EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKA KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK TANSZÉK

Adatintenzív alkalmazások grid-es környezetben

Schmidt János

Témavezető: Horváth Zoltán

Budapest, 2004. június 20.

Készült az IKTA 00064/2003 és 89/2002 keretében.

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. A problémakör leírása

A számítógépek fejlődéstörténetében mindig is voltak olyan feladatok, amelyek nagy falatnak tűntek a kor legjobb szuperszámítógépeinek is. Természetesen minden számítógép képes ideális körülmények között, - helyes program esetén - kellő idő alatt tetszőleges számítási feladatot elvégezni. Sajnos az emberek ideje erősen korlátos, így nem mindegy, hogy egy eredményre 100 évet, vagy csak 1 évet, esetleg pár napot, órát, percet, vagy másodpercet kell várni.

Ilyen feladatokkal látják el például a fizkiusok, a biológusok a számítógépeket. Ilyen volumenű feladat a cern-i kutatólaboratórium elektron gyorsítójából nyert adatok kiértékelése, a SETI (Search for Extraterrestrial Intelligence) projektből[1] nyert adatok kiértékelése, atombomba robbantás szimuláció, időjárás előrejelzés (kisebb területre lebontva, pontosabban, hosszabb időtartamra), vegyészeti kutatások, ütközésmodellezés, áramlástani formatervezés, ...

A dolgozat témája adatintenzív alkalmazások megvalósításának elemzése grid-es környezetben. Ezt az alábbi részekre bontva tárgyalom:

- Számítástechnika fejlődésének rövid története a Grid-ekig,
- az egyik legelterjedtebb Grid megoldás, és az egyik legelterjettebb job ütemező bemutatása,
- néhány párhuzamos programozási fejlesztőeszköz bemutatása,
- az elemenként feldolgozható függvények, és az azokat megoldó absztrakt programok ismertetése,
- egy konkrét elemenként feldolgozható függvény és futásának elemzése.

[1] SETI - földönkívüli intelligenciát kutató projekt, amely során az Egyesült Államok-beli Arizóna állam sivatagjaiban található hatalmas rádióteleszkópokkal hallgatják a mély űr "zajait", hátha valami intelligens jelsorozatra bukkannak

1.2. Témabejelentő

Adatintenzív alkalmazások közül elsőként az elemenként feldolgozható függvényekkel leírható feladatokat vizsgálom. Az elemenként feldolgozható függvények olyan gyakran használt függvények osztályát képzik, mint például összefésüléses rendezés, halmazok uniójának számítása, időszerűsítés, stb. Az elemenkénti feldolgozhatóság azt jelenti, hogy a függvény eredményét úgy kapjuk meg, hogy a bemenet minden egyes elemén ugyanazt a műveletet hajtjuk végre, egyiken a másik után. Általában az ilyen függvények értelmezési tartománya és értékkészlete egy rendezett sorozat, vagy egy halmaz.

A szuperszámítógépek között az 1990-es évek elején jelentek meg a hálózatba kötött és osztott rendszerként működő PC-k, az úgynevezett "PC cluster"-ek. Ezek gyorsan népszerűvé váltak a tudományos számítások terén olcsó kiépíthetőségük, nagy számítási teljesítményük és könnyű méretezhetőségük miatt.

Ezen rendszerek új megközelítését adta a grid technológia, amelyen a hálózatba kapcsolt rendszerek kommunikációs protokolljait, az általuk kínált szolgáltatásokat és az erőforrásaikat meghatározó infrastruktúrát értjük.

A dolgozat célja adatintenzív alkalmazások - pl. párhuzamos elemenkénti feldolgozás - megvalósításának elemzése grid-es környezetben és a kapott eredmények kiértékelése.

1.3. Köszönetnyilvánítás

Ezúton szeretném megköszönni dr. Horváth Zoltánnak, hogy elvállalta a diplomamunkám témavezetését, tapasztalatával ötleteket adott irodalmi kutatásokra, és a felmerülő problémák megoldását elősegíették a vele való beszélgetések.

Külön köszönöm családom, barátaim segítségét, akiknek egy-egy részproblémát elmagyarázva számomra is világosabbá váltak dolgok.

2. fejezet

Szuperszámítógépektől napjainkig

2.1. Szuperszámítógépek

A szuperszámítógépek olyan speciális célra készített gépek, amelyek alaklamasok nagy volumenű feladatok elvégzésére. A szuperszámítógépek fejlődésében az igazi áttörést a transzputerek[1] megjelenése (1984) hozta. Ezek segítségével viszonylag olcsón lehetett elosztott memóriájú párhuzamos rendszereket építeni, így azok "széles" felhaszálói körhöz eljutottak. A transzputerek sikerében nagy szerepe volt annak is, hogy nincs elvi működési különbség a két- vagy több száz transzputert tartalmazó rendszerek programozása között, és tetszőleges, egy adott feladathoz legjobban illeszkedő architektúra (pl.: háló, fa, gyűrű, kocka, ...) építhető fel vele.

A szuperszámítógépek főbb típusai: vektorprocesszorok, szimmetrikus multiprocesszorok, masszívan párhuzamos processzorok. Egy ilyen gép felépítése egyedi, csak néhány általános, köznapi haszálatban elterjedt dolog van benne. Ilyen például a processzor, de a processzorok közötti nagy sebességű, megbízható összeköttetés már egyedi. Egyedi dolgok kifejleszése és gyártása pedig költséges.

2.2. PC cluster-ek

Szükség van költséghatékony megoldásokra. A technika fejlődésével egyre gyorsabb és jobb eszközök terjednek el a mindennapi használatban, és válnak tömegcikké. Ilyenek például a 10, 100 megabites, majd 1 illetve 10 gigabites ethernet hálózati eszközök. Ezek gyors, megbízható, jó minőségű kapcsolatot biztosítanak számítógépek között. Innen jött az ötlet, hogy ne csak a processzorokat merítsék a gyártók a mindennapokból, hanem a kapcsolódási eszközöket is. Egy PC hálózat sokkal költséghatékonyabb, mint egy szuperszámítógép, ráadásul könnyebben skálázható és teljesítményben is felveszi vele a versenyt.

[1] transzputer - nagy teljesítményű RISC (Reduced Instruction Set Computer - csökkentett utasításkészletű számítógép) elvű processzor, saját memóriával és I/O felülettel, a párhuzamos működésű számítógépek egy építőeleme [4].

Így alakultak ki a úgynevezett "dedikált PC cluster"[2]-ek, melynek főbb jellemzői:

- **cél:** nagy teljesítmény és sebesség
- **módszer:** párhuzamos feldolgozás
- **tulajdon:** közös tulajdon
- **felépítés:** homogén (egyforma PC-k, egyforma operációsrendszer, . . .)
- **erős SSI (Single System Image)[3] követelmények**

A dedikált PC clusterek előzménye az úgynevezett NOW[4], ahol a pillanatnyilag nem használt munkaállomások szabad processzor idejét akarták kihasználni. A NOW főbb jellemzői:

- **cél:** szabad számítási ciklusok kihasználása
- **módszer:** háttérben futó feladatok kiosztása
- **tulajdon:** munkaállomások egyedi tulajdonban
- **felépítés:** általában homogén
- **korlátozott SSI követelmények**

2.3. Metaszámítógép, grid

Az elmúlt évek során az internet és a telekommunikációs eszközök robbanásszerű fejlődésével a Föld távoli pontjai egymáshoz "közel" kerültek. Szinte karnyújtásnyira kerültek egymáshoz PC clusterek, szuperszámítógépek, munkaállomások, otthoni PC-k, stb. Az internet segítségével ezek egy nagy hálózatba vannak kötve, innen az ötlet, hogy mi lenne, ha az internetre is megpróbálnánk ráhúzni az SSI követelményeket, hogy az egész egy nagy szuper-számítógép illúzióját nyújtsa.

[2]dedikált PC cluster - PC hálózat, amelyet azzal a céllal terveztek és készítettek, hogy tudományos számításokat végezzenek vele, mint egy szuperszámítógépen.

[3]SSI - A felhasználó felé egy képet mutat, számára úgy tűnik, mintha csak egy számítógép lenne, mintha az egész csak egy rendszer lenne.

[4]NOW - Network Of Worksations - munkaállomás hálózat; minden felhasználónak saját munkaállomása van, amikor nem használja (pl.: éjszaka, vagy elment ebédelni, . . .), akkor mások által írt programokat futtathat, ilyen például a SETI@home projekt

3. fejezet

Grid

3.1. Célok, és problémák

A grid egyik fő célja, egy nagy szuperszámítógép létrehozása, amelyet bárki, bárhol, bármikor elérhet, akár az elektromos hálózatot, innen is az elnevezés (az Egyesült Államokban az elektromos hálózatot grid-nek hívják). A grid főbb jellemzői:

- **cél:** szuerpszámítógép bárkinek, bárhol, bármikor
- **módszer:** interneten található gépek kihasználása
- **tulajdon:** minden egyedi tulajdonban
- **felépítés:** heterogén (mind harware, mind software szinten)
- **SSI**

A főbb jellemzőkből már következtetni is lehet néhény felmerülő kérdésre problémára. Például a heterogenitás: ha az én programom egy x86 processzor, linux operációsrendszerre van lefordítva, akkor az biztosan nem fog futni egy Machitos-on. Mi van, ha azt a gépet, ahol éppen fut a programom, a takarítónő véletlenül kirúgja a konnektorból? Kik futtathatnak az én gépemen programokat, kikben bízhatok? Ilyen és ezekhez hasonló kérdések és problémák merülnek fel a grid megvalósítása során.

A grid fejleszése során felmerülő kérdések és problémák a teljesség igénye nélkül:

- heterogenitás
- biztnságosság (pl: azonosítás)
- nagy válaszidők (az elektron is csak fénysebességgel halad)
- hibatűrés
- hol fut a program, honnan szedi a inputot, és hová kerüljön az eredmény
- ...

3.2. Mi a grid?

Nagyon sok féle, fajta gridről lehet olvasni manapság, például (megtartva az angol elnevezéseket): Comput Grid, Data Grid, Science Grid, Storage Grid, Access Grid, Knowledge Grid, Bio Grid, Cluster Grid, stb. Ha van egy ütemező egy lokális hálózaton, akkor az már "Cluster Grid". Ugyanezen a hálózaton egy file szerver szolgáltatás már "Storage Grid". Más szemszögből nézve egy munka állomás, amely magába foglal processzort, memóriát, merev lemezt, hálózati kártyát, az már egy "PC Grid". Minden számítógép rendszer egy grid [26] ?

Mielőtt belevágnánk a lehetséges grid megoldások ismertetésébe, tisztáznunk kell, hogy pontosan mit is értünk grid alatt. Az eddig leírtak csak homályos utalások voltak, hogy egy grid-nek milyen tulajdonságokkal kell(ene) rendelkeznie. A grid-et többen több féle képpen értelmezik.

Abban mindenki egyetért, hogy a grid az a middleware réteg, amely a heterogén eszözökre, rendszerekre (számítógépek, operációsrendszerek) úgymond ráülve egy homogén módon kezelhető felületet nyújt, ide értve az erőforrások nyilvántartását, -foglaltságát, -terheltségét, -jellemzőit, adatok titkosítását, stb. Ez nem egyszerű feladat egy olyan állandóan változó rendszerben, mint például az internet.

A middleware feladatai közé tartozik az is, hogy olyan "apróságokkal" foglalkozzon, mint számok-, stringek ábrázolási módja, egységes kommunikációs felület biztosítása, adatok kezelése (eljuttatása a szükséges hely(ek)re, szükség esetén titkosítása, . . .).

Napjaink grid kísérleteinek nagyrésze a Globus projekt által készített eszközöket használja, azokra épít.

3.3. Globus toolkit

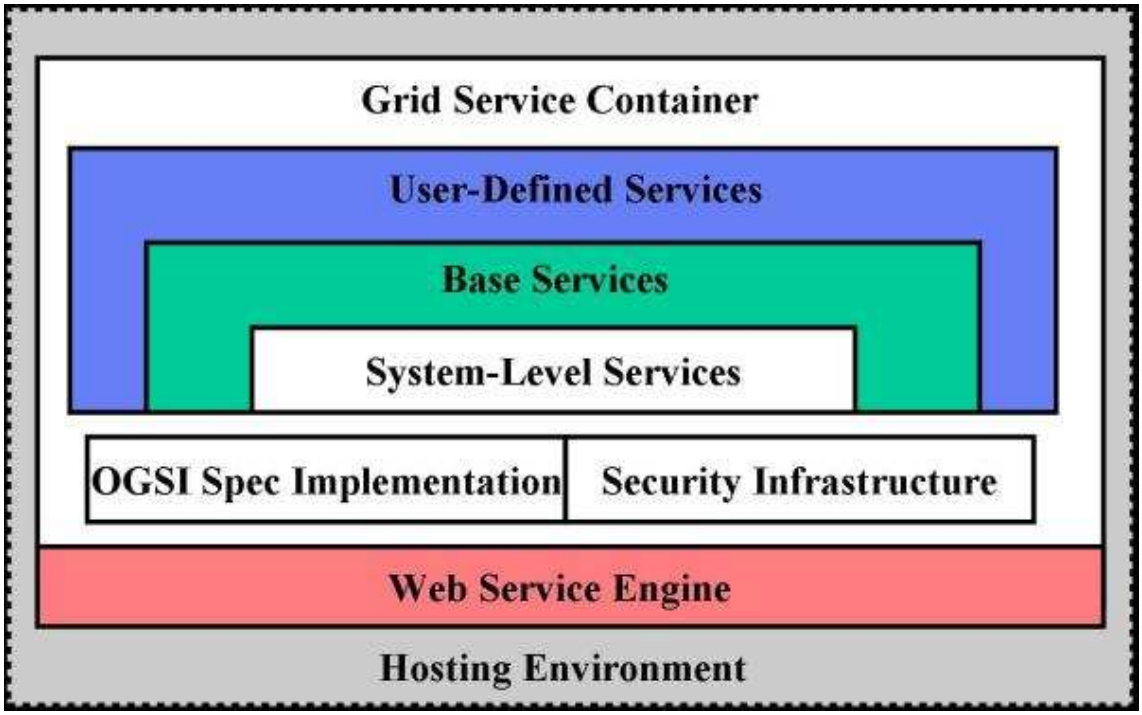
A Globus toolkit nagy teljesítményű elosztott rendszerek fejlesztését, építését elősegítő, támogató software komponensek együttese. A dolgozat írásakor a *http://www-unix.globus.org* oldalról letölthető GT[1] verzió: 3.2, így ennek rövid leírása olvasató az alábbiakban.

3.3.1. GT-core

A Globus Toolkit 3-as verziójának fő komponense olyan alap infrastruktúrát nyújt, amely nélkülözhetetlen grid szolgáltatások kiépítéséhez. Ez az alábbi alrészeket tömöríti magába:

- OGSI[2] specifikáció megvalósítása, bele értve az összes OGSI-ben leírt interface implementációját.

[1] GT - Globus Toolkit
[2] OGSI - Open Grid Service Infrastructure [6]



3.1. ábra. A Globus Toolkit 3 architektúrája

- Biztonsági infrastruktúra. Ez támogatást nyújt az üzenet szintű biztonsághoz, authentikációhoz és gridmap alapú authorizációhoz.
- Rendszer szintű szolgáltatások, amelyek megfelelnek az OGSI-ben leírt grid szolgáltatásoknak, és elegendőek ahhoz, hogy más grid szolgáltatások használják. Jelenleg három ilyen szolgáltatást nyújt a GT3[3]:
  - Admin Service
  - Logging Management Service
  - Management Serive

  Ezekről részletes leírás a [5]-ban olvasható.

3.3.2. Biztonsági protokollok

Grid Security Infrastructure

A GSI[4] nyílt kulcsú titkosítást használ és az alábbi alrészekből áll:

- Nyílt kulcsú kriptográfia

[3]GT3 - Globus Toolkit 3-as verziója
[4]GSI - Grid Security Infrastructure

- Digitális aláírások
- Tanúsítványok
- Kölcsönös jogosultság ellenőrzés (authentikáció)
- Titkos kommunikáció
- Privát kulcsok biztosítása
- Jogosultságok átruházása, egyszeri belépés

Community Authorization Service

A CAS[5] segítséget nyújt az erőforrás szolgáltatóknak, hogy közösségek számára, a közösség minden egyes tagjára egységes hozzáférési politikát alkalmazzanak. A CAS működési elve a következő:

- Egy CAS szerver létrehozása egy közösséghez.
- Az erőforrás szolgáltatók hozzáférést biztosítanak a közösség számára.
- A CAS szerver kezeli a közösség megbizható kapcsolatait, és biztosítja a hozzáférést az erőforrásokhoz.
- Amikor egy felhasználó hozzáférést szeretne kapni egy erőforráshoz, akkor ezt az igényét a CAS szerverhez nyújtja be. Amennyiben a felhasználónak van ahhoz hozzáférése, akkor az kiutal a számára egy igazolványt a szükséges jogokkal.
- Ezzel az igazolvánnyal a felhasználó bármely Globus eszközzel (pl.: GridFTP) az erőforráshoz fordulhat, amely eldönti, van-e a közösségnek hozzáférési joga.

3.3.3. Adatkezelés

GridFTP

A GridFTP egy nagy teljesítményű, biztonságos, megbízható adatátviteli protokoll, amely nagy sávszélességű WAN[6] hálózatokra optimalizáltak. A GridFTP alapja az FTP protokoll, és az alábbi jellemzőkkel bír:

- GSI biztosítja a vezérlő- és az adat csatornákat
- Több adat csatorna használata

[5] CAS - Community Authorization Service
[6] WAN - Wide-Area Network

- Részleges file átvitel
- Authentikált adatcsatornák
- Adatcsatornák újrafelasználhatósága
- parancsok csövezése

Reliable File Transfer

Az RFT[7] egy OGSA[8] alapú szolgáltatás, amely vezérlő és megfigyelő felületet biztosít olyan file átvitelekhez, amelyeket egy harmadik fél végez GridFTP szerverek segítségével. A kliens vezérelte átvitel egy grid szolgáltatáson belül helyezkedik el, ezért állapot modellek segítségével irányítható, és lekérdezhető annak ServiceData felülete segítségével, amely minden grid szolgáltatásban megtalálható. Ez a GT2[9]-ben található globus-url-copy eszköz egy megbízható és tovább fejlesztett változata.

Replica Location Service

Az RLS[10] végzi az információ leképzését az adatelemek logikai neveiről a cél névre. A cél nevek jelenthetnek egy fizikai helyet, vagy egy újabb RLS hivatkozást.

Ezt a szolgáltatást a Globus fejlesztők a DataGrid projekt fejlesztőivel közösen fejlesztik, és jelenleg kísérleti állapotban van.

A fejlesztők elképzelése, hogy az RLS egy szolgáltatás halmaz lesz a grid-beli adat tükrözések számára. Nem garantálja a másolatok konzisztenciáját, vagy az egyedi file bejegyzéseket a könyvtárakban. Úgy tervezik, hogy magasabb szintű grid szolgáltatások alapja legyen, amelyek majd ezeket (konzisztens másolatok, ...) biztosítják.

eXtensible Input Output

A XIO[11] egy kiterjeszthető I/O könyvtára a GT-nek. Egy egyszerű és könnyen kezelhető függvény könyvtárat (API-t) nyújt (open/close/read/write) a különböző I/O megvalósításokhoz. Két fő célja:

- Egyszerű felhasználói függvény könyvtár biztosítása minden grid I/O protokollhoz.
- Új protokollok létrehozásának és tesztelési idejének minimalizálása.

[7] RFT - Reliable File Transfer
[8] OGSA - Open Grid Services Architecture[8]
[9] GT2 - Globus Toolkit 2-es verzió
[10] RLS - Replica Location Service
[11] XIO - eXtensible Input Output

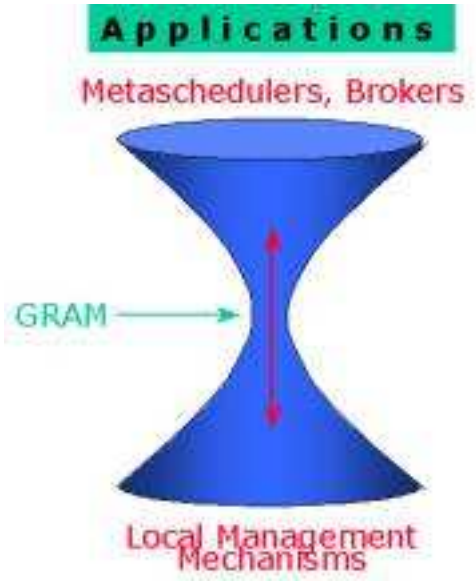
3.3.4. Erőforráskezelés

Globus Resource Allocation Manager

A Globus Toolkit egy szolgáltatás komponenens halmazt tartalmaz, amelyet GRAM[12]-nak nevezünk. A GRAM egyszerüsíti a "job"-ok végrehajtását és a távoli rendszerek használatát úgy, hogy egy egységes általános felületet biztosít a távoli erőforrások lekérdezésére és használatára. A GRAM legáltalánosabb (és egyben a legjobban támogatott) használata a távoli job submittálás és - felügyelet.

A GRAM feladata, hogy egy különálló általános protokoll, és - függvénykönyvtár legyen távoli erőforrások lekérdezésére és kezelésére úgy, hogy egy egységes, rugalmas felületet biztosítson a helyi job ütemezők számára. A GSI biztosítja mind a felhasználó, mind az erőforrás kölcsönös azonosítását. A GRAM ezen kívül egy egyszerű authorizációs mechanizmust biztosít a GSI azonosítókhoz, és egy leképző mechanizmust a GSI azonosító és a helyi felhasználó azonosító között.

A GRAM a távoli erőforrások használatához szükséges mechanizmusok számát igyekszik minimalizálni, de a helyi rendszerek ilyen mechanizmusok széles választékát használhatják (pl.: ütemezők, rendszerek lefoglalása, ...). A felhasználóknak, az alkalmazás fejlesztőknek tudniuk kell, hogy hogyan használják egyedül a GRAM-ot az erőforrások kezeléséhez, amelyik megefelően a többi GT komponens homokórában betöltött szerepével a homokóra nyakánál helyezkedik el.



3.2. ábra. GRAM homokóra beli szerepe.

[12]GRAM - Globus Resource Allocation Manager

3.4. Condor - High Throughput Computing

Számos tudós munkájának hatékonysága nagyban függ a számítási teljesítménytől. Könnyen található olyan probléma, amelynek megoldásához hetek vagy hónapok kellenek még számítógépek segítségével is. Az olyan rendszereket, amelyek hoszzú időn át sok számítást képesek elvégezni HTC[13] rendszereknek nevezzük. Ezzel szemben a HPC[14] rendszerek rövid időn keresztül óriási számítási teljesítményt nyújtanak. A fent említett tudósoknak az érdekül, hogy a lehető legtöbb job-ot tudják végrehajtani hosszú idő alatt, ezért nekik olyan rendszerekre van szükségük, amelyeknek nagy az áteresztő képessége (HTC) [10].

A Condor egy specializált feladatszétosztó rendszer számításigényes feledatokhoz. Mint bármely más batch rendszer, a Condor is biztosít feladatsort (ide kerülnek be a végrehajtandó feladatok, job-ok), ütemezési politikát, prioritási sémákat, erőforrások monitorozását és kezelését. A felhasználó odaadja a feladatot a Condornak, amely bekerül a feladat végrehajtási sorba, a meghatározott politika segítségével kiválasztja, hogy hol és mikor futtatja a job-ot, és miután a job lefutott, értesíti a felhasználót.

A Condor használható dedikált cluster-ek és NOW[15]-k működtetésére egyaránt. Condor segítségével kiépíthetünk egy grid stílusú számítási környzetet is, "flocking" technológiája segítségével több Condor rendszert is összekapcsolhatunk. Képes együtt működni számos grid-alapú számítási eljárással, protokollal. Ilyen például a Condor-G[9], amely képes teljesen együttműködni a Globus kezelte erőforrásokkal.

A Condor rendszer jellemzői:

- Elosztott, heterogén rendszerben mőködik,
- Alapvetően a szabad CPU ciklusok kihasználására tervezték,
- Képes egy működő feladatot áthelyezni az egyik gépről egy másikra (migráció),
- Képes az ún. *ClassAds* mechanizmussal a rendszerben lévő változó erőforrásokat az igényeknek megfelelően elosztani.

A *ClassAds* lényege, hogy a rendszerben található erőforrások jellemzőkkel bírnak, úgy mint teljesítmény, architektúra, operációs rendszer, bizalom, stb. Egy job összeállításánál ezekre a jellemzőkre lehet igényeket előírni, amelyeket a Condor megpróbál kielégíteni. A jellemzők között lehet preferenciákat is készíteni, amelyek akkor jutnak szerphez, ha több erőforrás is megfelel az igényeknek.

A Condor-ban számos univerzum létezik, amelyek meghatározzák, hogy a rendszerben a job-ok miket tehetnek, a job-oknak milyen tulajdonsaggal kell bírnia, stb. Ilyenek például:

- Standard univerzum:

[13] HTC - High Troughput Computing, nagy áteresztőképességű rendszer
[14] HPC - High Performance Computing, nagy teljesítményű rendszer
[15] NOW - Network Of Worstations, lásd a 7 oldalon a 2.2 bekezdésben.

  - nyomkövetés (checkpointing), automatikus migráció,
  - meglévő programokat újra kell fordítani, esteleg csak linkelni,
  - az alkalmazás nem használhat bizonyos rendszerhívásokat, pl: fork, socket, stb. ,
  - a Condor "elkapja" a file műveleteket.

- Vanilla univerzum:
  - nincs se nyomkövetés, se migráció,
  - meglévő futtatható kódot nem kell változtatni,
  - nincs korlátozás a rendszerhívásokkal szemben,
  - NFS, vagy AFS kell.

- PVM univerzum (PVM programok környezete):
  - binárisan kompatibilis,
  - PVM 3.4.2 + taszkkezeléshez kiegészítés,
  - dinamikus VM (virtual machine) kialakítás,
  - heterogén környezet támogatása,
  - egy felhasználó csak egy példányban futtathat daemon-t

- MPI univerzum (MPI programok környezete):
  - MPICH változtatás nélkül,
  - bináris kompatibilitás,
  - dimikusan nem változhat,
  - nem állhat meg,
  - NFS, vagy AFS kell

3.5. Hibatűrés

A hibatűrés fontos szerepet játszik elosztott rendszerek tervezésében. A hibatűrés a rendszernek az a jellemzője, amely megmutatja, hogy milyen mértékű meghibásodást képes elviselni, és a működését helyesen folytatni. Egy rendszer akkor hibatűrő, ha működését meghibásodások esetén is képes helyesen folytatni [21].

3.5.1. A hibatűrésről általában

A hibatűrő rendszerek a hibákat észlelve, bizonyos hibák esetén helyre "tudják" állítani magukat. Olyan nagy és összetett rendszerekben, mint például egy számítási grid, az erőforrások számának függvényében a meghibásodás valószínűsége már nem elhanyagolható. Gondoljuk végig: tegyük fel, hogy egy gép megbízhatósága 99%-os, és szeretnénk 10 ilyen gépből egy cluster-t csinálni. A kapott cluster megbízhatósága már csak 90 %-os, míg ha 100 ilyen gépből álló cluster-t nézünk, annak a megbízhatósága már csak $0.99^{100} = 0.366$, azaz majdnem 37 %-os. Minél nagyobb egy rendszer, annál nagyobb a valószínűsége, hogy valami meghibásodik benne [20].

Ezzel szemben, az esetleges hibát észlelve és lekezelve a teljes rendszer meghibásodásának valószínűsége sokkal kisebb, mintha csak egy gépünk lenne!

A hibatűrő rendszerekben a felépülés, a hibából való visszatérés a rendszer állapotának rendszeres feljegyzésén alapul. A hiba észlelése és elhárítása rengeteg adminisztrációs költséggel jár, de mégis nagyon hasznos lehet. Főleg az olyan számítási területeken, ahol sok adat készül, vagy nagyon bonyolultak a műveletek, és ezért sokáig tart a program futása. Ekkor egy kisebb hiba esetén az egész rendszer leállása nagy kiesést, nagy információ veszteséget okozhat. A nagy információ veszteség nem feltétlenül nagy mennyiségű adatot jelent, hanem nagy értékű információ vesztést, azaz olyan adatok elvesztése, amelyet költséges volt előállítani.

3.5.2. Alapgondolatok

A hibatűrő képesség szorosan összefügg az üzembiztos rendszer fogalmával, amely elosztott rendszerek számára sok hasznos követelményt fogalmaz meg. Ilyen például az elérhetőség (azonnali használhatóság), a megbízhatóság (milyen gyakoriak, illetve ritkák a meghibásodások), biztonságosság (akkor sem történik semmi végzetes, ha a rendszer ideiglenesen helytelenül működik), kezelhetőség (milyen könnyen javítható a meghibásodott rendszer), stb [19].

Fontos kérdés, hogy mikor tekintünk egy rendszert meghibásodottnak. Egy rendszer akkor meghibásodott, ha nem képes ellátni azt a feladatot, amiért készítették. Attól, hogy egy rendszerben előfordult egy hiba, még nem biztos, hogy az a rendszer meghibásodott.

Különbséget kell tenni a hibák megszüntetése és megelőzése között. Az egyik legfontosabb szempont a hibatűrő képesség, azaz a rendszer a hibák ellenére képes a feladatát tökéletesen ellátni.

A fellépő hibákat három csoportba osztályozhatjuk:

- alkalmi - a hiba csak egyszer következik be, a műveletet megismételve nem jelentkezik újra
- ismétlődő - bekövetkezése után a hiba eltűnik, majd ismét előjön, és így tovább
- állandó - a hiba bekövetkezésétől a megjavításáig tart

3.5.3. Meghibásodás

A meghibásodott rendszer nem képes azt a feladatot elvégezni, amelyre létrehozták. Elosztott rendszerek több alrendszerből épülnek fel. Ha a rendszer egy építőelemének helyes működése más építő elemktől függ, akkor a függőségi láncban előrébb elhelyezkedő elem meghibásodása okozhatja a láncban mögötte elhelyezkedő többi elem meghibásodását.

Ha a rendszer hibatűrő akar lenni, akkor legcélravezetőbb a hibák bekövetkeztét a többi folyamat elől elrejteni. A hibák elfedésében a redundancia a kulcstechnika, ennek három lehetséges fajtája van:

- információredundancia - az információhoz plusz információt adunk hozzá, amelynek segítségével észlelhető, eseteleg javítható a sérült információ. Ilyen például a paritás bit.
- időredundancia - végrehajtunk egy műveletet, majd ha adott időn belül nem érkezik válasz (időtullépés), akkor ismét végrehajtjuk.
- fizikai redundancia - külön eszközöket, folyamatokat helyezünk el a rendszerben, hogy meghibásodás esetén a kiesett komponenes szerepét átvéve áthidaló megoldást nyújtsanak. Ez megvalósítható mind hardware, mind software szinten.

A fizikai redundancia jól ismert technika a hibatűrés biztoítására, még a természetben is alkalmazzák, például az embernek két veséje van.

4. fejezet

Párhuzamos programozási eszözök

A párhuzamos feldolgozás (sok kis taszkkal oldunk meg egy nagy feladatot) kiemelkedő szerephez jut a modern számítások terén.

4.1. PVM

A PVM[1]egy integrált szoftver eszöz, és fejleszői könyvtár, amelyet heterogén rendszerekben végrehajtandó párhuzamos számításokhoz terveztek [12]. Célja egy rugalmas, könnyen skálázható, heterogén rendszerek összekapcsolására is alkalmas rendszer készítése, amelyeket összekapcsolva egy virtuális gépet lát a felhasználó. A PVM biztosítja az üzeneteket továbbítását, adatok konverzióját és taszkok ütemezését [13]. A PVM alapelvei a következők:

- Felhasználó által összeállított rendszer, ennek részét képezhetik egy-, illetve több processzoros gépek egyaránt, akár egyszerre is. A rendszerhez dinamikusan hozzáadhatók, illetve onnan el is távolíthatók az erőforrások.
- Processz alapú számítás, a PVM-ben a párhuzamosság egysége a taszk.
- Explicit üzenetátadás modell. A taszkok egymással üzenetek segítségével kommunikálhatnak, egy üzenet méretének a rendelkezésre álló memória szab határat.
- Heterogén rendszerek támogatása.
- Multiprocesszoros rendszerek támogatása.

4.1.1. PVM rendszer felépítése

Egy PVM rendszer két részből áll:

[1]PVM - Parallel Virtual Machine

- PVM daemon (pvmd3), melynek feladata a PVM rendszer összeállítása, gépek hozzáadása a rendszerhez, illetve elvétele a rendszerből, üzenetek továbbítása, job-ok indítása, kezelése, ...
- PVM rutinokat tartalmazó könyvtár.

A PVM letöltehtő a http://www.csm.ornl.gov/pvm/pvm_hmoe.html web oldalról (2004. június 10.)

4.1.2. Példa

A PVM jelenleg csak a C, a C++ és a Fortran nyelveket támogatja.

A PVM-mel készített programok problémája, hogy a megírt programot le kell fordítani minden egyes arhcitektúrára, amely a rendszerben szerepel, és a fordítás eredményét egy hozzáférhető helyen kell elhelyezni (ez általában: $(HOME)/pvm3/bin/$(PVMARCH)/). Futtatáshoz a felhasználó elindítja a PVM daemont (amikor a felhasználó a PVM-et indítja, és megjelenik a PVM prompt, akkor a PVM daemon is elindul automatikusan), majd futtatja a programját (PVM prompt-ból a spawn paranccsal).

PVM-ben általában master-worker programokat készítünk, azaz van egy taszk, amely létrehoz újabb taszkokat, és azok között szétosztja a feladatokat, majd összegyűjti az eredményeket.

Példa: a "Hello world!" probléma megoldása PVM rendszerben.

hello.c

```
#include "pvm3.h"
#include <stdio.h>

int main ( void )
{
    int mtid = pvm_mytid ();
    printf ( "\nHello, i'm t%x\n", mtid );
    int ctid;
    int cc = pvm_spawn ( "hello_other", ( char * * ) ( 0 ), 0,
"", 1, &ctid );
    if ( 1 == cc )
{
        char msg[128];
        pvm_recv ( ctid, 1 );
        pvm_upkstr ( msg );
        printf ( "Uzenet jott t%x -tol: \"%s\"\n", ctid, msg );
    }
    else
    {
```

```
        printf ( "Nem sikerult elinditani a \"hello_other\"
taszkot.\n" );
    }
    pvm_exit ();
    return 0;
}
```

hello_other.c

```
#include <string.h>
#include "pvm3.h"

int main ( void )
{
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent ();

    strcpy ( buf, "hello, world from " );
    gethostname ( buf + strlen ( buf ), 64 );
    msgtag = 1;
    pvm_initsend ( PvmDataDefault );
    pvm_pkstr ( buf );
    pvm_send ( ptid, msgtag );

    pvm_exit ();
    return 0;
}
```

Makefile

```
SRCS= hello.c hello_other.c
OBJS= $(SRCS:.c=.o)

CFLAGS= -c
LFLAGS= -lpvm3

CC= gcc

.SUFFIXES: .c
```

```
.c.o:
    $(CC) $(CFLAGS) -o $@ $<

all:
    make hello hello_other

hello: hello.o
    $(CC) $(LFLAGS) hello.c -o hello
    cp hello ~/pvm3/bin/$(PVM_ARCH)

hello_other: hello_other.o
    $(CC) $(LFLAGS) hello_other.c -o hello_other
    cp hello_other ~/pvm3/bin/$(PVM_ARCH)

clean:
    rm -f $(OBJS) hello hello_other
    rm -f ~/pvm3/bin/$(PVM_ARCH)/hello
    rm -f ~/pvm3/bin/$(PVM_ARCH)/hello_other

re:
    make clean hello hello_other
```

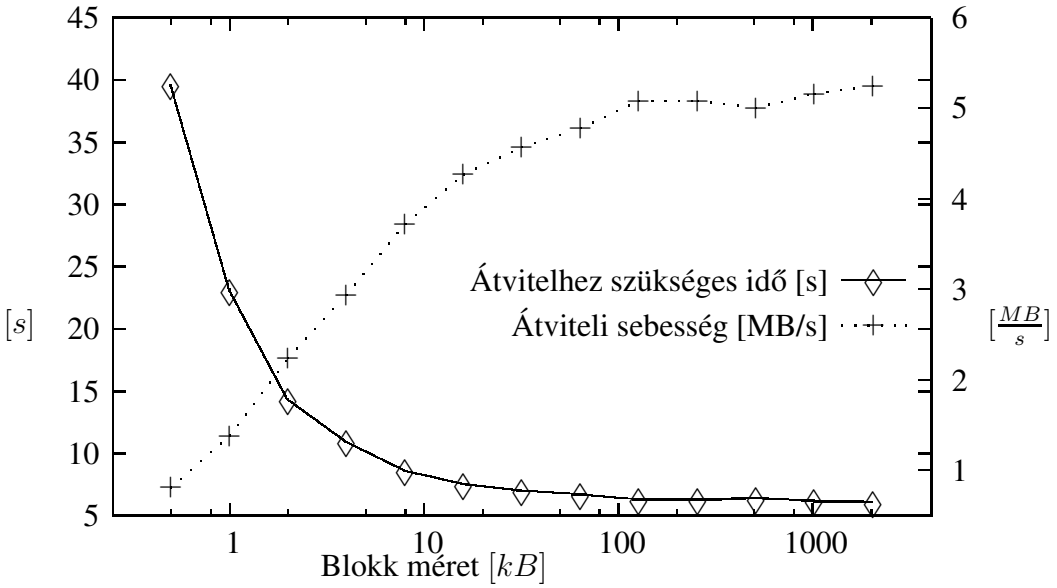
Futási eredmény

```
pvm> spawn -> hello
[1]
1 successful
t40002
pvm> [1:t40002]
[1:t40002] Hello, i'm t40002
[1:t40002] Uzenet jott t40003 -tol: "hello, world from
ozirisz"
[1:t40002] EOF
[1:t40003] EOF
[1] finished
```

4.1.3. PVM átviteli sebesség mérése

A mérés környezete: 2 db intel Pentium4-Celeron processzoros, 256MB DDR-SDRAM-os PC, 100 MBit-es hálózati kártyával összekötve, az ELTE-IK nyelvi laboratóriumában (nyl25, nyl26). A mérés dátuma: 2004. június.

A mérés eredménye a 22 oldalon a 4.1 ábrán látható. Az átviteli sebesség meghatározásához 32MB-nyi adatot küldtem át a PVM segítségével egyik gépről a



4.1. ábra. PVM sebessége 100MBit-es full-duplex hálózaton.

másikra, különböző blokk méretekkel. Minden mérést tízszer megismételtem, majd az eredények átlagát számoltam, hogy hiteles értékeket kapjak. A részletes mérési eredmények megtalálhatók a 56 oldalon a A.1 függelékben.

A grafikonról leolvasható, hogy szinte maximális átviteli sebességet lehet elérni már 128 KB-os blokk mérettel. Hiába növeljük meg a blokk méretet, sokkal gyorsabb átvitelt nem érünk el vele. Ha 2MB-os blokk méretettel küldenénk át az adatot, akkor is csak 3.35%-os sebesség növekedést érnénk el, míg a ráfordított memória költségünk 16-szoros.

A PVM képességeihez mérten aránylag jó átviteli sebesség érhető el már 32 KB-os blokkokkal, ha a 2 MB-os blokkot tekintjük maximumnak, akkor annak a 87 %-a, míg memóriaköltsége $\frac{1}{64}$-e annak!

A grafikonról az is leolvasható, hogy míg az 100 MBit-es hálózat elméleti áteresztő képessége 12.5 MB, addig PVM-mel ennek még a felét sem sikerült elérni.

Hibatűrés PVM-ben

PVM környezetben a virtuális gép üzeneteket küldhet bizonyos eseményekről, például egy host kiesése a virtuális gépből. Hogy milyen eseményeket szeretnénk figyelni, az a **pvm_notify** fügvvénnyel adható meg. Amennyiben egy ilyen esemény bekövetkezett, akkor arról a *pvmd* értesíti azt a taszkot, amelyik az esemény megfigyelését kezdeményezte úgy, hogy egy üzenetet küld neki az eseményről. Az üzenet tárgyát a pvm_notify függvényben kell megadni.

A pvm_notify-jal figyelhető események:

- Új taszk megjelenése a virtuális gépben

- Host kiesése a virtuális gépből
- Taszk kiesése

4.2. MPI

Az MPI[2] azért jött létre, mert a masszívan párhuzamos processzoros (MPP[3]) gépeket gyártó cégek megalkották a saját üzenet továbbító API[4]-jaikat, amelyek nem feltétlen voltak egymással kompatibilisek, így az elkészített programok csak adott típusú gépeken fordultak le, és futottak. Az MPI-t úgy tervezték, hogy az üzenet továbbítás szabványa legyen, és minden MPP gyártó implementálhassa a saját termékeire, ezzel egységes API-t biztosítson a párhuzamos programok számára. Az MPI-jal készített programok már hordozhatók.

Az MPI-1-gyel nem volt lehetőség arra, hogy kihasználjanak egy NOW hálózatot, mivel a szabvány nem írt elő olyan függvényt, amellyel szeparált host-on taszkot lehetne indítani.

Az MPI-2 1997-es specifikációjában számos új szolgáltatás jelent meg, például nem blokkolt kollektív kommunikáció, C++ nyelvi kapcsolat, MPI_SPAWN, stb. Ebből a változatból is hiányzik néhány hasznos szolgáltatás, mint például az erőforrások dinamikus feltérképezése, vagy amelyek segítségével hibatűrő alkalmazások készíthetők ([17], [20], [18]).

Az MPI letölthető a http://www.lam-mpi.org/download weboldalról (2004. június 10.)

4.2.1. Példa

Ez a példa a tipikus "Hello world!" példa MPI környezetben. Minden taszk kiolvassa, hogy milyen gepen fut, majd azt egy üdvözlő üzenetbe foglalva elküldi a 0-as taszknak, aki kiírja a kapott üzeneteket.

hello.c

```
#include <string.h>
#include <stdio.h>
#include "mpi.h"

int main ( void )
{
    int len, rank, pnum, src, dst, i, tag=0;
    char msg[256], buf[128];
```

[2]MPI - Message Passing Interface
[3]MPP - Massively Parallel Processor
[4]API - Application Programming Interface

```
    MPI_Status status;

    MPI_Init ( &argc, &argv );
    MPI_Get_processorname ( buf, &len );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    MPI_Comm_size ( MPI_COMM_WORLD, &pnum );

    if ( 0 != rank )
    {
        sprintf ( msg, "%s : Greetings from process %d!", buf,
rank );
        dst = 0;
        MPI_ Send ( msg, strlen ( msg ) + 1, MPI_ CHAR, dst,
tag, MPI_ COMM_ WORLD );
    }
    else     {
        printf ( "I am on %s : Hello World!\ n", buf );
        for ( i=1; i<pnum; ++i );
        {
            MPI_Recv ( msg, 256, MPI_ CHAR, i, tag, MPI_ COMM_
WORLD, &status );
            printf ( "%s\ n", msg );
        }
    }

    MPI_Finalize ();
    return 0;
}
```

Makefile

```
SRCS= hello.c
OBJS= $(SRCS:.c=.o)

CFLAGS= -c
LFLAGS=

CC= mpicc

.SUFFIXES: .c

.c.o:
    $(CC) $(CFLAGS) -o $@ $<
```

```
all:
    make hello

hello: hello.o
    $(CC) $(LFLAGS) hello.c -o hello

clean:
    rm -f $(OBJS) hello

re:
    make clean hello
```

Futási eredmény

```
$ mpirun N hello
I am on nyl17.nylab.inf.elte.hu : Hello World!
nyl09.nylab.inf.elte.hu : Greetings from process 1!
nyl11.nylab.inf.elte.hu : Greetings from process 1!
nyl12.nylab.inf.elte.hu : Greetings from process 1!
$
```

4.3. P-GRADE

A P-GRADE[5][3] az MTA-SZTAKI által fejlesztett grafikus keretrendszer párhuzamos programok szuperszámítógépekre, cluster-ekre, GRID rendszerekre fejlesztéséhez, és futtatásához. Segítségével könnyen és egyszerűen lehet párhuzamos programokat készíteni, még azok számára is akik korábban nem foglalkoztak ilyesmivel. Ugyanazt a környezetet használhatjuk a szuperszámítógépek programozásához, mint a cluster-ek programozásához, vagy mint Globus alapú GRID rendszerek programozásához.

A P-GRADE lehetőséget biztosít a futó folyamatok nyomkövetéséhez, a program futása közben lenyomatot vesz a programról, szükség esetén akár át is költözteti. A lenyomat vétellel képes egy félbeszakadt futást a vétel pillanatától folytatni, akár másik gépen is.

P-GRADE-es programok fejlesztéséhez szükséges a C nyelv bizonyos szintű ismerete és némi programozási képesség. Egy program fejlesztése piktogram alapú, azaz olyan, mintha egy folyamat ábrát raknánk össze. A fő építő elemek:

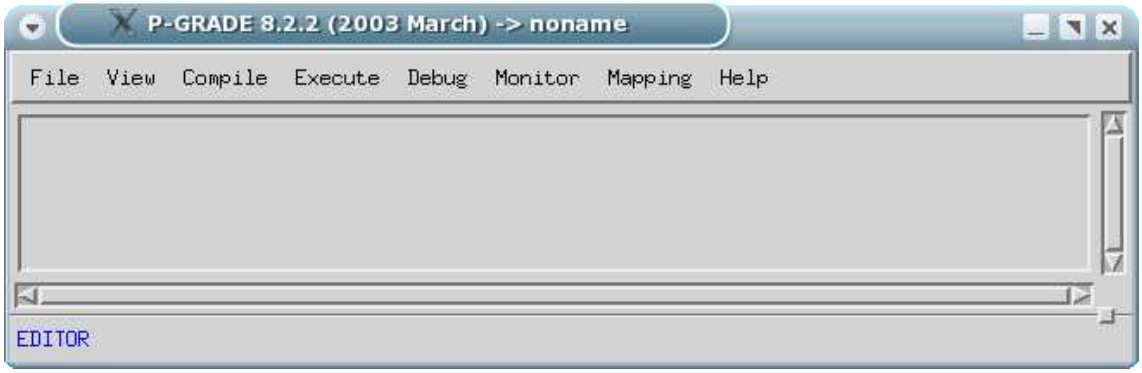
- taszk
- I/O csatornák
- szekvencia, elágazás, ciklus

[5] P-GRADE - Parallel Grid Run-time and Application Development Environment

Több elem összefogható, egységként kezelhető, ezzel segítve a program átláthatóságát.

A P-GRADE fordítás során C kódot generál, majd azt fordítja le, és futtatja. A generált programkód lehet PVM, vagy MPI alapú.

4.3.1. P-GRADE programfejlesztési eszközök



4.2. ábra. P-GRADE főablak

A 4.2 ábrán a P-GRADE főablaka, innen lehet indítani a fordítást, a futtatást, a monitorozást, ...

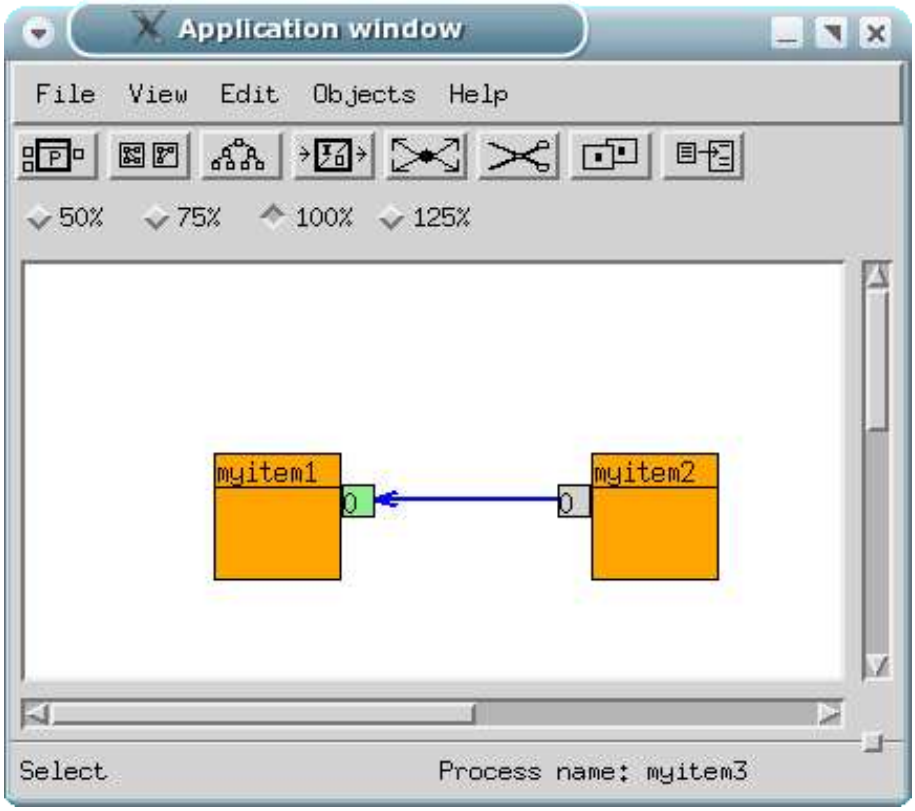
A 4.3 ábrán a P-GRADE alkalmzás szerkesző látható. A képen a két narancssárga folt (*myitem1, myitem2*) az alkalmazás két taszkja, melyek mellett a kommunikációs csatornák csatlakozása kis négyzettel vannak jelölve, zölddel, illetve a szürkével színezve (zöld - bemenet, szürke - kimenet), a két négyzetet összekötő nyíl a kommunikáció irányát mutatja.

A 4.4 ábrán egy taszk belső felépítése látható. A taszkbeli modulok a futás során fentről lefelé követik egymást. Ezen a példán a taszk egy inputra vár egy másik taszktól, majd egy feltétel kiértékelése után az igaz ágon egy ciklus fut le, a hamis ágon egy szekvencia.

Új építő elem úgy adható hozzá a programhoz, hogy egy függőleges vonalra kattintva a jobb egérgombbal kiválasztjuk, hogy milyen elemet szeretnénk oda beszúrni. A szerkesztő ablak alsó részében látható egy szövegszerkesztő, ott adhatjuk meg egy szekvencia működését, vagy egy ciklus -, illetve egy elágazás feltételét, stb. Azt az építő elemet szerkesztjük, amelyik ki van választva, azaz piros keret fogja körül.

4.3.2. P-GRADE példa program

A P-GRADE bemutatásához, egy példa programot készítek. Ebben a programban lesz két taszk, és az egyik elküld a másiknak egy üzenetet, a másik ezt megjeleníti.



4.3. ábra. P-GRADE alkalmazás szerkesző

P-GRADE alkalmazás szerkesztő

hello_slave

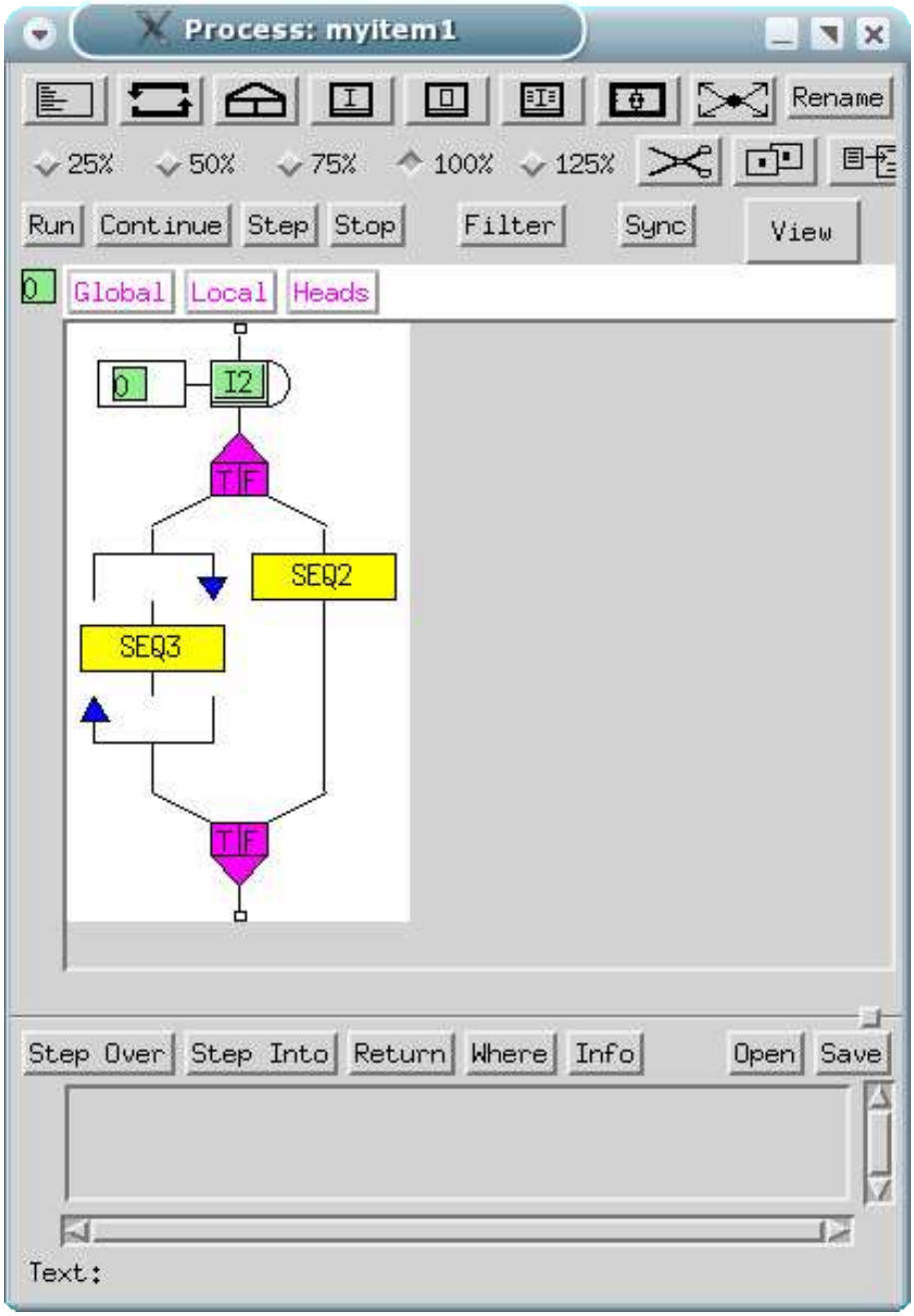
A 4.6 ábrán a hello_slave taszk látható, amint egy szekvenciából, és egy üzenet küldésből áll. A szekvenciában inicializálom a $data$ változót (ez nem látszik az ábrán, $data = 13$).

hello_master

A 4.7 ábrán a hello_master taszk látható, amint egy szekvenciából, egy üzenet fogadásból, és egy újabb szekvenciából áll. A szekvenciában inicializálom a $data$ változót (ez nem látszik az ábrán, $data = 0$), majd várok egy üzenetre (arra, amit a hello_slave küld). Miután megkaptam az üzenetet, megjelenítem azt a kijelzőn.

A program fordítása

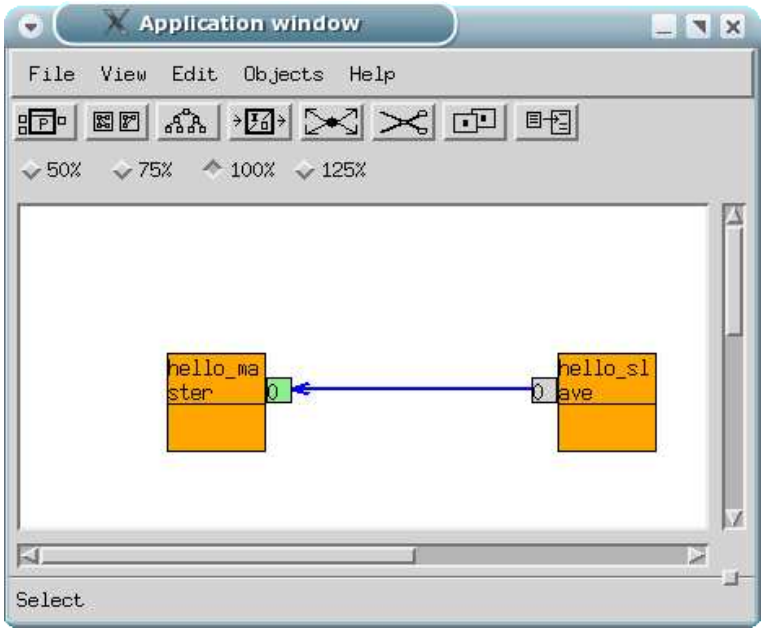
Miután elkészültünk a programok megírásával, le kell fordítani azokat. Ezt a P-GRADE főablakában (4.2 ábra) tehetjük meg. Fordítás előtt azonban érdemes eldönteni, hogy akarjuk-e megjeleníteni (monitorozni) a program futását. Amennyiben igen,



4.4. ábra. Egy taszk belseje

akkor be kell kapcsolni a főablak *Monitor* menü, *Mode* almenüjében a nyomkövetést (4.8 ábra).

A fordítás a főablak *Compile* menüjének *Build All* almenüpontjával történik. Ekkor feljön egy új ablak, amelyben a fordító program üzenetei vannak. Sikeres fordítás esetén a felbukkanó új ablak állapot kijelző sorában a *SUCCESS* felirat jelenik meg.

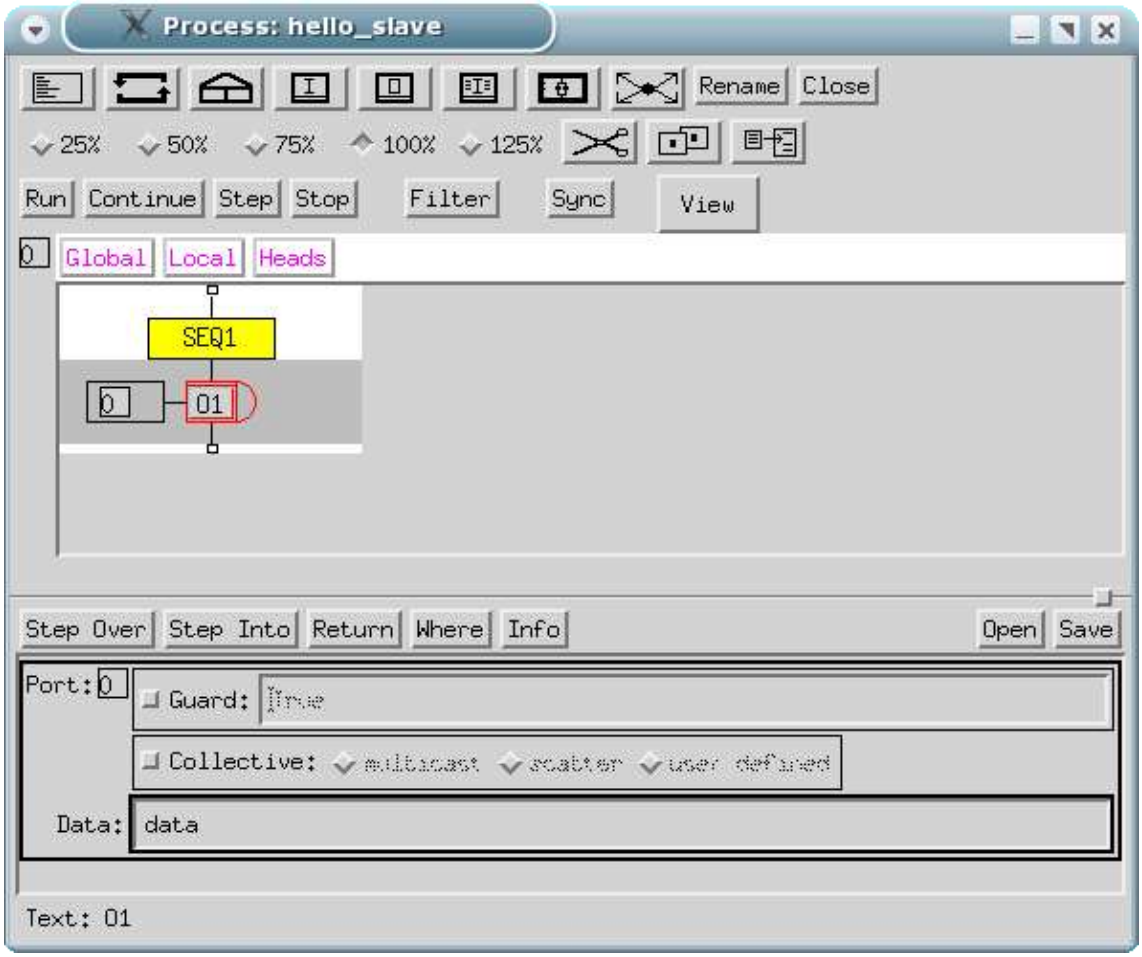


4.5. ábra. P-GRADE alkalmazás szerkesző

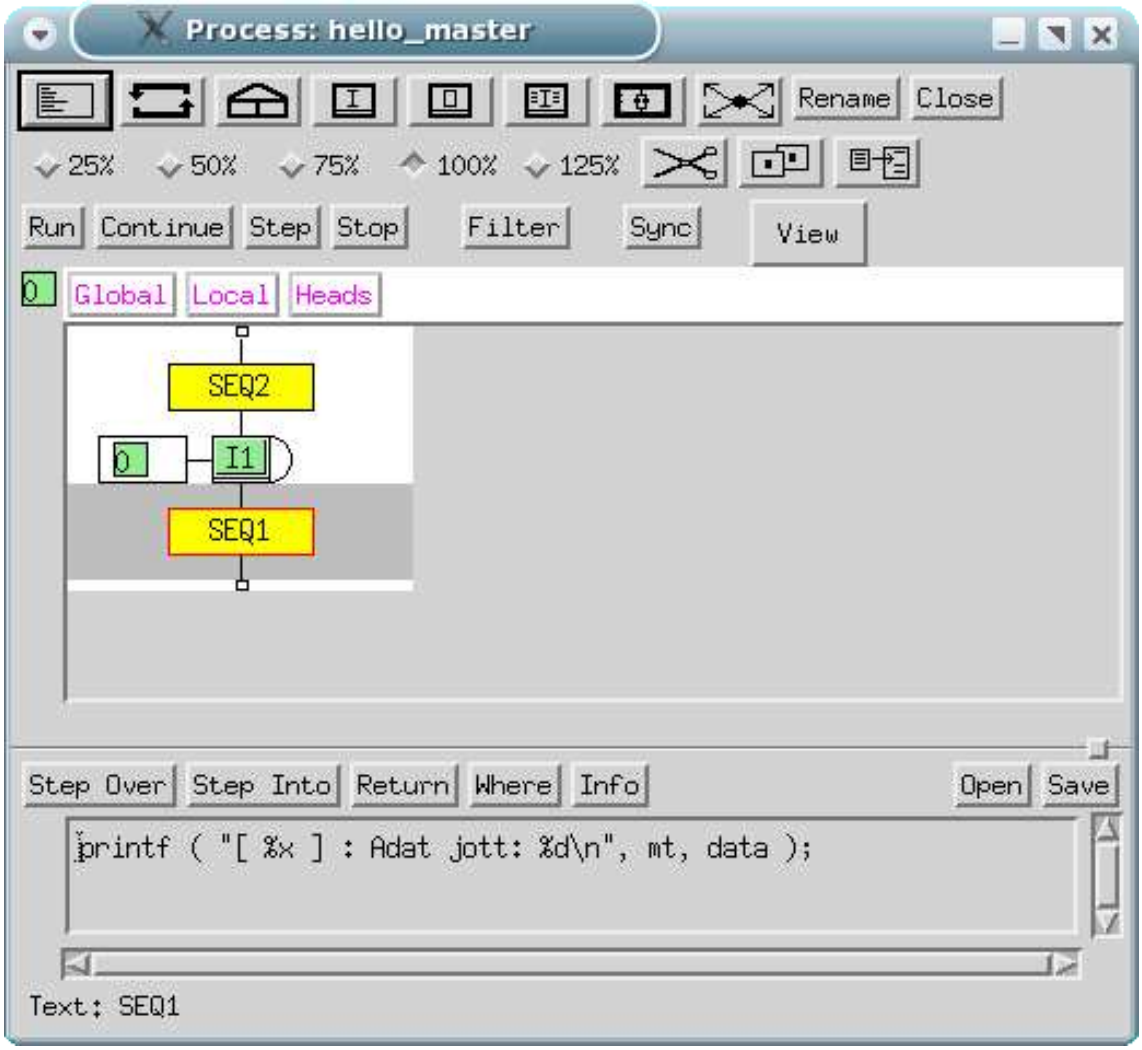
A példa program fordítási üzenetei a 4.9 ábrán láthatók.

Futtatás, monitorozás

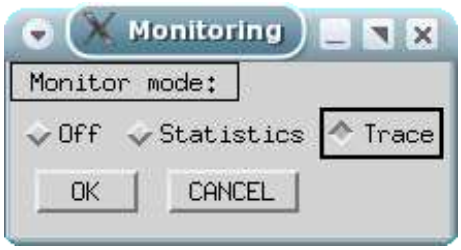
A futtatás a főablak *Execute* menüjének *Run All* almenüpontjával történik. Ekkor megjelenik egy újabb ablak, oda kerülnek a futási üzenetek (itt jelenik meg az általam kiírt üzenet is), ez látható a 4.10 ábrán. Futás közben már monitorozhatjuk is a programunkat a főablak *Monitor* menüjének *Visualize* almenüpontjával, A példa program monitorozásán (4.11 ábra) látható, hogy egy kommunikáció volt a program futása során, és hogy mely taszkok mely gépeken futottak, vagy esetleg éppen futnak.



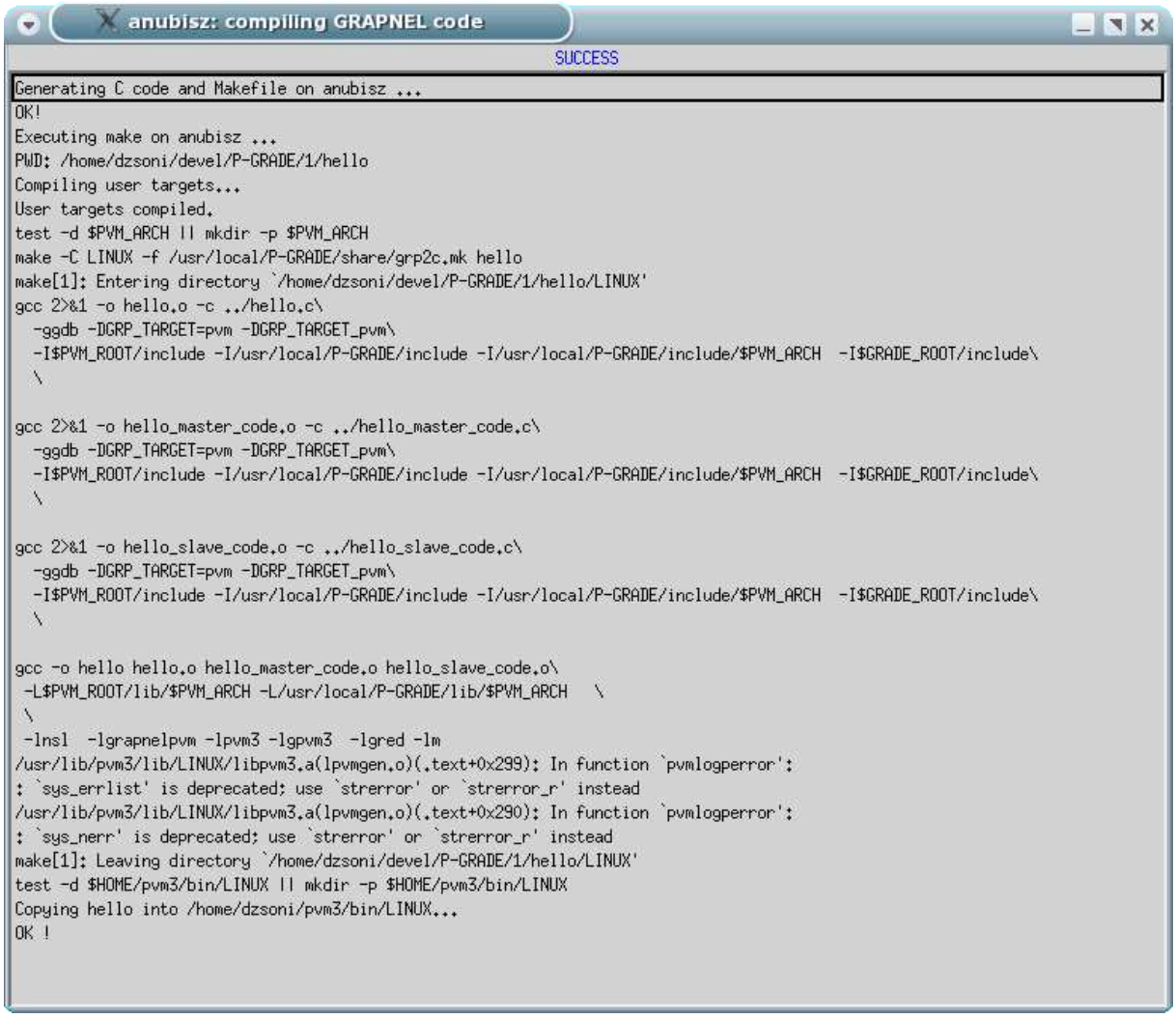
4.6. ábra. A hello_slave taszk



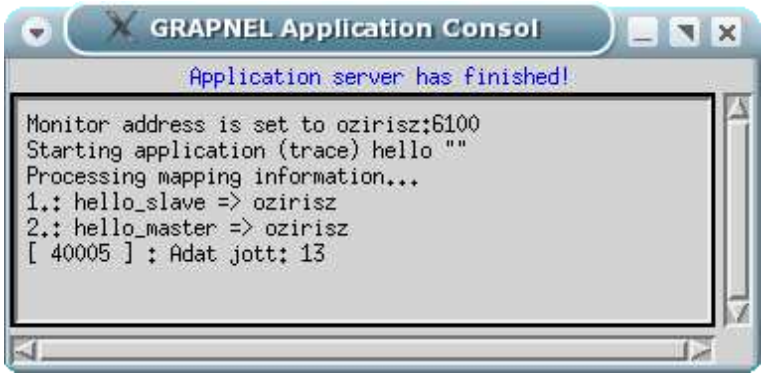
4.7. ábra. A hello_master taszk



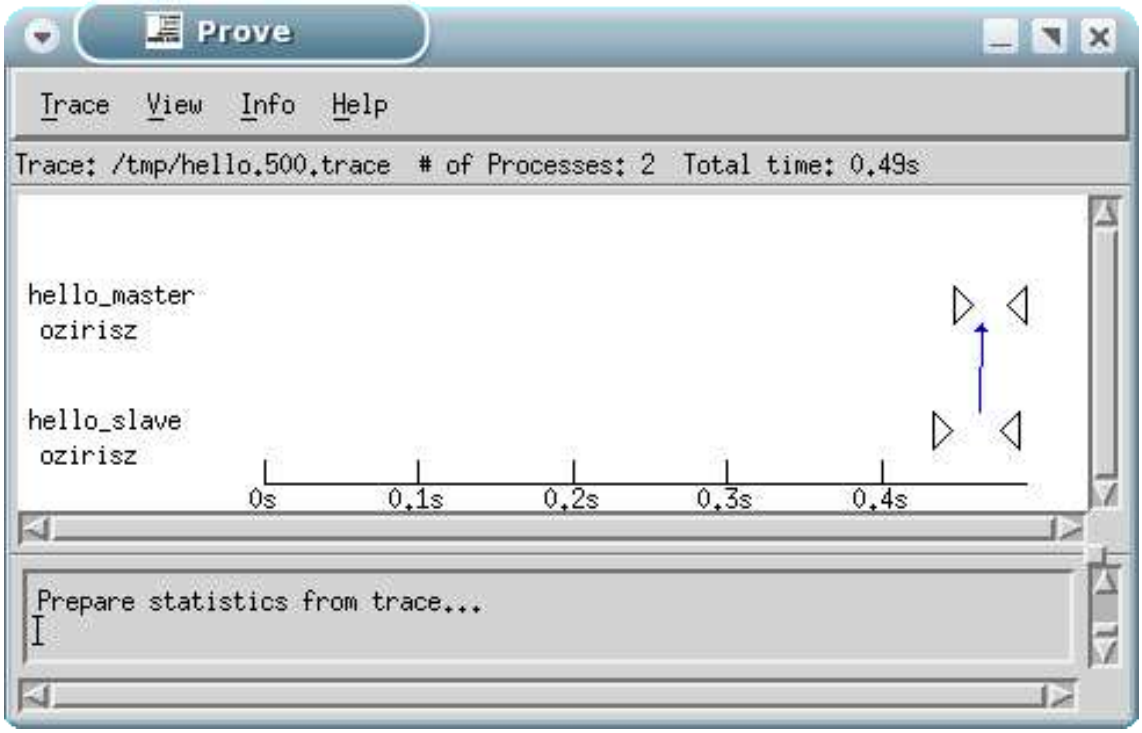
4.8. ábra. Nyomkövetés bekapcsolása



4.9. ábra. Fordítási üzenetek



4.10. ábra. Futási üzenetek



4.11. ábra. Monitorozás

5. fejezet

Párhuzamos elemenkénti feldolgozás

5.1. Leírás

Az elemenkénti feldolgozhatóság azt jelenti, hogy a függvény eredményét úgy kapjuk meg, hogy a bemenet minden egyes elemén ugyan azt a műveletet hajtjuk végre, egyiken a másikon után. Az ilyen függvények értelmezési tartománya és értékkészlete általában egy rendezett sorozat, vagy egy halmaz. Elemenként feldolgozható függvények olyan gyakran használt függvények osztályát képzik, mint például összefésüléses rendezés, halmazok uniójának számítása, időszerűsítés, stb.

5.2. Jelölések, adattípusok

A [15]-ben, és a [16]-ban megismert jelölési módszert fogom alkalmazni. A továbbiakban az ott bevezetett jelöléseket mutatom be.

5.2.1. Szekvencia

seq(T) jelölje a T típusú elemekből álló szekvenciát. A szekvenciáknál fontos az elemek sorrendje, így a « a, b, c » nem ugyan az, mint a « b, a, c ».

Legyen $T := seq(T_0)$, $t : T$ és t = « a_1, a_2, …, a_n », ahol $n \in \mathbb{N}$. Az alábbi függvényeket definiálom, és használom a továbbiakban a szekvencia típuson:

Név	**paraméterek**	**hatás**	**leírás**
dom	$T \to \mathbb{N}$	$dom(t) ::= n$	hossz
lov	$T \to T_0$	$lov(t) ::= a_1$, **ha** $dom(t) > 0$	fejelem
hiv	$T \to T_0$	$hiv(t) ::= a_n$, **ha** $dom(t) > 0$	utolsó elem
loext	$T \times T_0 \to T$	$loext(t, t_0)$ ::= « $t_0, a_1, \ldots, a_n$ »	alsó kiterjesztés
hiext	$T \times T_0 \to T$	$hiext(t, t_0)$ ::= « $a_1, \ldots, a_n, t_0$ »	felső kiterjesztés
lorem	$T \to T$	$lorem(t)$ ::= « $a_2, \ldots, a_n$ », **ha** $dom(t) > 0$	fejelem törlése
hirem	$T \to T$	$hirem(t)$ ::= « $a_1, \ldots, a_{n-1}$ », **ha** $dom(t) > 0$	utolsó elem törlése

Ezeken kívül vezessük be a következő jelölést is: $[t] = \{a_1, a_2, \ldots, a_n\}$, azaz a t szekvencia elemeiből képzett halmaz.

5.2.2. Specifikációkhoz szükséges jelölések, magyarázat

$k, l \in \mathbb{N}$

- H tetszőleges halmaz, melyen értelmezett a < rendezés
- $BLOKK = (s : \mathbb{N}, d : \mathbb{N}, f : H)$, egy blokkot leíró hármas, s a blokk első elemének indexe, d a blokk-rész mérete és f a blokk-rész medián értéke
- $BV = vektor[1..k] : BLOKK$, egy teljes blokk
- $BVS = seq(BV)$, teljesen diszjunk felbontás
- $BVSP = BVS^k$, az aktuálisan feldolgozott blokk
- $X = seq(H)^k$
- $Y = vektor[1..l] : \wp(H)$
- $YV = (vektor[1..l] : \wp(H))^k$, a részeredmények összegyűjtéséhez
- $range(s)$: intervalluma s-nek: $[lob(s)..hib(s)]$
- $CDD : X \times BVS \mapsto \mathbb{L}$
  $CDD(x, b) = cdd(x, ((x_i(b(j)[i].s \ldots b(j)[i].s + b(j)[i].d - 1))_{j \in range(b)})_{i=1}^{k})$

Ha nem definiálom felül másképp ezeket a jelöléseket, ott ezek érvényesek.

5.3. Teljesen diszjunkt felbontás

Miután az a célunk, hogy párhuzamos elemenkénti feldolgozást készítsünk, ahol az input minden egyes elemén ugyan azt a műveletet hajtjuk végre, így adott az az ötlet, hogy osszuk fel az inputot diszjunkt részekre, és az így kapott részeket dolgozzuk fel külön-külön processzorokon. Miután ezeket feldolgoztuk, már csak ismét össze kell fésülni azokat.

Legyen H egy halmaz, amelyen értelmezett a $<$ rendezés, $S = seq(H)$. Legyen $X = S^k$, ahol $k \in \mathbb{N}$. Az $x \in X$ egy teljesen diszjunkt felbontása az $x^{(1)}, x^{(2)}, \ldots, x^{(r)}$, $(r \in \mathbb{N}) \iff \forall i, j \in [1..r], (i \neq j)$-re és $\forall u, v \in [1..k]$-ra teljeülnek az alábbiak:

$$[x_u^{(i)}] \cap [x_v^{(j)}] = \emptyset \quad (5.1)$$

$$\exists i \in Perm(1, 2, \ldots, r) : \forall u \in [1..k] : x_u = x_u^{(i(1))} + + x_u^{i(2))} + + \ldots + + x_u^{(i(r))} \quad (5.2)$$

Az 5.1-es formula biztosítja a felbontás diszjunktságát, míg az 5.2-es a teljességet. A továbbiakban az x teljesen diszjunkt felbontására ez a jelölés is megtalálható: $cdd(x, (x^{(1)}, x^{(2)}, \ldots, x^{(r)}))$. Az $x^{(1)}, x^{(2)}, \ldots, x^{(r)}$ elemek azok a blokkok, amelyeket szét fogunk osztani a processzorok között. Amennyiben $k = 1$, azaz X egydimenziós, akkor a teljesen diszjunkt felbontás megegyezik a diszjunkt felbontással, de többdimeniós esetben a teljesen diszjunkt felbontás egy sokkal erősebb követelmény [15],[14].

Legyen N az $x \in X$ **csomag mérete**, M pedig a **halmaz mérete**, azaz

$$\begin{aligned} N &= \sum_{i=1}^{n} dom(x_i) \\ M &= |\bigcup_{i=1}^{k} [x_i]| \end{aligned}$$

5.3.1. Példa teljesen diszjunkt felbontásra

Legyen $S = seq(\mathbb{N})$, $X = S^4$, x_1 = « $1, 2, 3, 4$ », x_2 = « $2, 3, 4$ », x_3 = « $1, 3, 4, 5, 6, 7$ », x_4 = « $1, 2, 4, 5, 6, 8, 9$ ». Ennek egy lehetséges teljesen diszjunkt felbontása:

$$\begin{aligned} x^{(1)} &= (\ll 1, 2 \gg, \ll 2 \gg, \ll 1 \gg, \ll 1, 2 \gg) \\ x^{(2)} &= (\ll 3, 4 \gg, \ll 3, 4 \gg, \ll 3, 4 \gg, \ll 4 \gg) \\ x^{(3)} &= (\ll\gg, \ll\gg, \ll 5, 6, 7, 10 \gg, \ll 5, 6, 8, 9 \gg) \end{aligned}$$

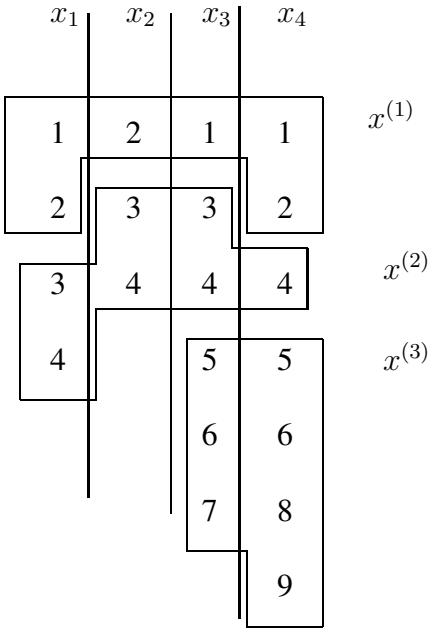
és a csomag -, illetve halmaz mérete:

$$\begin{aligned} N &= 20 \\ M &= 9 \end{aligned}$$

Amennyiben van p darab processzorunk, és egy bizonyos x inputot szeretnénk feldolgozni, akkor célszerű az inputot p darab teljesen diszjunkt részre felosztani, majd azokat a részeket szétosztani a processzorok között. Hogy minden processzor egyformán legyen terhelve célszerű, a processzorokhoz jutó blokkok csomag méretét $\frac{N}{p}$-re választani. A blokkok csomag méretének meghatározásához ki kell számítanunk az elemek unióját, amely szintén egy elemenként feldolgozható függvény.

Másik lehetőség, hogy egy blokk méretét $\frac{M}{p}$-nek vesszük. Ez esetben egyes blokkok halmaz mérete között nagy különbségek lehetnek, mint a fenti példában a blokkok csomag mérete nagyjából kiegyensúlyozott $(6, 7, 7)$, míg a halmaz méretük: $2, 2, 5$.

A processzorok egyforma terhelése céljából, az inputot blokkokra vágjuk, és ezeket a blokkokat dinamikusan osztjuk szét a processzorok között. Ha egy processzor elkészült a blokkja feldolgozásával, akkor egy újabb blokkot kap mindaddig, amíg van



5.1. ábra. Példa teljesen diszjunkt felbontásra

fel nem dolgozott blokk. Ez a megoldás, ha B a legnagyobb blokk csomag mérete, a p processzoron legfeljebb $\frac{M}{p} + B$ lépést igényel[15].

A programozó feladata B értékének kiválasztása, Kis B esetén sok blokk lesz, így megnő a blokkok terítéséhez szükséges kommunikációs költség. Nagy B esetén a processzorok terheltsége kiegyensúlyozatlanná válhat.

5.3.2. Program teljesen diszjunkt felbontásra

A teljesen diszjunkt felbontáshoz a $k \in \mathbb{N}$ dimenziós input minden egyes paraméterét, azoknak minden egyes részhalmazát egy hármassal fogjuk reprezentálni, amely hármas tartalmazza az inputon belüli kezdő indexet, a részhalmaz hosszát, és a részhalmaz medián értékét, azaz az alábbi hármassal leírható:

$$(s : \mathbb{N}, d : \mathbb{N}, h : H)$$

Például a 5.3.1 fejezetben található példa az alábbi hármasokkal reprezentálható:

$$((0, 4, 2), (0, 3, 3), (0, 6, 4), (0, 7, 5))$$

Egy ilyen leíróból ki lehet számolni az input zsákméretét ($N = 4+3+6+7 = 20$), és amennyiben N nagyobb, mint egy konstans B akkor az inputot két részre kell vágni. Az új kapott blokk részeket is vágni kell, amennyiben zsákméretük nagyobb, mint B. Az aprítás addig folyatatódik, amíg minden blokkrész zsákmérete kisebb nem lesz, mint B[15].

Vágás után két új blokk leíró keletkezik, a kettőnek az össz hossza megegyezik annak a blokk leírónak a hossz értékével, amelyiket felbontottuk.

A vágás során nincs ismeretünk az inputról, csak bizonyos medián értékeket ismerünk, és azok közül kiválasztjuk a középsőt remélve, hogy az nem áll messze a tényleges, azaz a teljes input medián értékétől.

Például, ha az előbbi példában vágunk a 4-es mentén, akkor az alábbi leírókat kapjuk:

$$((0,4,2),(0,3,3),(0,3,3),(0,3,2)),$$
$$((0,0,0),(0,0,0),(2,3,6),(2,4,6))$$

Feltéve, hogy $B = 8$, akkor az első 4-est ismét vágni kellene.

Az így kapott blokkleírók segítségével a teljes input ábrázolva van, és az egyes blokkleírók által képzett részhalmazok diszjunktak, tehát teljesen diszjunkt felbontáshoz jutottunk.

Inicializálás

Ahhoz, hogy eldönthessük mekkora részekre szeretnénk bontani az eredeti inputot tudni kell, hogy mekkora a zsákmérete.

1. Specifikáció:

$$\begin{aligned}
A &= X \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times BVS \times BVS && (5.3)\\
&\quad\ x \quad N \quad B \quad j \quad bb \quad blk \\
B &= X && (5.4)\\
&\quad\ x' \\
Q &= (x = x') && (5.5)\\
inv &: j \in [0..K] \wedge N = \sum_{i=1}^{j} dom(x'_i) && (5.6)\\
inv &: \forall (i \in [1..j] : blk\,[i] = && (5.7)\\
&\quad\ (lov(x_i), dom(x_i), x_i(\frac{lov(x_i) + dom(x_i) - 1}{2}))) \\
FP &\Rightarrow j = K \wedge B = opt(N) \wedge CDD(x', bb) \wedge bb = \ll blk \gg && (5.8)
\end{aligned}$$

A megoldó program:

$$s_0 \ : \ j, N := 0, 0$$

$$S: \begin{cases} \square & N, j, blk[j+1] := & N + dom(x_{j+1}), j+1, & \\ & & (lob(x_{j+1}), dom(x_{j+1}), x_{j+1}(\frac{lov(x_{j+1})+dom(x_{j+1})-1}{2})) & \textbf{ha } j < K \\ \square & B, bb := & opt(N), \ll blk \gg & \textbf{ha } j = K \end{cases}$$

Blokkok generálása

Miután meghatároztuk, hogy mekkora darabokra is vágjuk szét az inputot, kezdődhet a tényelges darabolás. 2. Specifikáció:

$$\begin{aligned} A &= X \times H \times BV \times BV \times BVS \times BVS && (5.9)\\ &\quad\ x \quad h \quad ub \quad lb \quad bb \quad blk \\ B &= X \times BVS && (5.10)\\ &\quad\ x' \quad bb' \\ Q &= (x = x' \wedge bb = bb') && (5.11)\\ inv &: CDD(x', b \circ bb) && (5.12)\\ FP &\Rightarrow bb = \ll\gg && (5.13) \end{aligned}$$

A megoldó program:

$$\begin{array}{lll} \textbf{while} & dom(bb) \neq 0 & \textbf{loop} \\ & h & := CutValue(lov(bb)) \\ & ub, lb & := Cut(x, lov(bb), h) \\ & bb, b & := Update(bb, b, ub, lb) \\ & lorem(bb) & \end{array} \qquad (5.14)$$

A *CutValue* algoritmusa megtalálható a [15]-ben, és könnyen implementálható egy szekvenciális programmal. Megkeressük a K medián közül azt, amelyik a mediánok átlagához legközelebb van, és annak a mentén kell vágni.

A *Cut* algoritmusa visszatér két blokk leíróval, amelyek a h menti vágás alsó, illetve felső fele. Mivel az input rendezett, ezért abban lehet binárisan keresni. A keresés lépésszáma: $O(log\ n)$.

Az *Update* pedig annak függvényében dönti el tovább kell-e bontani, hogy a kapott blokkok leírók (ub, lb) zsákmérete hogyan viszonyul B-hez (bb-be teszi vissza azt, amelyiket tovább kell bontani, egyébként b-be a már felbontottak közé).

5.4. Elemenként feldolgozható függvények

Legyen X ugyanaz, mint a 5.3 fejezetben, H' tetszőleges halmaz, $P = \wp(H')$, és $Y = P^l$, ahol $l \in \mathbb{N}$. Az $f : X \to Y$ függvény elemenként feldolgozható, ha

$\forall x, x^{(1)}, x^{(2)}, \ldots, x^{(r)} \in X : cdd(x, (x^{(1)}, x^{(2)}, \ldots, x^{(r)}) \wedge \forall i \in [1..l]$ -re teljesülnek az alábbi feltételek:

$$\bigcup_{j=1}^{r} F_i(x^{(j)}) = F_i(x) \tag{5.15}$$

$$\bigcap_{j=1}^{r} F_i(x^{(j)}) = \emptyset \tag{5.16}$$

5.4.1. Egy változós - egy értékű

Először az az eset látható, amikor mind X, mind Y egydimenziósak, azaz $k = l = 1$.

3. Specifikáció:

$$A = X \times Y \times H \times \mathbb{L} \tag{5.17}$$
$$x \quad y \quad e \quad ch$$
$$B = X \tag{5.18}$$
$$x'$$
$$Q = (x = x') \tag{5.19}$$
$$inv : cdd(x', (x, x' - x)) \tag{5.20}$$
$$inv : y = F(x' - x) \tag{5.21}$$
$$FP \Rightarrow (x = \ll\gg) \tag{5.22}$$

A 5.20 invariáns mutatja, hogy x (a még feldolgozandó rész), és az $(x' - x)$ (a már feldolgozott rész) teljesen diszjunkt felbontása az x' eredeti inputnak, míg a 5.21 invariáns biztosítja, hogy a feldolgozás eredménye az y-ba kerül. A 5.22 fixpont feltételt elérjük, amikor az összes x-et feldolgoztuk, azaz $x = \ll\gg$.

A megoldó program:

$$s_0 : y, ch := \emptyset, \downarrow$$

$$S : \begin{cases} \square \quad e, ch := lov(x), \uparrow & \textbf{ha } x \neq \ll\gg \wedge \neg ch \\ \square \quad x, y, ch := lorem(x), y \cup F(e), \downarrow & \textbf{ha } ch \end{cases}$$

5.4.2. Két változós - egy értékű

Ez esetben $k = 2$ és $l = 1$, azaz F függvénynek két paramétere van, azaz az x input két szekvenciából áll össze. Az egyik legismeretebb példa erre az esetre két halmaz

uniójának kiszámítása. Annak belátásához, hogy ez elemenként feldolgozható, elég belátni, hogy a 5.15 és a 5.16 teljesülnek minden teljesen diszjunkt felbontásra.

4. Specifikáció:

$$
\begin{aligned}
A &= X \times Y \times H \times \mathbb{L} && (5.23)\\
&\quad\ x \quad y \quad e \quad ch \\
B &= X && (5.24)\\
&\quad\ x' \\
Q &= (x = x') && (5.25)\\
inv &: cdd(x', (x, x' - x)) && (5.26)\\
inv &: y = F(x' - x) && (5.27)\\
FP &\Rightarrow (x = \ll\gg, \ll\gg) && (5.28)
\end{aligned}
$$

A megoldó program:

$$s_0 \ : \ y, ch := \emptyset, \downarrow$$

$$
S : \begin{cases}
\square \quad e, ch := min\left\{lov(x_1), lov(x_2)\right\}, \uparrow & \textbf{ha } x \neq (\ll\gg, \ll\gg) \wedge \neg ch \\
\square \quad x, y, ch := (lorem(x_1), x_2), y \cup F(e, \emptyset), \downarrow & \textbf{ha } ch \wedge e = lov(x_1) \wedge e \neq lov(x_2) \\
\square \quad x, y, ch := (x_1, lorem(x_2)), y \cup F(\emptyset, e), \downarrow & \textbf{ha } ch \wedge e \neq lov(x_1) \wedge e = lov(x_2) \\
\square \quad x, y, ch := (lorem(x_1), lorem(x_2)), y \cup F(e, e), \downarrow & \textbf{ha } ch \wedge e = lov(x_1) \wedge e = lov(x_2)
\end{cases}
$$

5.4.3. Egy változós - két értékű

Ebben az esetben $k = 1$ és $l = 2$.

5. Specifikáció:

$$
\begin{aligned}
A &= X \times Y \times H \times \mathbb{L} && (5.29)\\
&\quad\ x \quad y \quad e \quad ch \\
B &= X && (5.30)\\
&\quad\ x' \\
Q &= (x = x') && (5.31)\\
inv &: cdd(x', (x, x' - x)) && (5.32)\\
inv &: y = F(x' - x) && (5.33)\\
FP &\Rightarrow (x = \ll\gg) && (5.34)
\end{aligned}
$$

A megoldó program:

$$s_0 \;:\; y, ch := (\emptyset, \emptyset), \downarrow$$

$$S : \left\{ \begin{array}{llll} \square & e, ch & := & lov(x), \uparrow \quad \textbf{ha } x \neq \ll\gg \wedge \neg ch \\ \square & x, y, ch & := & lorem(x), (y_1 \cup F_1(e), y_2 \cup F_2(e)), \downarrow \quad \textbf{ha } ch \end{array} \right.$$

5.4.4. k változós - l értékű (általános) eset

$k, l \in \mathbb{N}$

6. Specifikáció:

$$\begin{array}{rcll} A & = & X \times Y \times H \times \mathbb{L} & (5.35) \\ & & x \quad y \quad e \quad ch & \\ B & = & X & (5.36) \\ & & x' & \\ Q & = & (x = x') & (5.37) \\ inv & : & cdd(x', (x, x' - x)) & (5.38) \\ inv & : & y = F(x' - x) & (5.39) \\ FP & \Rightarrow & (x = \ll\gg, \ldots, \ll\gg) & (5.40) \end{array}$$

A megoldó program:

$$s_0 \;:\; y, ch := (\emptyset, \ldots, \emptyset), \downarrow$$

$$S : \left\{ \begin{array}{llll} \square & e, ch & := & min\left\{lov(x_i) \mid i = 1..k \wedge x_i \neq \ll\gg\right\}, \uparrow \quad \textbf{ha } x \neq (\ll\gg, \ldots, \ll\gg) \wedge \neg ch \\ \square & (& & \\ & y, ch & := & (y_i \cup F_i(sl(x, e)))_{i=1}^{l}, \downarrow \| \\ & & & \|_{i=1}^{k} (x_i := lorem(x_i), \\ & & & \textbf{ha } x_i \neq \ll\gg \wedge e = lov(x_i)) \\ &) & \textbf{ha } ch & \end{array} \right.$$

, ahol $\forall j \in [1..k]$

$$sl(x, e)(j) ::= \left\{ \begin{array}{ll} \ll e \gg & \textbf{ha } x_j \neq \ll\gg \wedge e = lov(x_j) \\ \ll\gg & \textbf{különben} \end{array} \right.$$

6. fejezet

Adatintenzív alkalmazások

6.1. Leírás

Adatintenzív - vagy I/O (input/output) intenzív - alkamazások alatt az olyan feladatokat megoldó programokat értjük, amelyek a feladat megoldása során nagy mennyiségű adatot dolgoznak fel és/vagy termelnek. Ilyen például a naptevékenységeket megfigyelő műholdak kimenő adatainak feldolgozása, SETI projektből nyert adatok feldolgozása, fizikai, biológiai, vagy kémiai kísérletek szimulációja, stb.

Ezek az alkalmazások adatok terrabyte-jait manipulálják. Ahhoz, hogy ekkora mennyiségű adat feldolgozása közben ne kelljen állandóan várakozni megközleítőleg 10GB/s -os adatátviteli sebességre van szükség. Az ilyen alkalmazásoknál a futási idő nagy részét az adatok mozgatása teszi ki [19].

6.2. Milyen környezetben?

Mielőtt belevágnánk a közepébe tisztázni kell, hogy mik is az elvárásaink a programmal szemben. Milyen környezetben szeretnénk futtatni, hol, ott mik a lehetőségek, stb. E dolgozat témája "Adatintenzív alkalmazások grid-es környezetben", ezért heterogén környezetben szeretnénk futtatni, így olyan fejlesztő eszközt kell választani, amely sok rendszeren elérhető. Mivel a program nagyon sok adattot fog feldolgozni, ennélfogva akár napokig is futhat, ezért jó lenne, ha valamilyen szintű hibatűréssel is rendelkezne. Ilyen lehetőségeket biztosít például a C/C++ nyelv a PVM könyvtárral kiegészítve.

6.3. Fordítási környezet

Az absztrakt programokat C++ nyelven a PVM könyvtár (3.4 -es verzió) segítségével valósítottam meg. A program elméletileg más UNIX szerű operációs rendszeren is lefordul, én Debian 3, illetve SuSE Linux 9.1 alatt írtam és teszteltem, a GNU C Compiler (*g++*) 3.3.3-as verziójával fordítottam.

A feladat megoldásához több részporgramot is készítettem, amelyek együttes munkája adja az eredményt. Első feladat volt az absztrakt adattípusok megvalósítása, szemelőtt tartva, hogy a program adatintenzív lesz.

Az absztrakt adattípusokat nem konkrét típusokból építettem fel, hanem sablonokat alkalmaztam, így a H alaphalmazt megváltoztatva megváltozik az összes ráépülő típus is.

Mivel a program adatintenzív, ezért nem lehet a teljes inputot beolvasni a memóriába, hanem az a háttértérolón foglal helyet, így fontos annak a gyors elérése. Ezt figyelembe véve nem a C++-os $iostream$-et választottam filekezeléshez, nem is a C-s $stdio$-t, hanem rendszerhívásokra építettem a programot. Ezzel a megoldással csorbult a program hordozhatósága, ezért megírtam a file kezelő *stdio*-ra épülő változatát is. Hogy a program majd melyiket használja, azt fordítás során lehet megválasztani. A *g++* -nak átadva a -DFILE_WITH_STDIO paramétert az *stdio*-s megoldás fordul le, különben a rendszerhívásos.

6.4. Adattípusok

Mivel a program adatintenzív, és ezért sok adatot kezel, ezért indexeléshez bevezettem egy *t_size* változót, amely 64 bites előjel nélküli egész (*typedef unsigned long long t_size*). Ennek az előjeles párja a *t_ssize*. Az absztarkt típusoknál látott vektort és direkszorzatot a $vektor$ osztály segítségével valósítottam meg, amely szintén sablon.

6.4.1. Sablonok

A formális definíciók során minden típus és művelet egy H típusra épül, amely nincs konkrétan meghatározva. H bármi lehet, csak értelmezve legyen rajta a $<$ rendezés, azaz elemi sorbarendezhetőek legyenek. Ezért az adattípusok implementálása során sablonokat készítettem, melyekből tetszőleges, az előbb emlíetett tulajdonságú H halmazzal konkrét típusok generálhatók.

Nat

Ennek a típusnak az a szerepe, hogy egy olyan H alaptípus legyen, amely szükség esetén méretezhető. Ez egy egyszerű, egész számok abrázolásához készített sablon, amelyből a $Nat128$ (128bites előjel nélküli egészek) készült. A $Nat128$ képezte a programban a H alaptípust. A Nat-on értelmezett műveletek az értékadás, összehasonlító műveletek ($==, !=, <=, <, >=, >$), az összeadás és a kivonás. Amennyiben más H alaptípust szeretnénk használni a programhoz, akkor fontos, hogy azon a típuson értelmezve legyenek az összehasonlító műveletek, az értékadás és a kivonás!

seq

Ez egy egyszerű H típusú elemekből álló szekvenciát valósít meg az absztrakt típuson megismert ($lov, lorem, hiext, dom$) műveletekkel, és kiegészítve néhány új függvénnyel, amelyek a PVM kommunikáció során elvégzik a szükséges adatok be-, illetve kicsomagolását.

File

A file I/O-t végző osztály, amely H típusú elemeket olvas be, vagy szúr a file végére. A szekvenciánál ismert műveleteket itt is megvalósítottam, hogy a két típust egyformán lehessen kezelni.

BlockPart

Egy blokk részt leíró típus, amely a blokkról 3 információt tárol, ez az absztrakt $BLOKK$ megvalósítása.

6.4.2. Sablonokból képzett típusok

- T_SeqH - $seqh$
- T_BlockPart - $BLOKK$
- T_BV - BV
- T_BVS - BVS
- T_X - X
- T_Y - Y

6.4.3. Mérések az adattípusokkal

A nagy adatmennyiség miatt azok nem tárolhatók mind a memórában, ezért izgalmas kérdés számunkra a file kezelés, és azt vesszük górcső alá. Az íráshoz használt teszt program az *iotest*.

File írás

A tesztelés során két dolgra voltam kíváncsi. Az egyik, hatékonyabb-e, ha rendszer hívásokat használok az *stdio.h* helyett, és ha igen, mennyivel. A másik, mennyivel hatékonyabb, ha az adatokat blokkonként kiírni , mint egyesével. A mérést 12-szer végeztem el, az alábbi táblázatba az átlageredményeket írtam, a részletes mérési eredmények a 57 oldalon a A.2.1 függelékben olvashatók. A mérést egy 1.4 GHz-s

512 MB DDR-SDRAM -os Compal CL-50 notebook-on végeztem. A mérés során 1GB-nyi *Nat128* típusú elemet (2^{26} db elem) írattam ki file-ba, blokk írás során 2^{14} db 2^{12} byte méretű blokkot.

	sys i/o	*stdio.h*
seq<Nat128>	61.75	65.3
Nat128	230.083	343.583

6.1. táblázat. File írás, futási idő másodpercben

A tábláztaból kiolvasható, hogy szekvenciák írása folyamán a rendszerhívásos megoldás 5.7%-kal gyorsabb, míg ha az elemeket egyesével írom ki, akkor a gyorsulás majd 50 %-os. Ennek oka lehet, hogy míg rendszerhívásoknál, a file-t *O_APPEND* paraméterrel nyitom meg, akkor minden írás automatikusan a file végére történik, még akkor is, ha *O_RDWR | O_APPEND* móddal nyitottam meg és két írás között a file közepéről olvastam[1]. Másik esetben, nehogy a file közepére írjak, kénytelen vagyok minden írás előtt a file végére pozícionálni[2].

6.5. A program

6.5.1. Futtatási környezet

A programot az ELTE-IK nyelvi laboratóriumában futtatam, ahol 40 db intel Pentium4-Celeron PC található, egyenként 256 MB DDR-SDRAM memóriával felszerelve, 100 MBit-es full-duplex hálózatba kötve. A file kiszolgálást NFS file szerver végzi. Midnen gépen Debian Linux operációs rendszer fut, és midengyiken van PVM, és MPI könyvtár és Condor. Ez egy dedikált cluster (lásd 7. oldal 2.2 fejezet), amelyet az ELTE hallgatói használhatnak.

6.5.2. grid-es változat

Tervezés

A programok formális leírásából kiindulva (5.3.2) először egy alapváltozatot terveztem, amely két lépésben végez a feladatot. Először teljesen diszjunkt részhalmazokra bontja a inputot, majd feldolgozza azt.

Az egész folyamatot egy taszk (*master*) kezeli, amely a hozzá befutott adatokból dötni el, hogy kell-e, és ha igen, akkor hol vágni a halmazokat. Az inputot egy-egy adatkiszolgáló egység biztosítja majd, amelynek nincs más feladata, mint adatok

[1] lásd: LINUX manual 2 (*man 2 open*)
[2] lásd: LINUX manual 3 (*man 3 fopen*)

továbbítása az igénylők felé. Az adatokat nem a *master* dolgozza fel, hanem kiadja azokat más taszkoknak. A feladat ebből a szemszögből nézve tipikus **master-worker** probléma, azaz van egy taszk, amely szétosztja a feladatokat a más taszkok között.

Az adatfeldolgozó egységek - miután elvégezték a feladatukat - az eredményt visszaküldik a *master*-nak.

Áttervezés

Az első változat nyilvánvaló hibája, hogy amíg nincs kész az input teljes felbontásával, addig nem kezdődik el a feldolgozás. Ez így nem hatékony, hiszen amint elkészült egy olyan részhalmaza az inputnak, amelyet már nem bontunk tovább, az azonnal kiadható feldolgozásra, ezzel is csökkentve a program futási idejét.

Ezen kívül a *master* idejének tetemes részét a feldolgozó egységek által küldött eredmények mentése fogja kitenni, így nem halad sem a felbontással, sem pedig a feladatok szétosztásával. Ezért ezt a feladatot célszerű más taszkra bízni, amely semmi mást nem tesz, mint átveszi a feldolgozó egységek outputját, és elmenti azokat.

Újbóli áttervezés

A előbbi terv már majdnem tökéletes, de figyelembe kell venni a futási környezetet is, azaz ,hogy nem egy osztott memóriás gépünk van sok processzorral, hanem sok egy processzoros gépünk hálózatba kötve. Az input egy-egy gépen található, és az adatokat át kell küldeni a többi gépre.

Sok feldolgozó egység esetén, amíg az adatkiszolgáló egység egy feldolgozó adatszükségletét kielégíti, addig a többiek várnak, hogy mikor kerül rájuk a sor. Hasonlóan a *master* is vár! Ha csoportosítjuk a feldolgozó egységeket, és minden csoporthoz saját adatkiszolgálót rendelünk, csökkentjük az adatszerverek terheltségét, és mind a dolgozók, mind a *master* várakozási idejét.

Megvalósítás

A tervezés során a feladatot négy alrészre bontottam, ezért a megoldó program is négy részből épül fel. Lesz a *master*, az adatkiszolgáló a *data*, a feldolgozó a *worker* és az adatrögzítő a *reciever*.

A *master* nem csoportosítja ténylegesen a dolgozókat, hanem a dolgozók számának függvényében hoz létre adatkiszolgáló egységeket, és a munka terítése során minden dolgozónak megmondja, hogy kihez forduljon adatért.

A *master* taszk folyamat stuktogrammja a 6.1 ábrán látható a 48 oldalon.

6.1. ábra. *master* taszk

6.5.3. cluster-es változat

Tervezés

Miután elkészült a grid-es változat, azt optimalizálni kell, hogy a cluster-es futtatási környezetre. Figyelembe véve a mért PVM átviteli sebességet, és a környezet (100 MBit-es full-duplex hálózat) elméleti maximális áteresztő képességet ($100MBit/sec = 12.5$MB/s), úgy terveztem meg a cluster-es verziót, hogy a PVM kommunikáció minél kisebb legyen a taszkok között.

Így a feldolgozó egységek közvetlenül az input file-okból olvassák ki a számukra szükséges feldolgozandó adatokat, majd egy saját file-ba írják ki az eredményt. Ezzel csak a PVM -et kerüljük ki, hiszen - miután NFS file szerverről olvassák a feldolgozó egységek az adatokat - ezt az adatmennyiséget így is, úgy is át kell küldeni a hálózaton. Nincs szükség adatrögzítő egységre sem, mivel a már feldolgozott részeredményeket a dolgozók kiírhatják egy-egy file-ba. Ezzel a módszerrel rengeteg rengeteg PVM kommunikációs költséget lehet megtakarítani.

Ha nem is áttörő, de azért némi gyorsulás várható ettől a megoldástól.

Megvalósítás

Az adatkiszolgáló egységek funkcióját redukáltam a *master* kiszolgálására. A feldolgozó egységek csak a *master*-rel kommunikálnak, új feladatot kapnak, és jelzik, ha elkészültek. Adatrögzítő egységre nincs szükség. A *master* csak annyi számú *data*-t indít, ami ahhoz kell, hogy a felbontás folyamatosan haladhasson.

A *master* lényegében nem vátozott, ezért a stuktogrammja megegyezik grides változatéval (6.1 ábra)

6.5.4. Hibatűrés a programban

A program tervezése során fontos tényező a hibatűrés, ugyanis grid-es környezetben nő a rendszer egyes részeinek meghibásodási valószínűsége. A programnak olyannak kell lennie, hogy bizonyos hibák esetén ne álljon le, hanem a hibát észlelve, majd amennyiben lehet kijavítva folytassa a futását.

A hiba kezelését nem lehet egyszerűen egy taszkra bízni, hanem minden taszknak valamilyen szinten kezelnie kell a hibát. Ezt figyelembe véve a hibakezelés oroszlán részét a *master*-ra bíztam, míg a többi taszk megoldást vár a *mater*-től, ha érinti a hiba (például a kommunikációs partnere kiesett), míg más esetben egyszerűen nem foglalkozika dologgal.

A PVM lehetőségeihez mérten csak a taszkok kiesését figyelik a taszkok, egy hoszt kiesése implikálja a rajta lévő taszkok kiesését is. Mindkét változatban a *master* figyel minden általa indított taszkot, a többi taszk pedig csak azokat, amelyekkel közvetlen kapcsolatban van. A munka szétosztásánál a *master* feljegyzi, hogy melyik feldolgozó egységnek mit adott ki feldolgozásra, és ha egy olyan feldolgozó egység esik ki, amelyik valamin dolgozott, akkor annak a feladatát visszaveszi a feldolgozandó feladatok közé, majd megpróbál újat indítani helyette. Ha nem sikerül, akkor csökkenti a *worker*-ek számát. Ha már minden dolgozó elfogyott (egyik helyett sem sikerült újat indítani), akkor a program leáll, mivel úgysem haladna, a teljesen diszjunkt felbontás végeztével holtpontra jutna.

grid-es változat

Ebben a változatban az előbb leírtakhoz hozzá tartozik az is, hogy az adatrögzítőt is értesíteni kell arról, hogy ki esett ki, hátha attól vár adatot (ne várjon hiába, mert akkor holtpontra jutna a program).

Amennyiben egy adatszolgáltató esik ki, akkor azt pótolni kell, majd értesíteni azt a dolgozót, amelyik éppen a kiesett adatszolgáltatóval dolgozott, hogy honnan kérje ezután adatokat.

Ha az adatrögzítő esik ki, akkor elméletileg tovább futhatna a program (amennyiben sikerülne újat létrehozni helyette), de az eddig kimentett adatok elveszhetnek, vagy sérült lehet a kimentett állomány, ezért a program leáll.

Ha a *master* esik ki, mivel lényegében az koordinálja a műveleteket, valamennyi taszk leáll.

cluster-es változat

Ebben az esetben egyszerűbb a hibakezelés, mivel nem kell foglalkozni sem az adatrögzítővel, sem azzal, hogy a kiesett adatszolgáltató új címét szétterítsük, mivel azzal csak a *master* áll kapcsolatban.

6.6. Mérés

A host-ok jobb kihasználtsága érdekében minden egyes host-on 4 feldolgozó egységet indítok, így például a 16 host-os mérés során 64 feldolgozó egység van.

A program méréséhez 4, egyenként 256MB mértű halamaz unióját számoltattam ki a programmal. Minden halmazban $Nat128$ típusú elemek voltak, egy elem $128bit = 16byte$ méretű, azaz egy input file-ban $\frac{2^{28}}{2^4} = 2^{24}$ darab elem van.

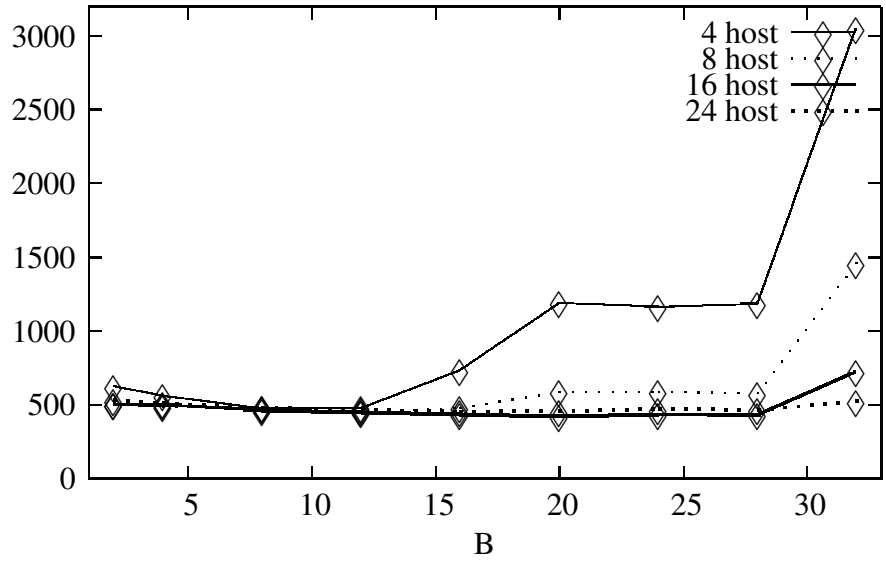
Az input file-okat egy segédprogrammal generáltattam úgy, hogy minden input file véletlenszerűen, maximum K-asával (négyesével) növekszik, így minden file növekvően rendezett halmaz.

A mérések részletes eredménye a A.4 függelékben a 58 oldalon található.

6.6.1. cluster-es változat

A 6.2 ábrán az összes cluster-es mérés átlag eredménye látható. Ebből kitűnik, hogy kb. 8-16000-es B érték mellett a 4 host-os mérésektől eltekintve az eredmény nagyából független a hostok számától, csak a felbontástól (B) függ, attól is alig. A felbontást tovább durvítva, a B értékét tovább növelve a feldolgozáshoz szükséges időt már erősen befolyásolja a felbontás mértéke, és a host-ok száma.

Kis B értékek mellett egy-egy részhalmaz áttöltése, feldolgozása, majd vissza mentése olyan kevés időt vett igénybe, hogy csak néhány dolgozó taszk volt, a többinek nem jutott munka. Mire a *master* kiosztotta az i-ediknek is a feladatát, és közben



6.2. ábra. DIA-Cluster teszt

tovább bontotta az inputot, addigra az első feldolgozó egység már végzett is a feladatával. Ezért az i.-edik feldolgozótól kezdve az összes többinek már nem jutott munka, mivel a *master* mindig a legelső "munkanélkülinek" ad munkát, így kárba veszett a lefoglalt erőforrás.

Mivel az adatok nem lokálisan helyezkednek el, hanem NFS file szerveren, ezért azokat is át kell húzni a hálózaton.

A 6.3 ábrán egy 22 host-os mérés látható 2K-128K -s B értékekkel.

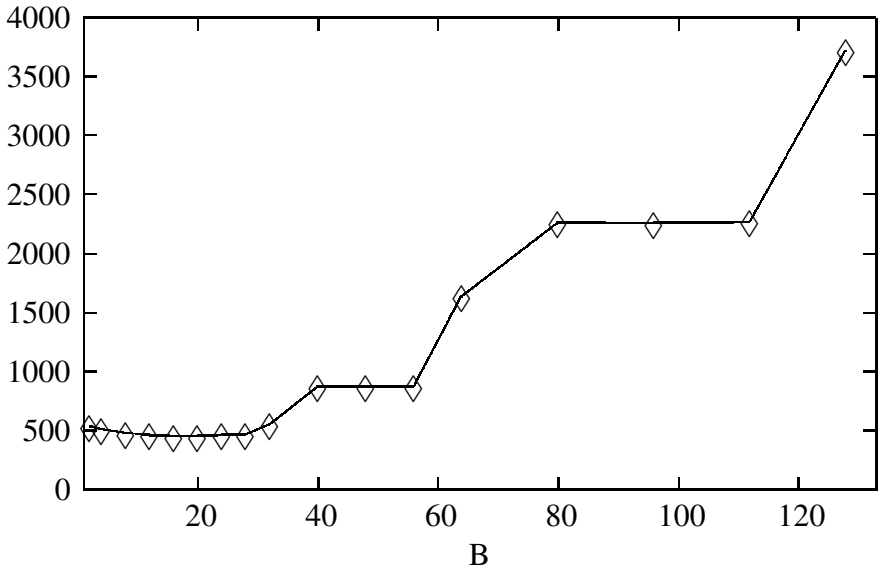
Összevetve a kapott eredményeket a [27]-ben találhatókkal, mindkét esetben látható, hogy ha az inputot a teljesen diszjunkt felbontás során egy bizonyos méretnél nagyobb részhlamazokra bontjuk fel, akkor a megoldás nem gyorsulni, hanem lassulni fog. Minél több processzorral dolgozunk, annál később jelentkezik ez a lassulás.

6.7. Optimalizálás

6.7.1. Hardware-, software finomhangolása

A hálózati infrastruktúra optimalizálásával (más ethernet, és/vagy nfs ablak méret a kommunikćió során), fejlesztésével (pl. Giagbit ethernet) gyorsabb lefutás várható, mivel a feldolgozó egységeknek csökken a várakozással töltött ideje. Gigabit ethernet esetén nagy adatmennyiség átvitele során 9000 MTU[3] értékkel kb. 50 %-os gyorsulás érhető el [24].

[3]MTU - ethernet csomagmérettel



6.3. ábra. DIA-Cluster teszt, 22 host

6.7.2. Az input speciális tulajdonságainak kihasználása

Mivel a feldolgozó egységek az idejük nagy részében inputra várnak, ezért ha rendelkezünk kellő erőforrással, akkor nagy adatmennyiség átküldése előtt az adatokat be lehetne tömöríteni. Ez több számítási erőforrást igényel, de ha az input jól tömöríthető, akkor jelentős mértékben lefaraghatunk a kommunikációs költségekből.

Például a mérésekhez készített inputot ábrázolhatnánk úgy, hogy kezdőérték, majd a növekmények sorozata. Így $n \in \mathbb{N}$ elem átküldése nem $n * sizeof(H)$ byte, hanem kihasználva, hogy az input szigorúan monoton növekvő, és a növekmény maximum K, tehát ábrázolható $\lceil log_2(K-1) \rceil$ biten: $sizeof(H) + \lceil n * \frac{\lceil log_2(K-1) \rceil}{8} \rceil$ byte. Ez addig jó megoldás amíg $\lceil log_2(K-1) \rceil < sizeof(H)$.

Az input H típusú elemek rendezett szekvenciája, és ez a szekvencia egy halmazt ábrázol, azaz egy elem csak egyszer fordul benne elő (emiatt szigorúan monoton növekvő sorozat az input). Az inputban a növekmény csak egyszer lehet akkora, hogy annak az ábrázolása $sizeof(H)$ byte-ot foglaljon, mert két ilyen elemnek az összege H-n belül már túlcsordulást okoz. Tehát feltehető, hogy a növekmények maximuma ábrázolás során kevesebb, mint $sizeof(H)$ byte-ot foglal.

Mivel az inputot teljesen diszjunkt részhalmazokra bontjuk ezért valószínűleg kevés olyan részhalmaz lesz, ahol a növekmény értéke nagy. Minden részhalmazban a növekmény felülről becsülhető az alábbi képlettel:

$$B \quad = \quad x_{n-1} - x_0 - n + 2$$

- ahol $n \in \mathbb{N}$ a részhalmaz hossza, x_0 az első eleme a részhalmaznak (ezért x_{n-1} az utolsó eleme). A fenti képletet úgy kapjuk, hogy ha n elemünk van, akkor $n-1$

növekményünk. Legrosszabb esetben van egy extremálisan nagy növekményünk, és az összes többi 1, azaz $n-2$ növekmény 1. Tehát az átvivendő byte-ok száma:

$$sizeof(H) + \lceil \frac{n * \lceil log_2 B \rceil}{8} \rceil + Z \tag{6.1}$$

- ahol Z olyan konstans járulékos költségek együttese, mint n és B értéke, hogy az átküldött adatokat vissza is tudjuk alakítani.

Ezzel a becsléssel konstans időn belül (n-től függetlenül) meghatározható egy felső korlát, persze ez lehet, hogy nem a legkisebb. A legkisebb felső korlát megkereséséhez maximum keresést kell végeznünk a növekmények között. Így hatékonyabb tömörítést érhetünk el, mint a konstans időn belüli becsléssel, de a maximum keresés $O(n)$ lépést igényel.

Ezzel a megoldással változó értékű B-kkel kell bitvektorokat kezelni, és azt nem is olyan egyszerű implementálni. Valamivel rosszabb tömörítéshez jutunk, ha mindig byte-határra rakjuk a növekményeket. Így az átvivendő byte-ok száma:

$$sizeof(H) + n * \lceil \frac{\lceil log_2 B \rceil}{8} \rceil + Z \tag{6.2}$$

Ezt sokkal egyszerűbb megvalósítani, de elveszítünk minden egyes növekménynél

$$V \;=\; 8 - (log_2 B \mod 8) \tag{6.3}$$

bitet, azaz összesen $\frac{n*V}{8}$ byte-ot.

A teszthez olyan file-okat készítettem, ahol a növekmény maximum négy, H 128 bit-en (16 byte-on) ábrázol természetes számokat, azaz egy n elemből álló sorozat átküldése, a járulékos konstans költségektől eltekintve:

$$S_e \;=\; n * 16 \tag{6.4}$$

$$S_t \;=\; 16 + \lceil \frac{n * \lceil log_2 4 \rceil}{8} \rceil = 16 + \frac{n}{4} \tag{6.5}$$

A fentiekben a 6.4 az eredeti megoldás során átvitt byte-ok száme, míg a 6.5 az új megoldásé. Az átvitt byte-ok aránya (új / régi megoldás):

$$\frac{S_t}{S_e} = \frac{16 + \frac{n}{4}}{16 * n} = \frac{1}{n} + \frac{1}{64}$$

- azaz körülbelül $\frac{1}{64}$-ére csökken a kommunikációs költség.

Ez azt jelenti, hogy nem kell $1GB = 2^{30}B$-ot átküldeni a hálózaton, hanem elég csak $2^{30} * 2^{-6} = 2^{24} = 16MB$-ot átküldeni.

Persze ezzel az ábrázolás móddal költséges lehet az $i \in \mathbb{N}$-edik elem kiolvasása, ha i kellően nagy, mert minden esetben összegezni kell az első i elemet, azaz

$$x_i = x_0 + \sum_{j=1}^{i} n_j \tag{6.6}$$

- ahol x_0 a kezdő érték, és $\forall j \in [1..i] : n_j$ a növekmény. Ezt a műveletet felgyorsíthatjuk, ha bizonyos i-nként eltároljuk a részletösszeget, persze ez is plusz erőforrást igényel.

Ha az inputot az eredeti formájában ábrázoljuk, és csak a kommunikáció során alakítjuk át nincs ilyen gond, viszont megnő az információ előkészítési költsége, és feldolgozási költsége, ugyanis az adatokat küldés előtt át kell alakítani, feldolgozás előtt pedig vissza kell alakítani.

Miután minden részhalmazt elemenként dolgozunk fel, nem kell egyszerre visszaalakítani a teljes inputot, hanem elég csak az éppen feldolgozandó elemet, amelyet megkaphatunk:

$$x_i = x_{i-1} + n_i \tag{6.7}$$

- a már kiszámolt elem és az i-edik növekmény összegeként.

Ezen megoldás során a feldolgozók nem olvashatják ki egy file szerverről az infomrációt, hanem szükség van egy adatkiszolgáló egység, amely közvetlenül a merevlemezről olvassa ki az adatokat, és elvégzi a szükséges átalakításokat. Ha az adatkiszolgáló egység nem közvetlenül ott helyezkedik el, ahol az adat van, akkor kétszer kell átküldeni a hálózaton a teljes adatmennyiséget.

Az utóbbi is lehet jó megoldás, ha nagy a sávszélesség a file szerver és az adatkiszoláló között, de a feldolgozó egységek felé már szükös a keresztmetszet. Ilyen esettel állunk szemben egy grid-ben futtatott alkalmazás esetén, ahol az adatkiszolgálók a file szerver közelében vannak, míg a feldolgozó egységek máshol.

6.7.3. Taszkok elhelyezése

A grid-es változatban a taszkok elhelyezésével lehet javítani a rendszer teljesítményét. Pédául, ha a program készítése során úgy osztjuk szét a taszkokat, hogy az adat- kiszolgálók és az adatrögzítő a file szerverhez minél közelebb helyezkedjen el. Ideális esetben a file szerveren vannak, és ténylegesen közvetlenül a merevlemezről olvasnak, illetve oda írnak.

7. fejezet

Összegzés

Adatintenzív alkalmazásoknak már az elnevezéséből is adódik, hogy rengeteg adattal dolgoznak. Az adatokat lokálisan cache-elni kell, hogy a program folyamatosan haladhasson, és ne kelljen állandóan várakoznia. Ezért az ilyen alkalmazásokat általában lokális erőforrásokon, cluster-eken futtatnak.

Az ilyen alkalmazaások nagy részét képzik az elemenként feldolgozható függvényekkel leírható feladatok, mint például összefésüléses rendezés, halmazok uniójának számítása, stb. Az elemeként feldolgozható függvények párhuzamos feldolgozásához az inputot teljesen diszjunkt részhalmazokra bontjuk, majd ezeket a részhalmazokat párhuzamosan feldolgozzuk. A feldolgozás során fontos kérdés, hogy mekkora részekre bontsuk az inputot, sok kis rész rengeteg kommunikációs költséggel jár, míg kevés nagy rész a processzorok kiegyensúlyozatlan terheléséhez vezet. Fontos kérdés a részhalmazok ideális méretének meghatározása.

Dolgozatomban a nagy adatmennyiség ideális részhalmaz méretét határoztam meg abban az esetben, amikor nagy mennyiségű inputtal négy halmaz unióját kellett kiszámítani. Kísérleti futtatásaimban azt tapasztaltam, ha nagyon kicsi volt a részhalmaz mérte, akkor egyes taszkok nem jutnak munkához. Mivel a taszkokat a PVM minden egyes host-on egyenletesen helyezi el, ez nem feltétlen jelenti azt, hogy az erőforrások egy része teljesen kihasználatlan, csak azt, hogy túl sok a feldolgozó, így azok feleslegesen foglalnak le erőforrásokat.

A feldolgozandó részhalmazok mértét növelve van egy ideálishoz közeli méret, ahol szinte függetlenül a lefoglalt erőforrások számától a feldolgozás nem gyorsul tovább. Ez valószínüleg a szűkös hálózati sávszélesség miatt alakul így.

A továbbiakban érdemes lenne megvizsgálni, hogy milyen gyorsulás érhető el, ha az input-ot és az output-ot tömörítjük mielőtt átküldjük a hálózaton.

A. Függelék

Mérések

A.1. PVM átviteli sebesség

Ebben a tesztben a PVM átviteli sebességet mértem meg. A mérés környezete: 2 db Pentium4-Celeron 1.7GHz, 256 MB DDR-SDRAM -os PC, 100MBit -es fullduplex hálózati kártyával összekötve. A teszt során 32MB-nyi adatot küldtem át egyik gépről a másikra PVM-en keresztül, különböző blokk méretekkel. Minden mérést tízszer ismételtem meg, a részeredmények itt találhatók.

Adat [MB]	Blokk méret [KB]	Idő [s]									
32	0.5	39	40	39	40	39	40	40	39	40	40
32	1	23	23	24	23	23	24	23	23	23	22
32	2	15	15	15	14	14	14	14	14	14	14
32	4	11	11	11	10	11	11	11	11	11	11
32	8	9	9	8	9	8	8	8	9	9	9
32	16	8	7	7	8	8	8	7	7	7	8
32	32	7	7	7	7	7	7	7	7	7	7
32	64	6	7	6	7	7	7	7	7	7	6
32	128	7	6	6	6	6	7	7	6	6	6
32	256	6	7	6	7	6	6	6	6	7	6
32	512	6	7	6	7	6	6	6	7	7	6
32	1024	7	6	6	6	6	6	6	6	7	6
32	2048	6	6	7	6	6	6	6	6	6	6

A.1. táblázat. PVM átviteli sebesség, 100MBit-es fullduplex hálózaton

A.2. File

A.2.1. Írás

1 GB-nyi, azaz 2^{26} db *Nat128* típusú adat kiírása file-ba. Mérési eredények másodpercben.

Futtatás sorszám	1	2	3	4	5	6	7	8	9	10	11	12
sys i/o	64	71	60	64	60	61	60	61	60	60	60	60
stdio.h	61	62	72	63	70	70	62	70	61	62	71	60

A.2. táblázat. *seq<Nat128>*

Futtatás sorszám	1	2	3	4	5	6	7	8	9	10	11	12
sys i/o	258	228	230	228	229	227	226	226	227	227	226	229
stdio.h	324	325	326	328	324	327	387	359	330	416	350	327

A.3. táblázat. *Nat128*

A.3. Cluester-es verzió

A program cluster-es verziójának bemérését 8, 16, 24 host-on csináltam meg. A részeredmények az alábbi táblázatokban olvashatók. A teszt folyamán az input file kezdetben 4x1GB volt, később ezt 4x256MB-ra csökkentettem. A 4x1GB-s mérés időket arányosítottam a 4x256MB-s mérésekhez.

A táblázat első oszlopa a host-ok száma, a második a B értéke és a harmadik oszlopban a mérések eredmény olvasható másodercben.

(*) A 24 host-os mérések során egy host kiesett, és nem sikerült másikkal pótolni, így ott az esetek majd 90 %-ában csak 23 host dolgozott.

Host	B	Mérések [s]
4	2K	600 650 626
	4K	560 553 561
	8K	493 466 468 464
	12K	474 473 481 484
	16K	745 715 744 724
	20K	1202 1194 1172 1197
	24K	1171 1138 1152 1206
	28K	1195 1157 1183 1210
	32K	3083 3017 3041 3046
8	2K	490 520 531 478
	4K	507 492 487 468
	8K	478 460 488 497
	12K	444 461 438 459
	16K	474 495 471 475 513
	20K	586 597 597 585 579
	24K	582 594 582 581 582
	28K	586 579 569 583 586
	32K	1472 1453 1460 1462 1459
16	2K	521 503 506 493
	4K	488 493 510 509
	8K	470 454 452 463
	12K	457 440 467 424 447
	16K	447 429 425 432 440
	20K	409 431 429 436 413
	24K	433 432 456 421 427
	28K	432 437 445 418 421
	32K	749 720 747 718 721 720 716
24*	2K	534 534 524
	4K	506 515 508
	8K	472 478 478
	12K	467 455 474
	16K	445 453 452
	20K	448 457 476
	24K	471 455 476
	28K	459 460 481
	32K	513 507 539 531 526

A.4. táblázat. A program cluster-es verziójának mérési eredményei

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] Wettl Ferenc - Mayer Gyula - Sudár Csaba: *LATEX kezdőknek és haladóknak*, Panem, 1998, [409], ISBN 963-545-141-5

[2] www.lpds.sztaki.hu 2004. június 8.

[3] http://www.lpds.sztaki.hu/pgrade 2004. június 8.

[4] http://mazsola.iit.uni-miskolc.hu/~juhasz2/intro.html 2004. június 8.

[5] http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf 2004. június 8.

[6] http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf 2004. június 8.

[7] http://www.cacr.math.uwaterloo.ca/hac/ 2004. június 8.

[8] http://www.globus.org/ogsa/ 2004. június 8.

[9] http://www.cs.wisc.edu/condor/condorg/versusG.html 2004. június 8.

[10] http://www.cs.wisc.edu/condor/overview 2004. június 8.

[11] http://www.cs.wisc.edu/condor/doc/condorg-hpdc10.pdf 2004. június 8.

[12] http://www.netlib.org/pvm3/book/node17.html 2004. június 8.

[13] ftp://netlib2.cs.utk.edu/pvm3/book/pvm-book.ps 2004. június 8.

[14] Fóthi Ákos, Steingart Ferenc: *Programozási módszertan (egyetemi jegyzet)*

[15] Ákos Fóthi / Zoltán Horváth / Tamás Kozsik: *Parallel Elementwise Processing - A Novel Version,* In Varga L., ed., Proceedings of Fourth Symposium on Programming Languages and Software Tools, Visegrád, Hungary, June 9-10, 1995, pp. 180-194. Vol 17, pp. 105-174, 1998

[16] Vincent Nikkelen: *MASTER'S THESIS - Implementation of abstract UNITY algorithms in PVM.*, Master's Thesis, Technical University Eindhoven and Eötvös Loránd University Budapest, October 2000.

[17] http://www-unix.mcs.anl.gov/mpi/ 2004. június 8.

[18] http://www.lam-mpi.org/overview/ 2004. június 8.

[19] Foster Kesselman: *The Grid: Blueprint for a New Computing Infrastructure* , Morgen Kaufmann Publishers, Inc., 1999, [677], ISBN 1-55860-475-8

[20] http://www.csm.ornl.gov/pvm/PVMvsMPI.ps 2004. június 8.

[21] Andrew S. Tanenbaum - Maarten van Steen: *ELOSZTOTT RENDSZEREK - Alapelvek és paradigmák*, Panem, 2002, [872], ISBN 963-545-387-6

[22] http://www-unix.mcs.anl.gov/~ bester/historical/dsl/nettestes.html 2004.június 8.

[23] http://cdfcaf.fnal.gov/doc/cdfnote_5962/node16.html 2004. június 8.

[24] http://sd.wareonearth.com/~ phil/jumbo.html 2004. június 8.

[25] http://publib16.boulder.ibm.com/pseries/en_US/aixbman/prftungd/netperf3.htm 2004. június 8.

[26] http://www-fp.mcs.anl.gov/~ foster/Articles/WhatIsTheGrid.pdf 2004. június 8.

[27] Horváth Z. - Hernyák Z. - Kozsik T. - Tejfel M. - Ulbert A.: *A Data Intensive Application on a Cluster - Parallel Elementwise Processing*, In P. Kacsuk, D. Kranzlmller, Zs. Nemeth, J. Volkert (Eds.): Distributed and Parallel System - Cluster and Grid Computing, Proc. 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems, Kluwer Academic Publishers, The Kluwer International Series in Engineering and Computer Science, Vol. 706, pp. 46-53, Linz, Austria, September 29-October 2, 2002.