának. Ez a rangsorolás mellett összeilleszti az igényeket és lehetőségeket. Az igények és lehetőségek találkozásakor az adott igénylő megkapja és használhatja a kért erőforrást. Más hasonló feladatú feladatkezelőktől eltérően a Condor az erőforrásokat egységes formában kezeli, így lehetővé válik azok rangsorolása még akkor is, ha fizikai értelemben nem összehasonlítható mennyiségek. Ezen rangsorra tetszőlegesen bonyolult szabály illeszthető, ami meglehetősen nagy rugalmasságot biztosít a konfigurálásban.

A Condor rendszerrel futtatható alkalmazásokat különböző csoportokba, ún. univerzumokba soroljuk aszerint, hogy hogyan illeszthetők be a rendszerbe. Párhuzamos programok futtatását a PVM ill. az MPI univerzum támogatja közvetlenül. Ezekben a PVM ill. MPI programok változtatás nélkül, esetleg minimális változtatással futtathatók. Így a párhuzamosan futó PVM vagy MPI alkalmazások is képesek a Condor által összegyűjtött szabad erőforrásokat – elsősorban CPU – hatékonyan kihasználni.

A PVM programok szempontjából nagyon fontos előnyt jelenthet a Condor használata, mert a virtuális gép felépítését és üzemeltetését a Condor magára vállalja, ami nagyméretű elosztott rendszer esetén meglehetősen komoly feladatot jelent. A dolgozatban bemutatott számítási feladatok egy részénél magam is alkalmaztam a Condor rendszert erre a feladatra.

A PVM és MPI programok támogatása mellett a Condor egy objektumorientált Master-Worker (MW) fejlesztői könyvtárral is támogatja a párhuzamos programok fejlesztését ill. futtatását. Ennek segítségével olyan master-worker alkalmazás hozható létre, amely ki tudja használni a Condor nyújtotta opportunista[7] környezetet, kellően hibatűrő és jól skálázható. A módszer illusztrálására vezetésemmel elkészült a dolgozatban bemutatott PSA alkalmazás MW könyvtárat használó változata is [51].

[7] A lehetőségekhez jól alkalmazkodó.

SGE (Sun Grid Engine)

A Sun Grid Engine [48] a SUN által kifejlesztett mag a Grid rendszerekhez. Lényegében a Condor rendszerhez hasonlóan egy hosszútávú ütemező és erőforráskezelő, amely kellően nyitott ahhoz, hogy a Grid rendszerekre jellemző heterogén, dinamikusan változó környezetben alkalmazható legyen. Nyitottsága az alkalmazott protokollokban és azokban a scripting technikákban rejlik, melyekkel könnyen összeintegrálható más rendszerekkel.

Az SGE alapvetően kétszintű Grid kialakítására ad lehetőséget. Az alsó szinten (Cluster Grid) egyszerű számítógépes klaszterek és/vagy azokhoz kapcsolódó szuperszámítógépek helyezkednek el. A felső szint az ún. Campus Grid, amely az alsó szinteket fogja össze. Ezen a szinten az összes erőforrás egységesen látszik. Az SGE ütemezője az általa felügyelt UNIX rendszerek standard ütemezőjére épül. Ütemezési stratégiája könnyen módosítható ill. paraméterezhető. Az egyes komponensei közötti adatcsere biztonságos protokollon keresztül, kódolva történik.

A rendszer működésének legfőbb eleme a *job* (feladat) és a *queue*. A job a felhasználók által létrehozott háttérben futtatható feladat. Érdekessége a rendszernek, hogy a feladat mindig valamilyen script, és sohasem bináris program. Ez a script indítja el architektúrának megfelelő bináris programot vagy programokat. A Condor-hoz hasonlóan az SGE is különböző futtatási környezeteket definiál, mint pl: párhuzamos környezet, checkpointing környezet, interaktív környezet stb. Az egyes feladatok számára a felhasználó definiálja a megfelelő környezetet ill. fogalmazza meg a feladat erőforrásigényét.

A queue lényegében egy olyan gyűjtő, amelyhez az adott gépen rendelkezésre álló erőforrások (CPU, memória, diszk, stb.) vannak hozzárendelve. Egy gépen több ilyen erőforrás queue is lehet. Rendszerint ilyenkor ezekhez különböző típusú erőforrások tartoznak. A rendszer által futtatandó feladatok a job által igényelt erőforrások leírásával kiegészítve egy közös várólistán helyezkednek el. Az SGE ütemezője időnként végignézi várólistán levő feladatokat és a queue-ban levő szabad erőforrásokat, és megpróbálja kielégíteni az igényeket a rendelkezésre álló erőforrások alapján. Amikor egy feladatot elindít, akkor a feladat által használt erőforrások számát a queue-ban a használatnak megfelelően csökkenti.

Megvalósítását tekintve az SGE-t különböző alrendszerekből áll. Ezek mindegyike futhat különböző gépeken, de akár egy gépen is. Az egyes alrendszerek a következő funkciókat biztosítják:

- Mester ütemező
  Ez a rendszer központi ütemezője.
- Végrehajtó
  Ez azokon a gépeken fut, ahol felhasználói feladatok futhatnak. Amikor a központi ütemező úgy dönt, hogy egy feladatot elindít, akkor értesíti a megfelelő gép végrehajtó alrendszerét és az elindítja a kívánt feladatot.
- Adminisztrációs alrendszer.
  Ez teszi lehetővé a rendszer egyszerű karbantartását, rendszergazdai felügyeletét.
- Feladat benyújtó (submit) alrendszer.
  Ennek segítségével lehet egy új igényt benyújtani. A Condor-hoz hasonlóan a futtatás és a feladat benyújtás jól szétválasztható. Így lehet olyan gép, amin csak futtatni ill. csak feladatot benyújtani lehet.

Globus

A Globus [35, 24] toolkit alapvetően a Grid rendszerek létrehozásához kifejlesztett programcsomag. A programcsomag alapvető feladata a heterogén, folyton változó Grid rendszerek elemeinek, erőforrásainak biztonságos kezelése. Ezért olyan komponenseket tartalmaz, amelyek megkönnyítik nagy számítási hálózatok kialakítását. Ezen komponensek olyan alapvető szolgáltatásokat nyújtanak, mint a biztonság, erőforrás-keresés, erőforrás-kezelés, adatelérés vagy kommunikáció. A programcsomag nem az uniform programozási modellt támogatja, hanem inkább egy szolgáltatáshalmazt biztosít, amelyek közül a fejlesztők kiválaszthatják a szükséges elemeket. Ezek a szolgáltatások jól elkülönülnek és jól definiált interfésszel rendelkeznek, ezért lehetőség van arra, hogy különböző alkalmazásokba lépésről lépésre építsék be azokat. Ezt támogatja a réteges felépítés is:

1. Az alsó réteg (fabric).
   Ezek a különböző számítógépeket, operációs rendszereket, helyi erőforrás-kezelő rendszereket, adattároló rendszereket stb. tartalmazzák.

2. Szolgáltatás réteg.
   Ez olyan modulokat takar, amelyek különböző szolgáltatásokat nyújtanak a következő réteg felé. (pl: információs rendszer, erőforrás-kezelő rendszer, biztonsági rendszer, távoli adatelérést biztosító modul, stb.) Minden modul jól definiált interfésszel rendelkezik, amelyen keresztül a magasabb szintekről elérhetők a kívánt szolgáltatások. A modulok az alacsonyabb szinten levő szolgáltatásokat használják működésük megvalósításához.

3. Eszközrendszer réteg.
   Az alkalmazás-fejlesztő rendszerek a Grid eszközök segítségével tudnak magasabb szintű szolgáltatásokat nyújtani egy megcélzott alkalmazás-osztály számára. (pl: a távoli programfuttatást lehetővé tevő parancsok, az MPICH-G2, ami az MPI Grid változata, nagy mennyiségű adatok elosztott kezelését megkönnyítő fejlesztőrendszerek, stb.)

4. Alkalmazás réteg.

A legfontosabb Globus szolgáltatások már implementálva vannak, de egyes rétegek még üresek. Az eddig implementált kiemelten fontos szolgáltatások a következők:

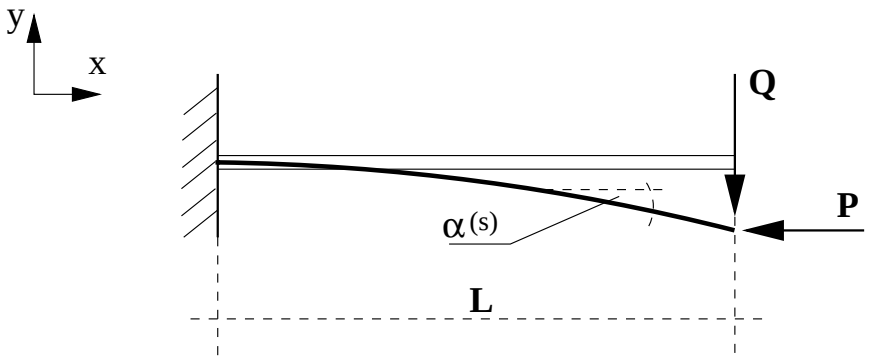
- Grid Security Infrastructure (GSI) A Grid erőforrások egymástól függetlenül szabhatják meg a használatukhoz szükséges biztonsági szabályokat. A Grid erőforrásokat használó alkalmazások számára a GSI könyvtár biztosítja ezen erőforrások eléréséhez szükséges biztonsági szolgáltatásokat.

- Grid Information Service (GIS) vagy Metacomputing Directory Service (MDS). Ez a szolgáltatás biztosítja a middleware információk eléréséhez szükséges közös interfészt.

- Globus Resource Allocation Manager (GRAM) Ez a könyvtár tartalmazza azt a közös felhasználói interfészt, ami ahhoz szükséges, hogy a Grid Fabric gépein programokat futtathassunk.

1.3. Peremérték feladatok általános bemutatása, jelentősége

A műszaki problémák egy jelentős része leírható közönséges differenciálegyenletek segítségével (KDE), melyek megoldását a probléma jellegéből adódó feltételek mellett keressük. Abban az esetben, ha ezek a feltételek, mint kezdeti feltételek állnak rendelkezésünkre, kezdetiérték feladatról



1.3. ábra. Egyik végén befogott rúd

(KF) beszélünk, amely valamilyen, rendszerint numerikus integrálási módszerrel, viszonylag egyszerűen megoldható. Azaz a differenciálegyenlet általános megoldása által kijelölt integrálgörbék (trajektóriák) közül a kezdeti feltételek egyértelműen meghatároznak egy trajektóriát, melynek egy meghatározott pontjából iterációs lépésekkel eljuthatunk a görbe egy másik pontjába. A gyakorlati problémák nagy részénél azonban nem ilyen egyszerű a helyzet, mivel a legritkább esetben adott minden feltétel egy pontban. Az esetek többségében a feltételek egy része nem a kiindulási pontban ismert, hanem pl. a végpontban, vagy esetleg egy vagy több közbülső pontban. Ezeket a feltételeket peremfeltételeknek vagy peremértékeknek nevezünk, és a megoldandó feladatot peremértékfeladatnak (PF) hívjuk. A dinamikai feladatok nagy része KF, de PF is előfordulhat. A statikus feladatok közül példaként tekintsünk egy hosszú, egyik végén befogott, egyenletes hajlítómerevségű rudat (1.3. ábra). Az ábra jelöléseit használva a rúd alakját a következő KDE írja le:

$$EI\alpha''(s) + P\sin(\alpha) + Q\cos(\alpha) = 0 \tag{1.1}$$

ahol:

E	-	a rúd anyagára jellemző állandó
I	-	a rúd keresztmetszetére jellemző állandó
P	-	a rúdra ható, tengelyirányú erő az $s = L$ pontban
Q	-	a rúdtengelyre merőleges erő
$\alpha(s)$	-	a rúdtengely és az x tengely által bezárt szög az ívhossz függvényében $s \in [0, L]$

A fenti KDE általános megoldását jelentő integrálgörbék közül egy adott P és Q mellett az $\alpha(0), \alpha'(0)$ kezdeti feltételek egyértelműen kijelölnek egy görbét, ami egyben a KF megoldását adja.

Példánkban azonban a két kezdeti feltételből az s=0 pontban csak az $\alpha(0) = 0$ feltétel adott. A szabad rúdvégből következik viszont, hogy a nyomaték ($M(s) = EI\alpha'$) az s=L pontban nulla, azaz $\alpha'(L) = 0$ érték adódik peremfeltételnek. A mechanikai problémát ezzel egy kétpontos PF megoldására vezettük vissza. Ha a fenti feladat megoldását rögzített tengelyirányú ($P = const.$) terheléssel, nemcsak egy adott Q mellett keressük, akkor egy úgynevezett egyparaméteres PF-hoz jutunk.

A gyakorlatban igen sok műszaki probléma vezethető vissza – a fenti egyszerű példához hasonlóan – egyparaméteres PF megoldására. Sokszor azonban a legegyszerűbb esetek is olyan bonyolult

differenciálegyenletre vezetnek, melyek analitikus megoldása reménytelen. Így numerikus eszközöket kell alkalmazni.

A numerikus megoldáshoz leginkább az ún. útkövetéses módszereket alkalmazzák, de a nagyobb számítási teljesítmények elérhetőségével megnyílt az út az ún. letapogató módszerek alkalmazása előtt is, mely számos előnnyel rendelkezik a korábbi módszerekhez képest.

A hagyományos útkövetéses módszerek lényege, hogy a kezdőpontból vagy más jól ismert megoldáspontból kiindulva megpróbálnak egy újabb, elfogadható hibahatáron belül levő megoldáspontba eljutni. Rendszerint a módszer valamilyen optimalizálási eljárással kombinálva iterációs lépések sorozatából áll [43].

Az útkövetéses eljárások rendszerint nem alkalmasak a feladat ún. globális vizsgálatára, mivel olyan új megoldási pontot nem képesek szolgáltatni, amely nincs folytonosan összekötve egyetlen ismert megoldással sem.

Merőben más, új módszert valósít meg az 1.5. alfejezetben bemutatott módszer [50], amely iterációs lépések kiküszöbölésével állítja elő a feladat megoldásait.

1.4. Globális módszerek lényege, célkitűzése

Globálisnak nevezünk egy módszert, ha a probléma matematikai modelljének felállítása után nem tesz elhanyagolásokat, hanem a modell megoldását elvileg tetszőleges pontossággal előállítja. Általában globális módszerre három fő ok miatt lehet szükség: [13]

- Az adott feladat esetében minden, a matematikai modellben megengedett megoldás elméleti vagy gyakorlati szempontból érdekes.
- Az adott feladat esetében nem jöhet létre fizikailag minden, a modellben megengedett konfiguráció, de éppen ennek vizsgálatára, tehát a matematikai modell érvényességi határának megismerésére alkalmazunk globális módszert.
- Az adott feladat vizsgálatánál beérjük ugyan a közelítő módszerek által szolgáltatott eredményeket, de annak megállapítására, hogy milyen tartományban fogadható el a választott közelítés, globális vizsgálatra van szükség.

A globális módszerek alkalmazásakor nem csupán a megoldás megkeresése jelent problémát, hanem azok tömör, egyértelmű leírása is. A probléma az, hogy a különböző paraméterektől és kezdeti feltételektől függő egyensúlyi alak ábrázolására a mérnöki gyakorlatban leginkább elterjedt módszer, miszerint az egyensúlyi alakot meghatározó függvény Furier-sorának együtthatóit tekintik koordinátáknak, nem adhat korrekt leírást a véges dimenziójú koordinátatérben. A rúdszerkezetek mechanikai vizsgálatához Domokos [12, 13] bevezetett egy ún. globális reprezentációs teret (GRT), amely alkalmas a globális megoldások tömör, pontos leírására, és egyben a PSA alapját is képezi. A GRT rövid bemutatásához tekintsük a következő KDE-t:

$$u'(s) = h(u(s), \lambda), \quad u \in \Re^k, \quad \lambda \in \Re, \quad h : \Re^{k+1} \to \Re^k, \quad s \in [0, L]) \tag{1.2}$$

A keresett megoldásgörbét az $u(0)$ kezdeti feltételek és a λ paraméter határozzák meg. Jelölje u_λ a λ paraméterrel bővített u vektort, amelynek így eggyel több komponense van, mint a KDE független változóinak száma, amit k-val jelöltünk. Az egyszerűbb leírás miatt tegyük fel, hogy ún. kétpontos, egyparaméteres peremérték-feladatunk van, azaz a peremfeltételek egy része az $s = 0$ pontban, más

része az $s = L$ helyen adott. Más szóval az u vektor egyes komponenseit az $s = 0$ és az $s = L$ helyen írjuk elő. Ha a KDE eleget tesz bizonyos feltételeknek, akkor a GRT terét az $u_\lambda(0)$ vektor azon komponensei (u_g) feszítik ki, melyekre nincs előírva peremfeltétel az $s = 0$ pontban. Ez a leírási mód azért előnyös, mert a matematikai modellből adódó KDE minden megoldásához egyértelműen hozzárendelhető a GRT egy pontja (unicitás), mely a konkrét fizikai konfigurációval egyértelműen összekapcsolható. Így a feladat megoldását jelentő egyensúlyi út a GRT segítségével korrekten, véges dimenziójú térben megadható ill. megrajzolható.

Ezek után vizsgáljuk meg az 1.3. ábrán bemutatott rúd viselkedését $P = 0$ tengelyirányú erő mellett. A fenti jelöléseket alkalmazva:

$$\begin{aligned} \lambda &= Q \\ u_\lambda(s) &= [\alpha(s), \alpha'(s), Q] \end{aligned}$$

A GRT-t pedig az

$$u_g = [\alpha'(0), Q] \tag{1.3}$$

vektor alkotja, hiszen adott az $\alpha(0) = 0$ kezdeti feltétel. Azaz ebben a kétdimenziós térben egyértelműen és korrekten ábrázolható a konzol globális egyensúlyi útja. A tér bármely pontjához megrajzolható egy olyan rúdalak, amely a KF egy megoldását jelenti, így az egyensúlyi úthoz tartozó alakok is egyértelműen kiszámíthatók.

A globális és lokális módszerek közötti különbségek egy példán történő megvilágításához a fenti mechanikai feladatot vizsgáljuk meg azzal a megszorító feltételezéssel, hogy csak kis elmozdulásokat vizsgálunk. Ezzel a feltételezéssel a PF megoldásainak vizsgálatát a triviális (terheletlen) megoldás kis környezetére szűkítjük, tehát az eredmények is csak megszorításokkal kezelhetők és nem globálisan. Ebben a tartományban azonban a rúd alakját leíró KDE linearizálható, így a linearizált egyenletet akár analitikusan is megoldhatjuk. Az 1.3. ábra jelöléseit használva a kis elmozdulás miatt a következő közelítéseket tehetjük:

$$\begin{aligned} \frac{dy}{ds} &= y' \approx \alpha, \\ \frac{dx}{ds} &= x' \approx 1 \end{aligned}$$

Ezeket felhasználva az

$$EIy''' + Q = 0 \tag{1.4}$$

lineáris differenciálegyenletet kapjuk, melyet a peremfeltételeknek megfelelően megoldva az

$$\begin{aligned} y''(0) &= \frac{QL}{EI} \\ y(x) &= \frac{Qx^3 - 3QLx^2}{6EI} \\ y(L) &= -\frac{QL^3}{3EI} \end{aligned} \tag{1.5}$$

összefüggéshez jutunk, ami természetesen csak a kis elmozdulások tartományában érvényes. Az 1.5. alfejezetben megmutatjuk, hogy a globális vizsgálat során kapott eredmények mennyire vágnak össze az itt kapott eredménnyel.

1.5. A szimplex módszer ismertetése

A GRT fogalmát felhasználva Domokos és Gáspár [14] egy olyan számítási módszert javasolt, amely alkalmas globális vizsgálatra. Tételezzük fel, hogy rendelkezésünkre áll egy KF megoldó függvény, amely a KF megoldását adja a [0, L] intervallumban. Ezen megoldó (integrátor) függvénnyel az s=0 pontból a megfelelő kezdeti értékekkel kiindulva előállíthatók az s=L peremre vonatkozó értékek, azaz feltételezésünk szerint létezik az

$$f_u(u_\lambda(0)) = u(L), \quad u_\lambda \quad \in \Re^{k+1}, \quad f_u : \Re^{k+1} \to \Re^k \tag{1.6}$$

függvény. Mivel az $u_\lambda(0)$ vektor egyes komponenseire az $s = 0$ peremre vonatkozóan van előírásunk, így azok valójában konstansként jelennek meg a fenti függvény változói között. A GRT meghatározásához viszont csak azon komponenseket használtuk, amelyekre nem volt előírásunk az $s = 0$ pontban. Helyezzük ezeket a szabad komponenseket az x vektorba. Ez a GRT dimenziószámával (n) egyező számú változót jelet. Az $u(L)$ azon komponenseit pedig, amelyekre csak az $s = L$ pontban van előírás, helyezzük az u_L vektorba. Az említett konstans paraméterek formális elhagyásával bevezethetjük az f_g függvényt, amely formálisan nem függ az $s = 0$ peremen megadott peremértékektől. Miután u_L vektor is csak konstansokat tartalmaz formálisan azt is elhagyhatjuk, így az 1.7 nemlineáris algebrai egyenletrendszerhez jutunk az $s = L$ peremre vonatkozóan.

$$f(x) \quad = \quad f_g(x) - u_L = 0, \quad x \in \Re^n, \quad u_L \in \Re^{n-1}, \quad f : \Re^n \to \Re^{n-1} \tag{1.7}$$

Ahol f függvény a fent bevezetett megoldó függvény azon változata, amely formálisan nem függ a peremértékektől. (A konstans paraméterek formális elhagyása miatt $n \leq k$.) Ezen egyenletrendszer megoldása egyértelműen és korrekten szolgáltatja az eredeti PF megoldását. Az így kapott megoldások a GRT segítségével könnyen kezelhetők és tárolhatók, melyekből az egyensúlyi alak egyszerűen előállítható.

Az 1.3. alfejezetben ismertetett egyszerű konzol esetében az 1.7 képlet egyetlen skalár egyenletet jelent, melynek megoldásával az eredeti mechanikai feladat globálisan vizsgálható és leírható:

$$f_1(\alpha'(0), Q) = \alpha'(L) = 0 \tag{1.8}$$

Ezzel az eredeti PF globális vizsgálatát egy nemlineáris skalár egyenlet megoldására vezettük vissza. Ez utóbbi feladatot több ismert numerikus eljárással is megoldhatjuk.

A fenti gondolatmenet során feltételeztük, hogy a PF jellegéből és megfogalmazásából adódóan az 1.7 egyenlet létezik, és van megoldása. Megjegyezzük, hogy a dolgozatban bemutatott önálló eredmények az egyenletrendszer hatékony megoldására kidolgozott numerikus módszerekre ill. algoritmusokra és, nem a függvények előállítására vonatkoznak. Ezért a továbbiakban feltételezzük, hogy a feladat matematikai modelljéből adódó egyenletek az n dimenziós GRT-ben rendelkezésünkre állnak.

Domokos és Gáspár [12] a fenti egyenlet megoldására a PL (Piecewise Linear) algoritmust [1] javasolja. Ezzel a GRT egy kijelölt részében elvileg az összes megoldás numerikusan előállítható. A módszer lényege, hogy a vizsgálandó teret egy ortogonális hálóval n dimenziós, egybevágó téglatestekre bontjuk, majd a téglatesteket úgy bontjuk szimplexekre, hogy azok a téglatesteket hézag- és átfedésmentesen kitöltsék. Ezek olyan alakzatok, melyeknek pontosan eggyel több csúcspontjuk és oldaluk van, mint a tér dimenziószáma[8]. A csúcspontokon kapott függvényértékeket felhasználva linearizáljuk a függvényt a szimplex tartományában, azaz a megoldandó egyenletrendszert a szimplex

[8]Egy n dimenziós szimplexet $n + 1$ csúcspont határoz meg, 2 dimenziós esetben a szimplex egy háromszög, 3 dimenziós szimplex neve tetraéder, stb.

belsejében $n-1$ darab lineáris függvénnyel helyettesítjük:

$$f(x) \approx f_i^L(x) = \sum_{j=1}^{n} a_{ij}x_j + a_{i0}, \quad x \in \Re^n \quad i = 1..n-1 \tag{1.9}$$

A szimplex tartományában az $n+1$ csúcsponton ismert függvényértékhez egyszerűen kiszámíthatók a helyettesítő függvény együtthatói, hiszen pontosan $n+1$ együtthatóra van szükségünk.

Ha az így kapott lineáris egyenletrendszernek van megoldása az adott szimplexen belül, akkor az egy n dimenziós térben levő egyenes szimplexbe eső szakasza kell hogy legyen, mivel az n dimenziós térben $n-1$ lineáris egyenlet megoldásaként állítottuk azt elő. Azaz ez az egyenes a szimplex egyik oldalán be-, a másik oldalán kilép. Az így nyert döféspontok által meghatározott egyenes viszont nem más, mint az eredeti egyenletrendszer közelítő megoldása.

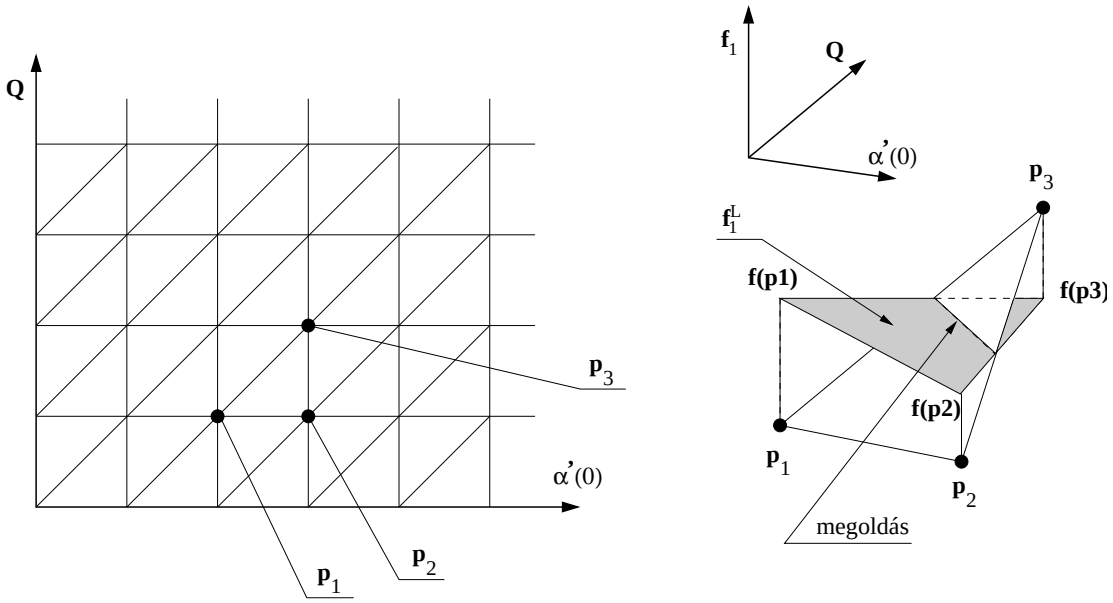
A kérdéses egyenes és a szimplex oldallapjainak döféspontját egyszerű egyenletmegoldó módszerrel meghatározhatjuk. Ehhez a linarizálás során nyert $n-1$ darab lineáris egyenletet rendre kiegészítjük a szimplex egy-egy oldalának egyenletével, és az így kapott n darab lineáris egyenletből álló n ismeretlenes egyenletrendszert megoldjuk. (Ez legfeljebb $n+1$ darab n változós lineáris egyenletrendszer numerikus megoldását jelenti.)

A szimplexek előállításához Gáspár és Domokos [14] egy igen jól algoritmizálható módszert javasol. Ennek lényege, hogy az ortogonális hálóval kialakított n dimenziós téglatest egyik csúcsából a téglatest éleinek mentén megpróbálunk eljutni a téglatest szemközti csúcsába úgy, hogy minden koordináta irányába csak egyszer haladunk. A keresett szimplex csúcsait az érintett csúcsok jelölik ki. Bizonyítható, hogy ha az összes lehetséges úton eljutunk a szemközti csúcsba a fenti módon, akkor az így keletkező pontosan $n!$ szimplex hézagmentesen és átfedés nélkül kitölti az n dimenziós téglatestet.

Ha a korábban bemutatott egyszerű konzol esetére alkalmazzuk a fent bemutatott módszert, akkor az 1.3 összefüggés alapján az $\alpha'(0)$ és Q koordináták által kifeszített síkon keletkeznek az 1.4. ábrához hasonló szimplexek, melyek esetünkben háromszögek. Az ismertetett módszer szerint ezen háromszögek csúcsain ki kell számítani az 1.8 képletben szereplő f_1 függvényt, és a kapott függvényértékek alapján az adott szimplexben a függvény linearizált változatát elő kell állítani. Ezt követően a kapott lineáris egyenletet a szimplex oldalainak egyenletével kiegészítve meg kell oldani. Amennyiben a lineáris egyenletnek van az adott szimplexben (háromszög) megoldása, úgy az egy egyenes, mely a szimplex egyik oldalán belép, a másik oldalán kilép.

Könnyen belátható, hogy a lineáris függvény előállítása lényegesen egyszerűsödik, ha csúcsukkal az origóba transzformált, egységnyi méretű szimplexeket alkalmazunk. Egyszerű példánkban ez egységnyi oldalhosszúságú háromszöget jelent, mely fölött úgy kell előállítani az 1.9 képletben szereplő együtthatókat, hogy a linearizált függvény függvényértéke a csúcsokon megegyezzen az eredeti függvény függvényértékével. Ebből a $p_1 = (0,0)$, $p_2 = (0,1)$, $p_3 = (1,1)$ koordinátájú háromszögre az 1.10 összefüggéssel megadott értékek adódnak.

$$\begin{aligned}
f_1^L(\alpha'(0), Q) &= a_{11}\alpha'(0) + a_{12}Q + a_{10} \\
f_1(0,0) = f_{10} &= f_1^L(0,0) \\
f_1(0,1) = f_{11} &= f_1^L(0,1) \\
f_1(1,1) = f_{12} &= f_1^L(1,1) \\
a_{10} &= f_{10} \\
a_{11} &= f_{11} - a_{10} \\
a_{12} &= f_{12} - a_{11} - a_{10}
\end{aligned} \tag{1.10}$$



1.4. ábra. Szimplexekre bontás és linearizálás 2 dimenzióban

A szimplexet határoló egyenesek egyenletei pedig rendre:

$$\begin{aligned} Q &= 0, \\ \alpha'(0) &= 1, \\ \alpha'(0) &= Q \end{aligned}$$

Példában szereplő szimplex vizsgálatához a linearizálás után 3 db 2 változós lineáris egyenletrendszert kell megoldani.

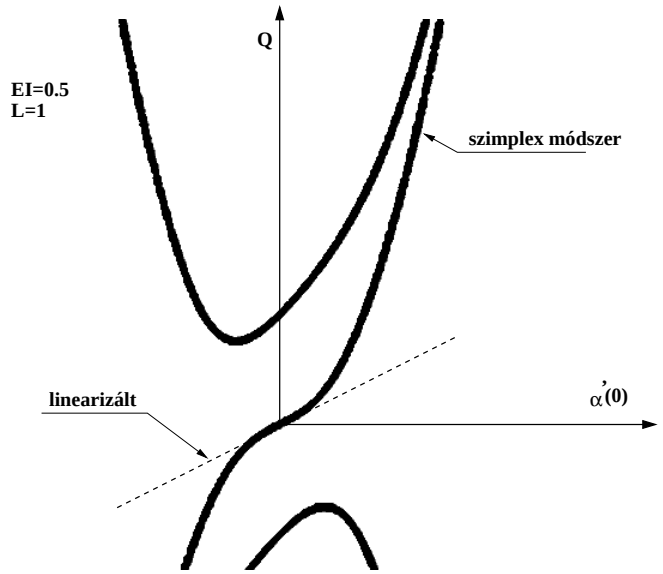
A módszert a vizsgálandó tér minden szimplexére elvégezve, elvileg tetszőleges pontossággal előállíthatók az eredeti PF megoldását egyértelműen megadó pontok az $\alpha'(0)$ és Q koordináták (GRT) által kifeszített síkon. Az 1.5. ábrán együtt ábrázoltuk a szimplex módszerrel számított eredményeket és az 1.5 összefüggéssel nyert, kis elmozdulásokra érvényes eredményeket.

A fentiekben bemutatott szimplex letapogató módszer tömör összefoglalását az 1.1. algoritmusleírás szemlélteti. Fontos kiemelni, hogy a módszer nem tartalmaz iterációs lépéseket, valamint képes olyan egyensúlyi utakat is felfedezni, amelyek nincsenek egymással, ill. a triviális megoldással összekötve. Ezért különösen alkalmas peremérték-feladatok globális vizsgálatára. A módszer pontosságát alapvetően két tényező befolyásolja:

- A KF megoldó integrátor pontossága, ami az alkalmazott módszertől és lépésszámtól függ.
- A GRT linearizálása során elkövetett hiba, ami alapvetően az ortogonális háló sűrűségétől függ. Az itt elkövetett hibák nagysága az eredeti egyenletbe való visszahelyettesítéssel ellenőrizhető.

Megjegyezzük, hogy egyes speciális esetekben külön kérdésként merülhet fel, hogy milyen módon változnak meg az eredeti nemlineáris egyenletrendszer tulajdonságai a véges és diszkrét[9] számábrázolás miatt. Ezzel a kérdéssel azonban jelen dolgozatban nem foglalkozunk.

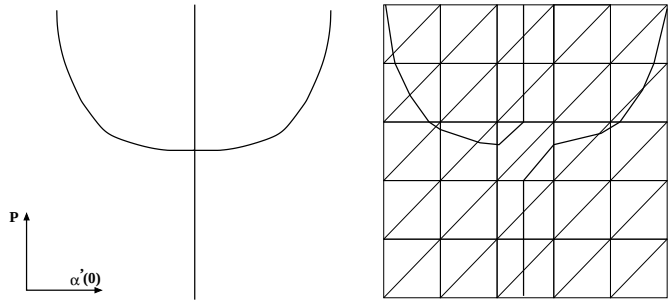
[9] A valós számok számítógépekben történő ábrázolása nem folytonos és véges pontosságú.



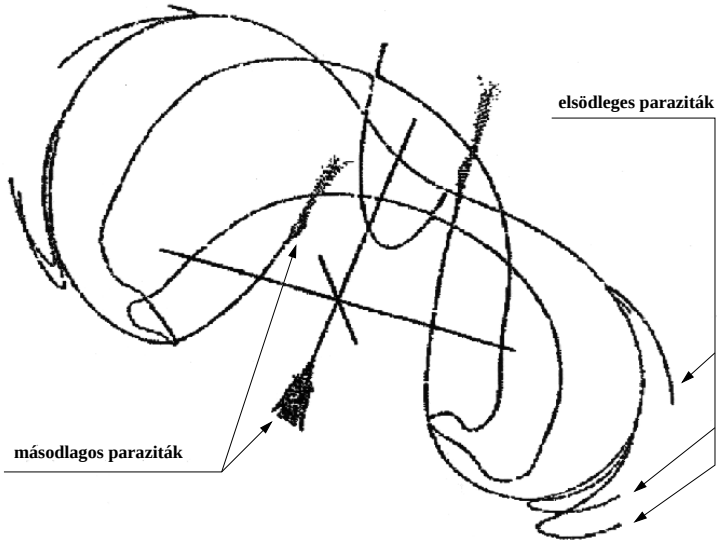
1.5. ábra. A rúd egyensúlyi útja $P = 0$ feltétel mellett

```
Az n dimenziós GRT ortogonális hálóval való lefedése.
while van feldolgozandó téglatest do
    A téglatest lineáris transzfomációja (egységnyi oldal).
    A kocka szimplexekre bontása.
    while van feldolgozandó szimplex do
        A szimplex csúcsaihoz tartozó függvényértékek előállítása.
        Linearizált függvény előállítása.
        while van feldolgozandó oldalfüggvény do
            Következő oldalfüggvény előállítása.
            Egyenletrendszer megoldása.
            if a megoldás rajta van az oldalon then
                Eredmény ideiglenes letárolása.
        end
        if pontosan 2 döféspont van és az eredmények elég pontosak then
            Lineáris transzformáció a valós térbe.
            Megoldáspontok közlése.
        end
        Ideiglenes eredménytár törlése.
    end
end
```

1.1. algoritmus: Szimplex letapogató módszer



1.6. ábra. A rúd bifurkációs diagramja $Q = 0$ feltétel mellett, és a linearizálásból keletkező hiba.



1.7. ábra. Első- és másodlagos paraziták a két végén befogott rúd bifurkációs diagramján.

Ha a KF integrálásakor alkalmazott lépésköz nem kellően kicsi, akkor a bifurkációs diagramban keletkezhetnek új ágak, melyeket elsődleges parazita megoldásoknak nevezünk. Ha a GRT linearizálásakor használt lépésköz nem megfelelően kicsi, akkor sok apró zárt poligonnal találkozunk, ezeket másodlagos parazita megoldásoknak nevezzük [26]. Mindkét fajta parazita megjelenésére mutat példát az 1.7. ábra. A parazitákat egyrészt a linearizálás finomításával, másrészt a függvénybe való visszahelyettesítéskor számított hibák alapján lehet eltávolítani ill. kiszűrni (ld. 3.2. alfejezet).

A linearizálásból adódóan a szimplexek belsejében a megoldásgörbe nem ágazhat el, hiszen egy szimplexbe csak egy egyenes szakasz eshet. Ebből adódóan a megoldásgörbe a bifurkációs pontoknál kettészakad. Ilyen esetet szemléltet az 1.6. ábra. Bal oldalt az 1.3. alpontban bemutatott egyszerű konzol bifurkációs diagramja látható $Q = 0$ feltétel mellett, a jobb oldalt pedig szimplex módszer során kapott eredményt ábrázoltuk. A rácsméret csökkentésével a szétszakadó görbedarab egyre közelebb kerül egymáshoz, így elvben a kívánt pontossággal meghatározható a tényleges bifurkációs pont helye a GRT-ben.

2. fejezet

A szimplex letapogató algoritmus párhuzamos implementációja

Jelen fejezet az algoritmus párhuzamosításának fontosabb lépéseit, ill. a párhuzamosítás lehetséges módozatait mutatja be. Ismerteti a megvalósított program programrészei közötti kapcsolatokat, valamint a megvalósítás során felmerülő problémákat. Rámutat a hatékony megvalósítás fontosabb algoritmusaira. A feldolgozási teljesítmény további növelésének lehetőségeit részletesen a 3. fejezet ismerteti. Ehhez a fejezethez az alábbi tézisek kapcsolódnak:

1. Tézis. *A Szimplex Módszer letapogató változatához (PSA) – a párhuzamosítási lehetőségek feltárása és számszerű egybevetése alapján – párhuzamos algoritmust dolgoztam ki és implementáltam. Az elkészített programot heterogén elosztott környezetben és párhuzamos architektúrán is eredményesen teszteltem.*

2. Tézis. *Analitikus közelítő modellt dolgoztam ki a párhuzamosítás hatékonyságának vizsgálatára. A modell érvényességét mérésekkel igazoltam. A kidolgozott modell lehetővé teszi, hogy a letapogató módszerhez a konkrét feladattól függetlenül előzetesen becsülni lehessen a leghatékonyabb feladatfelosztást.*

2.1. Párhuzamos algoritmus alkalmazásának lehetősége

Az 1.5. alfejezetben ismertetett szimplex algoritmus igen számításigényes, mivel a linearizálás miatt csak megfelelően kis részekre osztott GRT esetén várhatunk kellően pontos eredményt. Ez azt jelenti, hogy nagyon sok rácspontunk keletkezik. Ugyanakkor a számítás során rácspontokban a függvényértékeket minél pontosabban kell előállítani. Ezért a függvényértékek előállítását végző numerikus integrátornak is kellően kicsi lépésközzel kell dolgoznia, amely tovább növeli az algoritmus számítási igényét. Az általunk eddig vizsgált feladatok GRT-je 2-6 dimenziós, de bonyolultabb technikai problémák esetében nem ritka a 7 vagy magasabb dimenziószám sem.

A letapogató algoritmus számításigényének számszerű becsléséhez a következő algoritmusrészeket vizsgáltuk:

- Függvényértékek előállítása egy rácsponton.
  Az n dimenziós GRT minden rácspontján $n-1$ darab skalár értéket kell előállítani a KF integrálásával. A numerikus integrálási módszerek számítási igénye a lépésszámtól (k) lineárisan

függ. A tényleges számítási idő nagyban függ a konkrét KF-től. A további számításokban ezt az időt $k \cdot T_f(n)$ értékkel közelítjük, ahol $T_f(n)$ az n változós KF függvényeinek egy pontban való kiértékelésének ideje.

- *Összes függvényérték kiszámítása.*
  Ha feltételezzük, hogy a GRT minden koordinátáját l részre osztjuk, akkor az ortogonális hálóval való felosztás során nyert rácspontok száma a szélső rácspontokkal együtt $(l+1)^n$. Így az összes függvényérték kiszámítása $T_{fo} = (l+1)^n \cdot k \cdot T_f(n)$ számítási időt jelent.

- *A számítások elvégzése egy kockában.*
  A számítások elvégzéséhez az ortogonális hálóval való felosztás során nyert n dimenziós kockákat szimplexekre osztjuk. A felosztás során pontosan $n!$ szimplex keletkezik, így egy kockán belül $n!$-szor végezzük el a szimplex számítást.

- *Egy szimplex számítása.*
  Az algoritmusban alkalmazott Gauss-elimináció műveletigénye a dimenziószám köbével arányos. A továbbiakban jelölje $T_G(n)$ az n változós egyenlet megoldásának idejét. A Gauss-eliminációt a szimplex minden oldalával el kell végezni, így egy szimplex számítási ideje $T_s = (n+1) \cdot T_G(n)$ alapján becsülhető.

A fentieket figyelembe véve a teljes számítás időigénye:

$$\begin{aligned} T &= T_{fo} + l^n \cdot n! \cdot T_s = \\ &= (l+1)^n \cdot k \cdot T_f(n) + l^n \cdot n! \cdot (n+1) \cdot T_G(n) = \\ &= (l+1)^n \cdot k \cdot T_f(n) + l^n \cdot (n+1)! \cdot T_G(n) \end{aligned} \tag{2.1}$$

Ebből a közelítő összefüggésből jól látható, hogy a számítás ideje különösen érzékeny a vizsgált GRT felosztására (l), és az integrálás lépésszámára (k), valamint a dimenziószámra (n). A gyakorlatban eddig vizsgált feladatokban n értéke 10 alatt maradt, ezzel szemben l és k nagyságrendje 10^2-ra, vagy ennél nagyobb értékre adódott. T_G viszonylag jól mérhető érték és a vizsgált KF-tól független. Ezzel szemben T_f erősen feladatfüggő. Méréseink alapján kijelenthetjük, hogy $T_f \gg T_G$, amiből következik, hogy a 2.1 összefüggés 2. tagja csak kis mértékben befolyásolja a számítási időt.

T_G hatását tovább csökkenti, hogy nem kell minden szimplexben kiszámítani, ugyanis a szimplexben végzendő számítások megkezdése előtt célszerű ellenőrizni, hogy várható-e megoldás az adott szimplexben. Így a Gauss-eliminációt csak akkor kell végrehajtani, ha az adott szimplex csúcsain előjelet váltanak a függvények. Ezzel jelentős számú műveletvégzést takaríthatunk meg, mivel a térnek csak kis részében számítunk megoldásra. A továbbiakban ezért T_G hatását elhanyagoljuk. Ez azt jelenti, hogy a számítás ideje gyakorlatilag csak a KF függvényértékeinek számítási idejétől függ:

$$T = (l+1)^n \cdot k \cdot T_f(n) \tag{2.2}$$

Elsősorban l és k nagyságrendje miatt a módszer nagy számítási teljesítményt igényel ahhoz, hogy alkalmazásával reális méretű feladatokat reális idő alatt meg tudjunk oldani. Példaként vegyünk egy $n = 5$ dimenziós GRT-t $l = 100$-as felosztással, és $k = 500$ integrálási lépésszámmal. Ekkor $T = 101^5 \cdot 500 \cdot T_f(5) = 5.2 \cdot 10^{12} \cdot T_f(5)$ érték adódik.

A számítási kapacitás növelésének és a számításra fordított valós idő csökkentésének egyik módja a párhuzamosítás, azaz a feladatot több részre bontjuk, és azokat a lehetőségeknek megfelelően párhuzamosan végezzük el. Megvizsgálva az 1.1. algoritmust, több partícionálási lehetőség is adódik:

1. *A függvényértékek párhuzamos előállítása több rácsponton egyszerre.*
   A függvényértékek kiszámítása a numerikus integrálás miatt jelentős ideig tart. A pontok azonban egymástól függetlenül számíthatók, így azok párhuzamosan is kiértékelhetők. Ez a fajta feladatmegosztás gyakori adatcserét igényel a függvényértékeket előállító és azokat felhasználó programrészek között.

2. *A numerikus integrálás párhuzamosítása.*
   Általában a numerikus integrálási módszerek jól párhuzamosíthatók, ugyanakkor az előző megoldáshoz hasonlóan a módszer alkalmazása a feladatban gyakori adatcserét igényel.

3. *A lineáris egyenletrendszer megoldásának párhuzamosítása.*
   Elsősorban sok ismeretlen esetén várhatunk sebességnövekedést a különböző egyenletmegoldó módszerek párhuzamosításával. Esetünkben azonban viszonylag kevés – kevesebb, mint 10 – ismeretlennel számolunk.

4. *A GRT felosztása kisebb térrészekre és az egész feladat párhuzamosítása.*
   Az 1.5. alfejezetben ismertetett számítás a GRT bármely részében a szomszédos részektől teljesen függetlenül elvégezhető, ezért kézenfekvőnek tűnik ez a fajta felosztás. Megjegyezzük, hogy így a térrészek határánál levő pontokban a függvényértékeket többször is kiszámítjuk, ami a párhuzamosítás hatásfokát rontja.

Feladat neve	Kubusok száma	Integr. lépes (k)	Teljes T_t[s]	Algor. T[s]	Függv. $T_{fo}[s]$	Szimpl $T_{so}[s]$	$\frac{T_{fo}}{T_t}$	$\frac{T_{so}}{T_t}$	r_{sgn}
fraser30	$10908 \cdot 10^3$	500	650.5	650.3	620.2	3.1	0.95	0.004	0.003
fraser31	$10589 \cdot 10^3$	500	3768.7	3768.4	3551.0	9.3	0.94	0.002	0.001
telm0	$64 \cdot 3^6$	10	227.0	226.9	23.0	201.7	0.10	0.889	0.145
telm1	$729 \cdot 3^6$	10	970.1	970.0	272.3	671.8	0.28	0.693	0.041
telm606	$32 \cdot 6^6$	100	2599.0	2598.7	1932.3	617.8	0.74	0.237	0.017
telm706	$64 \cdot 6^6$	500	17216.6	17216.3	16012.9	1112.5	0.93	0.065	0.014

2.1. táblázat. Mérési eredmény: Az algoritmus egyes részeinek időbeli viselkedése.

A fenti párhuzamosítási módoktól várható előnyök ill. hátrányok összevetéséhez a következőket vizsgáltuk meg:

a.) *Várható sebességnövekedés aszimptotikus becslése.*
A maximális sebességnövekedés közelítő meghatározásához az Amdahl-féle becslés – a konkrét hardver architektúra ismerete nélkül is – jól alkalmazható (ld. az 1.2.2. alfejezetet). Ehhez megbecsültük az egyes módszerek alkalmazása során keletkező kód párhuzamosítható részének arányát. A becsléshez méréseket végeztünk a GNU profiler [31] segítségével, ami részletes képet adott a program letapogató változatának időbeli viselkedéséről. Néhány ilyen mérési eredményt szemléltet a 2.1. táblázat. A mérés során a következő mennyiségeket vizsgáltuk:

- T_t - teljes futási idő.
- T - megoldó algoritmus futási ideje (a nem párhuzamosítható részek nélkül)
- T_{fo} - az összes függvényérték előállítási ideje
- T_{so} - szimplexekben végzett összes számítás ideje
- k - integrálási lépésszám
- r_{sgn} - az összes szimplex és az előjelváltó szimlexek aránya

A mérést több különböző feladaton is elvégeztük. Várakozásainknak megfelelően a PF jellegétől és dimenziószámától függően eltérő eredményeket rögzítettünk. A mérés során a $telm0$ és $telm1$ feladatok integrálási lépésszámát rendkívül alacsonyra választottuk (10-re). A többi esetben a gyakorlatban szokásos 100 és 500 közötti értékkel számoltunk. A mérések során kapott eredmények igazolták korábbi feltevéseinket. Az eredményekből levonható legfontosabb következtetések a következők:

1. A függvényértékek kiszámítási T_{fo} ideje igen jelentős. Ez a teljes futási idő 30-96%-át teszi ki.
2. A függvényértékek kiszámítása egy rácsponton több időt – rendszerint lényegesen több időt – vesz igénybe, mint egy szimplex számítási ideje ($k \cdot T_f \gg T_s$).
3. A függvényértékek kiszámítása egy rácsponton több időt vesz igénybe, mint egy kocka számítási ideje ($k \cdot T_f > n! \cdot T_s$).
4. Azoknak a szimplexeknek az aránya, melyek csúcspontjain a függvények előjelet váltanak kicsi, és a felosztás növelésével csökken (T_{sgn}).
5. Az egyéb programrészek futási ideje szinte elhanyagolhatóan kicsi. Ez csupán a teljes futási idő 0.5-2%-át teszi ki.

A mérési eredmények jellegének figyelembevételével a párhuzamosítható részek arányára a következő becslés adható:

- Az első 2 párhuzamosítási lehetőségre azt találtuk, hogy a párhuzamosítható rész aránya erősen feladatfüggő. Ez nem meglepő, hiszen lényegében mindkét esetben a feladattól függő függvényértékek előállításának párhuzamosításáról van szó. Becslésünk szerint a párhuzamosítható rész a 30-96%-os.
- A 3. lehetőség is erős feladatfüggőséget mutat, bár a függőség jellege éppen fordított. Ennek oka az, hogy amíg az első két esetben az erősen feladatfüggő rész párhuzamosításáról volt szó, addig a 3. esetben a feladattól nem, vagy csak kis mértékben függő rész párhuzamosítása jön szóba. Ezért az első két esetben a feladatfüggő rész számítási igényének növekedése a párhuzamosíthatóság arányát növeli, míg a 3. esetben csökkenti. A viszonylag kis számú ismeretlen miatt azonban a párhuzamosítható rész aránya még nagyon egyszerűen kiszámítható KF esetén is biztosan 60% alatt marad. Ezt az arányt tovább rontja, hogy a GRT felosztásának növelésével egyre kevesebb szimplexben kell alkalmazni az egyenletmegoldás algoritmusát.
- A 4. esetben lényegében a teljes algoritmus párhuzamosítható, csupán néhány szervező utasítás és az adatcsere egy része marad a szekvenciális kódban. Ezek számításigénye alacsony, és a feladattól gyakorlatilag független. Így a feladatfüggő részek számítási igényének függvényében a párhuzamosítható rész aránya 98-99.5%-os. A hatékonyságot azonban csökkenti, hogy a térrészek szélein levő rácspontokon a függvényértékek többször is kiszámítódnak. Ez a hatás magasabb dimenziószám mellett jelentős is lehet.

b.) *Egyes részek közötti kommunikációs igény.*
A kommunikációs igény nagysága alapvetően befolyásolja a konkrét implementáció hardver igényeit, és ezen keresztül komoly korlátozó tényező is lehet az alkalmazható architektúrával szemben. Fontos kérdés lehet például, hogy a megvalósítás alkalmazható-e lazán csatolt architektúrán.

- Az első 3 párhuzamosítási lehetőség megvalósítása viszonylag nagy mennyiségű adatcserével jár, mivel a függvényértékek nem ott keletkeznek, ahol majd azok felhasználásra kerülnek.
- A 4. esetben a számítások nem igényelnek a térrészek között adatcserét. Csupán a térrészeket előállító és az eredményeket begyűjtő programrész (szervező vagy mester) és a számításokat végző programrészek között van adatkapcsolat. Ennek mértéke kicsi, mivel alapvetően kevés megoldást várunk egy-egy térrészben.

c.) *Terhelés-kiegyenlítés (Load balancing) lehetősége és kivitelezhetősége.*
Fontos vizsgálati szempontnak tekintettük, hogy a párhuzamos architektúrán természetesen kialakuló terhelési különbségek jól kiegyensúlyozhatók legyenek, és minden processzor a lehető legjobban kihasználható legyen.

- Az első 3 párhuzamosítási lehetőség esetében külön algoritmus alkalmazásával megoldható a terhelés kiegyenlítése. A nem jól kiegyenlített rendszerben azonban a teljes számítás leállhat ill. lelassulhat.
- A 4. esetben automatikus a terhelés-kiegyenlítés megvalósulása. Ez abból adódik, hogy az egyes térrészek számítását vezérlő mesterprogram nem fog (nem tud) újabb feladatot adni a foglalt vagy erősen terhelt processzoroknak. Azok a processzorok, amelyek hamarabb befejezik a feladatukat, viszont ezalatt újabb térrész számításba kezdhetnek. Ehhez nem kell más, mint az, hogy az eredeti térrész több darabra legyen felosztva, mint ahány feldolgozó egységünk (processzor) van. Természetesen a túl sok darab a redundánsan számolt pontok számát növeli, valamint a mester kommunikációs csatornái is túlterhelődhetnek ezáltal. Fontos hangsúlyozni, hogy a többi esettel ellentétben ennél a módszernél nem fordulhat elő, hogy egy terheletlen processzor a terhelt processzor miatt várakozzon.

d.) *Újrafelhasználhatóság.*
Itt azt vizsgáltuk, hogy egy adott PF-hez elkészített implementáció mennyire könnyen alakítható át más feladatokhoz. Joggal feltételezhető, hogy egy olyan programrész átírása, módosítása és nem utolsó sorban tesztelése lényegesen bonyolultabb, amelyiknek van kapcsolata a párhuzamos környezettel, mint amelyiknek nincs.

- Az 1., 3. és 4. eset szerinti párhuzamosítás nem érinti a feladatspecifikus függvényeket. Így azok megjelenésükben, konkrét megvalósításukban nem kell, hogy különbözzenek egy nem párhuzamos környezethez készített verziótól.
- A 2. eset szerinti párhuzamosítás az integrálközelítő összeg kiszámítását végző algoritmust érinti. Ez a leginkább feladatspecifikus rész, így egy újabb feladathoz ezt biztosan módosítani kell. Megjegyezzük, hogy egy általános párhuzamos integráló algoritmus alkalmazásával a feladatfüggő rész függetleníthető lenne a párhuzamosítástól, de ez felesleges megkötéseket jelenthetne a függvényekre vagy éppen az integráló algoritmusra.

e.) *Skálázhatóság.*
Itt azt vizsgáltuk, hogy az elkészült implementáció mennyire illeszthető a feladat nagyságához, illetve a rendelkezésre álló erőforrás (processzorszám) nagyságához.

- Az 1. esetben látszólag jól skálázható rendszerhez juthatunk, hiszen a függvényértékek a rácspontokon szabadon és tetszőleges sorrendben kiszámíthatók. A probléma azonban párhuzamosan kiszámolt értékek feldolgozásával van, hiszen azt nem párhuzamosítjuk.

– A 2. és 3. esetben a skálázhatóság nagyban függ az integrálás lépésszámától ill. a GRT dimenziószámától.

– A 4. eset nagy GRT ill. sok rácspont esetén különösen jól skálázható.

	Párhuzamosítás módja			
Vizsgálati szempont	**1.**	**2.**	**3.**	**4.**
S_N^A	< 10	< 10	< 2.5	> 50
kommunikációs igény	magas			alacsony
terhelés-kiegyenlítés	megoldható			automatikus
újrafelhasználhatóság	jó	problémás	jó	jó
skálázhatóság	korlátos	feladatfüggő		jó

2.2. táblázat. Lehetséges párhuzamosítási módok összevetése

Vizsgálatunk eredményét a 2.2. táblázatban foglaltuk össze. Figyelembe véve a különböző partícionálási lehetőségek előnyeit és hátrányait, a 4. módszer (domain dekompozíció) tűnik a legelőnyösebbnek, ezért ezen elvek szerint került megtervezése és megvalósításra a 2. fejezetben bemutatott párhuzamos szimplex algoritmus. Döntésünk mellett, illetve ellene a következő érvek szóltak:

1. Becsléseink szerint domain dekompozícióval lehet a legnagyobb párhuzamosítható arányt elérni, ezért a várható sebességnövekedés is ezzel a módszerrel a legnagyobb.

2. Domain dekompozícióval mind a feladatfüggő, mind a feladatfüggetlen részeket együtt párhuzamosítjuk, így a feladatfüggő rész számítási igényének változása nincs hatással az elérhető sebességnövekedésre.

3. A feladatfüggő rész a soros változattal azonos módon, változatlanul kerül át a párhuzamos programba, így annak elkészítése nem igényel külön párhuzamos programozásra vonatkozó ismereteket.

4. A keletkező program jól skálázható.

5. A terheléskiegyenlítés automatikus.

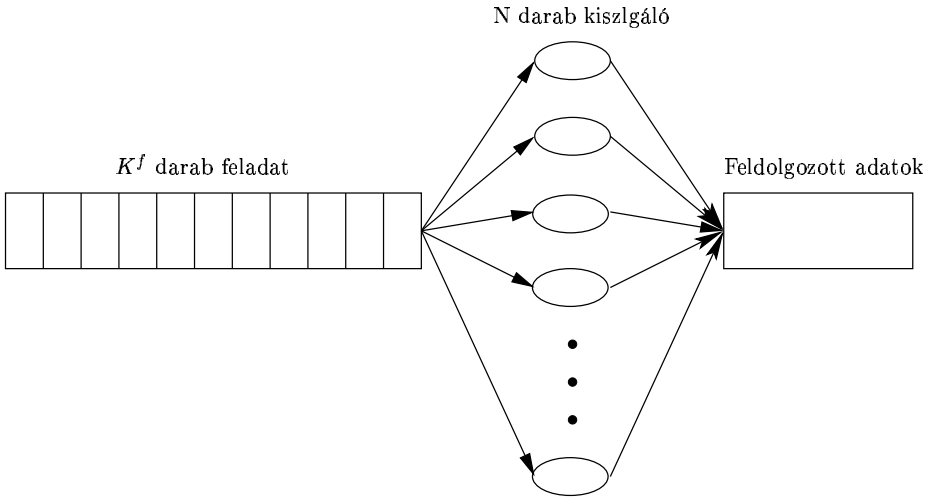
6. Nincs adatcsere a párhuzamos részek között.

7. Gyakorlati megvalósítási szempontból nagy GRT esetén a domain dekompozíció amúgy sem lenne elkerülhető (memória korlátok).

8. A fenti előnyökkel szembeni hátrány, hogy az egyes térrészek szélein levő rácspontokban a függvényértékek többször is kiszámítódnak, ami csökkenti a hatékonyságot.

Maga a partícionálási módszer alapvetően egy mester-szolga struktúrát sugall. A mesterprogram feladata a kijelölt GRT kisebb darabokra osztása és azok paramétereinek átadása a szolgák számára. A szolgák a kapott térrész letapogatása után visszaküldik eredményeiket a mesternek, ami újabb térrész számítását kéri a szabad szolgáktól.

A korábbi elemzésből kiderül, hogy ezzel a feladatmegosztással igen egyszerű az egyes processzorok terheltségének kiegyenlítése, de hátránya a módszernek, hogy túl kis térrészek esetén nagyon hamar felszabadulhatnak a szolgák, így azok esetleg a mesterre várnak újabb feladatért, vagy éppen



2.1. ábra. Választott párhuzamosítási módszer adatfolyam modellje

az eredmények átvételéért. Mint említettük, a feladat jellegéből adódóan alapvetően kevés helyen várunk megoldást a kijelölt térrészekben, ezért ez utóbbi nem jelent jelentős korlátot. A GRT sok kis részre vágásából adódó redundanciát egy közös függvényérték tárolóval lehetne csökkenteni, de lazán csatolt párhuzamos rendszer esetén ez nem valósítható meg hatékonyan.

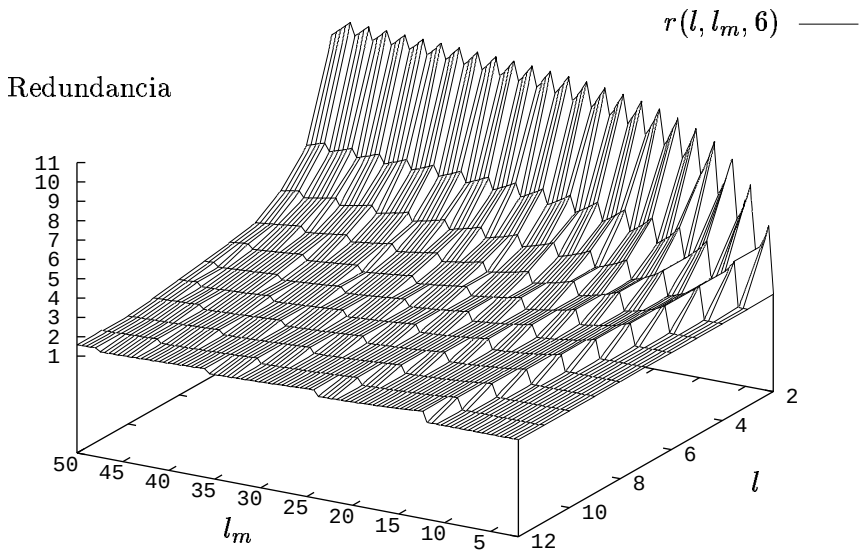
A többszörösen számolt pontok arányának meghatározásához tegyük fel, hogy az n dimenziós GRT minden koordinátáját l részre osztjuk, és a kisebb térrészekre való felosztás során is minden irányban azonos felosztást alkalmazunk. A másodlagos felosztást az elsőhöz hasonlóan egy ortogonális hálóval végezzük. Jelölje l_m a másodlagos rács rácsméretét, azaz azt, hogy az l részre osztott koordinátából hány rész esik egy-egy párhuzamosan számított térrészbe. Ekkor összesen $J(l,n) = (l+1)^n$ elsődleges rácspontunk van, de a másodlagos felosztás miatt egyes pontokban (a térrészek határán) többször is kiszámítjuk a függvényértékeket. Ezért összesen

$$J_m(l, l_m, n) = \left(l + \left\lceil \frac{l}{l_m} \right\rceil\right)^n \tag{2.3}$$

pontban értékeljük ki a függvényeket. Látható, hogy kis másodlagos rácsméret esetén jelentősen megnő a többszörösen kiértékelendő függvények száma. A számítás redundanciája (r) a számított pontok és a tényleges rácspontok arányaként képezhető:

$$r(l, l_m, n) = \frac{J_m(l, l_m, n)}{J(l, n)} = \left(\frac{\left(l + \left\lceil \frac{l}{l_m} \right\rceil\right)}{(l+1)}\right)^n \tag{2.4}$$

Ezen összefüggés alapján megállapítható, hogy kis másodlagos rácsméret esetén kis és nagy felosztás (l) mellett is meredeken megnő a többszörösen számított pontokból adódó redundancia. Nagyobb l_m értékeknél viszont kellően alacsony marad. Ez azt jelenti, hogy az algoritmus megfelelő paraméterezésével a redundancia megfelelő értéken tartható. A 2.2. ábra a felosztás és a másodlagos rácsméret függvényében ábrázolja a redundanciát $n = 6$ dimenziós esetben. A 2.4 összefüggésből az is megállapítható, hogy a dimenziószámtól nem függ a redundancia alakulásának jellege, de kis l_m értékeknél a kifejezés érzékeny a dimenziószám változására.



2.2. ábra. Többszörös számításból adódó redundancia (r) a felosztás (l) és a másodlagos rácsméret (l_m) függvényében ($n = 6$). A jobb láthatóság érdekében az ábrát elforgattuk, így az origó a jobb oldalon a távolabbi pontban van.

A választott párhuzamosítási módszer adatfolyam szinten a 2.1. ábrával szemléltethető. Ezek szerint a teljes feldolgozási folyamat egy olyan sorbanállási modellel modellezhető, mely bemenetén K^f darab a partícionálás során keletkező feladat van. N processzort feltételezve N darab feldolgozó (kiszolgáló) egységünk van. A teljes feldolgozási (áteresztési) idő becsléséhez feltételeztük, hogy $\{l/l_m\} = 0$, azaz a partícionálás maradék nélkül elvégezhető, így az összes $K_0^f = (l/l_m)^n$ darab feladat azonos méretű. Azonos teljesítményű kiszolgáló egységeket feltételezve, valamint figyelembe véve a 2.2 összefüggést, az azonos méretű feladatok kiszolgálási ideje azonosnak vehető. Így maradék nélküli felosztásnál a teljes számítási idő (T_{N_0}) a következő összefüggéssel becsülhető:

$$T_{N_0} = \left\lceil \frac{K_0^f}{N} \right\rceil \cdot T = \left\lceil \frac{\left(\frac{l}{l_m}\right)^n}{N} \right\rceil \cdot (l_m + 1)^n \cdot k \cdot T_f. \tag{2.5}$$

Amennyiben $\{l/l_m\}$ maradék nem nulla, akkor minden dimenzióban eggyel több résszel kell számolnunk, így összesen $K_1^f = \lceil l/l_m \rceil^n$ darab feladat keletkezik. A maradék miatt keletkező feladatok azonban kevesebb, mint $(l_m + 1)^n$ rácspontot tartalmaznak, így számítási idejük is kisebb. Ha azonban a számítási idők csökkenésétől eltekintünk, akkor a tényleges feldolgozási időt biztosan meghaladó T_{N_1} értéket kapunk, ami felső becslésnek felhasználható.

$$T_{N_1} = \left\lceil \frac{K_1^f}{N} \right\rceil \cdot T = \left\lceil \frac{\left\lceil \frac{l}{l_m} \right\rceil^n}{N} \right\rceil \cdot (l_m + 1)^n \cdot k \cdot T_f \tag{2.6}$$

Összesen $n \cdot (n-1)+1$ darab ilyen változó méretű térrészünk van. Ez a darabszám kis l/l_m aránynál

összemérhető az $(l_m + 1)^n$ pontot tartalmazó térrészek számával. Ezért jelentős hibát követhetünk el, ha a maradék térrészek számítási idejét azonosnak vesszük a többi térrész számítási idejével.

Más megközelítéssel is becsülhetjük a teljes feldolgozási időt: A választott párhuzamosítási modell automatikusan biztosítja a terheléskiegyenlítést, ezért a feldolgozó egységek kihasználtságát a legutolsó feldolgozási ciklust leszámítva 100%-osnak tételezhetjük fel.[1] Ezért a működés egy olyan modellel is közelíthető, amelyben nem térrészeket választunk ki, hanem a geometriai jelentéstől elvonatkoztatva csupán ponthalmazokat. Így a kiszolgáló egységek mindig maximális kapacitással tudnak dolgozni, leszámítva a legutolsó kiszolgálási ciklust. A kiszolgáló egységek maximális kapacitása $(l_m + 1)^n$ pont. Így ebben a modellben $K^f = \lceil J_m/J \rceil$ feladatot kell a kiszolgálóknak kiszolgálni, ahol J a rácspontok, J_m pedig a 2.3 összefüggés szerint a ténylegesen számított pontok számát jelenti. A kiszolgáló egységek számának függvényében a teljes feladat kiszolgálási idejére a:

$$T_N = \left\lceil \frac{K^f}{N} \right\rceil \cdot T = \left\lceil \frac{\left\lceil \frac{\left(l + \left\lceil \frac{l}{l_m} \right\rceil\right)^n}{(l_m+1)^n} \right\rceil}{N} \right\rceil \cdot (l_m + 1)^n \cdot k \cdot T_f \tag{2.7}$$

érték adódik. Belátható, hogy $T_{N_0} = T_{N_1} = T_N$, ha $\{l/l_m\} = 0$ azaz, ha maradék nélkül elvégezhető partícionálás, akkor a fenti közelítések azonos értéket adnak. A 2.5 és a 2.6 összefüggés esetén ez kézenfekvő. A 2.7 összefüggés esetében pedig K^f egyszerű átalakításával igazolható:

$$K^f = \frac{\left(l + \left\lceil \frac{l}{l_m} \right\rceil\right)^n}{(l_m + 1)^n} = \left(\frac{l + \frac{l}{l_m}}{l_m + 1}\right)^n = \left(\frac{\frac{l \cdot (l_m+1)}{l_m}}{l_m + 1}\right)^n = \left(\frac{l}{l_m}\right)^n = K_0^f$$

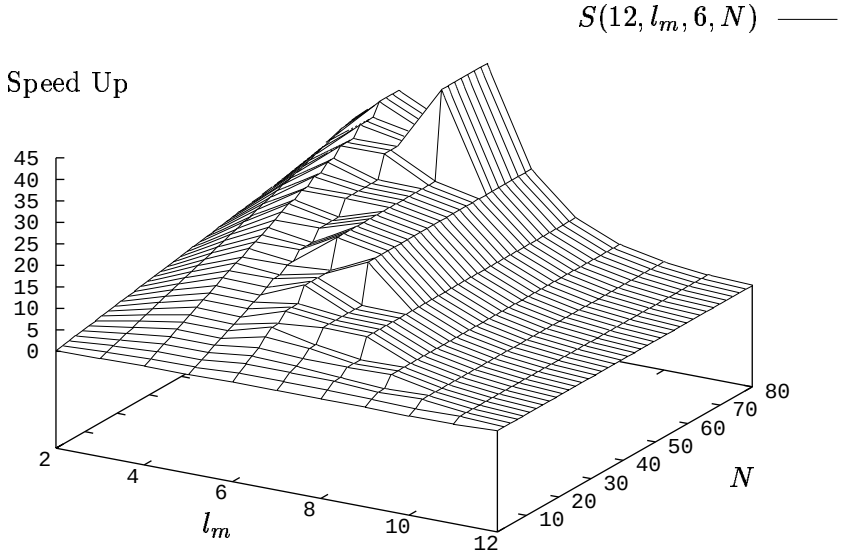
A felhasznált modellből adódóan $T_{N_1} \geq T_N$, amit számításokkal is igazoltunk. Megjegyezzük, hogy a 2.7 összefüggés nem veszi figyelembe, azt az esetet, amikor a legutolsó feldolgozási ciklusban csak egyetlen egy térrészt kell feldolgozni, és ez a térrész kevesebb pontot tartalmaz, mint a többi. Ez azonban lényegesen kisebb hibát okoz, mint a a 2.6 összefüggés hibája, mivel itt csak egy ilyen feldolgozási ciklus lehet és csak akkor, ha $\{|J_m/J|/N\} = 0$, és $\{J_m/J\} \neq 0$. A továbbiakban ezt a hibát elhanyagoljuk.

A feldolgozási idő fenti közelítő értékét felhasználhatjuk az 1.2.2. pontban bevezetett sebességnövekedés becslésére is:

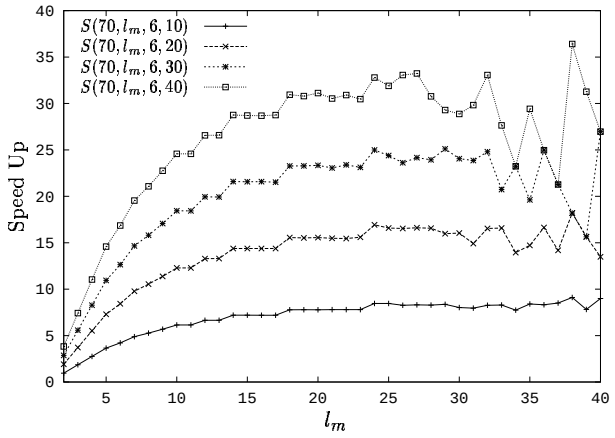
$$S_N = \frac{T_S}{T_N} = \frac{(l+1)^n}{\left\lceil \frac{\left\lceil \frac{\left(l + \left\lceil \frac{l}{l_m} \right\rceil\right)^n}{(l_m+1)^n} \right\rceil}{N} \right\rceil \cdot (l_m + 1)^n} = \frac{1}{\left\lceil \frac{\left\lceil \left(\frac{l + \left\lceil \frac{l}{l_m} \right\rceil}{l_m+1}\right)^n \right\rceil}{N} \right\rceil} \cdot \left(\frac{l+1}{l_m+1}\right)^n \tag{2.8}$$

A 2.8 összefüggés alapján igazolható, hogy $\lim_{l_m \to l} S_N = 1$, amint azt az adatfolyam modell alapján várjuk. Látható továbbá, hogy a várható sebességnövekedés görbéje l_m függvényében igen meredeken indul, majd N értékétől függően letörik. A 2.3. és a 2.4. ábrák ezt a jelenséget szemléltetik. A 2.4. ábra a valóságos l/l_m arányokhoz jobban hasonlító esetet mutat, így megfigyelhető a meredek felfutó szakasz. A 2.5. ábra az N processzorral elért hatékonyságot (E_N) szemlélteti a másodlagos rácsméret és a processzorszám függvényében. Az ábrán jól látható, hogy a felfutó szakaszban a hatékonyság a processzorszámtól gyakorlatilag nem függ.

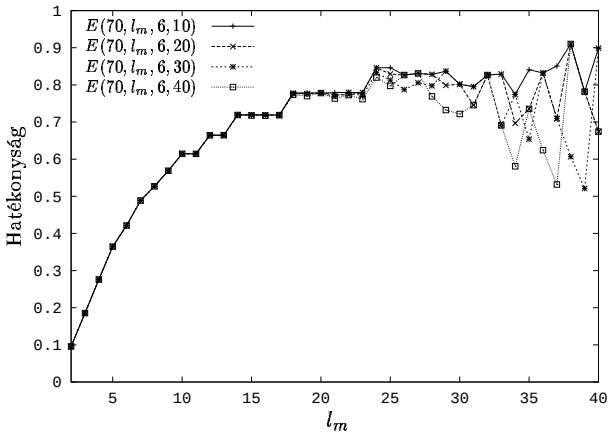
[1] A kisebb feladatokat rövidebb idő alatt dolgozzák fel, de azonnal kapnak egy új feladatot, amíg van feldolgozatlan feladat.



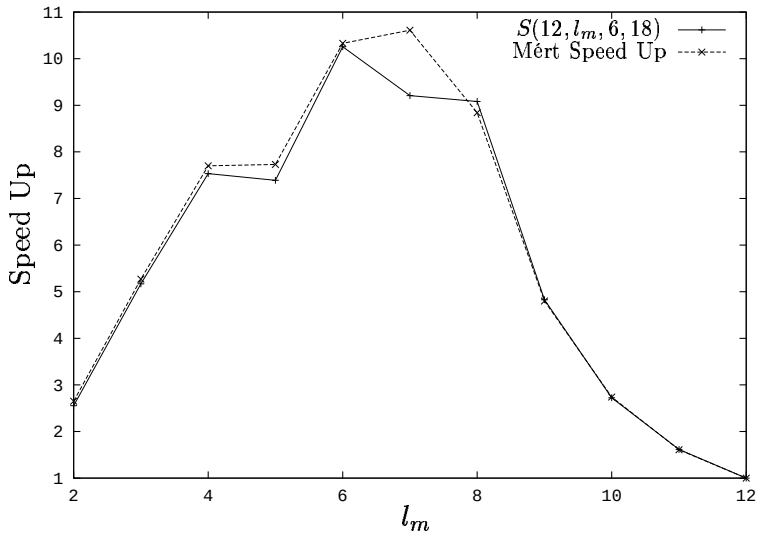
2.3. ábra. Sebességnövekedés változása a másodlagos rácsméret (l_m) és a processzorszám (N) függvényében ($l = 12$, $n = 6$). A kis felosztás miatt jól megfigyelhető a meredek növekedés, majd a letörés.



2.4. ábra. Sebességnövekedés a másodlagos rácsméret (l_m) és a processzorszám (N) függvényében ($l = 70$, $n = 6$).



2.5. ábra. Párhuzamosítás hatékonyságának változása a másodlagos rácsméret (l_m) és a processzorszám (N) függvényében ($l = 70$, $n = 6$).



2.6. ábra. A 2.8 összefüggés alapján számított és a program letapogató változatával mért sebességnövekedés változása a másodlagos rácsméret (l_m) függvényében ($l = 12$, $n = 6$, $N = 18$).

A sebességnövekedés fentiekben becsült összefüggését az elkészített párhuzamos program letapogató változatával végzett méréseink igazolták. A 2.6. ábra egy ilyen mérési eredmény és a 2.8 képlet alapján számított értékek összevetését mutatja. A mérést 18 processzorral 6 dimenziós feladaton végeztük. A mérés során viszonylag kis felosztást (12) alkalmaztunk, a másodlagos rács méretét pedig 2 és 12 között változtattuk. Megállapíthatjuk, hogy lényeges eltérést a két görbe nem mutat, kivéve az $l_m = 7$ pontban. Érdekes, hogy a legnagyobb sebességnövekedést nem az elméletileg optimális $l_m = 6$ értéknél, hanem az $l_m = 7$-nél mértük. Ez azzal magyarázható, hogy a mérés folyamán a processzorok teljesítménye ill. terheltsége nem volt azonos. A teljesen szimmetrikus felosztás viszont,

l_m	Pontok (J_m)	SpUp S_{18}	Slave idők[s] (szumma) real	comm	idle	Master idők[s] real	comm	idle
2	34012224	2.65	118344.36			6575.89		
3	16777216	5.27	59470.06	0.669	119.15	3309.76	0.428	3303.24
4	11390625	7.70	40429.51	0.115	331.76	2264.40	0.104	2263.16
5	11390625	7.73	40414.39	0.130	204.72	2256.45	0.181	2255.08
6	7529536	10.33	26946.06	0.014	3438.64	1687.96	0.114	1687.63
7	7529536	10.61	26956.02	0.017	2609.26	1642.44	0.097	1642.13
8	7529536	8.84	26964.22	0.013	8536.06	1972.20	0.054	1971.91
9	7529536	4.80	26915.09	0.020	38435.29	3630.54	0.062	3630.26
10	7529536	2.74	26870.95	0.014	87703.80	6365.24	0.034	6364.99
11	7529536	1.61	27043.06	0.018	167927.12	10831.59	0.099	10831.28
12	4826809	1.00	17451.62	0.005	296678.46	17451.71	0.019	17451.53

2.3. táblázat. Egy konkrét feladat párhuzamos letapogatásakor mért futási idők és Speed Up (S_{18}) (N=18, l=12). ($l_m = 2$ értéknél a kommunikációs és idle időket mérő alkalmazás (XPVM) hibája miatt nem kaptunk eredményt.)

mint amilyen az $l_m = 6$, nem teszi lehetővé a gépek más jellegű terheléséből, vagy éppen az eltérő teljesítményükből adódó optimális terheléskiegyenlítést.

A mérés egyéb eredményeit a 2.1. táblázat tartalmazza. A táblázatban feltüntettük a mért valós futási időket (real), valamint a kommunikációs és az *idle* időket is. Idle időnek azt az időt vettük, amíg a master vagy a slave azért várakozott, mert nem volt mit tennie. A szolgák esetében a mért időket összesítettük. Az eredményekből kitűnik, hogy a kommunikációs idők kis másodlagos rácsméretnél jelentősen megnőnek, de még így is alacsony értéken maradnak. Ez érthető, mivel a mester és a szolgák között átviendő adatmennyiség nem nagy. A mérés során az XPVM [5] rendszer segítségével minden üzenetet naplóztunk, majd a naplókat utólagosan feldolgoztuk. Itt jegyezzük meg, hogy $l_m < 3$ esetben az XPVM nem tudott kellő sebességgel naplózni, ezért nincs a táblázat első sorában ilyen adat.

2.2. A PSA környezete és programstruktúrája

A 2.1. alfejezetben tett megfontolások alapján GRT szintű párhuzamosítással mester-szolga programstruktúra kialakítása mellett döntöttünk, mivel ez a módszer tűnt a legelőnyösebbnek. A PSA implementálásához és kipróbálásához a PVM rendszert választottuk ki, mint párhuzamos környezetet, mivel a korábban ismertetettek alapján a módszertől lazán csatolt parallel rendszerekben is jó eredményeket reméltünk. A PVM kiválasztásánál fontos szempont volt az egyszerű elérhetősége és az, hogy heterogén hardver környezetben is kiválóan használható, mint azt az 1.2.3. pontban bemutattuk. Ma a PVM mellett számos elterjedten használt hasonló rendszer létezik, mint pl. az MPI, Harness, Condor stb., de 1994-ben, amikor az algoritmus első párhuzamos implementációja elkészült, szinte a PVM volt az egyetlen olyan rendszer, amely egyszerű eszközökkel lazán csatolt rendszereken párhuzamos feldolgozást lehetővé tett. Megjegyezzük azonban, hogy a jelenlegi fejlődési trendek szükségessé tették a program MPI változatának elkészítését is. Ez a változat az MPI-2 előírásait is figyelembe vevő MPICH [32] implementációra épül. A legutóbbi fejlesztések során pedig elkészült a program Condor rendszerrel együttműködő változata is [51].

A környezet kiválasztását követően első lépésként meghatároztuk a mester- és a szolgaprogram közötti feladatmegosztást és kommunikációt. Ezek alapján a mester feladatai:

- A konfigurációs állomány feldolgozása, a szolgaprogramok elindítása.
- A szolgák állapotának (szabad/foglalt/kihalt) nyilvántartása.
- A kijelölt GRT felosztása kisebb térrészekre és azok paramétereinek átadása a szabad szolgák számára.
- A szolgáktól érkező eredmények eltárolása.
- Terhelés-kiegyenlítés (load balancing)
- Adatok tárolása az újraindíthatósághoz (check-point restart).
- Statisztikai információk gyűjtése a szolgáktól.

A szolgák feladatai:

- A mestertől kapott térrészek letapogatása.
- A kapott eredmények visszaküldése a mesternek.
- Statisztikai adatok gyűjtése.

A mester és a szolga közötti viszonylag kis mennyiségű adatátvitellel járó kommunikációt teljes egészében a PVM üzenetrendszerére bíztuk. Egyes paramétereket ill. eredményeket közös fájlrendszer (NFS) segítségével is átadhattunk volna, de az csak megszorításokat jelentett volna a számítógépes környezetre, és bővítette volna a környezettel való kapcsolatot, így ezt a lehetőséget elvetettük.

Minthogy a bemutatott algoritmus peremérték-feladatok általános megoldására alkalmas, ezért a tervezés és az implementáció során törekednünk kellett arra, hogy a feladatfüggő programrészek jól elkülönüljenek a többi programrésztől. Ezt a program moduláris szerkezetével és az egyes modulok közötti jól definiált interfésszel valósítottuk meg. Ezek alapján a megvalósított program 4 fő modulból áll:

- Függvénymodul.
  Ebben kapott helyet a feladatra jellemző 1.7 alakú függvény, mely egy jól definiált interfészen keresztül kapcsolódik a program többi részéhez. Ilyen interfész a függvény független változóinak száma, ill. maga a függvény be ill. kimeneti paraméterei. Az 1.3. fejezetben a peremértékfeladatok bemutatásánál külön nem említettük, de gyakran van arra szükség, hogy a PF megoldását jelentő pontokban egy vagy több, a fizikai feladathoz kapcsolódó további függvény értékét is ki kell számítanunk. Ezért a függvénymodulba opcionálisan beépíthető egy kiegészítő függvény is, mely kiértékelése csak a letapogatás eredményeként kapott megoldáspontokban történik meg. Az így kapott értékek a megoldáspontokkal és a megoldás hibájával (visszahelyettesítés) együtt megjelennek a futási eredményben.

- Konfigurációs állomány beolvasását és értelmezését végző modul.
  Ebben a modulban kapott helyet az az egyszerű elemző, amely a konfigurációs állomány szintaxisát felismerve beolvassa az egyes konfigurációs paramétereket. Ennek különválasztását a funkcionális szétválasztás, ezen keresztül az újrafelhasználhatóság indokolta. A programcsomagot kiegészítő segédprogramok ugyanis (pl. a térrács sűrítését segítő program ld. 3.2. alfejezetben) szintén használják a konfigurációs állomány információit. Így célszerűnek látszott egy jól szeparált önálló modul kialakítása.

- Szolgaprogram (slave)
  Ez a modul lényegében egy önálló PVM program, amely az 1.5. fejezetben ismertetett szimplex algoritmus alapján elvégzi a kapott térrészben a letapogatást. Az algoritmus bemutatásánál láttuk, hogy egy-egy rácspont több szimplex csúcspontjaként is részt vesz a számításokban, így az azokban kiszámított függvényérték többször is felhasználásra kerül. A tervezésnél feltételeztük, hogy az egyes függvényértékek előállítása számításigényes. A hatékonyság növelése érdekében meg kellett oldani, hogy ezeket a függvényértékeket ne számítsuk ki többször feleslegesen. Ennek megoldásához felmerült, hogy valamilyen cache mechanizmussal tároljuk a már kiszámított értékeket. Így elérhető, hogy a már kiszámított és a cache-ben még bent levő értékeket ne számítsuk újra. A kiszámítandó függvény bonyolultságáról azonban előre nem túl sokat tudunk, felmerül az a veszély, hogy esetleg egy nagyon egyszerű függvény esetében egy bonyolultabb cache használata csak ront a hatékonyságon. Ezért a megvalósított programban azt az igen egyszerű módszert választottuk, hogy a szolgaprogram a térrész számításának megkezdésekor a térrészbe eső összes rácsponton kiszámítja a függvényt, és a kapott függvényértékeket eltárolja. Ezután kezdi meg valójában a letapogatást úgy, hogy a függvények értékeit ebből a tárból veszi ki. A tároláshoz természetesen magas dimenziószám és sok rácspont esetén nagy mennyiségű memóriára van szükség, de a szükséges tároló terület a tér nagyságából és a rácspontok sűrűségéből pontosan kiszámítható, így a probléma konfigurációs szinten jól kézben tartható. Ez azt jelenti, hogy a konfigurációs állományban megadható, hogy maximum hány pontot képes a szolga hatékonyan[2] tárolni, így a mester ennek megfelelő nagyságú térrészt fog átadni a szolgának.

  Megjegyezzük, hogy a 3. fejezetben bemutatott algoritmusok később lehetővé ill. szükségessé tették, hogy ne számítsuk ki minden rácspontban a függvényeket, ezért az itt leírt megvalósítás némileg módosult.

- Mesterprogram (master).
  Ez a modul lényegében egy önálló PVM program, amely a korábban ismertetett mester funkciókat látja el. A mester a konfigurációs paramétereknek megfelelően bontja szét a vizsgálandó

[2] A befogadó operációs rendszer képes virtuális tárkezelésre, de ennek használata hatékonyság csökkenéshez vezet.

GRT-t kisebb térrészekre. A térrészek csatlakozó határfelületén a függvények többszörösen is kiszámítódnak, ami redundanciához vezet (ld.2.1. alfejezet). Annak érdekében, hogy csökkentsük a térrészek határára eső rácspontokon fellépő redundanciát a mesterprogram megpróbálja a vizsgálandó teret úgy felosztani, hogy az minél kevesebb olyan rácspontot tartalmazzon, amely határfelületre esik. Ezt úgy éri el, hogy olyan térrészekre darabol, amelyekben minden koordináta-irányban azonos vagy közel azonos számú pont van, hiszen az ilyen kockához hasonló térrészek felülete lesz a legkisebb. Ez az algoritmus természetesen kikapcsolható. Ilyenkor a mesterprogram a konfigurációs állományból olvassa be a másodlagos rács rácsméretét.

Minden párhuzamos rendszerben, de a lazán csatolt rendszerekben különösen számolni kell azzal, hogy menet közben valamelyik feldolgozó elem kiesik. Ezért fel kell készíteni a programot az ilyen események kezelésére, azaz meg kell oldani az újraindíthatóságot. Az újraindíthatóság egyik fontos kérdése, hogy melyik az a pont, ahonnan a program képes újraindulni, ill. az elvégzett számítások eredményei hogyan használhatók fel. A PSA program tervezésekor úgy döntöttünk, egy esetleges újraindításkor a korábban megkezdett, de teljesen be nem fejezett térrészek számítását teljesen megismételjük. Ezzel lényegesen egyszerűsödik a szükséges adminisztráció, hiszen a mesterprogram az újraindíthatóságtól függetlenül tárolja a szükséges adatokat. A megoldás hátránya – amellett, hogy esetleg feleslegesen számolunk újra egy térrészt – az, hogy elképzelhetők olyan eredmények, amelyek többször előállnak, és így többször szerepelnek az eredményeket tároló állományban. Ez azonban az eredmények feldolgozásakor kiszűrhető. Az eredmények kiértékeléséhez amúgy is kell szűréseket alkalmazni, mert a pontosság által megkövetelt finom rács miatt igen sok adat keletkezik.

A fentiek alapján a mester olyan módon tartja nyilván a megkezdett, de még be nem fejezett térrészek paramétereit, hogy az egy szolga esetleges kiesésekor vagy akár a teljes program leállásakor a megkezdett térrészek számítása megismételhető legyen. Az újraindíthatóságot biztosító adatokat a mester rendszeresen fájlba írja, így a mester vagy az azt futtató gép leállása esetén is újraindítható a számítás.

A mester kialakítása olyan, hogy a szolgák paramétereit, állapotát dinamikus adatterületen tárolja, így a szolgák száma előre nem korlátozott.

2.3. A szolgaprogram részletes működése

A megvalósított rendszer központi eleme a szolga, mivel a lényegi számításokat ez a programrész végzi. A szolga implementálásakor külön figyelmet kellett fordítani a hatékonyságra és a feladatfüggetlenségre. Azaz a szolga csak az előírt interfészeken kapcsolódhat függvénymodulban realizált feladatfüggő részekhez. Fontos szempontként merült fel, hogy a szolgában csak olyan algoritmusokat és adatábrázolási módszereket lehet alkalmazni, melyek nem korlátozzák a függvénymodulban realizált függvények független változóinak számát. Ez azzal jár, hogy a feladattól függően más-más dimenziószámú adatokkal kell műveleteket (linearizálás, egyenletmegoldás, stb.) végezni.

Amint azt az előző pontban bemutattuk, a szolga a mestertől kapott adatok alapján első lépésben kiszámítja a kapott térrész minden rácspontjában a függvényértékeket, és azt letárolja. Ezt követően sorban végigmegy a rácspontok által kijelölt n dimenziós téglatesteken, és elvégzi a szimplexekre bontást, majd a szimplexekben az előírt számításokat.

2.3.1. A szolgaprogram felépítése

A megvalósított program funkcionálisan több alprogramra osztható:

Vezérlő programrész

Ez a programrész fogadja a PVM üzenetküldő mechanizmusán keresztül mesterprogram adatait ill. parancsait. Ezek a következők lehetnek:

- térrész paraméterek,
- számítás indítás,
- program leállítás,
- állapotkérés

Kockákra bontó programrész

Ez a programrész járja be a teret, és választja ki következő kockát a kapott paraméterek (felosztás) alapján, melyet átad az *egy kocka számítását* végző programrésznek. Az 1.5. alpont rámutatott, hogy egyes algoritmusok lényegesen egyszerűsödnek, ha a tér koordinátái helyett rácskoordinátákkal számolunk. Ezért a programban a tér koordinátáit nem valós koordinátákkal tároljuk, hanem a teret lefedő rács rácskoordinátaival. Így ezek az értékek integer típusban tárolhatók, ami az egyszerűsítés mellett gyorsabb műveletvégzést is biztosít. Ez azt jelenti, hogy a rácspontok, valamint az eredménypontok valós koordinátáit egy lineáris transzformációval kapjuk meg. Cserébe viszont az egységnyi méretű n dimenziós kockák, ill. szimplexek a számításokat nagymértékben egyszerűsítik. Így pl. egyszerűsödik a linearizált függvények, valamint a szimplex oldalfüggvényeinek előállítása is, továbbá könnyebben vizsgálható, hogy egy eredmény a szimplexben van-e (ld. 2.9 egyenlőtlenséget).

A futás gyorsítása érdekében egy kockában csak akkor kezdődik meg a számítás, ha ott várható megoldás, azaz a kocka csúcsain kiszámított összes függvényérték előjelet vált. Ebben az esetben az egy kocka számítását végző programrésznek úgy adjuk át a kockát számításra, hogy annak egyik csúcsát az origóba transzformáljuk. Ez tovább egyszerűsíti a feladat konkrét megoldását.

Egy kocka számítását végző programrész

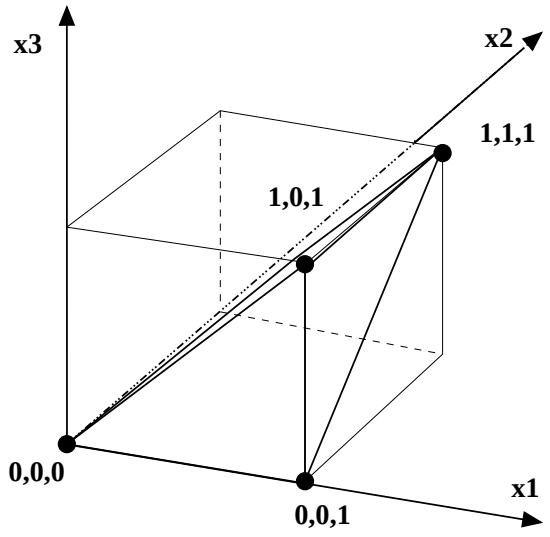
Ez a programrész a az 1.5. alfejezetben ismertetett módon szimplexekre osztja a kapott n dimenziós kockát, és meghívja az *egy szimplex számítását* végző programrészt, ha a szimplexben várható megoldás. A szimplexekre bontás egy egyszerű rekurzív permutációs algoritmussal történik, mely a koordináták megfelelő változtatásával előállítja az összes, a kockát kitöltő szimplexet. Ez úgy történik, hogy a kocka origóba helyezett csúcsából, a kocka élein haladva minden lehetséges úton megpróbálunk eljutni a szemközti csúcsba. Ez pontosan $n!$ különböző utat jelent, így az érintett csúcspontok $n!$ különböző szimplexet jelölnek ki. A további számításokhoz bevezetjük a bejárási sorrend fogalmát, amely megadja, hogy az origóból elindulva mely (úton) koordináták változtatásával haladhatunk végig. A 2.7. ábra egy 3 dimenziós egységoldalú kocka leírt módszerrel történő szimplexekre bontását mutatja be. A keletkező 6 szimplex csúcsait és az egyes szimplexek bejárási sorrendjét a 2.4. táblázatban foglaltuk össze.

Egy szimplex számítását végző programrész

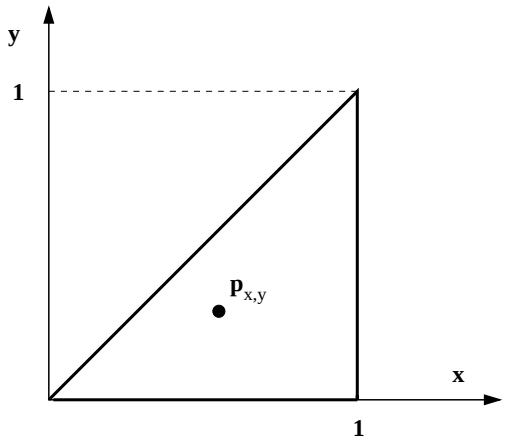
Első lépésben megtörténik a linearizált függvény együtthatóinak előállítása. A szimplex módszer ismertetésénél (1.5. alfejezet) láttuk, hogy a megoldáshoz szükség van szimplex oldalait leíró

Csúcsok koordinátái	Bejárási sorrend
000,001,011,111	1, 2, 3
000,001,101,111	1, 3, 2
000,010,011,111	2, 1, 3
000,010,110,111	2, 3, 1
000,100,101,111	3, 1, 2
000,100,110,111	3, 2, 1

2.4. táblázat. Bejárási sorrend 3 dimenzióban



2.7. ábra. 3 dimenziós kocka szimplexekre bontása



2.8. ábra. Pont helyzetének felismerése 2 dimenziós szimplexben

egyenletekre is. A megvalósítás egyszerűsítése érdekében nem ezt az egyenletet, hanem a szimplex oldallapját magába foglaló sík egyenletét állítjuk elő. Ezt a továbbiakban oldalfüggvénynek nevezzük. A linearizáláshoz és az oldalfüggvények előállításához felhasználjuk, hogy a szimplex bejárási sorrendje ismert, valamint azt, hogy a szimplex oldalai egységnyi hosszúságúak. Az így előállított $n-1$ egyenlethez és $n+1$ oldalfüggvényhez tartozó együtthatók képezik a Gauss-eliminációval működő lineáris egyenletrendszer-megoldó bemenetét. A Gauss-elimináció algoritmusát főelemkiválasztással kombinálva valósítottuk meg. Az algoritmus implementálásánál figyelembe vettük, hogy a linearizálás során nyert $n-1$ darab egyenletet minden oldalfüggvénnyel meg kell oldani, így az elimináció részeredményeit a következő oldalfüggvénynél újra fel tudjuk használni. Ez némileg gyorsít az algoritmuson, ugyanakkor hátrányként jelentkezik, hogy így a főelemkiválasztás nem teljes, mivel az oldalfüggvények nem vehetnek abban részt.

Mivel az oldalfüggvények a szimplex oldalát alkotó $n-1$ dimenziós hipersíkokat[3] jelentenek, és nem csupán a szimplex oldalát, így az egyenletek megoldása során kapott megoldáspontokról el kell döntenünk, hogy azok valóban a szimplex oldalára esnek-e. Ennek eldöntéséhez azt vizsgáljuk, hogy a kapott pont a szimplexbe esik-e, ahol a szimplex oldalait a szimplex részének tekintjük. A vizsgálat igen egyszerű, ha feltételezzük, hogy a szimplex a korábban ismertetett algoritmussal keletkezett és egységoldalú. Legyenek a vizsgált p pont koordinátái $p_1, p_2, \cdots, p_n$, mégpedig a szimplex bejárási sorrendjében. Ebben az esetben a p pont akkor esik a szimplexbe, ha bármely egymást követő koordinátájára igaz a következő egyenlőtlenség:

$$1 \geq p_i \geq p_{i+1} \geq 0, i = 1..n-1 \tag{2.9}$$

Példaként vegyük a 2.8. ábrát. Ez egy 3 dimenziós probléma 2 dimenziós szimplex-oldalát ábrázolja (a bejárási sorrend x, y). Ezen egyszerű esetben könnyen belátható, hogy a p pont csak akkor esik a szimplexbe, ha teljesül az alábbi egyenlőtlenség:

$$1 \geq p_x \geq p_y \geq 0$$

Ha az egyenletek megoldásával pontosan két megoldáspontot kapunk egy szimplexben, akkor az 1.5 pontban ismertetettek alapján az a PF megoldásgörbéjének egy szakaszát jelöli ki. A megoldás-

[3]Két dimenzióban ezek egyenesek.

pontoknak az eredeti koordinátatérbe való visszatranszformálása után az egyenletbe való visszahelyettesítéssel meghatározzuk a megoldás hibáját. Ha ez a kívánt érték alatt marad, akkor a nyert pontok bekerülnek abba a tárolóba, melyet a szolga majd visszaküld a mesternek, mint eredményt. Ezt az eredményt a szolga kiegészíti a visszahelyettesítésből származó függvényértékkel, mint hibával és a kiegészítő függvények adott pontban vett helyettesítési értékével. Az eredménybuffer akkor ürül, ha az betelik, vagy a kijelölt térrészben befejeződött a számítás.

2.3.2. A szolgaprogram algoritmusa

Az 1.1. algoritmus megfelelő módosításával egyszerűen előállítható a szolgaprogram algoritmusleírása (2.1.). A leírásból látható, hogy szolga az adott térrész számítása közben nem vagy minimális mértékben kommunikál a mesterrel. A működés leírásánál is érzékelhető, de a konkrét implementáció is megerősítette, hogy a párhuzamosítás miatt szinte alig kerültek be új feladatok a programba. (A tér paramétereinek beolvasását és az eredmények visszaadását vagy letárolását a program soros változatában is meg kell oldani.)

A 2.1. algoritmus-leírásban 2 ill. 3 címkével ellátott sorok látszólagosan felesleges adatmozgatást végeznek, hiszen az 1 jelű sorban már minden függvényérték előállt. A feleslegesnek tűnő lépést azonban több szempont is indokolja:

1. Információ rejtés. Ez azt jelenti, hogy nem célszerű, ha a szimplex letapogatást ténylegesen végző programrészek olyan információt is tudnak, ami nem szükséges a működésükhöz.

2. A legtöbb mai számítógép hatékony hardver gyorsítótárral (cache) rendelkezik. Ezek működése a leggyakoribb megvalósításban olyan, hogy a memóriában közvetlenül egymás mellett levő adatokra hatékonyabban működnek, mint az egymástól távol levőkre. A fenti algoritmus a jelzett utasításokkal biztosítja, hogy a később többször is használt adatok egymás mellé kerüljenek a tárban.

3. A 3. fejezetben bemutatott algoritmusok miatt az 1 jelű utasítást a későbbiekben elhagytuk. Így a 2 jelű utasítás feladata lett, hogy a még ki nem számított függvényértéket előállítsa, és azt a megfelelően nagyra választott tárolóba letárolja.

Összességében elmondható, hogy a választott partícionálással olyan rendszerhez jutottunk, melyben a nem párhuzamosítható rész aránya nagyon kicsi. A konkrét érték meghatározása nem egyszerű, mert nagyban függ a PF jellegétől, de függ a megoldás során alkalmazott felosztástól ill. integrálási pontosságtól is (ld. a 2.1. táblázat mérési adatait). A 2.1. alfejezetben ismertetett vizsgálatok alapján leszögezhetjük, hogy PF bonyolultságából ill. számítási pontosság növeléséből adódó számítási igény növekedése csak a párhuzamosítható rész arányát növeli. Ez azt jelenti, hogy a GRT partícionálása alapján párhuzamosított alkalmazás elméleti sebességnövekedési korlátja a PF bonyolultságával és a megkívánt pontosság növelésével nő.

Gyakorlati tapasztalataink és mérési eredményeink azt mutatják, hogy a nem párhuzamosítható rész számítási igénye olyan kicsi, hogy a mért sebességnövekedés tág processzorszám határok között is lineáris. Ezt támasztja alá a 2.5. táblázatban bemutatott mérési eredmény is, melynek során egy feladatot 30, 60, 120 processzorral futtattunk egy SP2 típusú számítógépen. A kapott eredmények még 120 processzor esetén is közel lineáris sebességnövekedésről tanúskodnak.

Miután az 1.5. pontban bemutatott szimplex módszerhez nem készült soros algoritmussal működő program. Ezért mérésünk eredményeként klasszikus értelemben vett sebességnövekedésről nem

```
   while nem jön megállási parancs a mestertől do
   |  Térrész paraméterek átvétele a mestertől.
1  |  Térrész minden függvényértékének kiszámítása.
   |  while van még feldolgozandó kocka do
2  |  |  Következő kocka csúcsaihoz tartozó függvényértékek elővétele.
   |  |  if előjelet váltanak a függvények a csúcsokon then
   |  |  |  while van még feldolgozandó szimplex do
   |  |  |  |  Következő szimplex előállítása.
3  |  |  |  |  A szimplex csúcsaihoz tartozó függvényértékek elővétele.
   |  |  |  |  if előjelet váltanak a függvények a csúcsokon then
   |  |  |  |  |  Linearizált függvény előállítása.
   |  |  |  |  |  Részleges főelem kiválasztás
   |  |  |  |  |  while van meg oldalfüggvény do
   |  |  |  |  |  |  Következő oldalfüggvény előállítása.
   |  |  |  |  |  |  Egyenletrendszer megoldása.
   |  |  |  |  |  |  if a megoldás rajta van az oldalon then
   |  |  |  |  |  |  |  Eredmény ideiglenes letárolása.
   |  |  |  |  |  end
   |  |  |  |  |  if pontosan 2 döféspont van és az eredmények elég pontosak then
   |  |  |  |  |  |  Lineáris transzformáció a valós térbe.
   |  |  |  |  |  |  Kiegészítő függvények kiszámítása
   |  |  |  |  |  |  Eredmények letárolása az adópufferbe.
   |  |  |  |  |  |  if az adóbuffer megtelt then
   |  |  |  |  |  |  |  Eredmények küldése a mesternek
   |  |  |  |  |  end
   |  |  |  |  |  Ideiglenes eredménytár törlése.
   |  |  |  |  end
   |  |  |  end
   |  |  end
   |  end
   |  Eredmények közlése a mesterrel
   end
```

2.1. algoritmus: A szolgaprogram egyszerűsített algoritmusa

Processzor-	idő[s]		Térrészek		Párhuzamos		Számított	
szám	User	Real	Min.	Max.	Speed Up	Hatékonyság	Speed Up	Hatékonyság
30	26330	26517	55	56	30.0	1.00	29.1	0.97
60	13131	13235	27	28	60.1	1.00	58.1	0.97
120	6617	6663	13	14	119.3	0.99	112.1	0.93

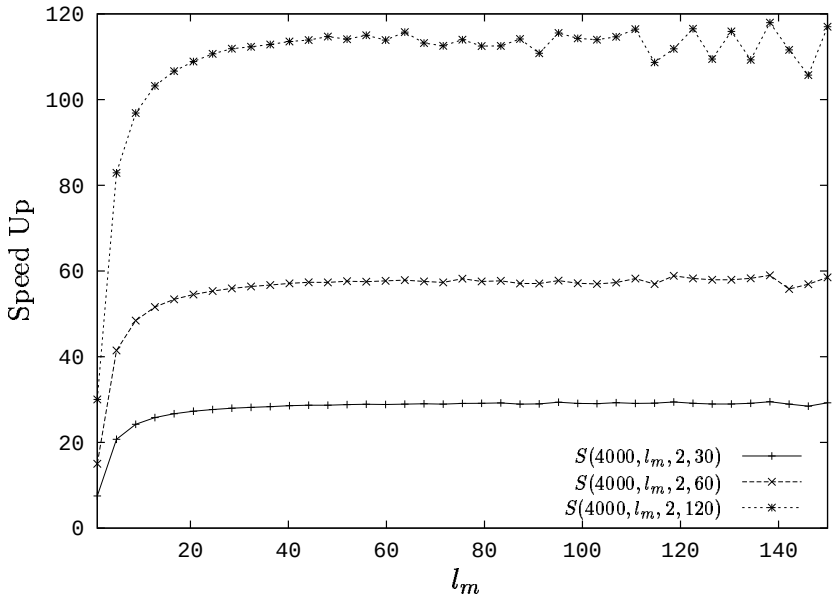
2.5. táblázat. Sebességnövekedés

tudunk beszélni, csupán olyan relatív értékekről, melyek a párhuzamos változat futási idejéhez viszonyítanak. Ezért a továbbiakban a T_S időt a T_1 idővel helyettesítjük, ami az egy processzorral működő, vagy egy processzorra vetített párhuzamos program futási ideje. A 2.5. táblázatban ezt a viszonyítási alapot a 30 processzorral mért érték 30-szorosa adta. (Ettől kevesebb processzorral nem volt módunk futtatni az adott gépen, mert a maximális időkorlát 12 óra volt.) Az így számított sebességnövekedést ezért *párhuzamos sebességnövekedésnek* nevezzük.

Fontos megjegyezni, hogy a partícionálás miatt többszörösen számított rácspontok hatékonyságrontó hatása az így számított sebességnövekedésben nem jelenik meg, hiszen a viszonyítási alapként választott mérésben is pontosan annyi pontot számolunk többszörösen, mint az N processzoros esetben. Annak ellenére, hogy látszólag az így kapott eredmény hamis, mégis fontos, mert:

- A párhuzamos megvalósításban alkalmazott algoritmusok hatékonyságát jól méri.
- A gyakorlatban előforduló feladatok felosztása (l), és dimenziószáma (n) olyan, hogy egyszerre memóriakorlátok miatt nem célszerű[4] letapogatni.

A 2.5. táblázatban bemutatott méréshez kapcsolódó $n = 2$ dimenziós feladat felosztását $l = 4000$-re, a másodlagos rácsméretét pedig $l_m = 100$-re választottuk. Így a többszörösen számolt pontokból származó redundancia a 2.4. összefüggés alapján $r = 1.019$, ami kellően kicsi érték ahhoz, hogy a 2.8 összefüggés alapján számított sebességnövekedés közel legyen a párhuzamos sebességnövekedéshez. (A táblázat utolsó két oszlopában a számított adatokat is feltüntettük.) A 2.9. ábra a 2.8. összefüggés felhaszálásával számolt sebességnövekedést ábrázolja. Az árbáról leolvasható, hogy a választott másodlagos rácsméret ($l_m = 100$) a görbék ellaposodó szakaszára esik. 120 processzor esetén ez már beleesik a letörési szakaszba, amit a mért eredmény is alátámaszt.



2.9. ábra. Sebességnövekedés a másodlagos rácsméret (l_m) függvényében ($l = 4000$, $n = 2$).

[4]Virtuális memória használata nagyon megnövelhetné a futási időt.

3. fejezet

A szimplex algoritmus általánosításai

Jelen fejezet a bemutatott PSA algoritmus feldolgozási teljesítményének növelésével ill. az algoritmus alkalmazási területének kiterjesztésével foglalkozik. Ehhez a fejezethez az alábbi tézisek kapcsolódnak:

3. Tézis. *A Szimplex Módszer letapogató változatához (PSA) gyorsító algoritmust készítettem, amely az n-dimenziós térrács lokális sűrítésén alapszik (FSA). Valós és teszt-feladatokon végzett mérésekkel igazoltam, hogy az eredeti, gyorsítás nélküli módszerhez képest – a feladat jellegétől és dimenziószámától függően – 2-150-szeres sebességnövekedés érhető el.*

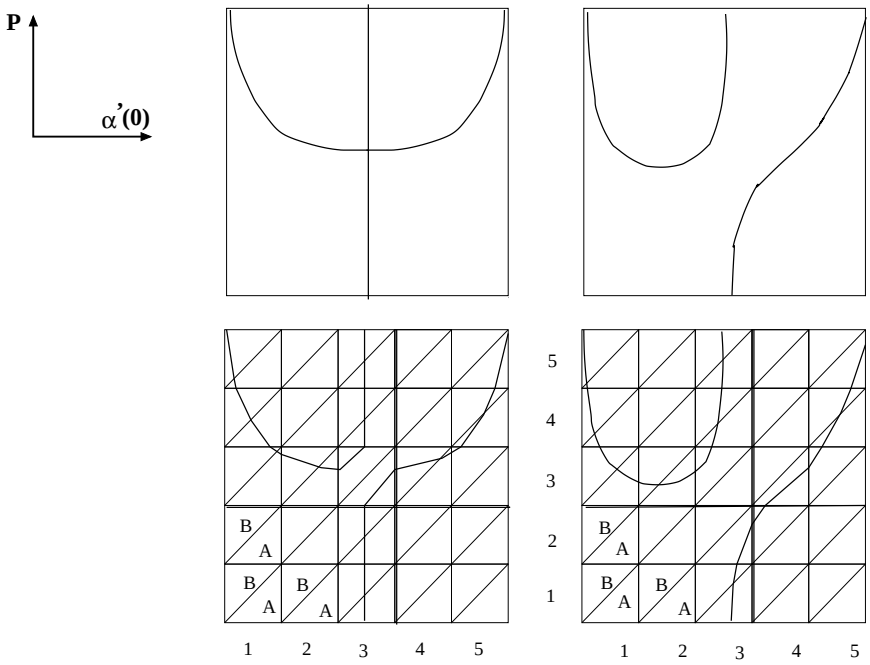
4. Tézis. *Kidolgoztam a Szimplex Módszer párhuzamos útkövető (PUA), véletlenszerű letapogató (VSA) és hibrid (HSA) változatait. Ezeket a korábban kidolgozott PSA általánosításaként fogalmaztam meg, és közös, paraméterezhető algoritmusként implementáltam. A módszert úgy egészítettem ki, hogy az szakadásos függvényekre is alkalmazható legyen.*

5. Tézis. *Kidolgoztam a Szimplex Módszer általánosított, többparaméteres feladatok megoldására is alkalmas változatát (ilyen feladat pl. a stabilitási határgörbék számítása), melyet egy magasabb dimenziószámú feladat megoldására vezettem vissza. Az új, magasabb dimenziószámú feladat megoldását az általánosított algoritmus rekurzív kiterjesztéseként fogalmaztam meg.*

A különböző módszerek feldolgozási teljesítményének összehasonlíthatósága végett az algoritmusok futási idejét különböző mérőszámokkal jellemeztük. Vizsgálataink során legjellemzőbb értéknek az adott feladatban kiszámítandó KF értékek száma adódott, pontosabban az, hogy hány pontban kell kiszámítani az 1.7 alakú, feladatspecifikus függvényt. Erre a következők figyelembe vételével jutottunk:

1. A számítások során a KF megoldása a legidőigényesebb feladat, amint ezt a 2.1 alfejezetben kifejtettük és mérésekkel is igazoltuk. (Megállapításunkat az 5. fejezetben bemutatott alkalmazásokkal szerzett tapasztalataink is megerősítették.)

2. Az egyes módszerek közel azonos számú, a KF kiszámításához képest kevésbé időigényes egyéb számítást (linearizálás, lineáris egyenletrendszer megoldása, stb.) végeznek. Ez abból a feltevésből következik, hogy a 3.1. alpontban leírt módszert leszámítva, minden módszer azonos számú szakaszt talál meg a görbéből. Ugyanakkor a 2.3.2. alpontban leírtak alapján csak azokban a szimplexekben történik meg a linearizálás ill. egyenletrendszer megoldás, ahol várható megoldás.



3.1. ábra. A rúd egyensúlyi útja $Q = 0$ és $Q = \epsilon$ esetben és az azokat lefedő szimplektikus háló

Így a további példákban a futási idő egységének egy függvényérték kiszámítási idejét tekintjük. Példaként vizsgáljuk meg az 1.3. ábrán bemutatott feladatot $Q = 0$ ill. $Q = \epsilon$ ($P \gg \epsilon$) erők mellett. A 3.1. ábra[1] felső részén a két esethez tartozó egyensúlyi út, míg az alsó részen azok szimplex módszerrel történő előállítása látható 5x5-ös rácsméret mellett. Az ábrán a rácsozat celláit sor, oszlop koordinátájukkal azonosítjuk, míg az azokban keletkezett szimplexeket A ill. B betűvel betűzzük. Így pl. az 1. sor 2. rácsának alsó szimplexére *12A* néven tudunk hivatkozni.

Tételezzük fel, hogy a fenti 5x5-ös ráccsal lefedett teret 4 szolga tapogatja le a 2. fejezetben leírt algoritmus alapján. Tételezzük fel továbbá, hogy a másodlagos rács mérete $l_m = 3$ (minden 3. pont válik másodlagos rácsponttá), így a szolgák közül egy 3x3-as teret, a többi pedig ennél kisebb teret kap. (A terek határát vastag vonallal jelöltük a 3.1. ábrán.) A 3.1. táblázat segítségével jól nyomon követhető a PSA algoritmus működési folyamata. A táblázatba azon cellák azonosítóit írtuk, melyek számítása az adott időpillanatban elkezdődik. Amennyiben a számítás nem fejeződik be egy egység alatt, akkor további időpontokhoz $\rightarrow$ jelet tettünk. Az adott szolga végleges befejeződését pedig $\oplus$ jellel jelöltük. Látható, hogy szolgánként az első cella számításainak elvégzéséhez 4, majd a további cellák számításához 2 ill. 1 időegységre van szükség. Ez abból adódik, hogy az egyszer már kiszámított függvényértéket nem számolja ki újra a szolga. A teljes számítás a 3. szolga befejeződésével, azaz 16 időegység múlva ér véget.

Megjegyezzük, hogy a 2.3.2. alpontban leírt megvalósítás szerint a szolgaprogram minden függvényértéket kiszámít és letárol, majd csak ezután kezdi meg a linearizálást, ill. egyenletmegoldást (ld. 2.1. algoritmust). Ez a működés az eredeti algoritmus futási idejét nem befolyásolja, hiszen előbb vagy utóbb minden függvényértéket ki kell számolni. Később azonban olyan algoritmus-változatokat is teszteltünk, melyek lényege az, hogy nem számítunk ki minden függvényértéket (ld. útkövetés 3.3.

[1] A baloldali ábrán jól megfigyelhető linearizálásból adódó kettéválás is, amit az 1.5. alfejezet részletesebben ismertet.

	Időegység															
Szolga	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1.	11	→	→	→	12	→	13	→	21	→	22	23	⊕	⊕	⊕	⊕
2.	14	→	→	→	15	→	24	→	25	⊕	⊕	⊕	⊕	⊕	⊕	⊕
3.	31	→	→	→	32	→	33	→	41	→	42	43	51	→	52	53
4.	34	→	→	→	35	→	44	→	45	54	→	55	⊕	⊕	⊕	⊕

3.1. táblázat. PSA feldolgozási folyamata 4 szolgával

alfejezet). Ezért az eredeti algoritmust úgy módosítottuk, hogy a KF függvényt csak akkor értékeljük ki, amikor arra szükség van, majd a kiszámított érték egy tárolóba kerül, így legközelebb már csak onnan kell elővenni.

Az eredmények kvantitatív értékeléséhez az 1.2.2. alpontban ismertetett mennyiségeket képezzük figyelembe véve, hogy a futási időt a fejezet bevezetőjében leírtaknak megfelelően definiáltuk. Definiáljuk továbbá a vizsgált GRT-ben levő megoldásszakaszok számát (L). Ez nem más, mint a tér adott ráccsal való felosztásakor a linearizálás során keletkező szakaszok száma. Ez azért lesz fontos, mert olyan módszereket is bemutatunk, melyek nem feltétlenül találnak meg minden ilyen szakaszt. Bevezetjük továbbá a sűrítési algoritmus (FSA) jellemzőit, melyekhez a részletes magyarázatot a 3.2. alfejezet adja meg. A következő mennyiségeket definiáljuk:

$S_N = \frac{T_1}{T_N}$ - N processzorral elért párhuzamos sebességnövekedés

$S_N^z = \frac{T_N^r}{T_N^z}$ - N processzoros fokozatos sűrítéssel elért sebességnövekedés

$E_N = \frac{S_N}{N}$ - N processzorral elért hatékonyság, ami a párhuzamosítás erőforrás-felhasználására vonatkozik

$E_N^a = \frac{L_N}{K_N}$ - algoritmus hatékonysága N processzorral

$R_N = \frac{L_n}{L}$ - algoritmus megbízhatósága N processzorral

ahol:

N - processzorok száma

T_N - futási idő N processzor esetén

T_N^z - összes futási idő a sűrítési lépésekkel együtt

T_N^r - a sűrítés során keletkező felosztáshoz szükséges futási idő sűrítési algoritmus nélkül

L - linearizálás során keletkező megoldásszakaszok száma

L_N - N processzorral megtalált megoldásszakaszok száma

K_N - N processzorral számított szimplexek száma

Amint már a 2.3.2. alfejezetben utaltunk rá, a PSA programnak nem készült soros változata, így a mérési eredményeken alapuló kvalitatív jellemzők kiszámításánál a párhuzamos program egy processzorral mért, vagy az egy processzorra vetített futási idejével (T_1) számolunk.

A fenti mennyiségek példánkra kiszámított értékeit a 3.2. táblázatban foglaltuk össze. A számításokhoz felhasználtuk, hogy a 3.1. ábra alapján $Q = 0$ esetben $L = 24$, $Q = \epsilon$ esetben pedig $L = 22$ megoldásszakaszunk van. A számításoknál a továbbiakban mindig feltételezzük, hogy 1 processzoros esetben a teljes teret egyetlen processzor kapja meg úgy, hogy ilyenkor nincs alterekre bontás.

A letapogatás módszeréből következik, hogy mindkét feladatnál ($Q = 0$ ill. $Q = \epsilon$) a megbízhatóságra 1.00 értéket kaptunk mind egy, mind négy processzoros esetben. Ez gyakorlatilag azt jelenti, hogy minden olyan megoldást biztosan megtaláltunk, amely legalább egy szimplex oldalát metszi,

Feladat	T_1	K_1	E_1^a	R_1	T_4	E_4^a	R_4	S_4	E_4
$Q = 0$	36	50	0.48	1.00	16	0.48	1.00	2.25	0.56
$Q = \epsilon$	36	50	0.44	1.00	16	0.44	1.00	2.25	0.56

3.2. táblázat. PSA algoritmussal elért jellemzők

azaz csak olyan zárt megoldásgörbéket veszíthettünk el, melyek egy szimplexen belül záródnak. Látható továbbá, hogy a megoldásszakaszok megtalálásának hatásfoka (E_N^a) mindkét feladatra egy és négy processzorral is azonos.

Az így kiszámított sebességnövekedés azonban lényegesen elmarad a 2. fejezetben előrevetített értéktől, illetve a 2.5. táblázatban bemutatottól. Ez abból adódik, hogy az egyszerűség kedvéért nagyon kevés rácsponttal számoltunk, így a másodlagos rácsméret (l_m) is kicsi. Emiatt egy-egy szolga csak néhány pontot tud számolni. Ebből adódóan a domain határokon duplán számolt pontok aránya nagy. A 2.1. alfejezetben bemutatott 2.8 összefüggés segítségével könnyen kiszámolható, hogy pl. 100x100-as rács és $l_m = 50$ értékkel már $S_4 = 3.9$ adódna. Megjegyezzük, hogy az 5. fejezetben bemutatott valós feladatok esetében lényegesen finomabb felosztást kellett alkalmazni a megkövetelt pontosság elérése érdekében.

3.1. Véletlenszerű szimplex letapogatás (VSA)

A letapogató módszer megbízhatósága 100%, de nagy feladatokra a számítás meglehetősen időigényes. A módszer gyorsítására vonatkozóan felvetődött az ötlet, hogy a rács celláit ne egymás után, hanem valamilyen módon véletlen vagy véletlenszerű módszerrel járjuk be. Ennek hatása a következőkben foglalható össze:

- Elképzelhető, hogy a megoldások egy része hamar megjelenik, mivel nagyobb eséllyel találunk meg egy-egy megoldást, mint szisztematikus bejárással.
- Ha meg kívánjuk tartani a 100%-os megbízhatóságot, akkor természetesen minden cellában el kell végezni a számítást, így nem várhatjuk az algoritmus hatékonyságának növekedését.
- A teljes letapogatás befejeződése előtt leállított keresés a megbízhatóságot rontja ugyan, de bizonyos feltételek mellett kompromisszumos megoldást jelenthet.
  Ilyen leállási feltétel lehet pl:
  - Megtalált eredmények grafikus kiértékelése során hozott döntés.
  - Megfelelő számú megtalált eredményszakasz elérése.
  - A letapogatott ill. le nem tapogatott cellák valamilyen statisztikai vagy geometriai jellemzőjének elérése.

A fentiek figyelembe vételével olyan kiválasztási algoritmust dolgoztunk ki, amely a letapogatás teljes befejeződése előtt is képes statisztikailag, ill. geometriailag értelmezhető információt szolgáltatni a le nem tapogatott cellákról. Ehhez minden cellát egy ún. *index* (i) értékkel jellemzünk. Ez az index nem más, mint a cella le nem tapogatott szomszédainak[2] száma, ill. -1, ha a cella már le lett tapogatva.

[2] Akkor tekintünk két cellát szomszédosnak, ha azoknak van közös oldaluk. Könnyen belátható, hogy egy n dimenziós kockának maximálisan $2n$ szomszédja lehet. Természetesen a tér szélein levő celláknak ettől kevesebb szomszédja van.

2	3	3	3	2
3	4	4	4	3
3	4	4	4	3
3	4	4	4	3
2	3	3	3	2

2	3	3	3	2
3	4	4	4	3
3	3	4	4	3
2	-	3	4	3
2	2	3	3	2

2	3	3	2	2
3	4	3	-	2
3	3	4	3	3
2	-	3	4	3
2	2	3	3	2

2	3	3	2	2
3	4	3	-	2
3	3	4	2	3
2	-	2	-	2
2	2	3	2	2

2	2	3	2	2
2	-	2	-	2
3	2	4	2	3
2	-	2	-	2
2	2	3	2	2

2	2	3	2	2
2	-	1	-	2
3	1	-	1	3
2	-	1	-	2
2	2	3	2	2

3.2. ábra. A kiválasztás első 6 lépése 5x5-ös rács mellett.

Jelölje h a szimplektikus hálóval lefedett GRT egy cellájának helyét és $idx[h]$ az ehhez a helyhez tartozó index értékét. Legyen I egy előre megadott érték, amit a továbbiakban leállási feltételnek nevezünk. Ezek felhasználásával a véletlen letapogatás algoritmusa a 3.1. algoritmus-leírással adható meg.

Indextábla feltöltése a szomszédok számával.
repeat
  $i_{max} \leftarrow$ indextáblában található maximális érték
  $h \leftarrow$ a rács egy véletlenszerűen választott helye, ahol $idx[h] = i_{max}$
  $idx[h] \leftarrow -1$
  foreach j, h *minden szomszédjára* **do**
    if $idx[j] > 0$ **then**
      $idx[j] \leftarrow idx[j] - 1$
  end
  h-nak megfelelő cellában elvégezzük a szimplex letapogatást
until $i_{max} < I$

3.1. algoritmus: Véletlenszerű letapogatás (VSA)

Ha I értékét 0-ra választjuk, akkor az algoritmus teljes letapogatást végez. Ettől eltérő értékek mellett azonban I azt adja meg, hogy maximálisan hány le nem tapogatott szomszédja lehet egy cellának. A 3.2. ábra egy 5x5-ös indextáblán mutatja be a kiválasztás első 6 lépését. (Az áttekinthetőség kedvéért az indextábla -1 értéket tartalmazó elemei helyére csak $-$ jelet tettünk.) Ha itt állítanánk le az algoritmust, akkor kijelenthetnénk, hogy nem maradt olyan cella, melynek 3-nál több le nem tapogatott szomszédja van.

Látható, hogy ezzel az egyszerű nyilvántartással a leállási feltételhez egy geometriai jelentés köthető. Még egyszerűbben jellemezhető feltételhez jutunk pl., ha $I = 1$ értéket írunk elő. Ez gyakorlatilag azt jelenti, hogy csak olyan cella maradhat ki a letapogatásból, melynek minden szomszédja le lett tapogatva (ld. 3.3. ábra).

Az algoritmus párhuzamosítása a PSA módszernél bemutatott megfontolások alapján, célszerűen domain dekompozícióval történhet. Ennek lényege, hogy a teljes letapogatandó teret egy másodlagos hálóval részekre osztjuk, majd ezeket a részeket átadjuk a szolgaprogramoknak. A szolgák működését

-	0	-	0	-
0	-	0	-	0
-	0	-	0	-
0	-	0	-	0
-	0	-	0	-

3.3. ábra. A kiválasztott cellák helyzete $I = 1$ leállási feltétellel

tekintve alapvetően két lehetőség kínálkozik:

a.) A szolgák kapott térrészt a beállított leállási feltétel figyelembevételével letapogatják. Könnyen belátható, hogy $I = 0$ leállási feltétel mellett az algoritmus jellemzői a PSA módszer jellemzőivel azonosak lesznek.

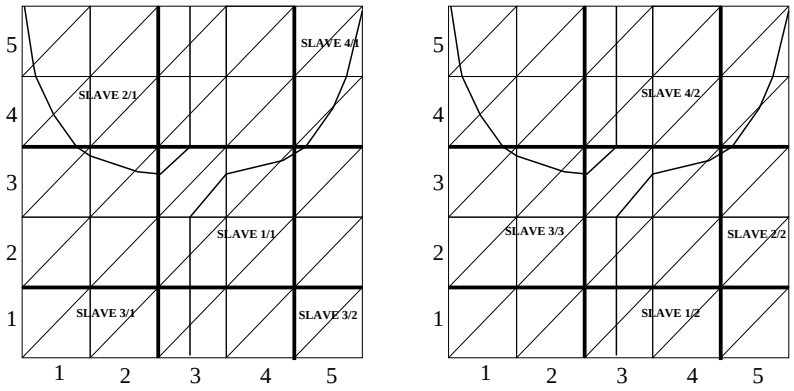
$I > 0$ esetén már nehezebb a jellemzők meghatározása. Sajnos a szolgák által számított térrészek szélein a leállási feltételből adódó geometriai jelentés is felbomolhat, mivel a szomszédos térrészt számító szolgának nincs információja a szélső cellák szomszédairól. A módszer egy konkrét implementációját ill. az azzal elért eredményeket részletesebben mutatja be egy a vezetésemmel készített diplomamunka [41].

$I > 0$ esetben a széleken jelentkező geometriai zavar hatása csökkenthető lenne, ha a tábla minden elemét kezdetben a maximális szomszédok számára ($2n$) állítanánk be. Ezzel ugyanis biztonság felé mozdulnánk, és megelőznénk, hogy a két egymástól függetlenül számított térrész illeszkedésénél a megkívánt geometriai jelentésnél rosszabb helyzet álljon elő.

b.) A szolgaprogramok továbbra is teljes letapogatást végeznek.
A mesterprogram a másodlagos ráccsal lefedett tér celláit a 3.1. algoritmusnak megfelelően választja ki. Így az algoritmus hatása a másodlagos rács méretének megfelelő térrészek figyelembe vételével jelentkezik.

A működés bemutatásához példaként tételezzük fel, hogy a 3.1. ábrán bemutatott feladat 5x5-ös terét úgy osztjuk fel egy másodlagos ráccsal, hogy a másodlagos rács osztástávolsága pont 2-szerese ($l_m = 2$) az elsődleges rácsénak (3.4. ábra). A felosztáshoz rendelt 3x3-as indextábla feltöltését a 3.5. ábra első táblázata szemlélteti. Tételezzük fel továbbá, hogy 4 szolga áll rendelkezésünkre. Az ábra második indextábláján az első 4 szolga elindítása utáni helyzet figyelhető meg. A harmadik és negyedik tábla az $I = 1$ ill. $I = 0$ leállási feltételkor kialakuló helyzetet ábrázolja. A szolgák nem azonos méretű tereket kapnak, ezért különböző ideig számolják az altereket. Amikor egy szolga befejez egy térrészt, akkor újabbat kap, amíg van – a leállási feltételt is figyelembe véve – kiszámolatlan térrész. A 3.4. ábrán a szolga sorszáma után feltüntettük azt is, hogy az adott térrészt hanyadik feladatként hajtja végre a szolga. Így pl. SLAVE 1/2 jelölés az első szolga 2. feladatát jelenti.

A feldolgozás egy lehetséges sorrendjét $I = 1$ leállási feltételig a 3.3. táblázat mutatja. A 3.4. táblázat pedig a folyamat folytatását ábrázolja $I = 0$ értékig. A kvantitatív jellemzők tekintetében belátható, hogy azonos másodlagos rácsméret mellett $I = 0$ leállási feltétellel a PSA algoritmussal megegyező értékeket kapunk. Egyszerű példánkon jól látható, hogy leállási feltétel növelésével a megbízhatóság csökken, de a sebesség nő. Ennek illusztrálására a merőleges erőtől mentes példához ($Q = 0$) kiszámítottuk a fontosabb jellemzőket különböző I értékek mellett. Ezt tartalmazza a 3.5. táblázat. Hangsúlyozzuk, hogy a kiszámolt értékek $l_m = 2$ esetre vonatkoznak, ezért $I = 0$ esetén is eltérést tapasztalunk a 3.2. táblázatban bemutatott



3.4. ábra. Szolgák feladatai 2x2-es másodlagos rács mellett (VSA)

2	3	2
3	4	3
2	3	2

-	0	-
0	-	1
-	1	2

-	0	-
0	-	0
-	0	-

-	-	-
-	-	-
-	-	-

3.5. ábra. A kiválasztott cellák helyzete $I = 1$ és $I = 0$ leállási feltétellel

eredményekhez képest. Várakozásainknak megfelelően I növelésével a megbízhatóság csökken, de a sebesség nő. Példánk egyszerűsége miatt több leállási feltételhez is azonos értékek adódnak, ami a kis méretű indextáblából adódik.

Könnyen belátható, hogy a másodlagos rács méretének $l_m = 1$ értékre történő megválasztásával a véletlenszerű letapogatás párhuzamosítására felvetett *a.* és *b.* módszer azonos. Belátható továbbá, hogy a VSA módszer megbízhatósága és sebessége a l_m és I paraméterek megválasztásával mindkét párhuzamosítási megoldással tág határok között paraméterezhető.

3.2. A fokozatos sűrítési algoritmus (FSA)

Az elkészített PSA implementációval végzett gyakorlati tapasztalatok hasznos eredményeket szolgáltattak a mikroszálak konfigurációinak vizsgálatához (5.3. alfejezet). Ezekhez a vizsgálatokhoz általános célú munkaállomások álltak rendelkezésünkre. Ezek sajnos nem csúcsteljesítményű gépek voltak. Ugyanakkor a feladat pontosabb kiértékelése viszont azt igényelte, hogy növeljük a rácspontok sűrűségét, ugyanis az eredményekben nagyon nagyszámmal jelentek meg másodlagos

	Időegység									
Szolga	1	2	3	4	5	6	7	8	9	10
1/1	23	→	→	→	24	→	33	→	34	
2/1	41	→	→	→	42	→	51	→	52	
3/1, 3/2	11	→	→	→	12	→	15	→	→	→
4/1	45	→	→	→	55	→				

3.3. táblázat. VSA feldolgozási folyamata 4 szolgával $I = 1$ feltétel mellett

	Időegység												
Szolga	7	8	9	10	11	12	13	14	15	16	17	18	19
1/2	..	..	..	13	→	→	→	14	→	⊕	⊕	⊕	⊕
2/2	..	..	..	25	→	→	→	35	→	⊕	⊕	⊕	⊕
3/3	..	..	..	..	21	→	→	→	22	→	31	→	32
4/2	43	→	→	→	44	→	53	→	54	⊕	⊕	⊕	⊕

3.4. táblázat. VSA feldolgozási folyamat folyatása a 7. időegységtől

Feltétel	T_1	K_1	E_1^a	R_1	T_4	E_4^a	R_4	S_4	E_4
$I = 0$	36	50	0.48	1.00	24	0.48	1.00	1.25	0.38
$I = 1$	36	50	0.48	1.00	24	0.48	1.00	1.25	0.38
$I = 3$	30	26	0.54	0.58	10	0.54	0.58	3.00	0.75
$I = 4$	30	26	0.54	0.58	10	0.54	0.58	3.00	0.75

3.5. táblázat. VSA algoritmussal elért jellemzők

parazita megoldások (ld. 1.5. alpontot). A feladatban vizsgált GRT 5 ill. 7 dimenziós volt, így a rácspontok sűrűségének növelése már oly mértékben növelte volna a számítási igényt, hogy a 10-15 munkaállomással végzett futtatások alapján azt becsültük, hogy a megnövelt pontossággal a feladat közel egy évig futna. Így más megoldást kellett keresni.

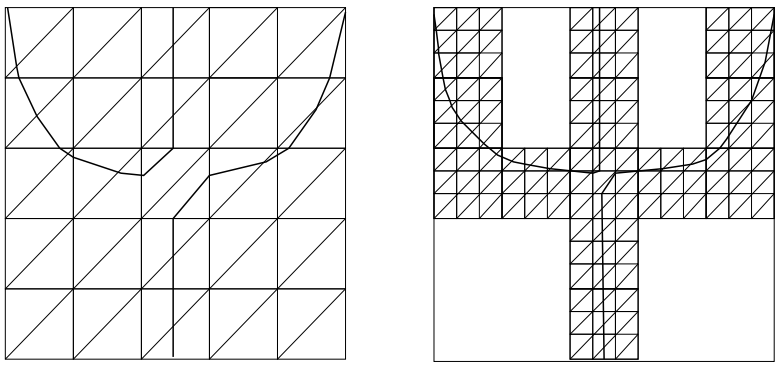
Az általunk vizsgált peremérték-feladatokban 1 paraméter van, ezért a keresett megoldáshalmaz tipikusan nullmértékű, ráadásul kodimenziója magas $(n-1)$. Ezért a GRT egyre finomabb felosztásával arányaiban egyre kevesebb olyan kockát ill. szimplexet kell találnunk, melyben megoldást remélhetünk. Ez viszont azt jelenti, hogy a felosztás finomításával egyre több olyan n dimenziós kockához jutunk, melyben teljesen felesleges a számítást elvégezni. Ha valamilyen módszerrel ki tudnánk szűrni ezeket a térrészeket, jelentős sebességnövekedést érnénk el.

Ez a gondolat vezetett a fokozatos sűrítési algoritmushoz, melynek lényege, hogy egy viszonylag durva felosztással meghatározzuk, hogy mely térrészekben várunk megoldásokat. Ezután ezen térrészekben a számítást finomabb felosztással újra elvégezzük. Ezzel a módszerrel természetesen megoldásokat veszíthetünk el, de ezt a veszélyt a hibakorlát lazításával, ill. az eredmények grafikus ábrázolása alapján tett felhasználói beavatkozással csökkenthetjük.

Az algoritmus működését a 3.6. ábrán figyelhetjük meg. Az ábra bal oldalán a már ismert egyszerű feladatunk egyensúlyi útja látható 5x5-ös ráccsal lefedve. A számítást elvégezve kiszűrjük azokat a cellákat, ahol kaptunk eredményt. Ez összesen 13 cellát jelent. Ezeket minden irányban 3-szor sűrűbb ráccsal lefedve újabb számítást indítunk (jobboldali ábra). Így eredményül 15x15-ös rácsnak megfelelő felosztáshoz jutottunk, de nem kellett a teljes teret ilyen felosztással letapogatni. A módszer természetesen ismételhető, így egyre pontosabb eredményekhez jutunk.

Feladat	T_1	K_1	E_1^a	R_1	T_4	S_4	E_4	T_4^z	S_4^z
5x5	36	50	0.48	1.00	16	2.25	0.56		
15x15	256	512	0.14	1.00	112	2.29	0.57		
13x3x3	208	234	0.31	1.00	64	3.25	0.81	80	1.4

3.6. táblázat. FSA algoritmussal elért jellemzők



3.6. ábra. Fokozatos sűrítési algoritmus két fázisa

Az eredmények értékeléséhez bevezetjük a T_N^z mennyiséget, amivel a fokozatos sűrítés során N processzorral mért összes futási időt jelöljük. Bevezetjük továbbá a T_N^r időt, ami a sűrítés során keletkezett végső felosztás eléréséhez kellene, ha nem alkalmaznánk a fokozatos sűrítés módszerét. Ezek felhasználásával a sűrítés módszerével elért sebességnövekedést az $S_N^z = \frac{T_N^r}{T_N^z}$ összefüggéssel definiáljuk. A 3.6. táblázatban 4 szolgát feltételezve feltüntettük az 5x5-ös felosztás, a 15x15-ös felosztás, valamint a sűrítés során keletkező 13x3x3-as méretű feladat jellemző értékeit ($Q = 0$). A táblázatban szereplő $S_4^z = 1.4$ számunkra azt jelenti, hogy a 15x15-ös felosztásnak megfelelő pontosságot 1.4-szer több idő alatt lehetett volna elérni, ha nem alkalmazzuk a fokozatos sűrítés módszerét. A táblázat alapján az is megfigyelhető, hogy a letapogató algoritmus hatékonysága E_N^a a rács sűrítésével a nullához közelít.

A fenti módszer gyakorlati kipróbálásához egy olyan program készült, mely a durva felosztással kapott eredményeket egy megadott sűrűségű térrács alapján osztályozza. Ennek során, amelyik térrácsba esik megoldás, annak koordinátáit a PSA program által feldolgozható formában egy konfigurációs állományba írja. Ezután a program javaslatot tesz, hogy mi legyen az újabb felosztás ahhoz, hogy az így keletkező n dimenziós téglatestek száma körülbelül azonos legyen a korábbi számításnál használt, durvább felosztásnál keletkező téglatestek számával. Ez a gyakorlatban azt jelenti, hogy így az újabb futási idő közel azonos lesz a korábbi felosztás futási idejével, de a kérdéses térrészekben már attól lényegesen pontosabb eredményre számíthatunk. Természetesen a javasolt felosztástól eltérő felosztás is választható az újabb futtatáshoz. A eljárást ismételve egyre szűkíthető az a térrész, ahol a megoldáspontokat keressük, a felosztás pedig növelhető anélkül, hogy a futási idő jelentősen megnövekedne.

Ezzel a segédprogrammal és a PSA program megfelelő módosításával (ne csak egyetlen térrész legyen megadható a konfigurációs állományban), jelentős sebességnövekedést sikerült elérni. A megfelelő térrész kiválasztását grafikus megjelenítő rendszerrel is ellenőriztük. Az egyes lépések eredményeinek grafikus ellenőrzése azért is fontos, mert a korábban említett másodrendű parazita megoldások grafikusan jól felismerhetők (ld. 4.2. ábra).

3.3. Letapogatással nyert adathalmazok pontosítása útkövetéssel

A 2. fejezetben bemutatott szimplex algoritmus ismeretében belátható, hogy nagy rácsméret mellett is keletkezhet bizonyos számú elfogadhatóan pontos megoldás. Az alkalmazás használata során szerzett tapasztalataink ezt a feltevést igazolták, így felvetődött, hogy a 3.2. pontban leírt

eljárással ellentétben ezeket a pontokat ne újabb letapogatás indításához használjuk fel, hanem próbáljuk az adott pontból követni a görbét.

3.3.1. Szimplex útkövetés algoritmusa (SUA)

Egy megoldáspont ismeretében az 1.5. pontban bemutatott PL módszert felhasználhatjuk az adott ponton átmenő megoldásgörbe újabb pontjának előállítására. A módszer a [14] cikkben bemutatott algoritmushoz hasonló, azonban itt nem tükrözéssel állítjuk elő az új szimplexet. Vegyük fel az ortogonális hálót úgy, hogy a szimplexekre bontás után az ismert megoldáspont két szimplex határoló felületére essen. Válasszuk ki a két szimplex közül az egyiket. Ezt a továbbiakban jelölje s_1, az ismert pontot pedig nevezzük belépési pontnak (p_{be}). A megoldásgörbe szimplexbe eső linearizált darabja az s_1 szimplex két oldalát fogja metszeni. A p_{be} ponttal ellentétes oldalon keletkező metszéspontot nevezzük el kilépési pontnak (p_{ki}). A szimplex azon szomszédját pedig, mely a kilépési pont felöli oldalán van, jelölje s_2. A kilépési pont egyben a következő szimplex belépési pontja is. Így előző lépések ismétlésével a kezdeti pontból kiinduló megoldásgörbe folyamatosan bővíthető, azaz követhető (ld. 3.7. ábra). A 3.2. algoritmus-leírás a kezdőpontból kiindulva az egyik irányba követi a megoldást.

```
   p_be ← kezdőpont
   s_1 kiválasztása
   while p_be vizsgált térrészben van do
   |  Linearizálás és egyenletmegoldás az s_1-ben.
   |  if nincs megoldás then
1  |  |  Exit
   |  p_ki és s_2 meghatározása
   |  p_be ← p_ki
   |  s_1 ← s_2
   end
```

3.2. algoritmus: Szimplex útkövetés (SUA)

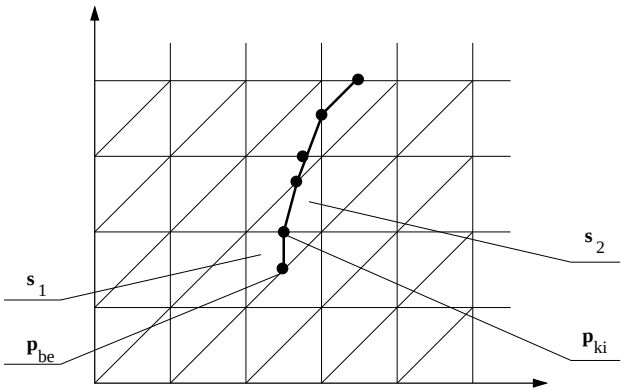
Mivel a két szomszédos szimplexnek mindig van egy közös oldala, ezért csak egy nem közös csúcsuk van. Így az egymást követő számítások csak egy integrálási lépést igényelnek. Ez egyben azt jelenti, hogy a PF egy újabb megoldáspontjához csupán egyetlen KF függvény megfelelő pontban való kiértékelését kell elvégezni. Meg kell jegyezni azonban, hogy a leírt algoritmus csak korlátozásokkal alkalmazható:

1. Az algoritmus nem képes kezelni az elágazásokat. Ez abból következik, hogy a linearizálás miatt egy szimplexbe csak egy belépési és egy kilépési pont eshet.

2. Az algoritmus nem robusztus, ugyanis numerikus hibákból és a linearizálásból adódóan előfordulhat, hogy egy belépési ponthoz nem találja meg a kilépési pontot. Ekkor viszont a leírt algoritmus leáll (ld. a címkével jelölt utasítást az algoritmusban).

3. Zárt görbe esetén nem áll meg az algoritmus, azaz végtelen ciklusban követi a görbét.

Fenti módszert példafeladatunkra alkalmazva mind a $Q = 0$, mind pedig a $Q = \epsilon$ esetben meghatároztuk a működés folyamatát, és a korábban bevezetett mennyiségeket. Ezeket a 3.7. ill. a 3.8.



3.7. ábra. Szimplex útkövetés

	Időegység												
Feladat	1	2	3	4	5	6	7	8	9	10	11	12	13
$Q = 0$	13A	$\rightarrow$	$\rightarrow$	13B	23A	23B	33A	34B	34A	35B	45A	45B	55A
$Q = \epsilon$	13A	$\rightarrow$	$\rightarrow$	13B	23A	24B	34A	35B	45A	45B	55A	$\oplus$	$\oplus$

3.7. táblázat. Szimplex útkövetés működési folyamata

táblázatban foglaltuk össze. A működési folyamatból jól megfigyelhető, hogy csak az egyik ágat tudja követni az algoritmus.

Az algoritmus párhuzamosítását illetően domain szintű párhuzamosítás csak abban az esetben jöhet szóba, ha a kiindulási pontból mindkét irányban követni tudunk, vagy esetleg több kiindulási pontunk van. Más, a 2.1. alfejezetben bemutatott módszerek természetesen alkalmazhatók lennének, de csak korlátozásokkal. Ezt az algoritmust változtatás nélkül csak szekvenciálisan alkalmaztuk a letapogatással nyert adatok további pontosítására (ld. 5. fejezet). Az algoritmus módosított, párhuzamosításra is alkalmas változatát a következő alfejezet mutatja be.

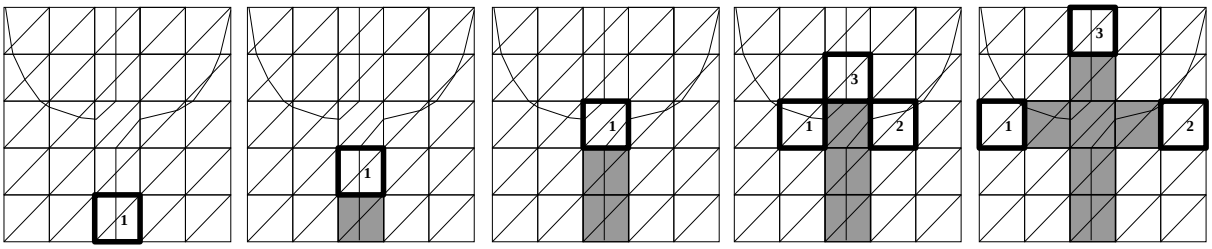
3.3.2. Párhuzamos útkövetés algoritmusa (PUA)

Az előző alfejezetben bemutatott algoritmus hatékonysága csökken ugyan, de megbízhatósága növekszik, ha szomszédos szimplexek helyett szomszédos kockákat választunk ki. Amíg egy szimplexnek csupán egy belépési és egy kilépési pontja lehet, addig egy n dimenziós kockának n belépési és n kilépési pontja lehet. Ez lehetővé teszi, hogy a kockára épülő algoritmus követni tudja a megoldásgörbe elágazásait is.

A szimplexen alapuló módszerhez képest így a numerikus hibákból adódó leállás veszélye is csökken, mivel a megfelelő szomszédos kocka minden szimplexe részt vesz a számításban. Azonban a

Feladat	T_1	K_1	E_1^a	R_1
$Q = 0$	11	11	1.00	0.46
$Q = \epsilon$	9	9	1.00	0.41

3.8. táblázat. Szimplex útkövetéssel elért jellemzők



3.8. ábra. Párhuzamos útkövetés (PUA)

Idő	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1. Szolga	13	→	→	→	23	→	33	→	32	→	31	→	41	→	51	→
2. Szolga									34	→	35	→	45	→	55	→
3. Szolga									43	→	53	→	⊕	⊕	⊕	⊕

3.9. táblázat. Párhuzamos útkövetés működési folyamata $Q = 0$ esetben

Idő	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1. Szolga	13	→	→	→	23	→	24	→	34	→	35	→	45	→	55	→

3.10. táblázat. Párhuzamos útkövetés működési folyamata $Q = \epsilon$ esetben

veszély továbbra is megmarad azokban az esetekben, amikor kilépési pont a kocka élére vagy csúcsára esik. Ezt úgy tudjuk kiküszöbölni, ha az élhez ill. csúcshoz tartozó oldallap minden szomszédját, valamint az él- és csúcsszomszédokat is bevonjuk a további számításokba. Ez sajnos tovább rontja az algoritmus hatékonyságát, mivel a minimálisan szükségesnél több kockában is el kell végezni a számítást.

A kockákra épülő algoritmus párhuzamosítására kézenfekvő az az ötlet, hogy a kilépési pont keresésekor a potenciálisan szóbajövő kockákban a számítást párhuzamosan végezzük el. Így a linearizálási hibából keletkező, szükségképpen elhaló ágak számítása a valódi megoldásgörbére eső kockák számításával egyidőben elvégezhetők. Elágazás esetén pedig, az egyes ágak követése párhuzamosan történhet.

A módszert 4 szolgával példafeladatunkra alkalmazva a $Q = 0$, mind pedig a $Q = \epsilon$ esetben meghatároztuk a működés folyamatát és a korábban bevezetett mennyiségeket[3]. A 3.9. táblázat alapján figyelemmel kísérhetjük a $Q = 0$ eset működési folyamatát. A 3.8. ábrán látható az elágazás pillanata. Ekkor további két párhuzamos ág indulhat. Az ábrán feltüntettük a párhuzamos példányok sorszámát is. $Q = \epsilon$ esetben viszont csupán az egyik ágat tudja követni az algoritmus, ami a 3.10. táblázat alapján jól megfigyelhető.

A fenti algoritmus további módosításával elérhető, hogy olyan „elágazásokat" is megtaláljunk, melyeknél a két görbének nincs közös pontja, de meglehetősen közel haladnak egymáshoz. Ez a vizsgált térrészre fektetett másodlagos ráccsal érhető el. Az útkövetés a másodlagos rács által kijelölt térrészek (másodlagos kockák) figyelembe vételével zajlik. A kockákon belül pedig egyszerű

[3]Az egyszerűség kedvéért feltételeztük, hogy a szolgák a függvényértékeket egy közös tárban tárolják, ezzel megoldható, hogy egy függvényérték csak egyszer legyen kiszámolva. Lazán csatolt párhuzamos rendszerben azonban ez a módszer nem kivitelezhető hatékonyan (ld. 2.1. alfejezetben).

Feladat	T_1	K_1	E_1^a	R_1	T_4	E_4^a	R_4	S_4	E_4
$Q = 0$	28	26	0.92	1.00	16	0.92	1.00	1.75	0.44
$Q = \epsilon$	16	14	0.64	0.41	16	0.64	0.41	1.00	0.25

3.11. táblázat. Párhuzamos útkövetés jellemzői

letapogatást végzünk.

Nyilvánvaló, hogy a másodlagos rács méretének (l_m) növelése csökkenti a hatékonyságot, ugyanakkor növeli a megbízhatóságot és annak valószínűségét, hogy minden megoldásgörbét megtalálunk. A másodlagos rács bevezetésével a teljes letapogatás módszere (PSA) az útkövetés egy olyan speciális esetének fogható fel, melyben a másodlagos rácsméret megegyezik a vizsgált tér méretével. A PUA hiányossága, hogy a zárt görbék esetén végtelen ciklusba kerül. Ezt az algoritmusba beépített adminisztrációval lehet kiküszöbölni. Egy megoldás lehet pl., hogy minden letapogatott másodlagos kockát nyilvántartunk egy adatbázisban. Egy új kockába való belépésnél pedig megvizsgáljuk, hogy járt-e már az algoritmus az adott kockában. Ez pl. a 3.1. alpontban bemutatott indextáblához hasonló megoldással valósítható meg. A konkrét megvalósítást a következő alfejezet mutatja be.

3.4. Párhuzamos hibrid algoritmus (PHA)

A letapogató algoritmus (PSA) megbízhatóságát és az útkövetéses módszer (PUA) hatékonyságát kívántuk ötvözni a hibrid (PHA) algoritmusban. A módszer alapötletét a véletlenszerű letapogatás (VSA) (3.1. alfejezet) adta. Az ott bevezetett indextáblát kiegészítettük egy újabb információval, amely azt adja meg, hogy az adott cellának van-e olyan letapogatott szomszédja, amelyből egy vagy több megoldásszakasz ide átlép. Így az indextábla minden celláról a következő információkat tárolja:

- a le nem tapogatott szomszédok számát ($0 \leq i \leq 2n$),
- azt a tényt, hogy az adott cellát már letapogattuk ($i = -1$),
- azt a tényt, hogy valamelyik letapogatott szomszédos cellában találtunk olyan megoldást, amelyik ebbe a cellába átlép. ($i = 2n + 1$).

A fenti információk tárolásához olyan kódolást választottunk, amely a legtömörebb tárolást teszi lehetővé. A további feldolgozás szempontjából felesleges kombinációk elhagyásával [4] a zárójelben megadott kódolás tűnt a legtömörebbnek. Ebben kihasználtuk azt, hogy egy n dimenziós cellának maximum $2n$ szomszédja lehet. A $2n + 1$ érték azt jelzi, hogy ha valamelyik szomszédból van átlépő megoldás. Mivel ez biztosan nagyobb minden más értéknél, ezért az algoritmus ezt az értéket fogja legközelebb választani. Ebből az következik, hogy ha sikerült egy megoldásszakaszra a véletlen kiválasztással rátalálni, akkor azt az algoritmus addig követi, amíg tudja. Minthogy az indextáblában a cella letapogatásának tényét is tároljuk, így a zárt görbék esetében sem kerülünk végtelen ciklusba. Az így módosított algoritmust a 3.3. leírás szemlélteti.

Az I leállási feltétel értékétől függően a fenti algoritmus:

- teljes letapogatást ($I = 0$),
- útkövetést ($I = 2n + 1$) vagy

[4] Pl. ha a cellát letapogattuk, akkor a le nem tapogatott szomszédok száma már nem érdekes.

```
Indextábla feltöltése a szomszédok számával.
foreach h, előre ismert megoldáspontok celláira do
  | idx[h] ← 2n + 1
end
repeat
  | i_max ← indextáblában található maximális érték
  | h ← a rács egy véletlenszerűen választott helye, ahol i_max = idx[h]
  | idx[h] ← −1
  | foreach j, h minden szomszédjára do
  |   | if idx[j] > 0 és idx[j] < 2n + 1 then
  |   |   | idx[j] ← idx[j] − 1
  | end
  | h-nak megfelelő cellában elvégezzük a szimplex letapogatást
  | if van megoldás then
  |   | foreach j, h minden szomszédjára do
  |   |   | if j-be belép a megoldás és idx[j] >= 0 then
  |   |   |   | idx[j] ← 2n + 1
  |   | end
  | end
until i_max < I
```

3.3. algoritmus: Hibrid algoritmus (PHA)

- útkövetést és letapogatást ($0 < I < 2n + 1$) végez.

Ezzel egy olyan módszerhez jutottunk, amely a letapogatás megbízhatóságát ötvözi az útkövetés gyorsaságával, ugyanakkor a megbízhatóság – a sebesség rovására – az I paraméter beállításával szabályozható. Az élek és csúcsok közelében kilépő megoldások megfelelő kezelésével, azaz az összes közeli szomszéd indexének $2n + 1$ értékre történő beállításával az algoritmus kellően robusztussá tehető.

A 3.1. alfejezetben megmutattuk, hogy I egyfajta geometriai jelentéssel rendelkezik. Ez a követés miatt némileg módosul. A külön kiemelt $I = 1$ érték jelentése pl. most azzal egészül ki, hogy minden olyan megoldásszakaszt megtaláltunk, amelyik egyik cellából átmegy a másikba, azaz csak a cellán belüli hurkok maradhatnak felderítetlenül. Látható, hogy a követés miatt már viszonylag magas I értékek is használható eredményeket nyújthatnak, ha pl. tudjuk vagy sejtjük, hogy a keresett bifurkációs görbe nem túl sok apró darabból áll.

Az algoritmus párhuzamos implementációja a legegyszerűbben a VSA módszernél ismertetett másodlagos rács segítségével történhet. A konkrét megvalósításnál 50. oldalon bemutatott, letapogatással kombinált módszert részesítettük előnyben, azaz a szolgák teljes letapogatást végeznek, míg a véletlen kiválasztás és útkövetés csak a másodlagos rács szintjén valósul meg. Döntésünk mellett a következő érvek illetve ellenérvek szóltak:

\+ Így lényegében csak a mesterprogramon kellett változtatnunk. A továbbra is teljes letapogatást végző szolgaprogramba csak azt kellett beépíteni, hogy küldje vissza a mesternek azt az információt, hogy az általa letapogatott teret mely irányokban hagyta el a megoldásgörbe.

2	3	3	3	2
3	4	4	4	3
3	4	4	4	3
3	4	4	4	3
2	3	3	3	2

2	3	3	2	2
3	4	5	-	5
3	5	-	5	3
2	-	5	-	2
2	2	3	2	2

2	3	5	2	2
3	2	-	-	5
5	-	-	-	5
2	-	-	-	2
2	2	5	2	2

3.12. táblázat. Hibrid algoritmus (PHA) indextáblája és az első 8 lépés

	Időegység															
Szolga	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1.	22	→	→	→	32	→	31	→	41	→	51	→	52	→	21	
2.	24	→	→	→	23	→	13	→	11	→	→	→	12			
3.	44	→	→	→	34	→	35	→	45	→	55	→	25	→	54	→
4.	33	→	→	→	43	→	53	→	15	→	→	→	14	→	42	→

3.13. táblázat. Hibrid algoritmus működési folyamata 4 szolgával ($I = C = 1$)

Feladat	T_1	K_1	E_1^a	R_1	T_4	E_4^a	R_4	S_4	E_4
$Q = 0$	48	36	0.67	1.00	12	0.67	1.00	4.00	1.00
$Q = \epsilon$	48	34	0.64	1.00	12	0.64	1.00	4.00	1.00

3.14. táblázat. Hibrid algoritmus számított jellemzői ($I = C = 1$)

+ Az indextábla mérete kisebbre adódhat, ami nem elhanyagolható a hatékony megvalósítás szempontjából.

– Az algoritmus hatékonysága a rácsméret növelésével csökken, ugyanakkor a túl kicsi rácsméret nagyon megnövelheti a mester és a szolgák közötti kommunikációt. Ez viszont a párhuzamos implementáció hatékonyságát csökkenti.

A hatékony implementáció érdekében célszerűnek látszott a mester által kezelt indextábla memóriában tárolt tömbként történő megvalósítása. Nagy méretű és több dimenziós GRT-k esetén ez azonban nem vagy csak nagyon rossz hatékonysággal kivitelezhető. Ezért ha az indextábla túl nagy méretű, akkor a mester a táblát egy igen egyszerű adatbázis-kezelő segítségével (GDBM) tárolja ill. kezeli. Így a tábla méretéből adódóan nem kellett korlátozásokat bevezetnünk a megvalósított programban.

Az algoritmus működésének bemutatására tekintsük a már jól bevált példánk 5x5-ös ráccsal való lefedését. Tételezzük fel továbbá, hogy a 3.3.2. alfejezetben ismertetett módon 4 szolgaprogramunk van, és a másodlagos rács mérete $l_m = 1$, így az indextábla is 5x5 méretű. Az indextábla értékeit és az első 8 cella kiválasztását a 3.12. táblázat segítségével, a működést pedig a 3.13. táblázat segítségével figyelhetjük meg. A 3.14. táblázat az egyszerű példánkra kiszámolt jellemző értékeket tartalmazza.

3.5. PSA szakadásos függvényekre

Az egyik alkalmazás, a folyadékhidak (5.2. alfejezet) vizsgálata során olyan feladat adódott, melyben a peremfeltételt adó függvény szakadásos függvényhez vezetett. A függvény érdekessége

a szakadáson kívül az volt, hogy több szakadási ponttal rendelkezett, és az egyes szakadási pontok között folytonos szakaszokban folytatódott. Ezeket a szakaszokat a szakadási pontok száma egyértelműen azonosítja, melyeket a továbbiakban szinteknek nevezünk. Az azonos független változó értékekhez tartozó, de különböző szinteken értelmezett függvényértékeket egyetlen integrálási ciklussal elő lehet állítani, ugyanis ugyanazon KF megoldását jelentik.

A PF megoldásához az azonos szintekhez tartozó függvényértékeket kellett kiértékelni. A korábban implementált PSA program képes volt ugyan a függvények értelmezési tartományát korlátozni, de a fenti probléma kezelésére nem volt alkalmas. Ez azt jelenti, hogy az egyes paraméter síkokon (szinteken) történő számítások külön paraméterrel megadhatók lettek volna, de minden szint vizsgálatához újabb futtatásra lett volna szükség. Ugyanakkor a függvény bemutatott jellegéből adódóan az egyes szintekhez tartozó értékek hatékonyabban voltak előállíthatók egyetlen futtatással. Ez abból adódott, hogy az újabb szintek függvényértékeinek kiszámításához minden előző szint kiszámítása is szükséges volt.

A függvénymodulhoz új interfész készült, amely képes több paramétersíkon is előállítani a függvényértékeket. A szintek maximális száma a függvénymodulban előre meghatározott. Az egyes szintekhez tartozó értékeket a függvénymodul egyszerre (egy függvényhívással) állítja elő. Ez azt jelenti, hogy az aktuális feladat függvénymoduljának tervezésekor jól kihasználhatók a szintek közötti számítási összefüggések. A függvénymodul minden szinthez előállít egy jelzőt is, amely a függvény adott szinten való értelmezettségét jelzi.

Ez a módosítás természetesen a szolgaprogram egyéb részeit is érintette, de ezzel előállt egy olyan program, amely a korábbi feladatok és a folyadékhidak vizsgálatánál felmerülő feladattípus vizsgálatához is hatékonyan alkalmazható. A módosítás során külön figyelmet fordítottunk arra, hogy a korábban tesztelt feladattípusok megoldási hatékonysága ne romoljon.

3.6. Többparaméteres PF megoldása rekurzív programozással

Az eddig bemutatott feladatok megoldásai különböző görbéket szolgáltattak a GRT-ben. A problémák vizsgálatánál felvetődhet az a kérdés, hogy ezen görbék kitüntetett pontjai, például a kritikus pontok hogyan vándorolnak, valamely – a PF leírásában – eddig konstansként kezelt paraméter függvényében. Ilyen problémához jutunk, ha pl. az 1.3. alfejezetben bemutatott rúd viselkedését vizsgáljuk különböző nagyságú tengelyre merőleges (Q) állandó erők mellett, és kíváncsiak vagyunk, hogy a Q erő függvényében hogyan vándorol a megoldásgörbe minimum pontja. (Az 5.2. alfejezet egy valós példát mutat az ilyen problémára.)

Klasszikus megközelítésben ez az analízis csak úgy végezhető el, hogy minden egyes paraméterhez meghatározzuk a megoldásgörbét, majd a kérdéses pontokat egy grafikonon összekötjük. Ez meglehetősen munkaigényes, ráadásul pontatlan módszer. A következőkben bemutatjuk az 1.5. alfejezetben ismertetett módszer rekurzív kiterjesztését, amely alkalmas az ilyen többparaméteres feladatok megoldásainak tetszőleges pontosságú numerikus előállítására.

A módszer alapja az, hogy az eredeti PF GRT-jét kiegészítjük a kérdéses paraméterrel (c). Az így kiegészített GRT-ben egyértelműen és véges dimenzióban megadható az adott paraméter változtatásával keletkező összes megoldás. Ezzel a problémát egy eggyel magasabb dimenziójú feladatra vezettük vissza [49]. Az új $n+1$ dimenziós feladat megoldásához n függvényre van szükségünk Ebből $n-1$-et változtatás nélkül átvehetünk az eredeti feladatból. Az n-edik, új függvénynek ott kell zérus értéket felvennie, ahol a megoldásgörbe kívánt tulajdonságú pontja van. A fentiek alapján kétfajta függvény fog szerepelni az új $n+1$ dimenziós rendszerben, az f és a g, ahol f az 1.7 össze-

függésben szereplő KF megoldó függvény, a g pedig egy alkalmas skalár értékű függvény, amely a megoldásgörbe valamely lokális tulajdonságát emeli ki.

Hasonló gondolatmenettel az így kapott megoldásgörbék viselkedését is megvizsgálhatnánk egy újabb paraméter függvényében. Ezen újabb paraméter miatt a dimenziószám ismételten nő, és egy újabb függvény bevezetésére van szükség, amely az előző eggyel alacsonyabb dimenziószámú feladat megoldásgöbéjének megfelelő tulajdonságát emeli ki. A gondolatmenet tetszőleges dimenziószámig rekurzívan folytatható.

A megoldás folyamatának elemzéséhez vizsgáljunk meg először egy $n+1$ dimenziós feladatot. A megoldás során a KF megoldó függvények (f) kiértékelése a rácspontokban nem jelent problémát, mivel azok a GRT minden pontjára értelmezettek. A g függvény numerikus kiértékeléséhez azonban ismerni kell *az adott rácsponton átmenő megoldásgörbét.* Néhány kivételtől eltekintve azonban egy adott rácsponton nem megy át megoldásgörbe, így g értékét nem tudjuk közvetlenül kiszámolni. Ezért az eredeti PF-ot a feladat általánosításaként úgy módosítjuk, hogy az adott rácspont éppen egy megoldáspont legyen. Könnyen belátható, hogy ez a végpontban előírt peremértékek megfelelő módosításával elérhető. A gyakorlatban ez azt jelenti, hogy egy olyan PF feladat megoldását kell megkeresnünk, amely $f = 0$ feltétel helyett az $f = f_p$ feltételt elégíti ki, ahol f_p éppen az adott pontban felvett függvényértéket jelöli. Ezen feladat megoldáspontja áthalad a vizsgált ponton, így g is értelmezhető rajta.

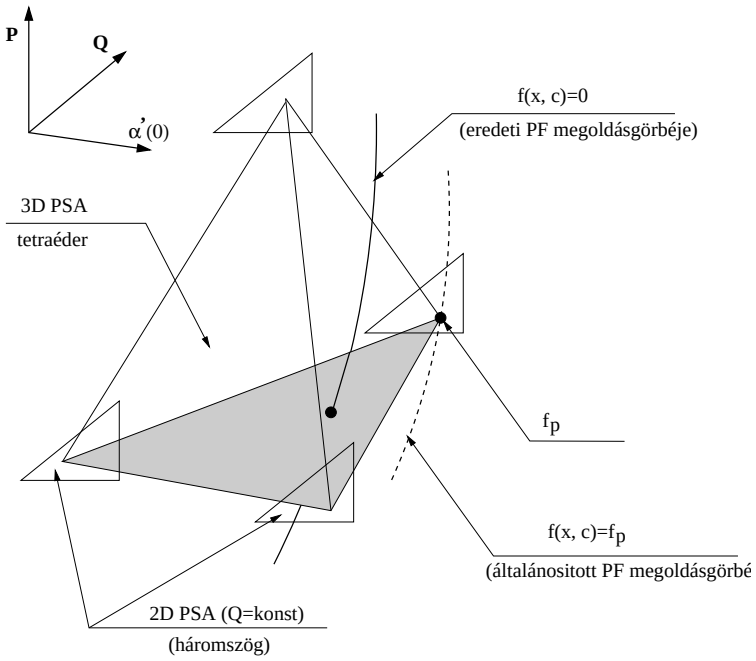
A fentiek alapján g értéke egy adott (x_p, x_p) rácspontban úgy számítható ki, hogy a rácspont kis környezetében meg kell oldani egy, a 3.1 összefüggés szerinti n dimenziós nemlineáris egyenletrendszert. Azaz az eredeti $n+1$ dimenziós feladat megoldása közben egy kis térrészre (egyetlen szimplex) ki kell számolni egy n dimenziós feladatot (ld. a 3.9. ábrát). Az eredményül kapott megoldásszegmens pontjai alapján a g függvény értéke ezek után már meghatározható.

$$f(x, c_p) - f(x_p, c_p) = 0, \quad x \in \Re^n, \quad c_p \in \Re, \quad f : \Re^{n+1} \to \Re^{n-1} \tag{3.1}$$

Ezt a gondolatmenetet általánosítva elkészült a 2. fejezetben bemutatott alkalmazás rekurzív változata. Ehhez a függvénymodult ki kellett egészíteni a g függvények definiálásának lehetőségével. Ezért némileg megváltozott a modul interfésze, de igyekeztünk úgy beilleszteni az új részeket, hogy a korábban készített függvénymodulok minimális változtatással felhasználhatók legyenek. Mivel a dimenziószámot korábban sem korlátozta az alkalmazás, a rekurzió mélységének korlátozása sem vált szükségessé.

A módszer bemutatásából látható, hogy a rekurzió lényegében azt jelenti, hogy egyes függvényértékek meghatározásához meg kell oldani egy alacsonyabb dimenziószámú feladatot, melynek mérete nagyon kicsi, csupán egyetlen szimplexet tartalmaz. Ebből következik, hogy a rekurzió implementálása miatti módosítás nem érintette a letapogatás, követés, stb. algoritmusait. Kisebb paraméter-átadási mechanizmusok módosulásától eltekintve a függvényértékeket számító programrész módosult. Ez a feladattól ill. a pillanatnyi rekurzió-szinttől függően vagy a függvénymodult hívja, vagy az egy szimplexet kiszámító programrészt (lényegében saját magát). Így nemcsak a megoldási módszer rekurzív, hanem maga a program is.

A rekurzió lehetőségével kiegészített alkalmazást sikeresen alkalmaztuk folyadékhidak egyensúlyi alakjainak meghatározására [20]. Ekkor még az útkövetéses algoritmus nem volt beépítve az alkalmazásba, így eredményeinket teljes letapogatással nyertük. Mivel maga a PF 2 dimenziós, így a rekurzió a 3. szintről indul, és csupán 1 mélységű. Ennek megvalósítása nem igényelt kiemelkedően magas számítási kapacitást (ld. 5.2. alfejezetben).



3.9. ábra. Rekurzió: 2 dimenziós feladat megoldása a 3 dimenziós szimplex csúcsain

3.7. Teljesítménynövelési módszerek összehasonlítása

Az algoritmus letapogató változatától (PSA) várható sebességnövekedés közelítésére a 2.1. alfejezet a gyakorlatban jól használható összefüggést adott. Ez az összefüggés azonban csak a teljes letapogatást eredményező változatoknál alkalmazható. Követéses, vagy követéssel kombinált hibrid algoritmusok esetében csak a tényleges feladat megoldáspontjainak ismeretében lehetne a 2.8 összefüggéshez hasonló analitikus modellt felállítani. Feltehetően célravezetőbb lenne egy valószínűségi modell felállítása, de ezzel jelen munka során nem foglalkoztunk, ezért az összehasonlítást egy konkrét feladat kapcsán végeztük el. Az így nyert eredményeket pedig megpróbáltuk oly módon osztályozni, hogy azokból általános érvényű következtetéseket vonhassunk le.

A mintafeladatra kiszámolt jellemzőket a 3.15. és a 3.16. táblázatokban foglaltuk össze. A számszerű összehasonlítás azonban mégsem egyszerű. Ennek oka az, hogy példafeladatunk a valós feladatokhoz képest túl kicsi, és az alkalmazott rács pedig túl ritka. Ez a legtöbb esetben jelentősen elnyomta a párhuzamosítás gyorsító hatásait, így a sebességnövekedés (S_N), valamint a párhuzamosítás hatásfoka (E_N) lényegesen elmarad a ténylegesen elérhető értékektől.

Ezek az eredmények azonban a módszerek osztályozására, nagyságrendi összehasonlítására mégis jó támpontot nyújtanak. Segítségükkel megfigyelhetjük, hogy az algoritmus hatékonysága a letapogató módszerek (PSA, VSA) esetén a legrosszabb, a feladat méretének növekedésével pedig egyre csökken. Az útkövető (SUA, PUA) és a hibrid (PHA) algoritmus esetében a hatékonyság nem vagy kevésbé függ a feladat méretétől, de az útkövető módszer csak a kiindulási ponttal összekötött megoldásokat képes megtalálni. A megbízhatóság terén a letapogató módszerektől és a hibrid algoritmustól várhatjuk a legjobb eredményt.

Eredményeink összehasonlítása alapján kijelenthetjük, hogy nagy méretű vagy sok dimenziós GRT esetén csak a hibrid algoritmus képes megfelelő sebességgel a nem összefüggő ágakat felfe-

dezni. Az is megfigyelhető, hogy a hibrid módszerrel értük el a legnagyobb sebességnövekedést. A bemutatott algoritmusok működése a 3.17., és a 3.18. táblázat segítségével, valamint a 3.10. ábra segítségével grafikusan is összehasonlítható.

Módszer	T_1	K_1	E_1^a	R_1	T_4	E_4^a	R_4	S_4	E_4
PSA	36	50	0.48	1.00	16	0.48	1.00	2.25	0.56
SUA	11	11	1.00	0.46					
PUA	28	26	0.92	1.00	16	0.92	1.00	1.75	0.44
PHA	48	36	0.67	1.00	12	0.67	1.00	4.00	1.00

3.15. táblázat. A módszerek jellemző értékeinek összehasonlítása $Q = 0$ esetben

Módszer	T_1	K_1	E_1^a	R_1	T_4	E_4^a	R_4	S_4	E_4
PSA	36	50	0.44	1.00	16	0.44	1.00	2.25	0.56
SUA	9	9	1.00	0.41					
PUA	16	14	0.64	0.41	16	0.64	0.41	1.00	0.25
PHA	48	34	0.64	1.00	12	0.64	1.00	4.00	1.00

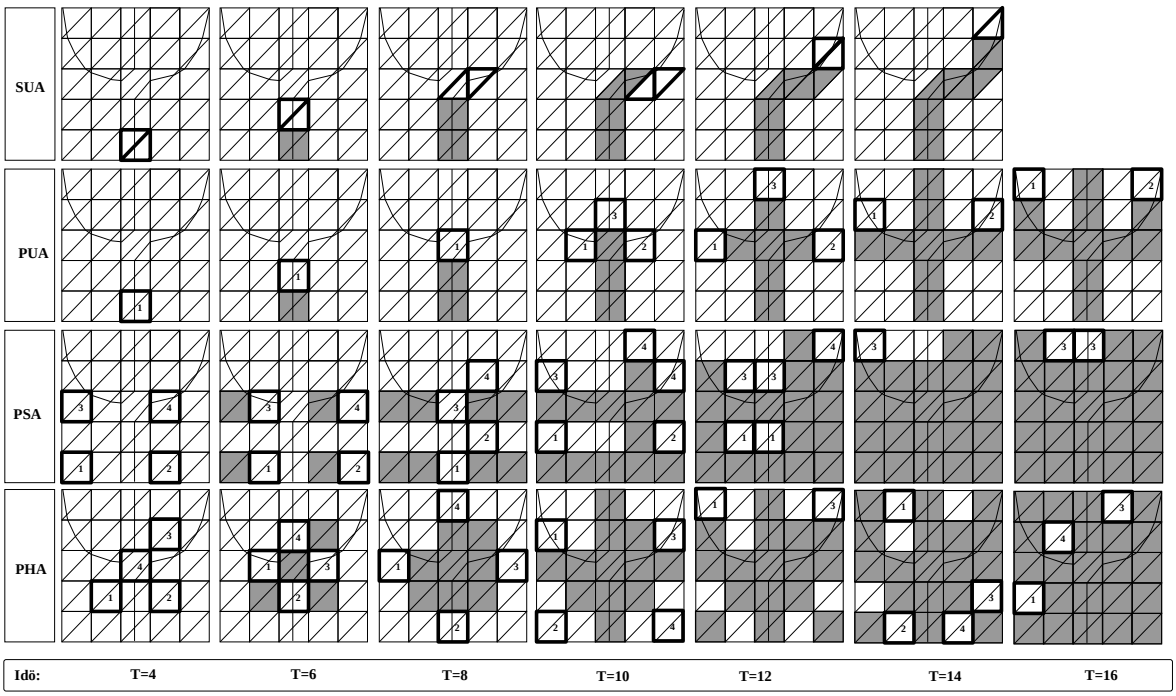
3.16. táblázat. A módszerek jellemző értékeinek összehasonlítása $Q = \epsilon$ esetben

	Időegység															
Mód.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PSA1	11	→	→	→	12	→	13	→	21	→	22	23				
PSA2	14	→	→	→	15	→	24	→	25							
PSA3	31	→	→	→	32	→	33	→	41	→	42	23	51	→	52	53
PSA4	34	→	→	→	35	→	44	→	45	54	→	55				
SUA1	13A	→	→	13B	23A	23B	33A	34B	34A	35B	45A	45B	55A			
PUA1	13	→	→	→	23	→	33	→	32	→	31	→	41	→	51	→
PUA2									34	→	35	→	45	→	55	→
PUA3									43	→	53	→				
PHA1	22	→	→	→	32	→	31	→	41	→	51	→	52	→	21	
PHA2	24	→	→	→	23	→	13	→	11	→	→	→	12			
PHA3	44	→	→	→	34	→	35	→	45	→	55	→	25	→	54	→
PHA4	33	→	→	→	43	→	53	→	15	→	→	→	14	→	42	→

3.17. táblázat. A módszerek működésének összehasonlítása $Q = 0$ esetben

	Időegység															
Mód.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PSA1	11	→	→	→	12	→	13	→	21	→	22	23				
PSA2	14	→	→	→	15	→	24	→	25							
PSA3	31	→	→	→	32	→	33	→	41	→	42	23	51	→	52	53
PSA4	34	→	→	→	35	→	44	→	45	54	→	55				
SUA1	13A	→	→	13B	23A	24B	34A	35B	45A	45B	55A					
PUA1	13	→	→	→	23	→	24	→	34	→	35	→	45	→	55	→
PHA1	22	→	→	→	23	→	13	→	11	→	→	→	12	→	14	
PHA2	24	→	→	→	34	→	35	→	45	→	55	→	25			
PHA3	44	→	→	→	43	→	53	→	52	→	15	→	→	→	54	
PHA4	33	→	→	→	32	→	31	→	41	→	51	→	21	→	42	

3.18. táblázat. A módszerek működésének összehasonlítása $Q = \epsilon$ esetben



3.10. ábra. A módszerek működésének grafikus összehasonlítása

4. fejezet

Alkalmazási modell

A kifejlesztett PSA program a kutatási ciklus egy fontos fázisát támogatja, de önmagában nem használható hatékonyan, csak egy integrált környezet részeként. A 2. és a 3. fejezetek bemutatták a PSA program felépítését algoritmusait. Jelen fejezet azt modellt és módszert mutatja be, melyet követve egy új feladat szisztematikusan, iteratív lépések sorozatával megoldható. Áttekintést kapunk arról a komplex folyamatról, és az azt támogató környezetről, amellyel a dolgozat eredményei egy nem informatikus képzettségű kutató számára is egyszerűen felhasználhatók a gyakorlatban. Ez fontos kérdés, mert minden egyes probléma más és más, ami miatt egyes feladatspecifikus részeket a konkrét probléma kutatójának kell elkészíteni. Így tulajdonképpen végleges rendszert maga a kutató állítja össze. Ehhez a fejezethez az alábbi tézis kapcsolódik:

6. Tézis. *Olyan alkalmazási modellt és környezetet dolgoztam ki, amely a függvénymodul és a konfigurációs állomány megadása után többlépcsős számítást, illetve elosztott hálózati környezetben több felhasználóval együttműködő interaktív megjelenítést tesz lehetővé.*

4.1. A probléma megoldásának folyamata

A teljes kutatási ciklus a matematikai modell felállításával kezdődik és az eredmények grafikus megjelenítésével és kiértékelésével, valamint dokumentálásával ér véget. A kezdet és cél között számos, esetenként többször is ismételt, iteratív lépés áll. A teljes problémamegoldási folyamatot egy olyan alkalmazási modellel írhatjuk le, melynek fő tevékenységei a következők:

1. Meg kell határozni az adott problémához a GRT koordinátáit és a peremfeltételeket biztosító függvények matematikai leírását.
2. El kell készíteni a feladatspecifikus függvénymodult, amit be kell építeni a PSA programba.
3. Le kell fordítani az új függvénymodult, és össze kell szerkeszteni a PSA program többi moduljával.
4. Elő kell állítani a feladatspecifikus konfigurációs állományokat.
5. Az elkészített és lefordított alkalmazást futtatni kell az adott párhuzamos környezetben.
6. Az eredményeket ki kell értékelni és dokumentálni kell oly módon, hogy azok reprodukálhatóak legyenek.

7. Az eredmények kiértékelése után a konfigurációs állomány egyes paramétereinek módosításával rendszerint újabb futtatásokat kell végezni (pl. fokozatos sűrítés), vagy esetleg módosítani kell a függvénymodult is. Ez utóbbi esetben értelemszerűen a 3. ponttól kell ismét végrehajtani a fenti tevékenységsorozatot.

Ennek az egyszerűnek tűnő tevékenységsorozatnak az elvégzését a következők nehezítik:

a.) A futtatás ill. egyes programrészek elkészítése párhuzamos és gyakran heterogén környezetben történik, ami kisebb-nagyobb mértékben eltér a napi gyakorlatban megszokott környezettől.

b.) A konfigurációs állományok elkészítésénél figyelemmel kell lenni az adott környezet adottságaira (memória, teljesítmény).

c.) Az eredmények nehezen kezelhető, nagyméretű szöveges állományban jelennek meg, melyek grafikus kiértékelése sem könnyű (2-10 dimenziós ponthalmaz).

d.) Az egyes iterációs lépések során számos futási eredmény keletkezik, melyet dokumentálni, archiválni kell.

A PSA program alkalmazása során szerzett tapasztalatok felhasználásával egy olyan környezetet definiáltunk, amely hatékonyan képes segíteni a PF-ok globális vizsgálatához kapcsolódó kutatói munkát. Meghatároztuk továbbá azokat a tevékenységeket, melyek számítógépes segítésével egy kényelmes környezet alakítható ki. Ezek három fő csoportba sorolhatók:

- *Számítást segítő keretrendszer, segédprogramok.*
  A számítások elvégzéséhez, azok előkészítéséhez és az eredmények előfeldolgozásához egy egységes keretrendszert definiáltunk, mely segédprogramjai elfedik a konkrét számítási környezet részleteit. A keretrendszer kialakításához olyan könnyen feldolgozható szöveges állományokat specifikáltunk, melyek öndokumentálóak, így hordozzák az ismételt előállításukhoz szükséges összes információt. A heterogén környezet támogatására alkalmas katalógusstruktúrát definiáltunk.

- *Eredmények interaktív megjelenítése.*
  Az eredmények kiértékeléséhez olyan grafikus rendszert definiáltunk amellyel nem csak az egyensúlyi út, hanem annak tetszőlegesen kiválasztott pontjához tartozó, a problémára jellemző fizikai alak is megjeleníthető.

- *Kutatói együttműködés támogatása.*
  A hatékony kutatói munka támogatására olyan környezetet definiáltunk, amely lehetővé teszi egymástól földrajzilag távol levő kutatók konkurens, interaktív együttműködését. Ennek lényege, hogy a képernyőn megjelenő információk mindkét helyszínen azonos időben és módon látszanak.

A módszer eredményességét mutatja, hogy a fenti koncepció alapján elkészített integrált rendszert az elmúlt időszakban számos probléma megoldására sikeresen alkalmaztuk különböző kutatási területek kutatóival együttműködve (ld. 5 fejezet). A továbbiakban az 1.3. alfejezetben ismertetett konkrét példa kidolgozásán keresztül mutatjuk be a fenti modell alapján elkészített rendszer fő elemeit. A példa részleteit is bemutató leírást nem felhasználói leírásnak szántuk, csupán a komplex feladat illusztrálásának.

4.2. Számítást segítő keretrendszer, segédprogramok

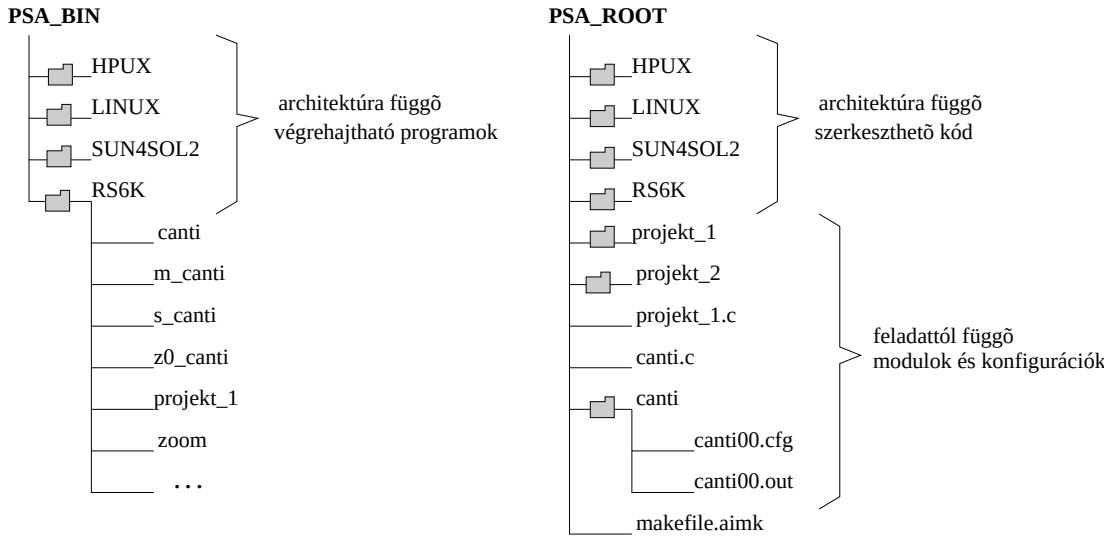
A GRT meghatározásától az eredmények előfeldolgozásáig tartó folyamatot támogató segédprogramok a feladatfüggő rész megadása után automatikusan generálódnak. Így a feladatfüggő rész megadására csak egyszer van szükség, illetve módosítása esetén a segédprogramok automatikusan módosulnak. Ilyen segédprogram pl. a fokozatos sűrítést és az eredmények előfeldolgozását segítő program, melynek működése az n dimenziós ponthalmazként előálló eredmények adott szempontok szerinti osztályozásán alapszik. A számítási folyamat főbb lépései következőkben foglalhatók össze:

1. Ki kell alakítani egy olyan katalógusstruktúrát, amely segíti az adott párhuzamos környezetben futó programrendszer komponenseinek paraméterezését, valamint a különböző gyakorlati feladatok adatainak, eredményeinek áttekinthető tárolását.
2. Meg kell határozni az adott problémához a GRT koordinátáit és a peremfeltételeket biztosító függvények matematikai leírását.
3. El kell készíteni a feladatspecifikus függvénymodult, melyet egy megadott vázra (*fxmodule.c*) kell ráépíteni.
4. Le kell fordítani az új függvénymodult, és össze kell szerkeszteni a PSA program többi moduljával. Ennek során előállnak a PSA rendszer futtatható moduljai.
5. Elő kell állítani a feladatspecifikus konfigurációs állományokat.
6. Elosztott rendszer esetén az 1. lépést minden gépen, heterogén rendszer esetén pedig ezen felül a 4. lépést minden architektúrán el kell végezni.
7. Elosztott rendszer esetén biztosítani kell azt is, hogy minden gépen elérhető legyen a 4. lépés során keletkező szolga végrehajtható kódja. (Ez olyan rendszerekben, ahol közös állományrendszer van, automatikusan biztosítódik.)
8. Az elkészített és lefordított alkalmazást futtatni kell az adott párhuzamos környezetben. A futtatást, a futtatások paraméterezését néhány egyszerű segédprogram támogatja, melyek automatikusan állnak elő függvénymodul fordításakor
9. Az eredmények (melyek egyszerű szöveges állományban állnak elő) kiértékelése, után a konfigurációs állomány egyes paramétereinek módosításával rendszerint újabb futtatásokat kell végezni (pl. fokozatos sűrítés), vagy esetleg módosítani kell a függvénymodult is. Ez utóbbi esetben értelemszerűen a 4. ponttól kell ismét végrehajtani a fenti tevékenységsorozatot.

4.2.1. Katalógusstruktúra

A programrendszerhez egy olyan katalógusstruktúrát terveztünk, amely hatékonyan támogatja a heterogén architektúrát, a kutatómunkát és egyszerre több különböző feladat vizsgálatára is alkalmas. A 4.1. ábrán megfigyelhető módon ez két főkatalógusból és további alkatalógusokból áll:

- A *PSA_BIN* katalógus tartalmazza a programok futtatható kódját, heterogén architektúra esetén minden architektúrára külön.
- A *PSA_ROOT* katalógus pedig a feladatspecifikus modulok forrásait és az egyes feladatokhoz tartozó konfigurációs állományok és eredmények katalógusait, valamint a programmodulok szerkesztéséhez, fordításához szükséges állományokat (itt is architektúra függően).



4.1. ábra. PSA program katalógusstruktúrája

Géptípus	IBM SP2
Architektúra (ARCH)	RS6K (homogén)
Parallel programozási környezet	PVM
Feladatkezelő	LoadLeveler
Közös állományrendszer	NFS
PSA telepítési helye (PSA_ROOT)	~/pvm3/psa
Futatható programok helye (PSA_BIN)	~/pvm3/bin/RS6K
A feladat függvénymodulja (MODUL)	canti

4.1. táblázat. Példához feltételezett környezet

A továbbiakban a 4.1. táblázat szerinti környezetet tételezzük fel.

4.2.2. Matematikai modell

A 2. tevékenységi pontban meghatározott feladat a PF leírásának matematikai részéhez és a módszer alkalmazhatóságához kapcsolódik, melynek részleteivel az elvek bemutatásán túl (ld. 1.5. alfejezet) a jelen dolgozatban nem foglalkozunk.

4.2.3. Függvénymodul

Alkalmazói szempontból a legfontosabb és felhasználói szempontból egyben legbonyolultabb számítástechnikai feladat a függvénymodul előállítása. Ez a C nyelven készített programrész tartalmazza az összes feladatspecifikus információt, amely 4 jól elkülöníthető részre osztható:

- konstans definíciók
  Ezekkel kell definiálni többek között a feladat dimenziószámát, a gépenként futtatandó szolga processzek számát, szakadásos függvények esetén a számítási szintek számát. Itt kell megadni a GRT-re és annak felosztására vonatkozó alapértelmezett konstansokat is.

- konfigurációs paraméterek definíciói
  Ezek értéke a konfigurációs állományban adható meg, melyeket a PSA program a program indításakor automatikusan beolvas.

- függvény definíciók
  A függvénymodulban 3 fajta függvényt kell definiálnia a felhasználónak:

  – A peremfeltételt adó 1.7 alakban megadott függvények (ezek zérushelyét keressük a GRT-ben).
  – Opcionális kiegészítő függvények. Ezek értéke a megtalált megoldáspontokban kiszámítódik és kiíródik az eredményeket tároló állományba.
  – Rekurzív feladat kiegészítő (g) függvényei, melyek zérushelye a rekurzívan megfogalmazott feladatok megoldási feltételeit szolgáltatják (ld. 3.6. alfejezetben).

  Programozás-technikailag mindhárom függvénycsoport egy-egy *void* típusú függvénnyel van megvalósítva. A függvények a paramétereket vektoros formában kapják ill. ilyen formában állítják elő a függvényértékeket. Így a GRT dimenziószámra nem kellett korlátozást bevezetni, az elméletileg tetszőlegesen nagy lehet.

- változó definíciók és kezdőértékeik
  Ez lényegében egy kötelező és nem módosítható része a függvénymodulnak. Ezen keresztül adódnak át a konstans definíciós részben definiált feladatspecifikus konstansok a program többi moduljának.

A függvénymodul előállításának rövid bemutatásához vegyük példaként az 1.3. alpontban ismertetett egyik végén befogott hosszú rúd problémáját $P = 0$ esetre. Az 1.3 összefüggés alapján a GRT 2 dimenziós, a feladat nem rekurzív, azaz pontosan DIM-1 explicit függvényünk van, és nincs kiegészítő függvényünk. Így a konstans definíciós rész a következő:

```
/*----------------------- Slaves ----------------------------*/
#define NSLAVE  1       /* Number of slaves per hosts */
#define NOLOCAL 1       /* No local slave(s) on local host */

/*----------------------- Dimension -------------------------*/
#define DIM     2       /* Number of dimensions */
#define LAYER   1       /* Number of layers */
#define FNNUM   1       /* Number of explicit functions */
#define FCNUM   0       /* Number of complementary functions */
```

A peremfeltételt az 1.8 függvény adja. Így a függvénymodulba egy olyan függvény kerül, mely az

$$f_1(\alpha'(0), Q) = \alpha'(L) = \int_0^L -\frac{Q}{EI}\cos(\alpha(s))ds$$

függvény értékét szolgáltatja, mely értéket a GRT minden pontjához, mint kezdeti feltételhez numerikus integrálással meghatározhatunk. A legegyszerűbb numerikus módszert alkalmazva a 4.1 összefüggés adódik:

$$\begin{aligned} \alpha''(s+\Delta s) &= \alpha'(s) + \alpha(s)''\Delta s = \alpha'(s) - \frac{Q}{EI}\cos(\alpha)\Delta s \\ \alpha'(s+\Delta s) &= \alpha(s) + \alpha(s)'\Delta s \end{aligned} \tag{4.1}$$

Ebből egyszerűen elkészíthető az 1.8 függvényt megvalósító kódrészlet, melyben célszerű az EI, L és ds konstansokat konfigurációs paraméternek választani. Így a konfigurációs paramétereket definiáló, valamint fenti függvényt megvalósító kódrészlet a következő:

```
/*--------------------- Parameters ---------------------------*/
#define NPAR    3       /* Number of parameters */
/*------------------------------------------------------------*/
#define EI params[0]    /* EI parameter */
#define L  params[1]    /* L parameter */
#define ds params[2]    /* ds parameter for the integration */
float params[NPAR] = {
        1,      /* default value of EI */
        1,      /* default value of L */
        0.1,    /* default value of ds */
};

/*---------------- Function for boundary condition -------------*/
#define ap0   p[0]      /* alpha'(0) */
#define Q     p[1]      /* Q */

void Fn(double *p, double *f, char *v, int layer)
{
double a, ap, s, dap;

a = 0;          /* alpha */
ap = ap0;               /* alpha' */
s = 0;          /* s */
do {
dap = - Q/EI * cos(a) * ds;
a += ap * ds;
ap += dap;
s += ds;
while (s < L);
f[0] = ap;              /* alpha'(L) */
v[0] = 1;               /* value is valid */
}
```

Példánkban nincs kiegészítő függvény, és szimplex függvények sincsenek, mivel a feladat nem rekurzív. Azonban ezeket is definiálnunk kell egy-egy üres függvénnyel. A teljes függvénymodult (*canti.c*) az A.1. melléklet tartalmazza, melyet a PSA program főkatalógusában (PSA_ROOT) kell elhelyeznünk. Ezután következhet a modul fordítása és szerkesztése.

4.2.4. Fordítás, szerkesztés

Az elkészített PSA programrendszer alapvetően heterogén környezetre készült. A különböző gépi architektúrákra történő fordítást a PVM rendszer *aimk* segédprogramjával oldottuk meg, amely a megfelelő környezeti változók beállítása után elindítja a *make* programot. Ez a függvénymodul lefordítása után előállítja az összes olyan programot, segédprogramot és alkatalógust, amely az új alkalmazás futtatásához kell. Jelölje *MODUL* az adott feladatot megvalósító függvénymodul nevét (.c kiterjesztés nélkül). Ekkor az aimk FX=MODUL parancs hatására a következő állományok és katalógusok jönnek létre:

- PSA_ROOT/MODUL - katalógus
  Ebben a katalógusban az egyes futtatások eredményei és konfigurációs állományai tárolódnak.
- PSA_BIN/MODUL
  Shell script a futtatások kényelmes indításához. A script feladata, hogy a konfigurációs állományban levő ütemezési és allokációs információkból előállítsa az adott párhuzamos számítási környezet job-leíró állományait, és elindítsa a PSA alkalmazást.
- PSA_BIN/m_MODUL
  Futtatható program, amely a mesterprogram kódját tartalmazza.
- PSA_BIN/s_MODUL
  Futtatható program, amely a szolgaprogram kódját tartalmazza.
- PSA_BIN/z0_MODUL
  Futtatható programállomány, amely a sűrítési algoritmus első lépését végzi.
- PSA_BIN/z1_MODUL
  Futtatható programállomány, amely a sűrítési algoritmus második lépését végzi.
- PSA_ROOT/l_MODUL
  Szöveges információs állomány, amely a modul elkészítésének tényét és pontos dátumát naplózza.

Példánk szerint a PSA_ROOT katalógusban kell kiadni az *aimk FX=canti* parancsot, melynek hatására a 4.2. táblázatban felsorolt állományok keletkeznek:

Hely/név	Típus
PSA_ROOT/canti	katalógus
PSA_BIN/canti	shell scripti
PSA_BIN/m_canti	mesterprogram
PSA_BIN/s_canti	szolgaprogram
PSA_BIN/z0_canti	program
PSA_BIN/z1_canti	program
PSA_ROOT/l_canti	naplóállomány

4.2. táblázat. Példafeladat fordítása során keletkező állományok (*aimk FX=canti*)

4.2.5. Konfigurációs állomány

A konfigurációs állomány egy egyszerű szöveges állomány, amely alapvetően a következő információkat tartalmazza:

1. A PF szimplex módszerrel történő megoldásához szükséges paraméterek:
   - GRT koordinátái, azok felosztási sűrűsége,
   - a peremfeltételt adó függvények paraméterei, számítási konstansai,
   - hibakorlátok.

2. A PSA program futtatásához, a feladatok szétosztásához, az ütemezéshez, terhelés-elosztáshoz szükséges paraméterek.

A fenti információkat nemcsak jellegük szerint, hanem felhasználási módjuk szerint is csoportosíthatjuk:

1. PSA program által közvetlenül felhasznált adatok ill. paraméterek.
   Ezek szintaxisa viszonylag egyszerű (kulcsszó = érték), melyet a PSA program mester modulja értelmez.

2. Az adott párhuzamos környezet feladatütemezője által felhasznált paraméterek, melyek szintaxisa erősen függ az adott környezettől. Ezek a mesterprogram számára megjegyzésként megjelenő sorok, így azok tartalma bármi lehet. A futtatást indító shell script azonban ezeket a sorokat feldolgozza, és annak megfelelően paraméterezi a feladat-kezelőt. Ezzel a megoldással csak egyetlen konfigurációs állományra van szükség, ugyanakkor a shell script az adott környezet igényeinek megfelelően könnyen módosítható.

A konfigurációs állományt a fordításkor létrehozott modul katalógusba kell elhelyezni *MODULxxx.cfg* néven, ahol *MODUL* a függvénymodul neve, *xxx* pedig tetszőleges karaktersorozat. A 4.2.1. pont feltételezései szerint vegyük példaként a következő egyszerű konfigurációs állományt *(canti00.cfg)*, melyben a '#' jellel kezdődő sorok megjegyzéssorok a mesterprogram számára. A futtatást indító shell script azonban a '# @' sorozattal kezdődő sorokból a feladat-kezelő (esetünkben a LoadLeveler) számára gyűjt össze paramétereket.

```
#
# canti00.cfg
#
# Parameters for LL must start with # @
# Other lines starting with # are comments
#
# @ wall_clock_limit = 1:00:0
# @ min_processors = 10
# @ max_processors = 50
#
FOLLOW = 0
EXTRAPAR 0 (EI) = 0.45
EXTRAPAR 1 (L) = 1
EXTRAPAR 2 (ds) = 0.0001

#ap0    P
-400    -5
400     35
800     40
```

A fenti konfigurációs állomány rövid elemzését kezdjük az első nem megjegyzéssortól. A konfiguráció előírja, hogy a feladatot letapogatással kell megoldani (FOLLOW = 0), valamint 0.45, 1.0 és 0.0001 értékeket definiál az EI, L, ds paramétereknek. Ezt követően definiálja a letapogatni kívánt GRT méretét (ap0=[-400, 400], P=[-5, 35]), és az egyes koordinátákon alkalmazott rács sűrűségét (800 ill. 40). Ez esetünkben azt jelenti, hogy az ap0 koordináta intervallumát 800 részre, a P intervallumot pedig 40 részre osztjuk. A # @ sorozattal kezdődő sorok a feladatkezelő számára maximum 1

órás futási időt, illetve minimum 10 és maximum 50 processzor allokálását írják elő. Ezekből a sorokból a feladat indítását végző script a feladatkezelőt paraméterezi. Ha az adott környezetben nincs feladatkezelő vagy annak nem adható időkorlát, úgy ezen sorok figyelmen kívül maradnak.

4.2.6. Futtatás

A konfigurációs állomány elkészítése után, a program futtatása a modulkatalógusból függvénymodul nevével azonos nevű shell script-tel indítható. A script egyetlen paramétere a konfigurációs állomány modulnév és kiterjesztés nélküli neve. Így példánkban az indítása a `canti 00` paranccsal történik, melynek hatására az indító shell script a `canti00.cfg` nevű konfigurációs állomány alapján előállítja a LoadLeveler feladatleíróját, és elindítja azt. Ez utóbbi a feladatkezelő megfelelő sorába irányítja a job-ot a megfelelő paraméterekkel.

4.2.7. Eredmények

A futás során három állomány épül:

- modulxxx.log
  Ez lényegében egy szöveges típusú naplóállomány, melyben az aktuális paraméterek mellett százalékosan kijelzésre kerül a már letapogatott térrészek aránya.

- modulxxx.out
  Ez szintén egy szöveges típusú állomány, melyben a táblázatos formában jelennek meg azok a megoldáspontok, melyek a hibaszűrési feltételnek eleget tettek. Az 1. fejezetben bemutatott szimplex algoritmusból adódóan a megoldáspontok egy-egy n dimenziós szakasz végpontjaiként állnak elő, melyek koordinátái az eredményeket tároló állomány két egymást követő sorában helyezkednek el. A koordináták mellett az adott pontban számított peremérték feltételt adó függvények, és a kiegészítő függvények helyettesítő értékei is szerepelnek. Statisztikai és naplózási célokra '#' jellel kezdődő sorokban a feladat paraméterei, és a futás során mért idők is az állományba íródnak.

- modulxxx.wrk Ebbe az állományba a mesterprogram rendszeresen elmenti a belső állapotát, melyből egy esetleges programhiba vagy rendszerleállás esetén a számítás a hibát megelőző állapotból újraindítható.

A futás végén egyes környezetekben a feladatkezelő, így a LoadLeveler is, elektronikus üzenetben értesít a feladat elvégzéséről. Példánk canti00.out állományának egy részlete a következő:

```
# Time: Mon Feb 11 17:49:30 2002 (3221223188)
# Config nr= 0
# Parallel Simplex Algorithm $Revision: 3.13 $
# Architectures=1 Processors=1 Slaves=1
# Follow=0,1 Ncubes=0 (3)
# Maximal number of solution points stored in slaves=500
# Dimension= 2 (800*40=32000)
# Sgrid =       0*0
# One cube =    316*316=99856
# Real space=
# -400,400; -5,35;
```

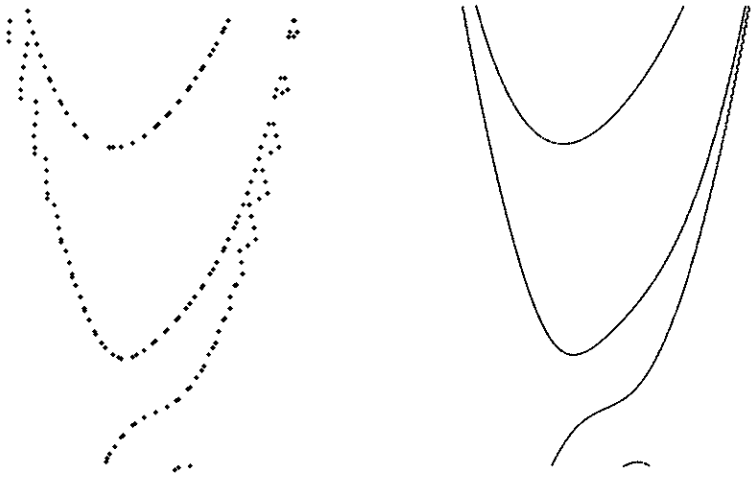
```
# Extra parameters: (3)
# 0.45  1       1e-04
# Form=err FcNum=0
# -1
# Delta:-1
# $Id: samplefx.c,v 1.3 2000/10/02 05:19:59 szebi Exp $
# FX Module for Cantilever beam
# ap0   P       f
6.59114 34.5911 -0.0456802
6.30851 34      -0.0450082
12      34.27   -0.0260502
12.082  34      -1.80448
12.117  34.117  -1.99491
12      34.27   -0.0260502
...
# All done.
#  tid=40052 (lapi      ) j=3     u=0 1:15.61  s=0 0:00.02  r=0 1:15.69
# Mtid=40051 (lapi      ) j=3        u=0 0:00.02  s=0 0:00.00  r=0 1:15.70
# Fxmax:        0
```

4.2.8. Eredmények pontosítása, fokozatos sűrítés

A 3.2. pontban leírt módszer szerint a felosztás fokozatos sűrítésével egyre pontosabb eredményekhez juthatunk anélkül, hogy a futási idő drasztikusan növekedne. Ehhez a durvább felosztással nyert eredmények feldolgozásával ki kell választanunk az eredeti GRT-ből azokat az altereket, amelyekben a durvább felosztás mellett eredményeket kaptunk. Ezt a feladatot segíti a feladatfüggő modul fordításakor automatikusan keletkező *z0_modul* és *z1_modul* program (*modul* a függvénymodul neve). (Példánkban ez a *z0_canti* és a *z1_canti*.) Ezek futtatását és paraméterezését a *zoom* script támogatja. Ez paraméterként kapja a feladatfüggő függvényeket tartalmazó modul nevét, a feldolgozandó eredményállomány azonosítóját, az eredeti konfigurációs állomány azonosítóját, valamint a sűrítési rács méretét (l_z), amit az eredmények osztályozásához használ.

A script eredménye egy olyan konfigurációs állomány, amely csak azon térrész koordinátáit tartalmazza, melyekben az osztályozás során esik megoldásszakasz. Az állomány formátuma olyan, hogy közvetlenül felhasználható a következő futtatáshoz, de alapértelmezett felosztásként az eredeti felosztásnak megfelelő értékek szerepelnek benne. Az új felosztás megválasztását a script futása során keletkező statisztikai információk segítik. A módszer hatékonyságát nagyban befolyásolja az eredmények osztályozására használt rács sűrűsége (l_z), ezért az új konfigurációs állomány előállítása néhány iterációs lépést igényel. Ez azt jelenti, hogy több különböző osztó értékkel kell futtatnunk a scriptet, hogy a statisztikai információk alapján ki tudjuk választani a legkedvezőbb rácsméretet. Minél kisebb osztót választunk ugyanis, annál több apró szegmensre esik szét a GRT, ami egyrészt kedvező, mert a kis térrészek gyorsan letapogathatók, ugyanakkor ha túl gyorsan befejezhető egy-egy térrész, akkor a mester-szolga modellből adódóan a mester terheltsége megnő, ami rontja a párhuzamos működés hatékonyságát.

Vizsgáljuk meg a 4.2.5. alfejezetben ismertetett konfiguráció felhasználásával előálló adatokat. A 4.2. ábra bal oldalán a futás eredményeként kapott adatokat ábrázoltuk. Látható, hogy az alkalmazott ritka rács miatt az eredmények hibája nagy, ugyankkor az is látszik, hogy a tér egy kis részében keletkeztek csak eredmények. Ezen részek sűrűbb letapogatásához elő kell állítani a következő lépes konfigurációját. A fentiek alapján a `zoom canti 00 00 4 > canti01.cfg` parancs hatására a `canti01.cfg` állományban a `canti00.cfg` egy olyan változata keletkezik, amely-



4.2. ábra. Canti00 és a sűrítéssel kapott eredmény

ben csak azok - a számításhoz használt rácsot alapul véve minden irányban 4 cellát tartalmazó - térrészek szerepelnek, amelyekbe esik megoldáspont. A scriptet lefuttatva a következő statisztikai információkat kaptuk:

```
# z0_canti: cfg:canti00.cfg hdr:canti00.hdr inp:canti00.out mul:1.00 div:4.00
-400    400     800     200
-5      35      40      10
Magnified/original/reduced space: 32000/32000/2000
Points: 560
# z1_canti: cfg:canti00.cfg inp:- mul:1.00 div:4.00
-400    400     800     200
-5      35      40      10
Nr of new domains= 38
Original/reduced nr of cubes= 32000/608 (1.90%)
Magnified/reduced nr of cubes= 32000/608 (1.90%)
Original/suggested nr of cubes= 32000/31958 (99.87%)
Suggested new resolution/domains: 29 (accumulated mul = 7.25)
```

Ebből az olvasható ki, hogy a 800x40=32000 cellából összesen 608-ba esett megoldás. Ezek a $l_z = 4$ osztású rács 38 különböző cellájába esnek. A program javaslatot tesz, hogy a kialakult térrészeket az új számításnál minden irányban 29 részre osszuk fel. Így a futási idő várhatóan hasonló lesz, mint az előző futtatáskor. Ennek megfelelően keletkező `canti01.cfg` állomány egy részlete a következő:

```
# z1_canti: cfg:canti00.cfg inp:- mul:1.00 div:4.00
....
#ap0    P
-400    -5
400     35
800     40
# z0_canti: cfg:canti00.cfg hdr:canti00.hdr inp:canti00.out mul:1.00 div:4.00
-16.00000 31.00000
-12.00000 35.00000
4       4
```

```
NEXTCUBE=
-12.00000 11.00000
-8.00000 15.00000
NEXTCUBE=
-12.00000 15.00000
-8.00000 19.00000
```

Feladat	Felosztás	Altér	T_1 [s]	T_1^z [s]	S_1^z
canti00	800x40	3	76		
canti01	38x32x32	38	105	181	25.85
canti02	6400x320	42	4678		

4.3. táblázat. Canti00 példa sűrítése során elért sebességnövekedés

Az előállított új konfigurációs állományban a `# z1_canti:` és a `# z0_canti:` sorok között az eredeti konfigurációs állomány áll változatlan formában. Ezt követi az első térrész koordinátája, majd az a felosztás, amivel számolva az eredeti felosztáshoz jutnánk. Jelen estben ez 4-4 mindkét irányban. Ezt kell a statisztikai információk alapján módosítni, amit 32-re írtunk át a javasolt 29 helyett, azaz minden irányban az eredetinél 8-szor sűrűbb rácsot (6400x320) definiáltunk. Ez nagyobb számítási feladatot jelent a javasoltnál, de a kis dimenziószám miatt ez az eltérés még nem okoz lényeges futási idő növekedést. Az újabb futtatás során előálló eredményeket a 4.2. ábra jobb oldalán figyelhetjük meg. Amint az várható volt, az eredmény pontosabb lett. A 4.3. táblázatban összefoglaltuk a futás során mért időket. Ebben feltüntettük a megnövelt felosztással a teljes térre elvégzett letapogatás valós futási idejét is. (A mérést egyprocesszoros környezetben 850Mhz-es Celeron processzorral végeztük.) Mérési eredményeink alapján $S_1^z = 25.85$ értéket kaptunk. Ennek a magas értéknek az a magyarázata, hogy az eredetileg felvett GRT nagy része nem tartalmaz megoldáspontokat.

4.2.9. Eredmények előfeldolgozása, szűrés

A PSA program működéséből adódóan egyes pontok többszörösen is megjelenhetnek az eredmény állományban. Ennek oka az, hogy a szomszédos szimplexeken áthaladó megoldás mindekét szimplex oldalán meghatározásra kerül. A többszörösen megjelenő pontok az amúgy sem kis méretű állományok méretét feleslegesen növelik. A *dxconf* segédprogrammal az eredménypontok osztályozhatók, és az azonos, vagy nagyon közeli pontok elhagyhatók. A program eredménye egy önleíró szövegfile, ami minden információt tartalmaz ahhoz, hogy az elvégzett számítás reprodukálható legyen.

4.3. Az eredmények interaktív megjelenítése

A sikeres probléma megoldáshoz elengedhetetlen az eredmények grafikus kiértékelése. Ennek oka alapvetően az, hogy a keletkező nagy mennyiségű adat szövegesen aligha tekinthető át, legfeljebb néhány alapvető jellemző (minimum, maximum) lenne ily módon kezelhető. A grafikus kiértékelés nagy előnye, hogy könnyebben felfedezhetők benne az esetleges szabályosságok vagy éppen rendellenességek, míg ez egy szövegesen megjelenő számhalmazban lényegesen nehezebb feladat. A grafikus megjelenítést nehezíti, hogy az esetek nagy részében sokdimenziós ponthalmaz az eredmény, aminek csak 2 dimenziós vetületét tudjuk grafikusan megjeleníteni.

A gyakorlati feladatok megoldása során szerzett tapasztalatink alapján meghatároztuk a grafikus megjelenítőkkel szemben támasztott legfontosabb követelményeket:

- Alkalmasnak kell lenni nagy mennyiségű, sok dimenziós (2-10) adathalmaz gyors megjelenítésére.
- Biztosítson lehetőséget a bemenő adatok különböző szempont szerinti szűrésére.
- Tegye lehetővé több ponthalmaz egymásra vetítését.
- Tegye lehetővé a kép forgatását, nagyítását.
- Tegye lehetővé, hogy a képernyőn megjelenő ponthalmazból interaktívan választani lehessen.
- Tegye lehetővé a kiválasztott ponthoz tartozó fizikai alak azonnali megjelenítését.
- Tegye lehetővé megjelenő grafikus ábra szabványos grafikus állományba történő elmentését.

Ezek figyelembe vételével 3 különböző megjelenítőt vizsgáltunk meg részletesen: Gnuplot, Gloves, IBM DX.

4.3.1. Gnuplot alkalmazása

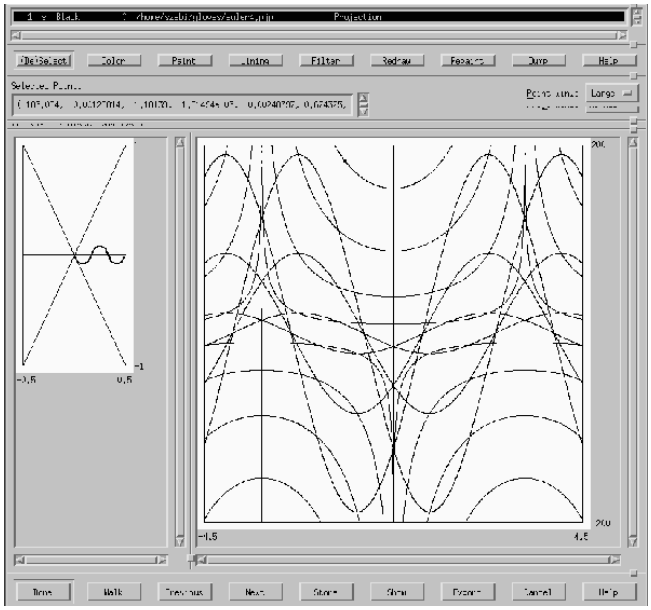
Egyszerű, gyors megjelenítést tesz lehetővé a Gnuplot [54] elnevezésű interaktív rajzoló program. A program több platformon is elérhető. Egyszerű, C nyelvhez hasonló kifejezéseivel könnyen állíthatók be szűrő feltételek az input adatokhoz. Nagy mennyiségű adatot is gyorsan megjelenít. Hátránya, hogy nem teszi lehetővé az interaktív pontkiválasztást. Ennek ellenére gyors ellenőrzésekre kitűnően alkalmas. Előnye, hogy segítségével a képernyőn megjelenő grafikus kép számos grafikus formátumban elmenthető. A program közvetlenül képes olvasni a PSA program eredmény állományait.

4.3.2. Gloves alkalmazása

Az eredmények kiértékeléséhez vezetésemmel egy Gloves [47] elnevezésű megjelenítő program készült, amely tetszőleges dimenziószámú ponthalmaz kiválasztott vetületeit meg tudja jeleníteni. A program közvetlenül a PSA eredményformátumát olvassa. Egyes esetekben ez az adathalmaz lényegesen részletesebb, mint ahogyan az megjeleníthető. A felhasználó ezért különböző szűrési feltételek definiálása mellett kiválaszthatja azokat a koordinátákat, melyek 2D vetületét meg akarja jeleníteni. A kiválasztott koordináták mellett lehetőség van egy harmadik koordináta kiválasztására is, ami a megjelenített pont színét befolyásolja. Ez felhasználható pl. arra, hogy az eltérő pontosságú pontokat eltérő színek jelöljék, vagy arra, hogy az eredményállományban tárolt az ún. kiegészítő függvények értékeit is kijelezzük. Egy ponthalmazhoz több ilyen vetítési összerendelés is definiálható, így a megjelenítéskor ezek egymásra vetíthetők.

A kicsinyítéssel, nagyítással, forgatással megjeleníthető adatok közül a felhasználó kiválaszthat egy tetszőleges pontot, melynek GRT-beli koordinátáit a program visszakeresi[1] és kijelzi. Mivel az így kapott értékek egyértelműen meghatározzák a KF-t, ennek alapján az adott ponthoz tarozó fizikai alak is egyértelműen előállítható és felrajzolható. Ezt a Gloves program úgy támogatja, hogy

[1]Magasabb dimenziószám esetén több pont vetülete egybeeshet. Ilyen esetben véletlenszerű, hogy melyik pontot választjuk. A vetítés irányának megválasztásával azonban a pontok szétválaszthatók.



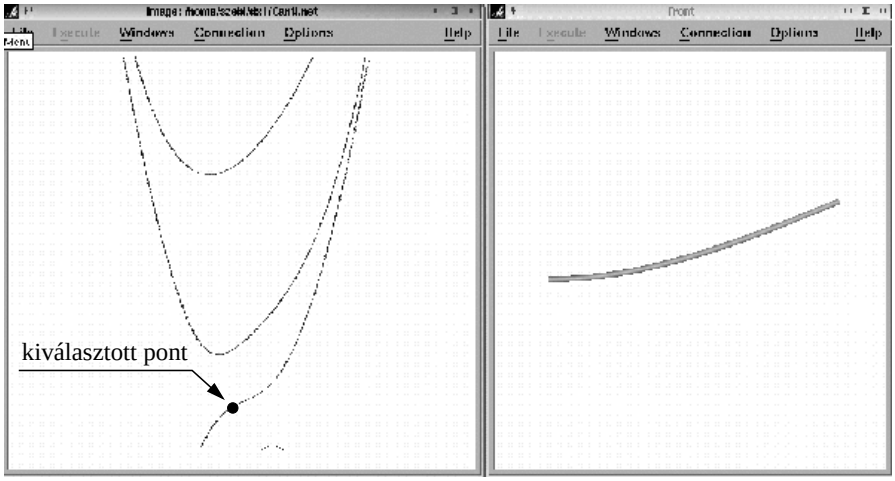
4.3. ábra. Gloves program alkalmazása az Euler feladatra

a kiválasztott ponttal, mint paraméterrel képes egy megadott, külön modulban elhelyezett függvényt meghívni. Ez a függvény az adott ponthoz rendelhető, a fizikai feladatra jellemző (pl. alak) 2 dimenziós pontsorozatot egy jól definiált felületen adhatja vissza, amit a Gloves egy külön ablakban felrajzol. Maga a függvény a PSA program függvénymoduljából könnyen előállítható. Több ilyen függvény is megadható a Gloves megfelelő moduljában, melyek közül a program futtatásakor menü segítségével lehet választani. Ez a lehetőség különösen szemléletes a mechanikai alkalmazások esetében, de más problémák, pl. DNS láncok alakjának megfigyelésére is jól alkalmazható. A 4.3. ábra a program egy alkalmazását szemlélteti. Az implementált program a függvények beépítésének csak azon lehetőségét támogatja, miszerint a programot, és a függvényt statikusan össze kell szerkeszteni. Ez a későbbi felhasználásban komoly hátránynak mutatkozott.

4.3.3. IBM DX rendszer alkalmazása

Az eredmények grafikus megjelenítéséhez legeredményesebben az IBM Data Explorer [36] elnevezésű grafikus eszközét alkalmaztuk. Ez az eszköz, annak ellenére, hogy több évvel ezelőtt fejlesztették, igen korszerű és hatékony eszköznek bizonyult. A rendszer dataflow elven programozható és objektumorientált elven működő külső modulokkal bővíthető. A DX az IBM üzleti szoftverei között sajnos már nem szerepel, talán ez az alapvető oka annak, hogy kevésbé elterjedt, Európában szinte alig ismert. Jelenleg a program ún. Open Source és egy viszonylag kis létszámú fejlesztőgárda fejleszti ill. portolja a különféle platformokra. Az Open Source változat manapság már sokkal ismertebb, mint az eredeti.

A rendszert kifejezetten nagy mennyiségű adat megjelenítésére fejlesztették, elsősorban fizikai kísérletek és szimulációs programok eredményeinek kiértékelésére. Külön erőssége a programnak, hogy a megjelenítés olyan grafikus ablakokban történik, melyekben a különféle grafikus transzformációk (zoom, pan, rotate, stb.) automatikusan rendelkezésre állnak, és azokhoz nem kell extra programrészeket készíteni. Egyszerű használhatósága és rugalmas programozhatósága miatt könnyen elkészít-



4.4. ábra. Mintafeladat vizsgálata a DX rendszerrel

hettünk egy olyan DX programot, amely a 4.3.2. alpontban bemutatott Gloves program funkcióit megvalósítja. Ez a program szintén alkalmas arra, hogy a bifurkációs diagram egy kiválasztott pontja alapján a tényleges fizikai feladatot megjelenítse. Ehhez egy olyan függvény beépítésére van szükségünk, amely a KF megoldását adja, és a megoldás során generálja a fizikai feladat geometriáját jellemző pontokat (pl. a rúd alakját leíró pontokat). A Gloves programmal ellentétben az így előálló pontok nem csak 2 dimenziósak lehetnek, hanem a tényleges fizikai modellnek megfelelőek.

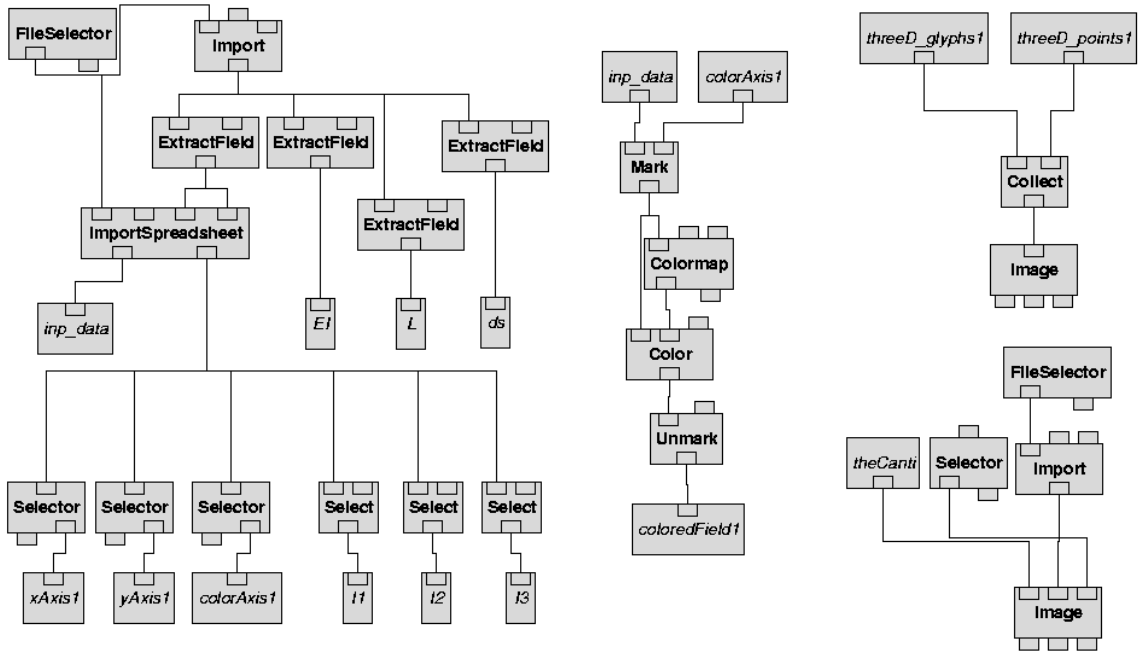
Felhasználói függvény beépítésére a DX 3 lehetőséget biztosít:

- Külső programmal, mellyel állományokon keresztül kommunikál.
- Külső programmal, mellyel valamilyen processzek közötti kommunikációs eszközzel (pl. pipe) kommunikál.
- A DX rendszerbe futás közben betölthető ún. loadable modullal. Ezzel úgy kapcsolhatók felhasználói függvények a DX rendszerhez, mintha azok a rendszer szerves részét képeznék.

A hatékonyabb megvalósítás miatt ez utóbbi lehetőséget választottuk. A megfelelő modul elkészítéséhez a DX rendszer egyszerűen kezelhető segédprogramot tartalmaz (builder), amellyel a modul interfésze könnyen definiálható, majd a definíciók alapján a program C nyelven egy keretprogramot generál. Ebbe kell beépíteni a felhasználói függvényt, ami esetünkben a PSA program függvénymoduljából előállítható. Ahogyan már utaltunk rá az előző pontban, a KF megoldó függvény ugyanis része a PSA program függvénymoduljának, csupán nem generálja, egyes esetekben csak nem tárolja el a megoldás során a geometriát jellemző pontokat.

Kézenfekvőnek tűnt, hogy a PSA függvénymodulját eleve úgy alakítsuk ki, hogy illeszkedjen a DX rendszerhez is. Ezt a C nyelv feltételes fordítási opciójának felhasználásával valósítottuk meg. Az A.1. melléklet egy ilyen módon felépített függvénymodult tartalmaz. Ebben megfigyelhetők azok a `#ifdef DX` és `#ifndef DX` alakú sorok, melyek a feltételes fordítást vezérlik. Ezzel a megoldással a PSA függvénymodul forrásából közvetlenül, a megfelelő fordítási opció felhasználásával azonnal előállítható a grafikus rendszer betölthető modulja.

A 4.4. ábra a bemutatott rúd viselkedésének vizsgálatát szemlélteti a DX rendszerhez készített függvénymodul felhasználásával. A bal oldali ábrán a sűrítéssel nyert bifurkációs diagram, a jobb

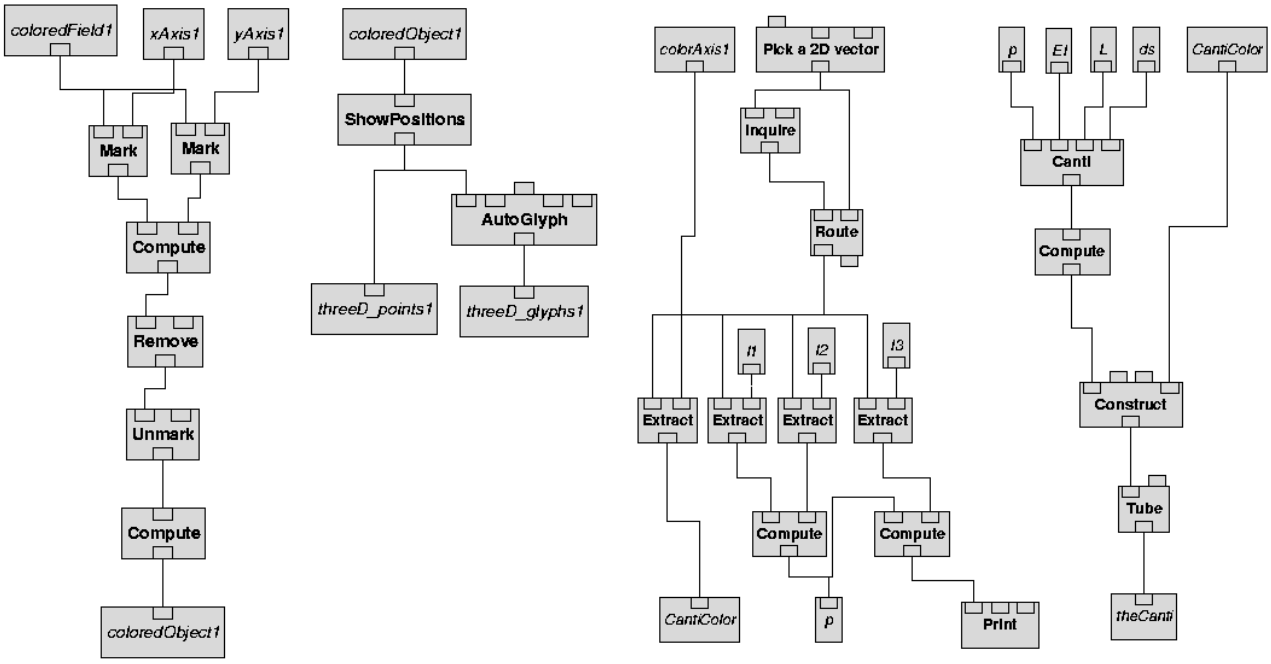


4.5. ábra. DX program: Adatok beolvasása és a megjelenítés

oldali ábrán pedig annak egy kiválasztott pontjához tartozó alak látszik. A modul használatára, egyben a DX hatékony programozására mutat példát a 4.5., valamint a 4.6. ábra. A két ábra a feladathoz készített teljes DX programot tartalmazza. Az ábrákon jól látható, amint az adatfolyam-elvnek megfelelően a beolvasott adatok az egyes modulokon végighaladva eljutnak a megjelenítésig. A 4.6. ábra jobb oldalán található a `Canti` modul, amely a korábban ismertetett módon, a megoldó függvénnyel együtt áll elő. Ez paraméterként kapja a GRT kiválasztott pontjának koordinátáit (p), valamint a konkrét feladat megoldásához felhasznált konfigurációs paramétereket (EI, L, ds). Ez utóbbiakat beolvasó programrész (ld. 4.5. ábra bal oldalán) olvassa be PSA program kimeneteként előálló állományból, egészen pontosan ennek egy előfeldolgozott változatából. Az előfeldolgozást a `dxconv` script végzi. Ez egyetlen állományba csomagol minden olyan információt, ami később a grafikus megjelenítéshez kell. A script a csomagoláson kívül egyszerűbb szűrést is végez, ugyanis megadható, egy a fokozatos sűrítésnél használt rácshoz (ld. 3.2. alpont) hasonló rácsméret (l_z), mely alapján a script osztályozza az eredménypontokat, és minden cellából csak egy pontot hagy meg.

A DX rendszert mind a DNS molekulák mechanikai modellezése során (ld. 5.4. alfejezet), mind a folyadékhidak egyensúlyi vizsgálata során (ld. 5.2. alfejezet) eredményesen alkalmaztuk. Mindkét feladatnál szükségessé vált, hogy a megoldáspontok közül kiválasztott kezdeti feltételeknek megfelelő fizikai elrendezést (a folyadékhíd alakját, ill. a DNS gyűrű alakját) külön ablakban megjelenítsük. Ehhez mindkét alkalmazásnál a fent leírt módon elkészítettük a megfelelő betölthető modulokat.

A DX rendszer alkalmazásával szerzett tapasztalatok alapján, a rendszerhez egy olyan program-modult definiáltam, amely a továbbiakban egyszerűen felhasználható a feladatspecifikus programrészek automatizált elkészítésére.



4.6. ábra. DX program: A kép-objektumok előállítása és a Canti modul

4.4. Kutatói együttműködés támogatása

Az eredmények kiértékelésében gyakran több, esetleg egymástól földrajzilag távol levő kutató együttműködésére van szükség. Ezért elsősorban az eredmények grafikus megjelenítésével kapcsolatban felmerült az az igény, hogy több helyszínen azonos időben kellene látni a megjelenítő képernyőjét úgy, hogy az egyes kutatók konkurensen be tudjanak avatkozni a megjelenítésbe (forgatás, kicsinyítés, adott pont kiválasztása).

A megvalósítását tekintve ez a lehetőség lehet magának a megjelenítő programnak a speciális szolgáltatása is, de mi más megoldást választottunk. A teljes rendszer megvalósításában törekedtünk a nyílt rendszerelemek felhasználására. Ezért a teljes rendszer, beleértve a megjelenítést is egyszerűen összeintegrálható más nyílt rendszerrel. Így pl. olyan grafikus környezettel, mint a VNC (Virtual Network Computing) programcsomag [42], amely támogatja a többfelhasználós, konkurens megjelenítést is. A VNC lehetővé teszi távolsági hálózaton keresztül is a közös interaktív munkát. Ez azt jelenti, hogy azonos képernyőképet láthatunk akár több ezer Km-re levő munkaállomásokon. A rendszer alkalmazásához a kliens oldalon nem szükséges speciális programot telepíteni. Egy egyszerű böngésző is elegendő használatához. Így a számítások grafikus eredménye szinte bárhonnan megtekinthető pl. egy NoteBook-ról, egy PDA-ról, vagy akár egy repülőtéri váróterem nyilvános böngészőjéről.

5. fejezet

A programrendszer gyakorlati alkalmazásai

A kifejlesztett programokat illetve a 3. fejezetben bemutatott algoritmusokat elmúlt évek során számos probléma vizsgálatához alkalmaztuk, melyek eredményeit számos cikkben és előadásban publikáltuk. Az 5.1. táblázatban összefoglaltuk ezeket az alkalmazott módszer és a feladat dimenziószáma alapján. A továbbiakban ezen alkalmazások közül mutatunk be röviden néhányat a teljesség igénye nélkül.

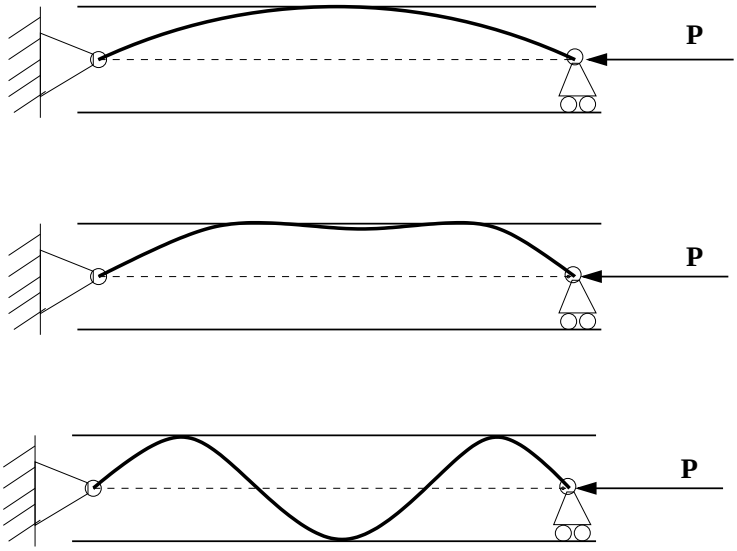
DIM.	2	3	4	5	6	7	8	9	10
PSA	[21, 25], [17, 18]	[12, 33], [20]	[15, 28], [39]	[34]					
SUA		[34, 38]	[34]	[19, 34]	[34]	[14, 34]	[27, 34]	[27, 34]	[27]
PHA					[16]				

5.1. táblázat. Alkalmazások összefoglalása

5.1. Gátolt érintkezési feladatok

Eredményesen alkalmaztuk a programrendszert vékony, rugalmas szálak viselkedésének vizsgálatához. A probléma a gyakorlatban olyan polipropilén szálak előállításához kapcsolódik, melyeket nem szövött anyagok gyártásához használnak fel. Az 5.1. ábra a feladat egyszerűsített mechanikai modelljét szemlélteti. A tengelyvonaltól azonos távolságra levő rögzített falak között az egyik végén csuklós befogású, a másik végén görgős alátámasztású rudat P erővel nyomjuk. A nyomóerő nagyságától függően a rúd kihajlik, és egy vagy több pontban a falnak támaszkodik. Könnyen belátható, hogy a a kihajlások száma és nagysága alapvetően meghatározza a gyártott anyag tulajdonságait.

A feladat matematikai modellje a támaszkodási pontoktól függően 3 vagy többdimenziós GRT segítségével írható le. Az elkészített program PSA algoritmust megvalósító változatát 7 dimenzióig alkalmaztuk. Tekintve, hogy ezzel a feladattal 1997-98-ban foglalkoztunk, a mai lehetőségekhez képest lényegesen kisebb számítási kapacitások álltak a rendelkezésünkre. A magasabb dimenziószámú feladatokhoz szinte minden elérhető erőforrást be kellett vonnunk. Az így kialakult igen heterogén rendszerben a PVM jó választásnak tűnt. A rendszer heterogenitásának és elosztottságának bemutatására tekintsük az egyik 1997 őszén futtatott 5 dimenziós feladat naplóállománya



5.1. ábra. Gátolt érintkezési feladat

alapján készített 5.2 táblázatot. A feladat megoldásában 6 különböző hardver architektúrájú gép vett részt, melyek helyileg sem ugyanott voltak. A legtávolabbi gép az IIF-ben, mintegy 15 Km-re volt az egyetemtől. A feladatban a tér minden irányban 50 részre volt felosztva (50^5 rácspont). A teljes futási idő 3 óra 10 perc volt úgy, hogy az egyik gép az első 10 percben kiesett. Így összesen 32 processzor számolta végig a feladatot. A táblázat utolsó oszlopa a konkrét feladat megoldásában nyújtott teljesítmény nagyságára utal géptípusonként csoportosítva. Egységnek a MIPS ill. a VUP mértékegységéhez hasonlóan a DEC VAX-780-as gép teljesítményét vettük, ami nagyon közel állt VAX 3800-as teljesítményéhez.

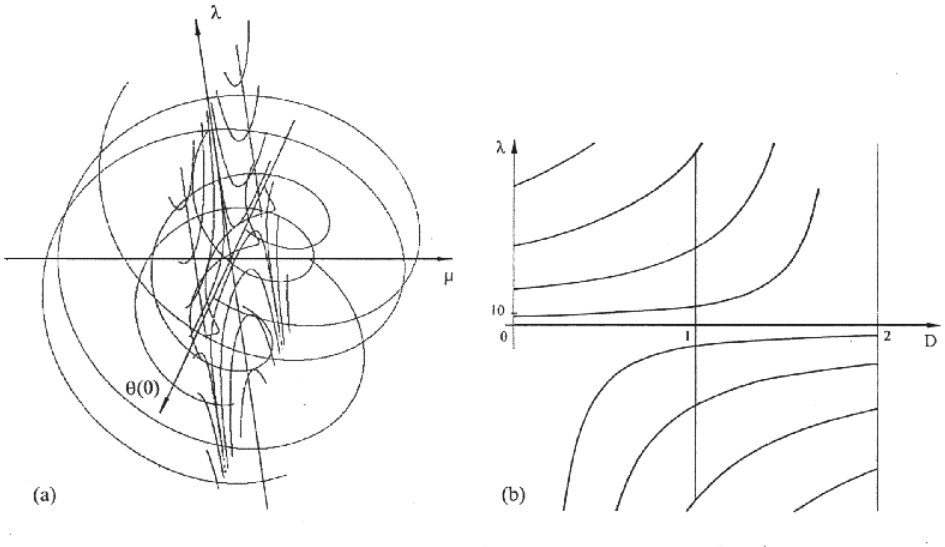
A vizsgálati eredményeinkből, mely alapján több cikk ill. konferencia-előadás készült, a következőkben a teljesség igénye és részletesebb magyarázat nélkül bemutatunk néhány érdekes ábrát. Ezek részletes kiértékelése megtalálható a [34] cikkben.

Az 5.2. ábra olyan konfiguráció egyensúlyi útját mutatja, melynek nincs érintkezési pontja a fallal. Az 5.3. ábra esetében, az 5.1. ábra felső elrendezéséhez hasonlóan csak egy ponton érintkezik a rugalmas rúd a fallal. A letapogatás eredménye láthatóan sok parazita megoldást tartalmaz. Ennek oka az, hogy a feladat egy érintkezési pont esetében is már csak 5 dimenziós térrel írható le, így meglehetősen ritkán felosztott ráccsal kellett számolnunk ahhoz, hogy reális futási időket kapjunk. A feladatok megoldásánál ezért rendszeresen használtuk a fokozatos sűrítés algoritmusát (3.2 alfejezet). Az 5.3. ábra felső részén ritkább ráccsal számolt eredmények, az alsó ábrán pedig a sűrítés eredménye látható. Megjegyezzük, hogy ekkor még nem volt implementálva sem a párhuzamos útkövetés (PUA), sem a hibrid (HSA) módszer. A letapogatással nyert eredményeket ezért gyakran egyszerű útkövetéssel tovább finomítottuk, az 5.4. ábrán ennek eredménye látható. A számozott feliratok a rúd jellegzetes geometriai alakját jelölik, mely alakokat a korábban ismertetett vizualizáló eszközzel (4.3.2. alfejezet) ellenőriztünk.

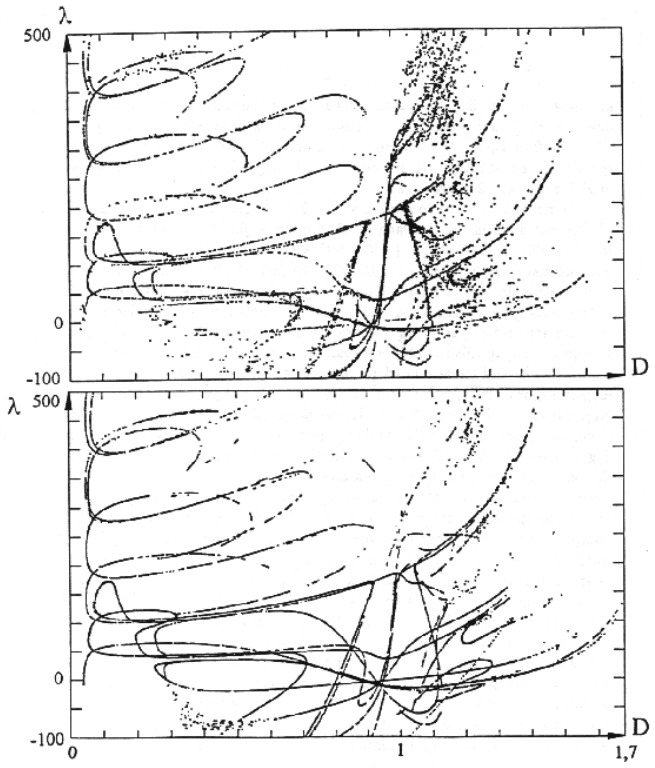
Típus/Modell	Architektúra	Darab	Processzor	Hely	Telj.
IBM RS6000	RISC-6000	2	1	FSZ, TTT	4
HP 9000	PA-RISC	6	1	FSZ, ME	1
IBM SP2	RISC-6000	1	8	ELTE	20
DEC VAX-3800	DEC VAX	1	1	FSZ	1
IBM PC	Intel Pentium	1	2	FSZ	10
IBM PC	Intel Pentium	1	4	FSZ	27
DEC ALPHA	Alpha	1	1	FSZ	5
SUN SUN-Ultra	Ultra-Sparc	1	2	IIF	6
SUN SUN4	Sparc II	2	2	HSZK, FSZ	5
SUN SUN2	Sparc II	3	1	VGT, VMT, SCH	3

ELTE - ELTE Számítógépközpont
FSZ - BME Folyamatszabályozási Tanszék (ma IIT)
IIF - Információs Infrastruktúra Fejlesztési Iroda (ma NIIF)
SCH - BME Schönherz Zoltán Kollégium
TTT - BME Távközlési és Telematika Tanszék
VGT - BME Villamos Gépek és Hajtások Tanszék
VMT - BME Villamosművek Tanszék

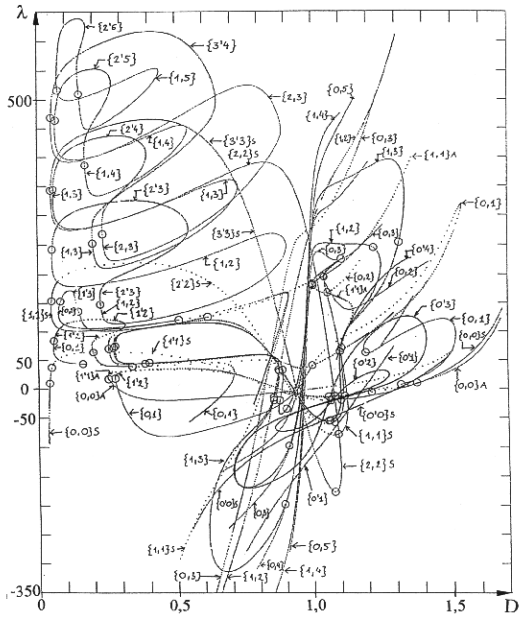
5.2. táblázat. Érintkezési feladatok számításához használt heterogén környezet



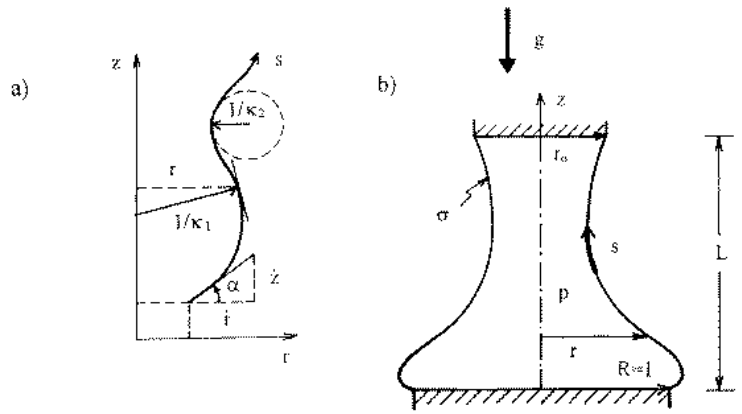
5.2. ábra. Mindkét végén csuklóval ellátott rúd egyensúlyi útja (nincs érintkezési pont)



5.3. ábra. Csuklós rúd egy érintkezési ponttal ($h = 0.125$)



5.4. ábra. Útkövetéssel finomított eredmények (egy érintkezési pont $h = 0.125$)



5.5. ábra. Folyadékhidak geometriai jellemzői; (a) KF; (b) PF

5.2. Folyadékhidak egyensúlya

A Cornell Egyetemmel kialakult együttműködés során folyadékhidak alakját vizsgáltuk a PSA módszerrel. Kapilláris méretű folyadékhidak alakját alapvetően a kialakuló felületi feszültség határozza meg. A feladat alapösszefüggéseit a Young-Laplace egyenletek adják. Ha a gravitáció hatásától eltekintünk, akkor a feladatot a következő KDE írja le (5.5. ábra):

$$\begin{aligned} \alpha'(s) &= p - \sin(\alpha(s))/r(s) \\ r'(s) &= \cos(\alpha(s)) \\ z'(s) &= \sin(\alpha(s)) \end{aligned} \tag{5.1}$$

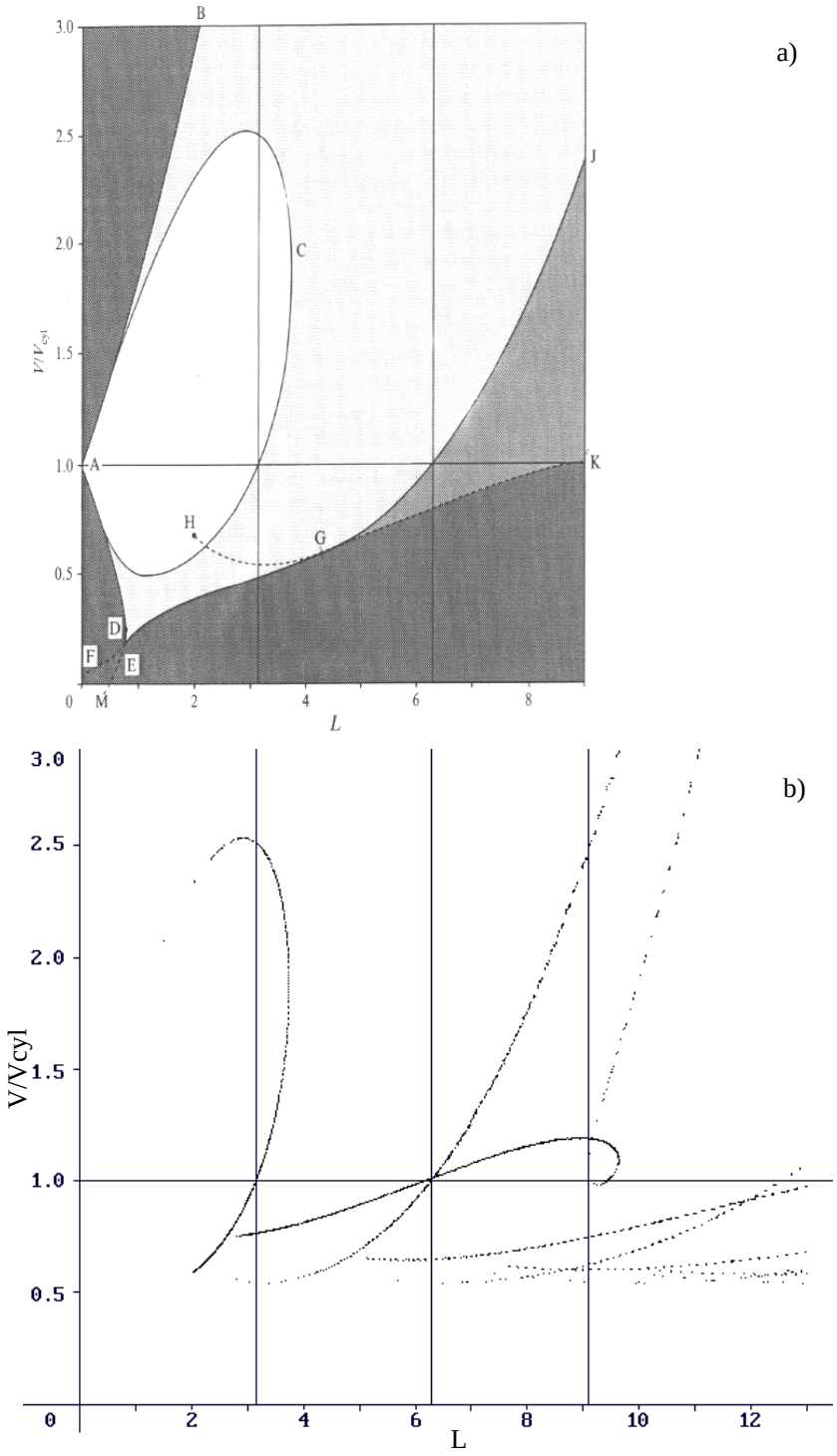
ahol:

- r - a szimmetriatengelytől mért sugár
- p - a folyadék belsejében mért nyomás
- $\alpha(s)$ - folyadék felülete és a sugár által bezárt szög az ívhossz függvényében $s \in [0, L]$

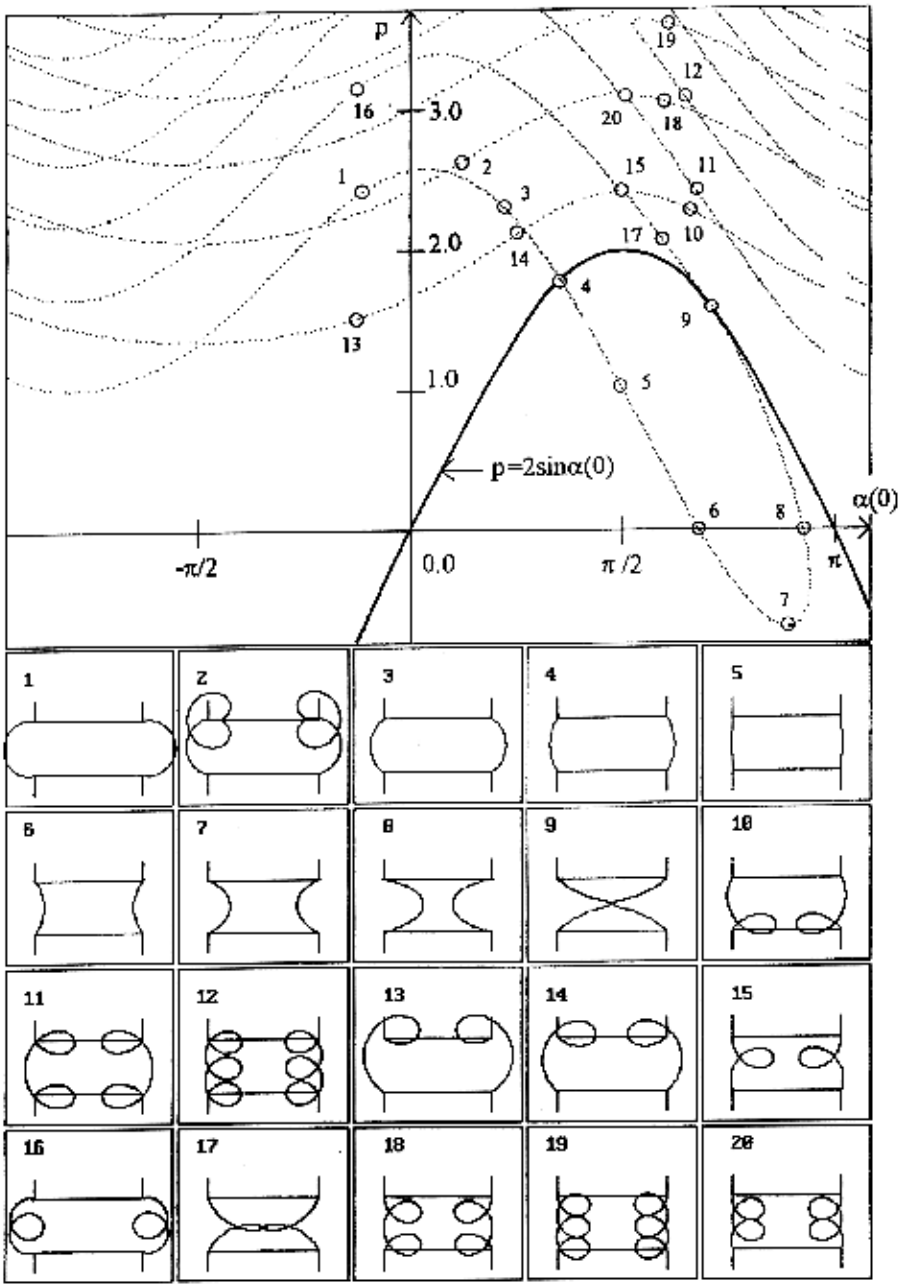
Az 1.5. alfejezet alapján a 2 dimenziós GRT-t a $[\alpha(0), p]$ változók feszítik ki. A peremfeltételt a $r(0) = r|_{z=L} = 1$ kifejezés adja. Ebből a peremfeltételt megfogalmazó függvényre az 5.2 összefüggés adódik:

$$f_1(\alpha_0, p) = r|_{z=L} - 1 = 0 \tag{5.2}$$

A fentiek alapján már elkészíthető a megfelelő függvénymodul, és a feladat futtatható. A vizsgálat során azonban nemcsak egy adott L mellett kialakuló alakra voltunk kíváncsiak, hanem a nyomás, ill. a térfogat lokális szélsőértékének alakulására is az L függvényében, azaz egy többparaméteres feladatról van szó, melynek megoldását a 3.6. alfejezetben ismertetett rekurzív algoritmussal végeztük el. A 2 dimenziós GRT-t kiegészítettük az L változóval, és meghatároztuk a 2. függvényt, amely a szélsőérték feltételt fogja adni, azaz a $\frac{\partial p}{\partial \alpha_0} = 0$ helyet keressük az L függvényében. Az így felépített 3 dimenziós feladatot a Cornell Egyetem SP2 számítógépén futtattuk. Az eredmények megfeleltek az ottani kutatócsoport korábbi eredményeinek, sőt a globális módszernek köszönhetően újabb, eddig nem ismert megoldásokat is sikerült megtalálnunk. A korábbi eredmények és a programunk által számított eredmények összehasonlítását az 5.6. ábra segíti.



5.6. ábra. A térfogat minimumának változása az L paraméter függvényében. (a) korábbi eredmények [37], (b) rekurzív PSA programmal számított eredményeink



5.7. ábra. Folyadékhidak stabilitási görbéje és alakja

Az eredmények megjelenítéséhez a DX vizualizáló rendszert használtuk. A folyadékhidak alakjának megjelenítéséhez a 4.3.3 alfejezetben ismertetett módon PSA program függvénymoduljának felhasználásával állítottuk elő. Az 5.7. ábrán együtt jelenítettük meg a bifurkációs görbét és a görbe néhány kiválasztott pontjához tartozó alakot. Ezek között látható néhány olyan, amely a valóságban nem jöhet létre. Ez abból adódik, hogy a matematikai modell nem tartalmaz bizonyos fizikailag releváns korlátokat.

Feladat	Felosztás	Rácspont	Proc.	$T_N[s]$	$T_N[s/10^6 pont]$	S_N	E_N
askrk05	300x300x250	$22.5 \cdot 10^6$	20	2122	94.3	20	1.00
askrk07	350x600x650	$135.5 \cdot 10^6$	30	8549	62.6	30.1	1.00
askrk09	10000x10000x2	$2 \cdot 10^8$	40	13696	68.5	27.5	0.69
askrk12	2000x1000x200	$4 \cdot 10^8$	50	9138	22.8	82.5	1.03
askrk13	300x300x300	$27 \cdot 10^6$	50	1058	39.2	48.1	0.96

5.3. táblázat. Mikroszálak vizsgálatánál elért sebességnövekedés

5.3. Mikroszálak konfigurációi

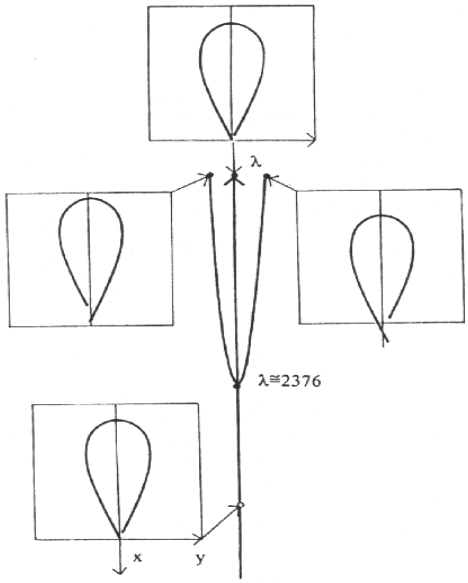
Megtámasztás nélküli, potenciálban úszó rugalmas szálak egyensúlyi helyzeteit vizsgáltuk, melyek mesterségesen növesztett nanoszálak mechanikai modelljeként szolgálnak [33], [4]. Szemben a klasszikus Euler-feladattal, a globális letapogatás (PSA) során másodlagos, szimmetriasértő elágazásokat (5.8. ábra), valamint úgynevezett „vegyes" módusú egyensúlyi alakokat (5.9. ábra) sikerült azonosítanunk. A GRT dimenziószáma 3 volt, ami meglehetősen finom letapogatást tett lehetővé. A párhuzamos letapogató algoritmust IBM SP2 típusú gépen futtattuk 20-80 processzorral. Az alkalmazott processzorszámtól és a tér felosztásától függően 30-250 perces futási idők adódtak. A sebességnövekedés a processzorszámtól való függése – várakozásainknak megfelelően – még 80 processzor esetén is lineáris volt. Az 5.3. táblázatban összegyűjtöttünk néhány jellemző konfigurációs paramétert és mért futási időt. Ezek összehasonlítása a processzorszámtól függő sebességnövekedés szempontjából nem egyszerű, mivel minden feladat más-más rácsmérettel, más-más GRT részre futott[1]. Ezért időegységnek a 10^6 rácspontból álló térrész letapogatásához (feldolgozásához) szükséges választottuk, amit a rácspontok és a futási idők ismeretében minden esetre kiszámítottunk. Ezek után önkényesen a 20 pocesszorral mért idő (táblázat 1. sora) 20-szorosát tekintettük a 10^6 rácspont egy processzorral történő kiszámításához szükséges időnek (T_1). A táblázat alapján jól megfigyelhető, hogy az egyik esetben kiugróan alacsony az elért hatékonyság (E_N), mivel a tér alakja előnytelen (az egyik irányban csak 2 részre van osztva). Az 1.00-nál nagyobb hatékonyság magyarázata az, hogy önkényesen választottuk ki az egyprocesszoros futási idő alapjául szolgáló askrk05 esetet.

5.4. DNS molekulák mechanikai modellje

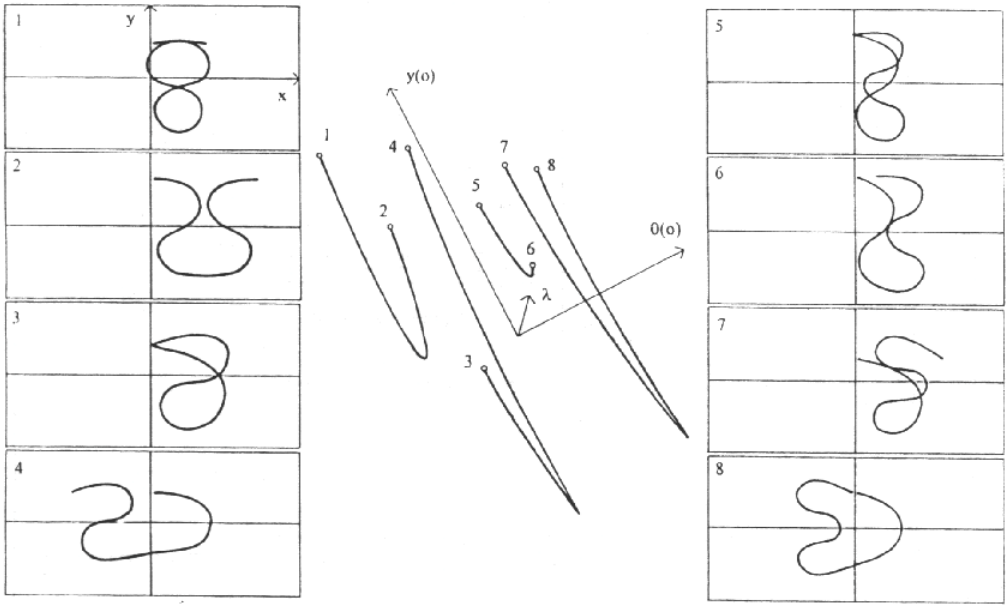
DNS molekulák egyes tulajdonságai mechanikai modellekkel is vizsgálhatók. Csavart, gyűrű alakúra összehajlított ún. Kirchhoff rudakkal [10], [44] modellezett DNS molekulák egyensúlyi alakját vizsgáltuk [15] a PSA program segítségével (ld. 5.10. ábra). A letapogató algoritmust IBM SP2 típusú számítógépen futtattuk 20-60 processzorral. Tipikus futási időnek 2-3 óra adódott a rács sűrűségétől, integrálási paraméterektől és a processzorszámtól függően. A kellő pontosság eléréséhez a fokozatos sűrítés módszerét (FSA) alkalmaztuk, amivel a 4 dimenziós GRT-ben jelentős sebességnövekedést érhetünk el. Az 5.4. táblázat egy $16 \cdot 10^6$ rácspontból álló tér letapogatását, és az azt követő sűrítési lépés futási idejét szemlélteti. A vizsgálatot 60 processzoral végeztük[2]. A 3.2. alfejezetben leírt módszerrel az eredeti teret $l_z = 5$ méretű ráccsal fedtük le, és így 309 térrészben

[1] A konkrét feladat megoldásánál már nem a teljesítménymérés volt a célunk, annál is inkább, mivel korlátos volt a számunkra felhasználható CPU idő.

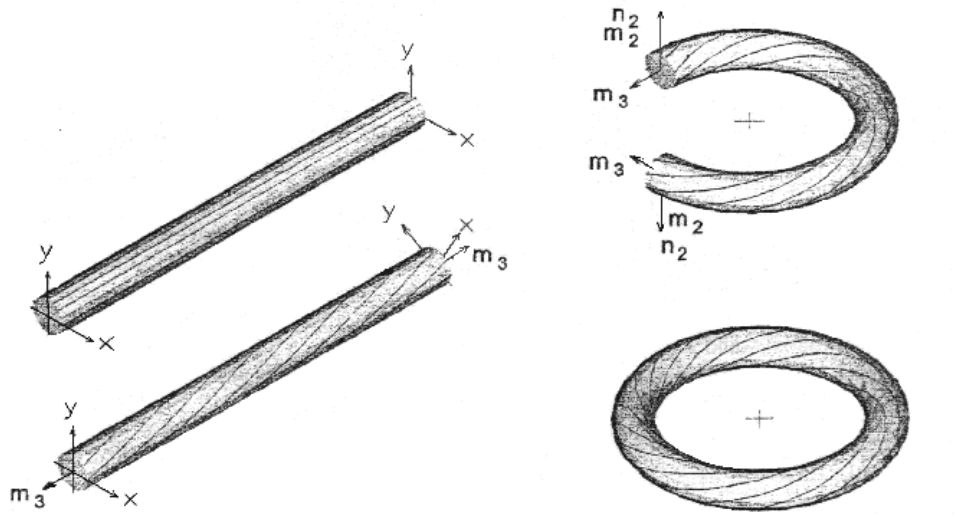
[2] A rendelkezésünkre álló környezet nem tette lehetővé, hogy a mester programot futtató processzoron (node-on) is futtassunk szolga programot, így valójában csak 59 szolga dolgozott.



5.8. ábra. Mikroszálak vizsgálata: másodlagos szimmetriasértő elágazás



5.9. ábra. Mikroszálak vizsgálata: vegyes módusú egyensúlyi alakok



5.10. ábra. DNS vizsgálata: csavart rúdmodell

Feladat	Felosztás	T_{59} [s]	T_{59}^z [s]	S_{59}^z
sring10	40x40x100x100	14007 (3h 53:27)		
sring12	309x15x15x15x15	13741 (3h 49:01)	27748	40.9
sring..	120x120x300x300	$\approx$ 1134567 (315h 9:27)		

5.4. táblázat. DNS molekulák vizsgálata során FSA módszerrel elért sebességnövekedés

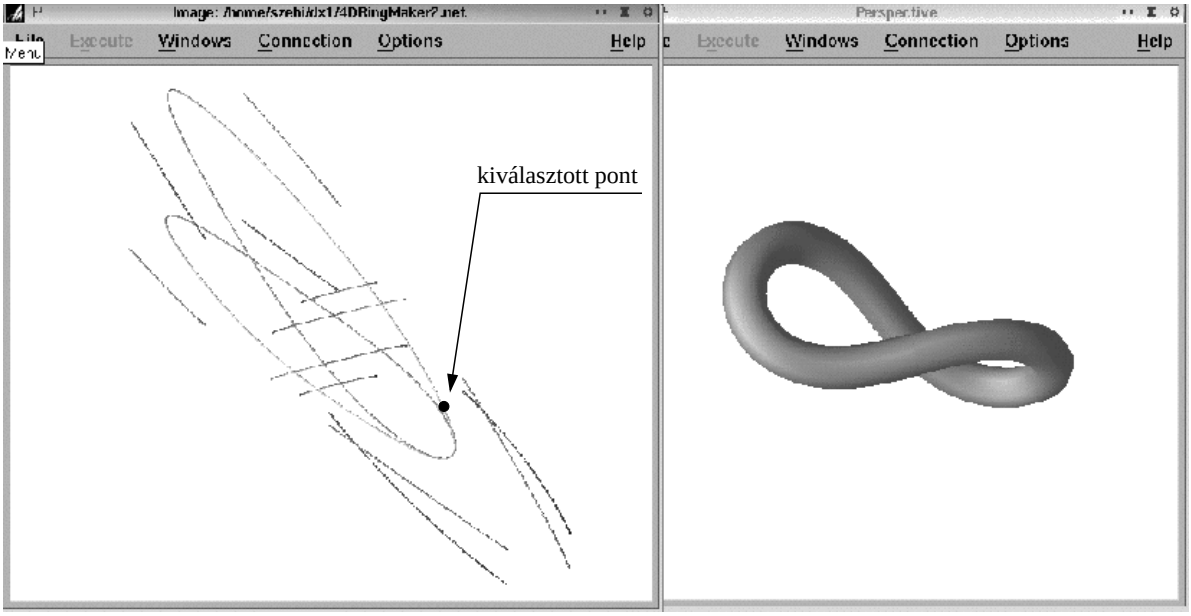
találtunk megoldást. Ezután ezek mindegyiket 15x15x15x15-os ráccsal lefedve közel azonos futási időt kaptunk, ugyanakkor a kapott eredmények az eredeti felosztás 3-szorosának megfelelő pontossággal keletkeztek. A dimenziószámot figyelembe véve ezt kb. 81-szeres[3] futási idő növekedéssel lehetett volna csak elérni az FSA alkalmazása nélkül.

5.5. Növényi indák mechanikai modellezése

A növényi indák feszültségmentes állapotban is rendelkeznek görbülettel, ezért ilyen típusú rudakkal kell őket modellezni. Az ilyen rudak több spirálból összetett egyensúlyi utat produkálnak húzóerő hatására. A spirálok közötti átmenetre a botanikusok már régen felfigyeltek. Az eddigi mechanikai vizsgálatok csak olyan egyensúlyi helyzeteket tudtak reprodukálni, amelyek 2 ellentétes irányultságú spirálból álltak.

A feladat 6 dimenziós GRT-jének belátható idő alatt történő letapogatása reménytelen feladatnak tűnt, mivel előzetes becslésekkel nem lehetett a vizsgálandó teret kellően kis méretűre választani. Ezért ennél a feladatnál alkalmaztuk elsőként a PHA módszert, melynek segítségével sikerült több

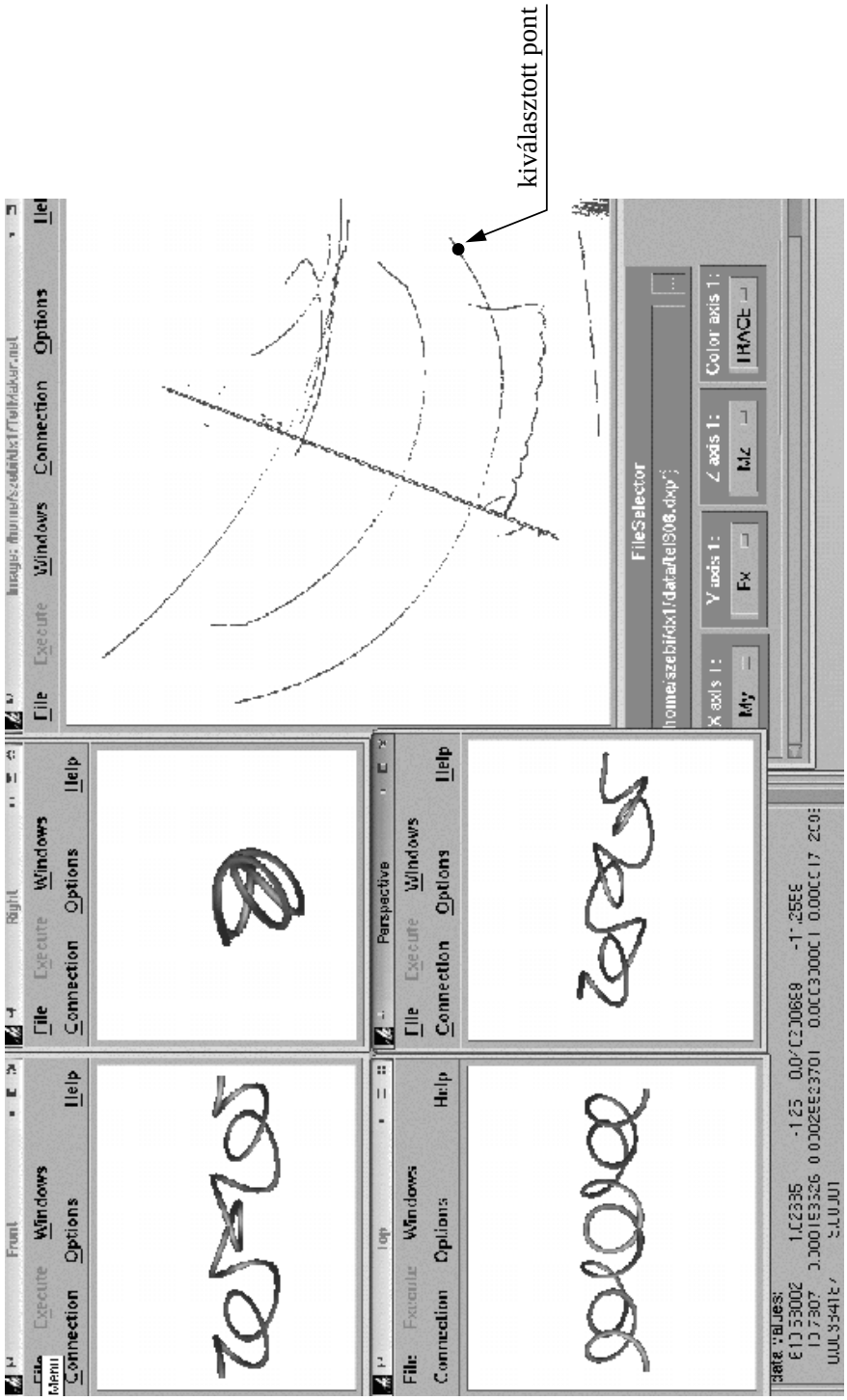
[3]Ezzel a rácsmérettel valójában nem futtattuk a feladatot, így a futási idő a táblázatban is (sring..) csak becsült idő.



5.11. ábra. DNS vizsgálata: bifurkációs görbe és a görbe egy pontjához tartozó egyensúlyi alak

mint 2 spirálból álló helyzeteket is azonosítanunk. Ilyen helyzeteket kísérletileg is meg lehet figyelni.

A futási eredmények 6 dimenziós ponthalmazként álltak elő, amely megjelenítéséhez, kiértékeléséhez a DX rendszert használtuk (5.12. ábra). Az 5.2. alfejezetben leírt alkalmazáshoz hasonlóan a fizikai alakot leíró ponthalmazt a függvénymodul feltételes fordításával nyert függvény segítségével állítottuk elő. Így a keresett, több ellentétes spirált tartalmazó egyensúlyi utak grafikusan azonosíthatók. Az 5.12. ábra jobb oldalán látható a PHA módszerrel előállított megoldásgörbe perspektivikus képe, melyet a 6 dimenziós ponthalmaz 3 kiválasztott koordinátája (Mx, Fx, Mz) alapján rajzolt fel a DX rendszer. A bal oldalon a megoldásgörbe egy kiválasztott pontjához tartozó inda-alak 3 vetülete és perspektivikus képe látható.



5.12. ábra. Növényi indák vizsgálata során keletkezett bifurkációs görbe és a görbe egy pontjához tartozó alak megjelenítése a DX rendszerrel

Tézisek összefoglalása, további kutatási célok

Dolgozatomban röviden áttekintettem a párhuzamos programozás történetét, a fontosabb programozási eszközöket és környezeteket, különös tekintettel az elosztott párhuzamos rendszerekre. Ezt követően bemutattam egy olyan módszert, melyet a nemlineáris mechanika területén fejlesztettek ki a közönséges differenciálegyenletekkel leírható peremérték-feladatok megoldására. A módszer jelentősége abban rejlik, hogy különösen alkalmas a problémák globális vizsgálatára, ugyanakkor használata korábban azért nem merült fel, mivel igen nagy számítási kapacitást igényel. A módszert megvizsgálva jól párhuzamosíthatónak találtam, ezért annak alapján kifejlesztettem a módszer párhuzamos változatát, melynek implementációját több gyakorlati feladatra is alkalmaztam. A gyakorlati alkalmazásokhoz az eredeti algoritmus általánosításaként új algoritmusokat készítettem és teszteltem. Az alkalmazási feladatok általánosításaként kidolgoztam egy alkalmazási modellt, mely alapján a peremérték-feladatok globális vizsgálatához támogató környezetet terveztem és implementáltam.

Új, önálló eredményeimnek az alábbiakat tartom:

1. A Szimplex Módszer letapogató változatához (PSA) – a párhuzamosítási lehetőségek feltárása és számszerű egybevetése alapján – párhuzamos algoritmust dolgoztam ki és implementáltam. Az elkészített programot heterogén elosztott környezetben és párhuzamos architektúrán is eredményesen teszteltem.

2. Analitikus közelítő modellt dolgoztam ki a párhuzamosítás hatékonyságának vizsgálatára. A modell érvényességét mérésekkel igazoltam. A kidolgozott modell lehetővé teszi, hogy a letapogató módszerhez a konkrét feladattól függetlenül előzetesen becsülni lehessen a leghatékonyabb feladatfelosztást.

3. A Szimplex Módszer letapogató változatához (PSA) gyorsító algoritmust készítettem, amely az n-dimenziós térrács lokális sűrítésén alapszik (FSA). Valós és teszt-feladatokon végzett mérésekkel igazoltam, hogy az eredeti, gyorsítás nélküli módszerhez képest – a feladat jellegétől és dimenziószámától függően – 2-150-szeres sebességnövekedés érhető el.

4. Kidolgoztam a Szimplex Módszer párhuzamos útkövető (PUA), véletlenszerű letapogató (VSA) és hibrid (HSA) változatait. Ezeket a korábban kidolgozott PSA általánosításaként fogalmaztam meg, és közös, paraméterezhető algoritmusként implementáltam. A módszert úgy egészítettem ki, hogy az szakadásos függvényekre is alkalmazható legyen.

5. Kidolgoztam a Szimplex Módszer általánosított, többparaméteres feladatok megoldására is alkalmas változatát (ilyen feladat pl. a stabilitási határgörbék számítása), melyet egy magasabb dimenziószámú feladat megoldására vezettem vissza. Az új, magasabb dimenziószámú feladat megoldását az általánosított algoritmus rekurzív kiterjesztéseként fogalmaztam meg.

6. Olyan alkalmazási modellt és környezetet dolgoztam ki, amely a függvénymodul és a konfigurációs állomány megadása után többlépcsős számítást, illetve elosztott hálózati környezetben több felhasználóval együttműködő interaktív megjelenítést tesz lehetővé.

További kutatási célok

Mint az 5. fejezet is rámutatott, a kutatás eredményei igen jól hasznosíthatók a gyakorlatban. A gyakorlati alkalmazások jelentősége miatt számos olyan további kutatási cél merült fel, melyek az elkészített rendszer továbbfejlesztéséhez kapcsolódnak. A továbbiakban ezek közül emelünk ki néhányat:

- *PUA és PHA algoritmusok modell alapú hatékonyságvizsgálata.* A 3. fejezetben bemutatott útkövető és hibrid algoritmusok modell alapú hatékonyságvizsgálatához valószínűségi alapokon nyugvó sorbanállási modellt kell felállítani. Ennek alapján számítható ki a sebességnövekedés várható értéke.

- *A hibrid (PHA) algoritmus továbbfejlesztése.* Az algoritmus továbbfejlesztéseként célszerű lenne, egy olyan algoritmust kidolgozni, amely dinamikusan képes változtatni a tér felosztását, és a tér méretét is. Egy ilyen algoritmus egyszerűsíthetné a komplex, sok változós, nagy számításigényű feladatok megoldását. Ez azonban több nehézséget is felvet, mint pl. a tér jelenlegi statikus transzformációjának dinamikussá válását.

- *Grid rendszerhez történő felkészítés.* Akadémiai viszonylatban egyre több lehetőség nyílik Grid-szerű környezetek elérésére. Az ilyen környezetek előnye, hogy nagyon sok számítógépből állnak, de azok rendelkezésre állása szélsőséges határok között változik (1.1.3. alfejezet), így erre fel kel készíteni a felhasználói programokat. A továbbfejlesztés szempontjából ez alapvetően a következő fő feladatcsoportokat jelöli ki:

  - A mester program felkészítése az opportunista környezetre, a Grid erőforrások kezelésére. Ez egyből felveti a hatékonyságvizsgálathoz felállított modell módosítását is. A változó rendelkezésre állás miatt ugyanis a kiszolgáló csak p valószínűséggel ér a feladat végére, és $1 - p$ valószínűséggel újraindul.
  - A kommunikációs séma és protokoll esetleges megváltoztatása.
  - A 4. fejezetben vázolt segédprogramok további integrálása, a felhasználói felület egyszerűsítése. Ehhez leginkább egy integrált Web felület létrehozása tűnik a legmegfelelőbbnek.

- *A párhuzamosítás hatékonyságvizsgálatának kiterjesztése.* Megoldás hatékonyságvizsgálatához egy olyan analitikus modellt állítottam fel, amely sorbanállási modellen alapszik. A modell felállításánál csak kevés feladatspecifikus megkötést tettem, így az más mester-szolga felépítésű párhuzamos programokra is kiterjeszthető lenne.

Köszönetnyilvánítás

A dolgozat részben közös munkákon, cikkeken alapszik. Minden társszerzőmnek szeretnék köszönetet mondani segítségükért. Kiemelt köszönet illeti tudományos vezetőmet, Domokos Gábort a munka irányításáért, a különböző mechanikai problémák kézzel fogható elmagyarázásáért, a folyamatos ösztönzésért. Köszönetem fejezem ki Gáspár Zsoltnak, aki tanácsokkal segített a numerikus módszerek megvalósításában. Köszönöm Philip Holmes és Paul Steen segítségét a számítási eredmények kiértékelésében illetve a mechanikai problémák megfogalmazásában. Köszönöm továbbá Chris Pelkie segítségét a DX rendszer megismerésében. Köszönöm tanszékvezetőm, Arató Péter, valamint minden kollégám támogatását. Végül, de nem utolsó sorban köszönöm családomnak a türelmet, amivel munkámat segítették.

Irodalomjegyzék

[1] E.L. Allgower and K. Georg. *Numerical Continuation Methods: an Introduction.* Springer-Verlag, Berlin, 1990.

[2] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings, second edition, 1994.

[3] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Poiceedings*, volume 30, page 483, Montvale, NJ, 1967. AFIPS Press.

[4] I.A. Askay, M. Trau, S. Manne, I. Honma, N. Yao, L. Zhou, P. Fenter, P.M. Eisenberger, and S.M. Gruner. Biomimetic pathways for assembling thin organic films. *Science*, 273:892–898, 1996.

[5] A Beguelin, J Dongarra, A Geist, R Manchek, and V Sunderam. Recent enhancements to pvm. *International Journal of Supercomputing Applications and High Performance Computing*, 1995.

[6] Allan Bricker, Michael Litzkow, and Miron Livny. CONDOR technical summary. Technical report, Computer Sciences Department, University of Wisconsin, 1998.

[7] T. Brown, K. Jeong, B. Li, S. Talla, P. Wyckoff, and Shasha D. *PLinda User Manual.* New York University, 1997.

[8] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[9] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, 1990.

[10] B.D. Coleman, D. Tobias, and Swingon I. Theory of the influence of end conditions on sef-contact in dna loops. *J.Chem.Phys*, 103:9101–9109, 1995.

[11] Mario Dal Cin, Andreas Grygier, Helmut Hessenauer, Uwe Hildebrandt, Joachim Hönig, Wolfgang Hohl, Edgar Michel, and András Pataricza. Fault tolerance in distributed shared memory multiprocessors. In Arnd Bode and Mario Dal Cin, editors, *Parallel Computer Architectures. Theory, Hardware, Software, Applications*, LNCS 732, pages 31–48. Springer, Berlin etc, 1993.

[12] G. Domokos. Global description of elastic bars. *Zeitschr. Angew. Math. und Mech.*, 74 (4):T289–T291, 1994.

[13] G. Domokos. *Rudak és rúdszerkezetek globális vizsgálata.* Akadémiai doktori értekezés, MTA, 1995.

[14] G. Domokos and Zs. Gáspár. A global, direct algorithm for path-following and active static control of elastic bar structures. *Int'l J. of Structures and Machines*, 23(4):549–571, 1995.

[15] G. Domokos and T. Healey. Hidden symmetry of global solutions in twisted elastic rings. *J. Nonlinear Sci.*, 11:47–67, 2001.

[16] G. Domokos and T. Healey. Helical perversion as a boundary value problem. *manuscript, in preparation*, 2002.

[17] G. Domokos and P. Holmes. Euler's problem and Euler's method, or the discrete charm of buckling. *J. Nonlin. Sci.*, 3:109–151, 1993.

[18] G. Domokos and P. Holmes. On non-inflexional solutions of non-uniform elasticae. *Int. J. Nonlin. Mech.*, 28 (6):677–685, 1993.

[19] G. Domokos, P. Holmes, and B.S.H. Royce. Constrained Euler buckling. *J. Nonlin. Sci.*, 7:1–34, 1997.

[20] G. Domokos, I. Szeberényi, and P. Steen. Parallel, recursive computation of global stability charts for liquid bridges. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 7th EuroPVM/MPI2000*, pages 64–71. Springer (Lec. Notes in Comp. Sci., Vol 1908), 2000.

[21] G. Domokos, I. Szeberényi, and P. Steen. Simulatenously resolved bifurcation diagrams: a novel global approach to liquid figures of equilibrium. *J. Comp. Phys.*, 159:38–57, 2000.

[22] M. Flynn and D. Gelernter. Some computer organisations and their effectiveness. *IEEE Transactions on Computres*, C-21:948–960, 1972.

[23] I. Foster. *Dessigning and Bulilding Parallel Programs.* Addison-Wesley, 1994.

[24] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufnann, 1999.

[25] Zs. Gáspár and G. Domokos. Global investigation of discrete models of the euler buckling problem. *Acta Techn. Hung.*, 102:227–238, 1989.

[26] Zs. Gáspár, G. Domokos, and I. Szeberényi. A parallel algorithm for the global computation of elastic bar structures. *Comp. Assist. Mech. and Eng. Sci.*, 4:55–68, 1997.

[27] Zs. Gáspár and P. Nédli. Global numerical analysis of elasto-plastic frames. *Comp.Ass. Mech. and Eng. Sci.*, 5:93–100, 1998.

[28] Zs. Gáspár and R. Németh. A special shape of a twisted ring. In *Proc. 2nd European Conference on Computational Mechanics*, page 11. Compact Disc, Cracow University, 2001.

[29] A. Geist, A. Beguelin, J. Dongara, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's guide and reference manual.* Oak Ridge National Laboratory, 1993. Technical Report ORNL/TM-12187.

[30] D. Gelernter. Multiple tuple spaces in linda. In *Proceedings of Parallel Architectures and Languages*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27, Berlin/New York, 1989. Springer-Verlag.

[31] GNU Free Software Foundation. *Gprof User's Guide*, 1988.

[32] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[33] P. Holmes, G. Domokos, and G. Hek. Euler buckling in a potential field. *J. Nonlinear Science*, 10:477–505, 2000.

[34] P. Holmes, G. Domokos, J. Schmitt, and I. Szeberényi. Constrained Euler buckling: An interplay of computation and analysis. *Comp. Meth. in Appl. Mech. and Eng.*, 170 (3–4):175–207, 1999.

[35] C. Kesselman I. Foster. Globus: A metacomputing infrastructure toolkit. *J. Supercomputer Applications*, 11(2):115–128, 1997.

[36] IBM. *IBM Visualization Data Explorer User's Guide*, 1997.

[37] B.J Lowry and P.H Steen. Capillary surfaces: Stability from families of equilibria with application to the liquid bridge. *Proc. R.Soc. London Ser.A*, 449:411–439, 1995.

[38] R. Macskási. Investigation of masonry arches. *Proc. Estonian Acad. Sci. Eng.*, 8, 3:147–161, 2002.

[39] S. Pajunen and Zs. Gáspár. Study of an interactive buckling model - from local to global approach. In J. Rondal, D. Dubina, and V. Gioncu, editors, *Proc. 2nd Int. Conf. on Coupled instabilities in metal structures*, pages 35–42. Imperial College Press, London, 1996.

[40] ParaSoft Corporation. *Express User's Guide*, 1990.

[41] B. Pákai. Párhuzamos számítási módszerek közönséges differenciálegyenletek megoldására. Diplomamunka, BME Villamosmérnőki és Informatika Kar, 1998.

[42] T. Rihardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[43] E. Riks. An incremental approach to the solution of snapping and buckling problems. *Int. J. Solids Structures*, 15:529–551, 1979.

[44] Y. Shi and J.E. Hearst. The kirchhoff elastic rd, the nonlinear schrodinger equation, and dna supercoiling. *J.Chem.Phys*, 101:5186–5200, 1994.

[45] D. Sima, T. Fountain, and Kacsuk P. *Advanced Comuter Architectures: A Design Space Approach.* Addison-Wesley, 1997.

[46] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1998.

[47] Z. Stejić. Developing a graphical interactive tool for displaying solution sets of boundary value problems. Diploma work, TUB Faculty of Elecrical Engineering and Informatics, 1998.

[48] Sun Microsystems. *Sun Grid Engine 5.3 Administration and User's Guide*, 2002.

[49] I. Szeberényi and G. Domokos. Solving generalized boundary value problems with distributed computing and recursive programming. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 6th EuroPVM/MPI1999*, pages 267–274. Springer (Lec. Notes in Comp. Sci., Vol 1697), 1999.

[50] I. Szeberényi, G. Domokos, and Zs. Gáspár. Solving structural mechanics problems by PVM and the simplex algorithm. In *Proc. 2nd EuroPVM'95*, pages 247–252. Hermes, Paris, 1995.

[51] P. Szulman. A master-worker számítási modell felhasználása számításigényes feladatok megoldására. Tdk dolgozat, BME Villamosmérnöki és Informatikai Kar, 2001.

[52] Condor Team. *Condor 6.4 User's Guide*. University of Wisconsin-Madison, 2002.

[53] LAM Team. LAM/MPI parallel computing. www.lam-mpi.org. Web pages.

[54] Thomas Williams and Colin Kelley. *Gnuplot An Interactive Plotting Program*. GNU Free Software Foundation, 1998.

A. Függelék

Programrészletek

A.1. canti.c függvénymodul

```
char *idenstr = "\
# $Id: canti.c,v 1.3 2000/10/02 05:19:59 szebi Exp $\n\
# FX Module for Cantilever beam\n\
# ap0   P       f";
/*********************************************************************
*  $Id: canti.c,v 1.3 2000/10/02 05:19:59 szebi Exp $
*  $State: Exp $
*  RCS history:
*    $Log: canti.c,v $
*
*********************************************************************/
#include <stdio.h>
#ifdef DX
# include <math.h>
#else
# include "fxmodule.h"          /* interface definitions */
#endif

#ifndef DX
char *fxmodulname = NAME;       /* module name, do not change */
#endif
static char rcsid[] = "$Id: canti.c,v 1.3 2000/10/02 05:19:59 szebi Exp $"; /* rcs ID */

#ifndef DX
/*------------------------ Slaves -----------------------------*/
#define NSLAVE  1               /* Number of slaves per hosts */
#define NOLOCAL 0               /* No local slave(s) on local host */

/*------------------------ Dimension --------------------------*/
#define DIM     2               /* Number of dimensions */
#define LAYER   1               /* Number of layers */
#define FNNUM   1               /* Number of explicit functions */

/*------------------------ Phase space ------------------------*/
long space[]={  100,            /* Number of points on x1 axis */
```

```
       100,            /* Number of points on x3 axis */
};
struct range realspace[]={      /* The real coordinates */
{  0, 10, 0 },          /* min_x1, max_x1, 0 */
{  0, 10, 0 },          /* min_x2, max_x2, 0 */
};

/*--------------------- Extra parameters ----------------------*/
#define NPAR    3               /* Number of extra parameters */
                        /* minimum value is 1 */
/* ------------------------------------------------------------ */
#define EI      params[0]       /* EI parameter */
#define L       params[1]       /* L parameter */
#define ds      params[2]       /* ds parameter for the integration */
float params[NPAR] = {
1,                      /* default value of L */
1,                      /* default value of EI */
0.1,                    /* default value of ds */
};

#endif
/*--------------------- Local variables ----------------------*/

/*---------------------- Function init-------------------------*/
/* FnInit: called by the slave program at the configure phase. */
/* When the slaves gets a configure message from the master the*/
/* slaves stores the new parameters and FnInit is called.      */
/*-------------------------------------------------------------*/

void FnInit()
{
/* No special initialization needed */
}

/*------------------------- Functions -------------------------*/
/* Fn: computes all function values for a point                */
/* WARNING: the caller allocates only the necessary amount of  */
/* memory for the return values !                              */
/*                                                             */
/* This function is called by the slave in two different       */
/* situations:                                                 */
/* First the slave calls with layervalue -1, in this case the  */
/* function should compute all the values from layer           */
/* 0 to layer-1. When the layer parameter is not negative, the */
/* function should compute the values for the given           */
/* layer only. In this case the return value should place to   */
/* the place of the first value set:                           */
/*         f[0], f[1], ..., f[DIM-2], v[0]                     */
/* and no memory allocated for the other sets.                 */
/*                                                             */
/* input:  the coordinates of a point (p), and the layer       */
/*         p[0], p[1], p[2], ..., p[DIM-1]                     */
/*         layer                                               */
```

```
/*                                                                  */
/* output: the function values (f), and a logical value (v)         */
/*         v is true if the function values are valid at the        */
/*         given point and layer                                    */
/*                                                                  */
/*      if layer < 0                                                */
/*         f[(Layer-1)*(DIM-1)+0], f[(Layer-1)*(DIM-1)+1], ...,     */
/*         v[0], v[1], ... v[Layer-1]                               */
/*      if layer >= 0                                               */
/*         f[0], f[1], ..., f[DIM-2]                                */
/*         v[0]                                                     */
/*                                                                  */
/*------------------------------------------------------------------*/
#define ap0     p[0]    /* a'(0) */
#define Q       p[1]    /* Q */

/*
*  Equation System:
*  EI a"(s) + Q cos(a(s)) = 0
*  IC: a(0) = 0,
*  BC: a'(L) = 0,
*  GRS: [a'(0), Q]
*  f: a'(L) = 0
*
*  a'(s+ds) = a'(s) + a"(s) ds = a'(s) - Q/EI cos(a(s) ds
*  a(s+ds) = a(s) + a'(s) ds
*
*/
#ifdef DX
/* p is 2-vector input;
EI, L, ds are float inputs;
PTS is [nn][2] output */
void Fn(float *p, float EI, float L, float ds, float PTS[][2])
#else
void Fn(double *p, double *f, char *v, int layer)
#endif
{
double a, ap, s, dap;

#ifdef DX
int i;
double x, y;
int nn;

FnInit();
nn = (int)ceilf(L/ds) + 1;

/* Store the first Canti profile point */
x = y = 0;
PTS[0][0] = x;
PTS[0][1] = y;
i = 1;
#endif
```

```
a = 0;
ap = ap0;
s = 0;
do {
dap = - (Q/EI) * cos(a) * ds;
a += ap * ds;
ap += dap;
s += ds;
#ifdef DX
x += cos(a) * ds;
y += sin(a) * ds;
/* Store the Canti profile points */
if (i < nn) {
PTS[i][0] = x;
PTS[i][1] = y;
i++;
}
#endif
} while (s < L);
#ifndef DX
f[0] = ap;              /* function value */
v[0] = 1;               /* function value valid */
#endif
}
#undef ap0
#undef Q

#ifndef DX
/*-------------------- Complementary functions ----------------*/
#define FCNUM 0         /* Number of complementary functions */

/*--------------------------- Fc -----------------------------*/
/* Fc: computes all complementary function values for a point. */
/* When the slave finds a solution point,calls the Fc function */
/* to compute the complementary function values.               */
/*                                                             */
/* input:  p[0], p[1], p[2], ..., p[DIM-1]                     */
/*         layer                                               */
/* output: f[0], f[1], f[2], ..., f[FCNUM-1]                   */
/*                                                             */
/* Fc is called always after the Fn function call, with        */
/* the same parameters, so the Fc may use the variables        */
/* set by preceding call of Fn.                                */
/*-------------------------------------------------------------*/

void Fc(double *p, double *f, int layer)
{
/* No complementary funtion is used in this application */
}

/*--------------------------- SFn ----------------------------*/
/* SFn: computes the simplex function value for a solution line*/
/*                                                             */
```

```
/* Simplex functions are complemantary functions for the       */
/* recursive algorithm. The main algorithm calls the Fn for    */
/* computing the explicit functions.                           */
/* A recursive simplex algorithm will be applied, when number  */
/* of explicit functions (FNNUM) are less than DIM-1.          */
/* The recursive simplex algorithm calls the SFn function.     */
/* input:  p[0], p[1], p[2], ..., p[d-1]                       */
/*         p[d], p[d+1], p[d+2] ..., p[2*d-1]                  */
/*           the start, and endpoint of the solution line      */
/*         xo[0], xo[1], xo[2], ..., xo[Dim-1]                 */
/*           base coordinates of the simplex                   */
/*         fy[0], fy[1], fy[2], ..., fy[d-2]                   */
/*         fy[d-1], fy[d], fy[d+1], ..., fy[(d+1)*(d-1)-1]     */
/*           function values on the simplex vertices           */
/*         x[0], x[1], x[2], ..., x[Dim-1]                     */
/*           coordinates of the grid point                     */
/*         ly,                                                 */
/*           current layer                                     */
/*         d,                                                  */
/*           current dimension                                 */
/* output: f[0]                                                */
/*           return value                                      */
/*         v[0]                                                */
/*           flag - false if the f[0] is not valid             */
/*                                                             */
/*-------------------------------------------------------------*/

void SFn(double *p, double *f, char *v,
    double *xo, double *fy, double *x, int ly, int d)
{
/* No simplex funtion is used in this application */
}

/* !!!!!! DO NOT CHANGE THE NEXT PART OF THE FILE !!!!!!!!!!!! */
/*------------ Inicialization of the global variables ---------*/
int Dim=DIM;                    /* Dim variable inic, do not change */
int Npar = NPAR;                /* Npar value inic, do not change */
int FcNum = FCNUM;              /* FcNum value inic, do not change */
int Nslave = NSLAVE;            /* Number of slave procs, do not change */
int NoLocal = NOLOCAL;          /* No local slave(s) on (master) local host */
int Layer = LAYER;              /* Number of layers */
int FnNum = FNNUM;              /* Number of explicit functions */
/*---------------- End of function module ---------------------*/

#endif
```

A.2. canti (starup) shell script

```
#!/bin/sh
# Startup script for simplex algorithm
# $Id: startup,v 1.6 2001/11/25 17:53:44 szebi Exp $
#
```

```
# RCS history:
# $Log: startup,v $
# Revision 1.6  2001/11/25 17:53:44  szebi
# dbm
#
# Revision 1.5  1998/06/19 11:02:00  szebi
# ' hiba
#
# Revision 1.4  1998/06/18 13:44:03  domokos
# MP_PROCS javitas
#
# Revision 1.3  1998/06/17 17:52:51  domokos
# grep javitas
#
# Revision 1.2  1998/06/17 13:58:57  szebi
# wrk file nev miatti valtozas
#
# Revision 1.1  1998/06/17 12:52:10  szebi
# Initial revision
#
#
# This script is generated automaticly when the programs are recompiled.
# The name of the script inherited from the name of the FX module.
# !!
# !! Please take care if you intend to modify this script because
# !! this is only a link to the original startup script.
# !!
#
# Parameters
#  1:   The config number
#  2:   Debug level
#
# The script starts (or submits to the LL) the master program with the
# NAMEx.cfg file with the following command:
#
# m_NAME -c NAMEx.cfg -o NAMEx.out > NAMEx.log &
#
# where:
#       NAME    - the name of the script
#       x       - the first parameter of the script
#
# If the NAMEx.wrk file is exist the script starts the master with the
# -recover flag.


i=`basename $0`

if [ x$1 = x ]; then
echo usage: $i cfg_nr
exit 1
fi

DBGLEV=0$2
```

```
MASTER=m_$i
PRJ=$i$1
CMD=$PRJ.cmd
CFG=$PRJ.cfg
LOG=$PRJ.log
OUT=$PRJ.out
WRK=$PRJ.wrk
HST=$PRJ.hst
DBG=$PRJ.dbg
DBM=$PRJ.dbm

warn()
{
if [ -f $1 ]; then
echo $1 exists
echo -n remove '?'
read answ
if [ x$answ = xy -o x$answ = xY ]; then
rm -f $1
return
fi
echo "$0 aborted"
exit 2
fi
}

if [ -f $WRK ]; then
echo $WRK file found!
ls -l $WRK
echo -n Recover '?'
read answ
if [ x$answ = xy -o x$answ = xY ]; then
R=-recover
fi
fi

if [ ! -f $CFG ]; then
echo $CFG doesn"'t" exist
exit 1;
fi

if [ x$R = x -a -f $LOG ]; then
warn $LOG;
fi

if [ x$R = x -a -f $OUT ]; then
warn $OUT
fi

rm -f $HST
rm -f $DBG
rm -f $DBM
```

```
if [ $DBGLEV -ge 1 ]; then
echo > $HST
fi

if [ $DBGLEV -ge 2 ]; then
echo > $DBG
fi

if [ x$PVM_ARCH != xSP2MPI ]; then
$MASTER -c $CFG $R -o $OUT >> $LOG 2>&1 &
echo "$MASTER -c $CFG $R -o $OUT >> $LOG        command started."
else
WDIR=‘pwd‘
MINPROCS=10
TIME_LIMIT=2:0:0

cat << END_OF_TEMPLATE > $CMD
#!/bin/csh
# ----------------------------------------------------------------
# $CMD:
# This cript generated automaticly by startup command of the simplex
# project. The script generated from the pvm-sp2mpi.cmd template:
#
# ----------------------------------------------------------------
# pvm-sp2mpi.cmd:
# A template for creating a job command file to submit to EASY-LL or
# LoadLeveler. This script runs a PVM 3 program using the SP2MPI
# architecture over the switch.
#
# Batch keywords must start with "# @".
# Other lines starting with "#" are comments.
#
# To use this template as your job command file, you will need to
# uncomment some LoadLeveler commands and to provide some additional
# information.  Instructions are provided within the template.
#
#               Updated: 1/17/95 RYL for LL 1.2
#                        7/15/96 SMJP for AIX 4 and PVM 3.3.10
#
#               Modified:6/17/98 ISZ for simplex project and automatic gen.
# ----------------------------------------------------------------
#  This section contains LoadLeveler/EASY-LL keywords
#  Modify initialdir to specify your own directory path
#  Specify wall_clock_limit as HH:MM:SS; e.g. 2:00:00 means 2 hours
#
# @ environment = "LL_JOB=TRUE"
# @ initialdir = $WDIR
# @ error       = $PRJ.cerr
# @ output      = $PRJ.cout
# @ wall_clock_limit = $TIME_LIMIT
# @ job_type = parallel
# @ min_processors = $MINPROCS
```

```
# @ notification = always
# @ notify_user = $USER
END_OF_TEMPLATE
  grep '^# @' $CFG >> $CMD
  cat << END_OF_TEMPLATE >> $CMD
# @ queue
# ------------------------------------------------------------------
#    Shell commands to be executed by the batch job go below this line.
tokens
set echo
date
#
#    mhost stores the switch address for the local host.
set mhost = \`hostname | cut -d. -f1\`-css
#
#    The node list (created by LL) has -css added to each hostname.
awk -F"." '{\$1 = \$1 avar; print \$1}' avar="-css" \\
    /tmp/\$LOADL_STEP_OWNER.\$LOADL_STEP_ID.hostfile > \\
    /tmp/\$LOADL_STEP_OWNER.\$LOADL_STEP_ID.hostfile2
cat /tmp/\$LOADL_STEP_OWNER.\$LOADL_STEP_ID.hostfile2
#
#    Set an environment variable to tell POE where to find your node file.
setenv MP_HOSTFILE /tmp/\$LOADL_STEP_OWNER.\$LOADL_STEP_ID.hostfile2
#
#    Set an environment variable to tell the number of processors
set wctmp = (\`wc \$MP_HOSTFILE\`)
setenv MP_PROCS \$wctmp[1]
#
#    Start PVM over the switch.
pvmd -n\$mhost &
#
#    Give PVM time to start.
sleep 120
#
#    Start the simplex application
#
$MASTER -c $CFG $R -o $OUT >> $LOG
date
#
#    Kill PVM.
pvm<<EOF
halt
EOF
#    Check for stray messages from child processes.
set myuid = \`id -u\`
cat /tmp/pvml.\$myuid
#
echo job done
END_OF_TEMPLATE

  echo "$MASTER -c $CFG $R -o $OUT >> $LOG"     command subbmitted.

  llsubmit $CMD
```

```
fi
```

A.3. dxconv shell script

```
#!/bin/sh
#
# dxconv script
#
# $Id: dxconv,v 1.4 2001/11/25 17:53:44 szebi Exp $
#
# RCS history:
# $Log: dxconv,v $
# Revision 1.4  2001/11/25 17:53:44  szebi
# dbm
#
# Revision 1.3  2001/10/31 10:04:39  szebi
# HDR javitas
#
# Revision 1.2  2001/10/29 22:37:36  szebi
# Comment lines
#
# Revision 1.1  2001/10/28 17:06:04  szebi
# Initial revision
#
#

dxhdr=yes
if [ x$1 = x-n ]; then
  dxhdr=no
  shift
fi

if [ x$1 = x -o x$2 = x -o x$3 = x -o x$4 = x ]; then
  echo "usage: $0 [-n] modul_name cfg_nr out_nr div [mul]"
  echo "       In case of the multiple zoom the original cfg_nr is reguied."
  echo "       mul is the accumulated zoom factor."
  exit 1
fi

if [ x$5 = x ]; then
  mul=1
else
  mul=$5
fi

MOD=$1
div=$4

INP=$MOD$3.out
CFG=$MOD$2.cfg
OUT=$MOD$3.dxp
HDR=$MOD$3.hdr
```

```
TMP=$MOD$3.tmp
TMP2=$MOD$3.tmp2
HD=$MOD$3.hd
DATE=‘date +%D‘
CLINES=0

rm -f $HDR $HD $OUT $TMP $TMP2

echo "Converting $INP file to DX-PSA format. Zoom faktor: $mul/$div"

if [ $dxhdr = yes ]; then
  echo "# DXconv: $DATE cfg:$CFG inp:$INP Zoom:$mul/$div" > $HD
  echo "Please give the comment lines (endig with empty line):"
  while read comm
  do
    if [ "x$comm" = x ]; then
      break;
    fi
    echo "# $comm" >> $HD
  done
  set not ‘wc -l $HD‘; shift;
  CLINES=$1
fi

z0_$MOD -z -c $CFG -h $HDR -d $div -m $mul -i $INP 2>$TMP2 | sort -u +0 -1 | cut -f2- > $TMP

set not ‘wc -l $TMP‘; shift;
LINES=$1

awk ’
BEGIN {
        nam1 = "’$TMP2’";
        nam2 = "’$HD’";
        lines = "’$LINES’" - 1;
        clines = "’$CLINES’";
}
/^# Dimension=/ {
        print >> nam2;
        clines++;
        dim=$3;
        next;
}

/^# Sgrid/ {
        print >> nam2;
        clines++;
        next;
}

/^# Real space=/ {
        print >> nam2;
        clines++;
        getline < nam1;
```

```
        for (i = 0; i < 5; i++) {
          getline < nam1;
          printf("#>%s\n", $0) >> nam2;
          clines++;
        }
        next;
}

/^# Extra parameters:/ {
        getline;
        Pars=NF-1;
        printf("# Reduced datalines = %d\n", lines) >> nam2;
        printf("points = 1\n") >> nam2;
        printf("type = int, int, int") >> nam2;
        for (i = 2; i <= NF; i++) {
          printf(", float") >> nam2;
        }
        printf("\nfield = Hdr, Dim, Pars") >> nam2;
        for (i = 2; i <= NF; i++) {
          printf(", Par%d", i-2) >> nam2;
        }
        printf("\nend\n") >> nam2;
        printf("%d\t%d\t%d", clines+10, dim, Pars) >> nam2;
        pr = 0;
        for (i = 2; i <= NF; i++) {
          printf("\t%s", $i) >> nam2;
        }
        printf("\n\n") >> nam2;
        next;
}

/^# \$Id:/ {
        print >> nam2;
        getline;
        print >> nam2;
        getline;
        printf("%s", $2) >> nam2;
        for (i = 3; i <= NF; i++) {
          printf("\t%s", $i) >> nam2;
        }
        printf("\n") >> nam2;
        exit;

}' $HDR

if [ $dxhdr = yes ]; then
  cat $HD > $OUT
  echo DX-PSA header created
fi
sed '/^#/d' $TMP >> $OUT
rm -f $HDR $HD $TMP $TMP2
ls -l $OUT
```

A.4. Makefile.aimk (architektúra független make)

```
########################################################################
#
#  Allapotegyenlet-rendszer megoldasa simplex modszerrel
#  $Id: Makefile.aimk,v 3.10 2001/11/25 17:53:44 szebi Exp $
#  RCS history:
#    $Log: Makefile.aimk,v $
#    Revision 3.10  2001/11/25 17:53:44  szebi
#    dbm
#
#    Revision 3.9  1999/05/08 21:21:52  szebi
#    cp->mv
#
#    Revision 3.8  1999/05/08 16:03:23  szebi
#    disztribucios atszervezes
#
#    Revision 3.7  1999/01/25 15:13:43  szebi
#    rekurzio jav.
#
#    Revision 3.6  1999/01/24 23:25:37  szebi
#    Rekurzio
#
#    Revision 3.5  1998/10/31 08:37:35  szebi
#    check
#
#    Revision 3.4  1998/06/19 11:02:00  szebi
#    cfgkeys, zoom
#
#    Revision 3.3  1998/06/17 14:39:01  domokos
#    Ismet Cornell
#
#    Revision 3.2  1998/06/17 13:08:56  szebi
#    Startup script
#
#    Revision 3.1  1998/05/29 19:00:47  szebi
#    Cornellrol hazahozott
#
#    Revision 3.0  1998/05/10 13:47:49  szebi
#    FX modultol fuggo neveken generalodik a slave es a master
#    Cornell-en fejlesztve.
#
#    Revision 2.9  1998/04/29 13:37:13  szebi
#    Makefile generated at Cornell
#
#    Revision 2.4  95/10/11  23:25:35  23:25:35  szebi (Szeberenyi Imre)
#    master ->simplex, slave->s_slave
#
#    Revision 2.3  1995/09/28  00:12:09  szebi
#    color
#
#    Revision 2.2  1995/09/27  18:11:15  szebi
#    segedfuggveny kiirasa az eredmenybe
```

```
#
#    Revision 2.1  1995/05/26  18:33:30  szebi
#    Hibaszamitas a masterben
#
#    Revision 2.0  1995/05/17  12:22:37  szebi
#    Configuracios file a fuggveny parameterezesehez
#
#    Revision 1.9  1995/05/12  10:11:43  szebi
#    Londonban osszerakott valtozat
#
#    Revision 1.7  1995/02/18  21:46:31  szebi
#    azonos pontok kiszurese
#
#    Revision 1.6  1995/02/16  20:23:34  szebi
#    Output file nev = $(FX).out
#    check program miatti atszervezesek
#
#    Revision 1.5  95/02/12  22:24:00  22:24:00  szebi (Imre Szeberenyi)
#    Eredmeny fileba is kiirodik az info es a report
#
#    Revision 1.4  1995/02/12  21:32:32  szebi
#    Log display
#    Keszenleti grafikon,
#    FX modul elnevezes.
#
#    Revision 1.3  1995/02/11  20:22:26  szebi
#    A slave tarolja a megoldaspontokat, es tobbet kuld vissza egyszerre
#    a masternek. (2. inditasi parameter = Maxmegold)
#
#    Revision 1.2  1995/02/11  20:18:25  szebi
#    Nem vizsgal minden kubust, (elojelvaltas)
#
#    Revision 1.1  1995/02/11  19:41:10  szebi
#    Initial revision
#
##########################################################################
#
# Set PVM_ROOT to the path where PVM includes and libraries are installed.
# Set PVM_ARCH to your architecture type (SUN4, HP9K, RS6K, SGI, etc.)
# Set ARCHLIB to any special libs needed on PVM_ARCH (-lrpc, -lsocket, etc.)
# otherwise leave ARCHLIB blank
#
# PVM_ARCH and ARCHLIB are set for you if you use "$PVM_ROOT/lib/aimk"
# instead of "make".
#
# aimk also creates a $PVM_ARCH directory below this one and will cd to it
# before invoking make - this allows building in parallel on different arches.
#

SDIR    =       ..
BDIR    =       $(HOME)/pvm3/bin
XDIR    =       $(BDIR)/$(PVM_ARCH)
```

```
CC      =       cc
MPCC    =       cc
#MPCC   =       mpcc
OPT     =
OPTIONS =       -O $(OPT)
CFLAGS  =       $(OPTIONS) -I$(PVM_ROOT)/include $(ARCHCFLAGS)
LIBSm   =       -lpvm3 $(ARCHLIB) -lgdbm -lm
LIBSw   =       -lpvm3 $(ARCHLIB) -lm
#LIBSw  =       -lpvm3pe $(ARCHLIB) -lm

LFLAGS  =       -L$(PVM_ROOT)/lib/$(PVM_ARCH) $(ARCHFFLAGS)

FX      =       fxmodule
SLAVE   =       s_$(FX)
MASTER  =       m_$(FX)
ZOOM0   =       z0_$(FX)
ZOOM1   =       z1_$(FX)
CHECK   =       c_$(FX)
STARTUP =       $(FX)
Common  =       $(SDIR)/common.h $(SDIR)/new.h
MAKELOG =       $(SDIR)/l_$(FX)
MSUBS   =       cubedb.o
SUBS    =       new.o
FILES   =       $(SLAVE) $(MASTER) $(XDIR)/$(STARTUP) $(XDIR)/zoom \
                $(ZOOM0) $(XDIR)/dxconv $(XDIR)/dxcat $(ZOOM1) $(CHECK)

all:    newfx $(FILES)
        @echo Slave created from $(FX) at `date`
        @echo Slave created from $(FX) at `date` > $(MAKELOG)
        @awk 'NR==1,/^\*\*\*\*\*\*\*\*\*\*\*\*/' $(SDIR)/$(FX).c >> $(MAKELOG)

newfx:
        - rm -f $(SLAVE) $(MASTER) $(FX).o

$(XDIR):
        - mkdir $(XDIR)

$(MASTER): simplex.o $(FX).o $(XDIR) $(SUBS) $(MSUBS)
        $(CC) -o $(MASTER) simplex.o $(SUBS) $(MSUBS) $(FX).o $(LFLAGS) $(LIBSm)
        mv $(MASTER) $(XDIR)

$(SLAVE): slave.o $(FX).o $(XDIR) $(SUBS)
        $(MPCC) -o $(SLAVE) slave.o $(SUBS) $(FX).o $(LFLAGS) $(LIBSw)
        mv $(SLAVE) $(XDIR)

$(FX).o: $(SDIR)/$(FX).c $(SDIR)/fxmodule.h $(Common)
        $(CC) -c $(CFLAGS) -DNAME=\"$(FX)\" -o $(FX).o $(SDIR)/$(FX).c

$(XDIR)/$(STARTUP): $(SDIR)/startup
        - rm -f $(XDIR)/$(STARTUP)
        ln -s `pwd`/$(SDIR)/startup $(XDIR)/$(STARTUP)

$(XDIR)/zoom: $(SDIR)/zoom
```

```
        - rm -f $(XDIR)/zoom
        ln -s `pwd`/$(SDIR)/zoom $(XDIR)/zoom

$(XDIR)/dxconv: $(SDIR)/dxconv
        - rm -f $(XDIR)/dxconv
        ln -s `pwd`/$(SDIR)/dxconv $(XDIR)/dxconv

$(XDIR)/dxcat: $(SDIR)/dxcat
        - rm -f $(XDIR)/dxcat
        ln -s `pwd`/$(SDIR)/dxcat $(XDIR)/dxcat

$(ZOOM0): zoom0.o $(FX).o $(SUBS)
        $(CC) -o $(ZOOM0) zoom0.o $(SUBS) $(FX).o $(LFLAGS) -lm
        mv $(ZOOM0) $(XDIR)

$(ZOOM1): zoom1.o $(FX).o $(SUBS)
        $(CC) -o $(ZOOM1) zoom1.o $(SUBS) $(FX).o $(LFLAGS) -lm
        mv $(ZOOM1) $(XDIR)

$(CHECK): check.o $(FX).o $(SUBS)
        $(CC) -o $(CHECK) check.o $(SUBS) $(FX).o $(LFLAGS) -lm
        mv $(CHECK) $(XDIR)

clean:
        - rm -f $(XDIR)/$(MASTER) $(XDIR)/$(SLAVE) $(XDIR)/$(STARTUP)
        - rm -f $(XDIR)/$(ZOOM0) $(XDIR)/$(ZOOM1) $(XDIR)/$(CHECK)
        - rm -f $(MASTER) $(SLAVE) $(ZOOM0) $(ZOOM1) $(CHECK) $(FX).o
#-------------------------------------------------------------------
```