EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

Programozási Nyelvek és Fordítóprogramok Tanszék

A PVM és az MPI új elemeinek vizsgálata

Farkas Gergely

Témavezető:
Horváth Zoltán

Budapest, 2004. tavaszi félév
Készült az IKTA 64/2003 és IKTA 89/2002 keretében.

Tartalomjegyzék

Köszönetnyilvánítás

Ezúton szeretném köszönetemet kifejezni mindenkinek, aki a diplomadolgozatom elkészítésében bármilyen módon segített.
Külön szeretném megköszönni a segítséget:

- **Horváth Zoltán**nak a témavezetői teendők elvállalásáért és a diplomadolgozat elkészítéséhez nyújtott segítségért.

1. fejezet

Bevezetés

1.1. A dolgozat témája

A dolgozat célja a PVM és az MPI fejlődése során megjelent változatok újdonságainak vizsgálata, különös tekintettel ezen eszközök nagy kiterjedésű, heterogén környezetekben használható változataira.

1.2. Elosztott rendszerek

„Az elosztott rendszer önálló számítógépek olyan összessége, amely kezelői számára egyetlen koherens rendszernek tűnik.” – Tanenbaum, Steen [1]
Ez a definíció két feltételt fogalmaz meg. Az első szerint az elosztott rendszer több számítógépből áll, amelyek lehetnek ugyanolyanok vagy különbözőek. Az előbbi esetén homogén, míg az utóbbi esetén heterogén a rendszerről beszélünk.
A másik feltétel szerint a rendszernek el kell rejtenie az első tulajdonságot, azaz több szempontból is egységes képet kell mutatnia a felhasználók felé. Ez utóbbi tulajdonságot nevezzük átlátszóságnak.

A hardver szempontjából valójában minden elosztott rendszer pusztán több CPU együttese. A rendszerek lehetnek a CPU-k összekötésének módja szerint busz alapúak vagy kapcsolóhálózat alapúak, de csoportosíthatóak a CPU-hoz tartozó memória elhelyezkedése szerint is. Ha minden CPU ugyanazt a memóriát látja, akkor közös memóriásnak nevezzük, míg ha minden CPU-hoz külön memória tartozik, akkor osztott memóriásnak nevezzük a rendszert.

Elosztott rendszerek építésének egyik legegyszerűbb lehetősége, ha több különálló számítógépet hálózattal kapcsolunk össze. Az elosztott programok futtatásának céljából összekötött számítógépek egy úgynevezett klasztert (*cluster*) alkotnak. A klaszterek állhatnak olyan munkaállomásokból, amelyekre az a jellemző, hogy a tulajdonosaik időnként használják őket, de az idő nagy részében csak szabadon állnak. Ezeknél a szabad számítási kapacitás kihasználása a cél.
Egy másik lehetőség, hogy kizárólag elosztott programok futtatásának céljából kötnek össze számítógépeket. Ezeket a klasztereket nevezzük dedikált klasztereknek. A dedikált klaszterek manapság felveszik a versenyt a szuperszámítógépekkel, azok árának töredékéért.
A dolgozatban tárgyalt eszközök leginkább ilyen, különálló számítógépekből felépített elosztott rendszerekben használtak.

1.2.1. A metaszámítógépek

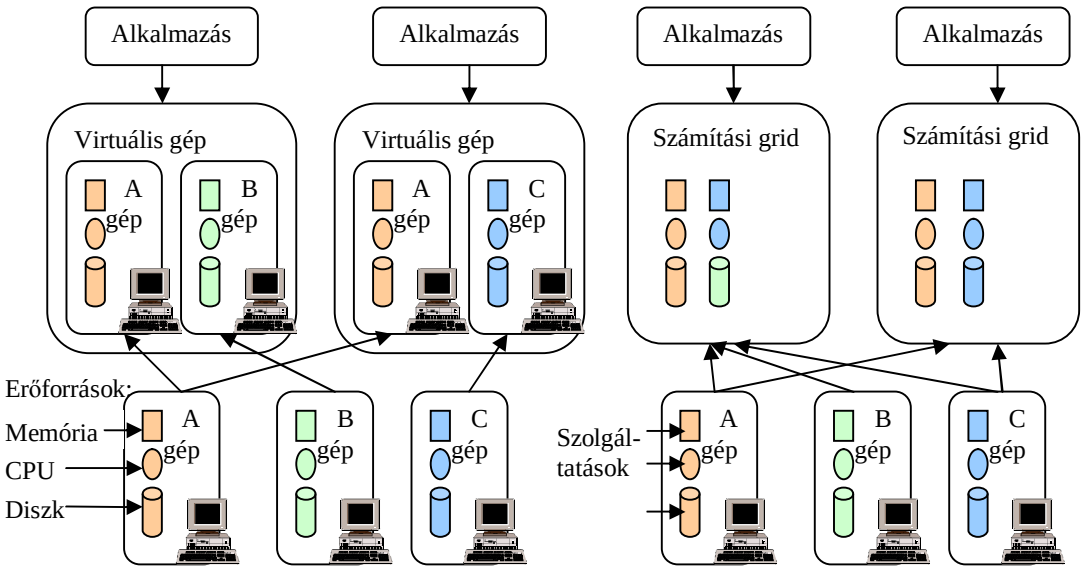
A hálózati technológia fejlődésével, a gyors hálózat révén a munkaállomások, a PC-k és a szuperszámítógépek egyre közelebb kerültek egymáshoz. A hálózattal

összekötött számítógépeken lehetőség nyílt az erőforrások egymás közötti megosztására is. Az ilyen módon megvalósított elosztott rendszereket, a különböző erőforrásokkal rendelkező, heterogén gépek alkotják (1. ábra). Ezekre a rendszerekre nagy fokú átlátszóság jellemző, azaz törekszenek az „egy számítógép kép” biztosítására. Hátrányuk, hogy a rendszer felhasználójának a rendszert alkotó minden egyes gépen képesnek kell lennie a bejelentkezésre, azért hogy a gép erőforrásait használni tudja.
A fenti tulajdonságokkal rendelkező rendszereket metaszámítógépeknek (*metacomputer*) nevezzük. Ilyen metaszámítógépre példa a PVM virtuális gépe és annak utóda, a Harness elosztott virtuális gép.

1.2.2. A grid

Az osztott számítógépes rendszerek teljesen új megközelítését hozta a grid technológia, melyen a hálózatba kapcsolt számítógépek kommunikációs protokolljait, az általuk kínált szolgáltatásokat és az erőforrásaik elosztását meghatározó infrastruktúrát értünk.
A grid a metaszámítógépek fogalmát általánosítja, mert itt az elosztott rendszert nem az erőforrásokat tartalmazó gépek alkotják, hanem az erőforrások maguk, a gépektől függetlenül (1. ábra). A grid koncepció szerint lehetőség van például néhány gép memóriáját és központi egységét, valamint egy másik gép hatalmas diszk kapacitását használva a CERN részecskegyorsítója (speciális erőforrás) által generált adathalmaz feldolgozására.
A felhasználó által elérhető erőforrások szolgáltatások formájában vannak jelen a gridben. Az erőforrások használatakor a grid esetében nincs szükség arra a megkötésre sem, hogy a felhasználónak minden gépen belépési lehetősége legyen. A felhasználónak csak a gridbe kell belépnie, a többit az infrastruktúra elrejti a grid felhasználója előtt.

A számítási gridek (*computational grid*) olyan osztott rendszerek, amelyekben nagyméretű elosztott számításokat végezhetünk. Ilyen számítási gridet megvalósító eszköz többek között a Globus Toolkit, melyről részletesebben is szó lesz a dolgozat hátralevő részében.



1. ábra A metaszámítógépek és a grid infrastruktúra

1.3. Elosztott programok

Azokat a feladatokat, amelyeket elosztott programokkal oldunk meg, az alábbiak szerint három csoportba sorolhatjuk.
Az első csoportba tartoznak azok a feladatok, amelyeknél a futási idő csökkentése a cél. Ezen a területen a párhuzamosság szinte minden szinten jelentkezik, a hardver felépítésétől a párhuzamos programok írását segítő programozási nyelvekig.
A második csoportba azok a feladatok tartoznak, amelyeket a természetükből adódóan csak elosztott programok segítségével valósíthatunk meg. Ide tartoznak például a földrajzilag elosztott erőforrásokat használó alkalmazások.
A harmadik csoportba azok a feladatok tartoznak, ahol a hibatűrő képesség megvalósítása a cél.
A három csoport feladatainak megvalósításához különböző lehetőségeket biztosító eszközök szükségesek.

1.3.1. Elosztott programok írását segítő eszközök

A következőekben felsorolt eszközöket a fenti felosztás szerint alapvetően az első csoport problémáira fejlesztették ki. A másik két csoport feladatainak megoldására ezek egyáltalán nem vagy csak korlátozott mértékben képesek.

Az elosztott programoknak, mint az elosztott rendszer gépein, párhuzamosan futó folyamatoknak a szinkronizációjára kétféle paradigma létezik. Az első a közös memóriás modell, melynél a program folyamatai egy közös memória segítségével kommunikálnak egymással. A megosztott memória lehet a hardver része (közös memóriás gépek), de lehet az elosztott rendszer gépein futó, elosztott operációs rendszer szoftveresen implementált szolgáltatása is. Közös memóriát használó eszköz például az OpenMP szabvány, amelyet OpenMP Architecture Review Board [2] bocsátott ki.
A második módszer az üzenetküldéses paradigma, amelyet olyan rendszereken használnak, ahol közös memória nem áll rendelkezésre. Itt ugyanis a folyamatok kizárólag üzenetek küldésével kommunikálhatnak.

A továbbiakban röviden ismertetem az üzenetküldéses paradigmát használó PVM és MPI rendszereket.

1.3.2. A PVM

A PVM (Parallel Virtual Machine) [3] egy olyan üzenetküldéses paradigmát alkalmazó függvénykönyvtár (application library, API), amelynek segítségével hálózatba kapcsolt számítógépeket egyetlen nagy virtuális számítógépként lehet látni, és kezelni.
A rendszer fejlesztése 1989-ben kezdődött meg az Oak Ridge National Laboratory-nál Vaidy Sunderam és Al Geist [4] közreműködésével. Azóta több verzió is napvilágot látott, és a PVM a párhuzamos programok írásának egyik szabványává vált. A legutóbbi változat a PVM 3.4 verziószámot viseli.

A rendszer eredeti változata publikus, ingyen elérhető [3], de sok hardverre létezik a gyártó által kibocsátott, optimalizált változat is.

A PVM egyik előnyös tulajdonsága a heterogenitása, amely annak a ténynek köszönhető, hogy a PVM számos hardverarchitektúrára elérhető, s ezek a változatok

egymással kommunikálni képesek. A rendszer másik hasznos tulajdonsága, hogy támogatja a virtuális gép dinamikus kezelését. Az elosztott programok futás közben kérhetik a rendszert a virtuális gép további gépekkel való bővítésére, de akár el is távolíthatnak belőle gépeket.

Nagy alkalmazásoknál fontos kérdés az elosztott program hibatűrő tulajdonsága. A PVM alapszinten támogatja a hibatűrő programokat, ugyanis képes figyelmeztetni az elosztott program folyamatait, ha a virtuális gép állapota változik, vagy ha egy folyamat elveszik.

1.3.3. Az MPI

Az MPI (Message Passing Interface) egy rutinkönyvtár szabvány, amely üzeneteken alapuló kommunikáció szintaxisát és szemantikáját definiálja.
Az 1992-ben megalakult MPI Forum, amelynek számos nagy hardvergyártó cég a tagja lett, 1994-ben kiadta a szabvány első változatát, az MPI-1-et, majd hamarosan annak javított változatát az MPI-1.1-et [5]. Az MPI kifejlesztésének célja az volt, hogy egy széles körben használható, hordozható szabványt készítsenek üzenetek küldésére.
A szabványban specifikált rutinkönyvtár implementációi két csoportba oszthatók. Az első csoportba a hardvergyártók által készített, az általuk gyártott hardverre erősen optimalizált változatok tartoznak. Ezzel ellentétes nézőpontot tükröznek a második csoport implementációi, ugyanis ezek fejlesztésének elsődleges célja a lehető legtöbb architektúra támogatása egy függvénykönyvtárban. Sajnos a különböző implementációival készített MPI programok egymással kommunikálni nem képesek, így a programok csak forrásszinten hordozhatóak.

Az MPI erősségei közé tartoznak a kommunikációs primitívek nagy száma, az erősen típusos üzenetküldés és a kollektív kommunikációs lehetőségek.
A szabvány használata során hamar jelentkeztek az MPI hiányosságait. Az egyik legfontosabb, hogy a szabvány nem definiál eljárásokat folyamatok indítására, illetve dinamikus kezelésére. Az 1997-ben kiadott MPI-2 [6] ezt és számos további hiányosságot pótol, azonban ennek a szabványnak máig nagyon kevés teljes implementációja létezik, s ezek is csak adott hardverre optimalizált változatok.
Hasonlóan a folyamatindításhoz a hibatűrés támogatása is csak a második változatban jelent meg, a PVM figyelmeztető lehetőségeivel analóg rutinok definiálásával.

Az MPI-1 szabvány előírja a C és Fortran nyelvek támogatását, ehhez az MPI-2 szabvány a C++ nyelv támogatását is hozzávette.

1.4. A dolgozat célkitűzései

A dolgozat célja annak vizsgálata, hogy miként lehet PVM és MPI programokat használni metaszámítógépes, illetve grid rendszerekben.
Ezekben a rendszerekben milyen lehetőségek vannak ezen eszközök használatára. Milyen új lehetőségeket támogatnak ezek az új eszközök, és melyek a használat során felmerülő problémák.

1.5. A dolgozat felépítése

A dolgozat 2. fejezetében ismertetem a dolgozatban használt elosztott rendszereket, név szerint a Harness, a Condor és a Globus rendszereket. A 3. fejezet tartalmazza az MPI használatának lehetőségeit ezekben a rendszerekben, majd a 4. fejezet a PVM használatát tárgyalja. Az 5. fejezetben öt szempont szerint megvizsgálom az ismertetett eszközöket, végül a 6. fejezet egy rövid összefoglalást tartalmaz a munkáról.

2. fejezet

A használt elosztott rendszerek ismertetése

2.1. A Harness rendszer

A Harness (Hererogeneous Adaptable Reconfigurable Networked SystemS) [7] az Emory University kísérleti metaszámítógép rendszere, amely a PVM virtuális gép koncepcióján alapul. Fejlesztésében részt vett többek között a PVM-et megalkotó Al Geist és Vaidy Sunderam is. A Harness 1.8 volt az első publikus változata, amely 2001-ben jelent meg. 2002-ben ezt követte a Harness 1.9 [8], amely már PVM programok futtatására is képes. Máig ez az utolsó megjelent változat.

A Harness virtuális gépét elosztott virtuális gépnek (Distributed Virtual Machine) nevezzük. Az elosztott virtuális gép örökölte elődjének dinamikusságát, azaz a PVM-hez hasonlóan lehetőség van a gép menet közbeni bővítésére és szűkítésére. Ennél azonban a Harness többre is képes. A virtuális gép dinamikus volta megjelenik a gép tulajdonságainál is, ahol a *plug-in* mechanizmus segítségével menet közben adhatunk hozzá, illetve távolíthatunk el olyan szoftverkomponenseket, mint például a PVM programokat futtató szolgáltatás.

A Harness architektúrájának tervezésekor nagy figyelmet fordítottak a rendszer hibatűrő képességekkel való felruházására, ezért a rendszer nem tartalmaz olyan pontot, amelynek meghibásodása a rendszer összeomlásához vezetne (*single point of failure*).

2.1.1. A Harness névszolgáltatása

Mint a legtöbb ma használatos elosztott rendszerhez, a Harnesshez is tartozik egy névszolgáltatás. A névszolgáltatás számontartja az általa biztosított névtérben levő elosztott virtuális gépek állapotát nyilvántartó státuszszervereket.

A Harnesst fejlesztő Emory University üzemeltet egy névszolgáltatást (elérhető a harness.mathcs.emory.edu:2000 címen), de ha nincs lehetőségünk ennek elérésére, vagy saját névteret szeretnénk, akkor indíthatunk mi is egyet, mert a szolgáltatás megtalálható a Harness letölthető verziójában.

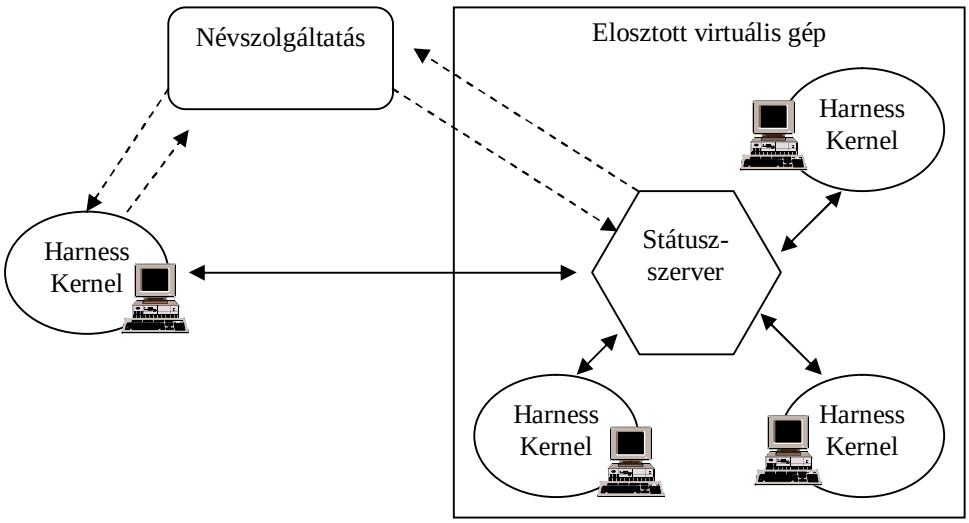
A központosított névszolgáltatás a Harness egyetlen olyan pontja, amelynek összeomlása esetén a bejegyzett elosztott virtuális gépek is elvesznek. Ennek elkerülése végett a Harness fejlesztői egy elosztott névszolgáltatást (Harness Distributed NameService) is implementáltak. Az elosztott névszolgáltatást legfeljebb 8 névszerver alkothatja, amelyek egymás között gyűrű topológiába rendezve terjesztik az információt. Az információ terjesztésének módja megtalálható Migliardi, Sunderam, Tyrakowski és Frisiani írásaiban [10,11].

Az Emory University elosztott névszolgáltatást is üzemeltet (harness.mathcs.emory.edu: 3000, harness1.mathcs.emory.edu:3000, harness2.mathcs.emory.edu:3000), de itt is lehetőségünk van saját szolgáltatás indítására.

2.1.2. A Harness elosztott virtuális gépe

A Harness architektúrában minden elosztott virtuális géphez egy státuszszerver és több kernel tartozik (2. ábra). A virtuális gépet alkotó számítógépeken a PVM démonjához hasonlóan egy kernel komponens található, amely a plug-in szolgáltatások betöltéséért és eltávolításáért felelős.
Ha bővíteni szeretnénk egy virtuális gépet vagy létre szeretnénk hozni egyet, akkor egy új kernelt kell indítanunk, amelynek paraméterként át kell adnunk az elosztott virtuális gép nevét. Az új kernel felveszi a kapcsolatot a Harness névszolgáltatásával és elkéri tőle az elosztott virtuális géphez tartozó státuszszerver elérhetőségét. A névszolgáltatás visszaadja a bejegyzett virtuális gép státuszszerverének címet, illetve ha a keresett néven nincs bejegyzett virtuális gép, akkor azt jelzi a kérdezőnek. Ennek a viselkedésnek köszönhetően a Harness névtérben nem lehet két azonos nevű virtuális gép.
Ha a kernel azt kapja válaszként, hogy nem létezik még a virtuális gép, akkor létrehoz egy státuszszervert az új elosztott virtuális gép. Létrejötte után a státuszszerver bejegyzi magát a névszolgáltatásnál. Ha a kernel már ismeri a státuszszerver elérési címét, akkor felveszi vele a kapcsolatot, és részévé válik a virtuális gépnek. A csatlakozást megvalósító protokoll részletesebb leírása megtalálható Migliardi és Sunderam írásában [12].

Az elosztott virtuális gépek státuszszervere egy központosított szolgáltatás a Harness architektúrában. Az ilyen központosított szolgáltatások két problémát vetnek fel. Az egyik, hogy szűk keresztmetszetet jelentenek *(bottleneck*), ami a rendszer teljesítményének romlásához vezet, a másik probléma pedig, hogy összeomlásuk esetén a virtuális gép elvesztését okozhatják. Az első probléma a Harness esetében nem áll fent, ugyanis a státuszszerverrel való kommunikáció elenyésző a kernelek közötti kommunikációhoz viszonyítva.



2. ábra A Harness architektúrája

A hibatűrés szempontjából nagyon fontos tulajdonság, hogy az elosztott virtuális gép státuszszerverének összeomlása után a rendszer képes egy új státuszszervert indítani, s ez a státuszszerver konzisztens módon tartalmazni fogja a virtuális gép aktuális állapotát.

2.1.3. A Harness plug-in szolgáltatásai

A Harness rendszer képes arra, hogy menet közben különböző heterogén számítási erőforrásokat vegyen hozzá a virtuális géphez (dynamic reconfiguration). Ezek az erőforrások a rendszerben plug-in-ek formájában jelennek meg. Egy új erőforrás hozzávétele vagy eltávolítása nem más, mint egy adott plug-in betöltése illetve eltávolítása a virtuális gépben.
A rendszer implementációjakor a fejlesztők a Java Technológia mellett döntöttek. Ennek legfőbb oka, hogy egy heterogén architektúrákból álló környezetben a Java virtuális gépek homogén alapot biztosítanak a Harness rendszernek, valamint az, hogy Java objektumok formájában egyszerűen megoldható a plug-in-ek betöltése és eltávolítása. További előnyöket Migliardi és Sunderam [9] írása részletezi.
Habár a Java homogén alapot biztosít a Harnessnek, mégis létezhetnek plug-in-ek, amelyek egy-egy architektúrán nem elérhetőek. A plug-in-eket a hordozhatóság alapján három csoportba oszthatjuk. Az első csoportba a speciális szolgáltatások (*specialized services*) tartoznak, amelyek nagymértékben függenek egy adott architektúrától. Ilyen lehet például egy adott hálózati felépítésre optimalizált üzenetátadó rendszer.
A második csoportba az olyan alapszolgáltatások tartoznak (*basic services*), mint például egy általános, hálózatfüggetlen üzenetátadó szolgáltatás. Ezek a szolgáltatások általánosan elérhetőek a legtöbb architektúrán.
A harmadik csoport a kernel szintű szolgáltatások (*kernel level services*). Ezek közé tartoznak azok a szolgáltatások, amelyek a virtuális gép állapotának változtatásáért felelősek, azaz azok, amelyek képesek betölteni és eltávolítani plug-in-eket. Ennek a csoportnak a tagjai bírnak a legnagyobb fokú hordozhatósággal.

2.1.4. A Harness és a biztonság

A biztonság az elosztott rendszerek egyik legbonyolultabb alapeleme, hiszen az egész rendszerre kiterjed. Alapvetően két területre bontható. Az egyik a rendszerben történő kommunikáció üzeneteinek biztonságos kezelése, a másik az elosztott rendszer felhasználóinak azonosítása, valamint a felhasználók jogainak biztosítása.
A Harnessben a felhasználók jogainak beállítását minden – a virtuális gépbe tartozó – gépen külön-külön, egy konfigurációs fájl segítségével állíthatjuk be.
Minden ilyen gépen van egy *root* felhasználó, amelynek mindenhez joga van. A *root* felhasználónak a nevét és jelszavát is a konfigurációs fájl tartalmazza. A rendszerben megadhatóak további felhasználók is, amelyeknél részletesen felsorolhatók a betölthető plug-in-ek, valamint felsorolhatók azok a már betöltött plug-in-ek is, amelyekhez a felhasználónak hozzáférési joga van.
A konfigurációs fájlban szerepelhet egy *NOBODY* nevű felhasználó is. Ha a rendszerbe egy a konfigurációs fájlban nem szereplő felhasználó akar belépni, akkor ahhoz a *NOBODY* felhasználó jogai rendelődnek. Ha nincs ilyen felhasználó, akkor a belépést a Harness megtagadja.

A Harness nyújtotta módszerrel részletesen leírhatók a virtuális gép felhasználóinak jogosultságai.

A biztonság előbb említett első területén, a biztonságos kommunikáció terén a Harness jelenlegi változata nem biztosít semmilyen használható mechanizmust. Az egyetlen, amit ilyen irányú igény felmerülése esetén tehetünk, hogy kihasználjuk a Harness rugalmasságát, és tervezünk, majd implementálunk egy új biztonságos csatornák feletti kommunikációt biztosító plug-in-t. Ennek virtuális gépbe töltésével már képes a rendszer a biztonságos kommunikációra.

2.2. A Condor rendszer

A Condor [29] egy lokális feladatkezelő (*jobmanager*) rendszer, amely elsősorban hosszú futási idejű, számításigényes feladatok végrehajtását segíti. Létrehozásának alapvető célja az volt, hogy a helyi hálózatok munkaállomásainak szabad CPU ciklusait kihasználják, de mára a rendszer több olyan lehetőséggel is kibővült, amelyeknek köszönhetően nagy számítási kapacitású, dedikált rendszerek feladatkezelőjeként alkalmazzák.

A Condor rendszer szolgál alapjául a magyar Klaszter Grid [13] kezdeményezésnek is, amely éjszakánként a magyar felsőoktatási intézmények számos számítógép laboratóriumát – többek között az ELTE Lovardáját is – Grid üzemmódban használja.

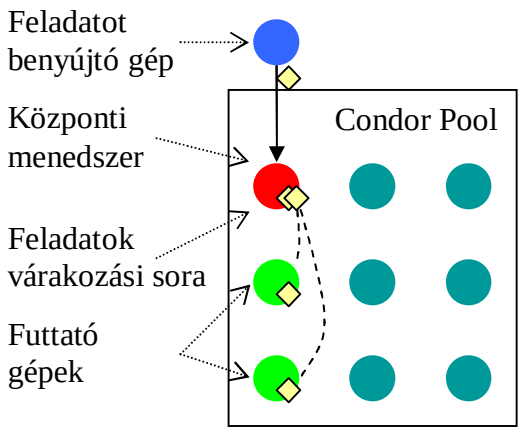
A Condor támogatja mind a PVM, mind az MPI programok futtatását, ezért a dolgozatban részletesebb ismertetést érdemel.

2.2.1. A Condor rendszer szereplői

A Condor rendszerben a szabad számítási kapacitásokkal rendelkező számítógépek egy úgynevezett „*Condor Pool*”-ba (3. ábra) szervezhetők. A Condor ennek a poolnak a gépein hajtja végre a lefuttatni kívánt programokat (jobokat).
Minden pool tartalmaz egy központi menedzsert (*Central Manager*), amely a pool gépein futó helyi démon program segítségével folyamatosan figyelemmel kíséri a gépek terheltségét.

A programot futtatni kívánó felhasználók a program futtatásának körülményeit részletesen meghatározva benyújtják a kérelmüket a Condornak.
Végrehajtandó feladat benyújtásakor a kliens gép felveszi a kapcsolatot a pool központi menedzserével, majd átadja neki a feladatleíró fájlt (*submit file*), amely a pool várakozási sorába kerül.

A Condor a várakozási sorban levő feladatok végrehajtásához történő erőforrás-allokációról egy, a rendszergazda által megadott politika szerint dönt. A feladat végrehajtásához választ egy szabadnak ítélt gépet a pool-ból, és elindítja rajta a programot. Az allokálási procedúra során nem csak az erőforrások állapotára, hanem a feladat igényeire is figyel.



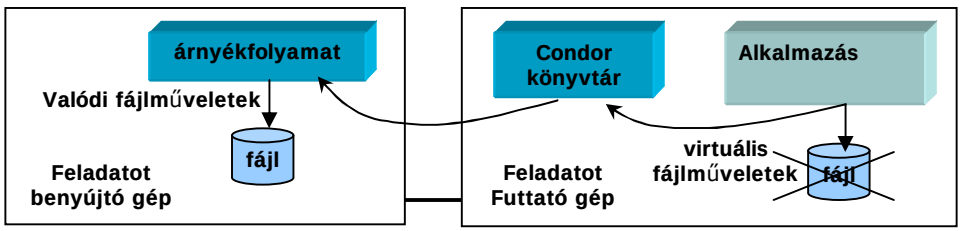
3. ábra A Condor rendszer szereplői

A leggyakoribb allokálási politika, amit Condor esetében alkalmaznak a következő. Minden olyan gép, melyet adott ideje nem használt annak tulajdonosa, a központi menedzser fennhatósága alá kerül, amely végrehajtandó feladatokat indíthat el rajta. Ha azonban megérkezik a tulajdonos, akkor őt illeti az elsőbbség, s a gépen a futó feladatokat meg kell szakítani. Bizonyos esetekben lehetőség van a feladat felfüggesztésére és egy másik gépen történő végrehajtására is.

A Condor jellegzetessége az univerzum fogalom, amely a rendszerben különböző futtatási környezeteket takar. A Condor jelenlegi változatában a felhasználók által a következő univerzumok használhatóak: Standard, Vanilla, PVM, MPI, Globus és Java.

A Standard univerzumban végrehajtott programok hibatűrők, mert a Condor rendszer állapotmentéseket (*checkpoint*) végez futás közben. Ha a program futása hardverhiba vagy a futtató gép tulajdonosának közreműködése miatt megszakad, akkor a lementett állapot segítségével a program futása egy azonos architektúrájú gépen folytatható tovább. A standard univerzum másik tulajdonsága, hogy a rendszerhívásokat, a feladatot benyújtó távoli gépre viszi át (*remote system calls*) egy árnyékfolyamat, a *condor_shadow* segítségével (4. ábra). Ha a futtató gépen egy rendszerhíváshoz (például fájlművelethez) ér a program, akkor azt a távoli gépen futó árnyékfolyamat hajtja végre (a távoli gépen lévő fájlon).

A standard univerzum használatához a meglévő programokat újra kell fordítani, hogy képesek legyenek a fenti lehetőség kihasználására. Ilyenkor a program tárgykódjához a Condor a saját Condor könyvtárát szerkeszti hozzá.



4. ábra A Standard univerzum távoli fájlműveletei

A standard univerzum használata azonban néhány megszorítással is jár. A legfontosabbak ezek közül a következők. Nem futtathatunk több folyamatból álló programokat, nem használhatunk néhány rendszerhívást (fork, exec, system) és nem használhatjuk a Unix változatok biztosította folyamatok közötti kommunikációs lehetőségeket (szemaforok, unix csővezetékek).

A Vanilla univerzum olyan programok futtatására használható, amelyeknél nincs lehetőség újrafordításra. Itt nincs lehetőség állapotmentésre és a rendszerműveletek átvitelére sem. Ez utóbbi miatt a program által használt fájloknak jelen kell lennie a futtató gépen is. Az elosztott fájlrendszerek használatával ez nem jelent problémát.

A másik megoldás, hogy a feladatleíró fájlban megkérjük a Condort a fájl átvitelére.

A PVM és az MPI univerzumok elosztott programok futtatására valók. A globus univerzum a Globus Toolkit nyújtotta lehetőségekre épít. Használatukról a dolgozatban később lesz szó.

A Java univerzum Java programok futtatására használható. A Condor gondoskodik a Java futtatókörnyezet elindításáról és a *CLASSPATH* paraméter beállításáról.

2.2.2. A feladat leírása a Condorban

A feladatleíró fájl meghatározza a feladat végrehajtásához szükséges erőforrásokat, így a központi menedzser eldöntheti, hogy a pool melyik gépén érdemes végrehajtani a feladatot.
A feladatleíró fájlban a kötelezően megadandó paraméterek:

- *executable* paraméterrel a végrehajtandó fájl neve, annak teljes elérési címével
- A másik a *queue* paraméter, amely a fájl által leírt feladatot helyezi bele a Condor várakozási sorába.

Ezek mellett megadható továbbá:

- a futtatáshoz használni kívánt architektúra (*requirements*)
- univerzum (*universe*)
- a futtatandó program paraméterei (*arguments*) és az alapértelmezett futtatási könyvtár (*initialdir*)
- a standard bemenetet tartalmazó fájl (*input*)
- a fájlba átirányított standard kimenetet (*output*) és a hiba célfájljai (*error*)

A felsoroltakon kívül számos más paraméter is megadható. Ezekről részletesen a Condor kézikönyvében [14] olvashat az érdeklődő.

A felhasználó a feladatleíró fájllal már kérheti a feladat végrehajtását a *condor_submit* parancs segítségével.

Egy példa feladatleíró fájlra:

```
# keresgelni fogunk a GNU grep segitsegevel
executable = /usr/bin/grep
universe = vanilla
# olyan Intel alapu Linuxos gep kell, aminek 20 MB-nal tobb
# memoria es 10 MB-nal tobb szabad diszk kapacitas van
requirements = (OpSys == "LINUX") && (Arch == "INTEL") && (Memory >
20) && (Disk > 10000)
# ha tobb ilyen is van, akkor a legnagyobb memoriaval
# rendelkezo kell
rank = Memory
initialdir = ~/
# bemeneti szoveg
input = ~/hosszuszoveg.txt
log = ~/log.out
# a keresendo szot parameterkent adjuk at a grep-nek
arguments = eztkellmegkeresni
# hibauzenetek ide kerulnek
error = ~/stderr1.out
# az eredmeny ide kerul
output = ~/megtalalta.txt
# berakjuk a varakozasi sorba
queue
# meg egy feladat beillesztese a varakozasi sorba
arguments = -v eztkikellhagyni
error = ~/stderr2.out
output = ~/kihagyta.txt
queue
```

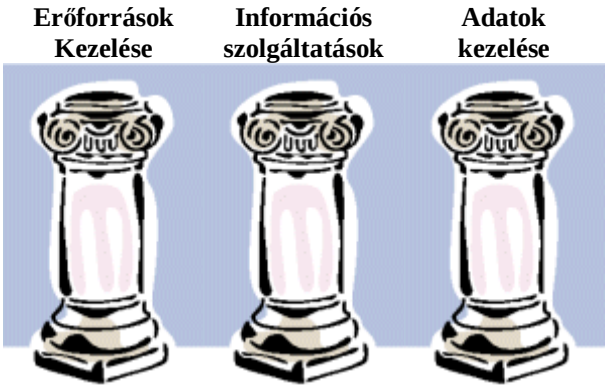
2.3. A Globus Toolkit

Ebben a részben először röviden ismertetem a Globus Toolkit [15] köztesréteg komponenseit, azok szerepét. Az áttekintés után részletesebben kitérek a dolgozat egy későbbi fejezetében ismertetett MPICH-G2 által használt komponensekre.

2.3.1. A Globus Toolkit

A Globus Toolkit egy nyílt forráskódú, nyitott architektúrájú köztesréteg szoftver számítási gridek felépítésére. A rendszert az Argonne National Laboratory munkatársai kezdtek fejleszteni, a jelenleg elérhető legújabb változat a 3.2 verziószámot viseli. A dolgozatban tárgyalt MPICH-G2 a 2.4 verzióval működik együtt, ezért annak architektúráját ismertetem részletesen.
A Globus Toolkit 3.0-val az egész Globus infrastruktúra megújult. A szolgáltatások kibővített Web szolgáltatások (Web Service) [16], úgynevezett grid szolgáltatások (Grid Services) formájában vannak jelen, amelyeket a Global Grid Forum által kiadott OGSI szabvány [17] ír le. A grid szolgáltatásainak felépítéséről egy másik Global Grid Forum szabvány az OGSA [18] rendelkezik.
Kompatibilitási okokból a Globus 3 változatlanul tartalmazza a 2.4 változat szolgáltatásait is, ezért az MPICH-G2 használható ezekkel a verziókkal is.

A Globus rendszer alapjául szolgáló komponensek három csoportba sorolhatók, az erőforrásokat kezelő komponensek, az adatokat kezelő komponensek és az információs szolgáltatások. Ezt szemlélteti az alábbi ábra.



5. ábra A Globus Toolkit építőpillérei

A három pillér feladata:

- Az első pillér a grid erőforrások kezelését, lefoglalását valósítja meg. Ehhez a pillérhez tartoznak a GRAM (Grid Resource Allocation Manager) és DUROC (Dynamically Updated Request Online Co-allocator) komponensek, amelyekről a következőekben még szó lesz.
- A második pillér az elosztott információs szolgáltatás, amely a kliensek számára szolgáltat információt a grid elérhető erőforrásairól. Ehhez a pillérhez tartoznak az MDS-t (Meta-Directory Service) alkotó GRIS (Grid Resource Information Service) és GIIS (Grid Index Information Service) komponensek.

- A harmadik pillér feladata az adatok elérése és kezelése grides környezetben. Ide tartoznak a GASS (Global Access to Secondary Storage) és GridFTP komponensek.

Mindhárom pillér alapjául szolgál a GSI (Grid Security Infrastructure) protokoll, amely a biztonságot hivatott biztosítani. Először erről ejtek néhány szót.

2.3.2. Biztonság a Globus-ban

A számítási gridek nagy kiterjedésű, heterogén rendszerek, amelyek több adminisztációs tartományt foglalnak magukba. A felhasználók és az erőforrások gyakran különböző adminisztrációs tartományokba esnek, ezért a biztonság kérdése alapvetően fontos.
A Globus Toolkit biztonsági mechanizmusát [24] a GSI (Grid Security Infrastructure) protokoll adja, amelynek létrehozásakor az elsődleges célok a következők voltak:

- Biztonságos kommunikáció biztosítása a számítási grid elemei között.
- Biztonság megvalósítása szervezeti határok mentén úgy, hogy elkerülhető legyen egy központi elem bevezetése.
- A grid felhasználói számára egyszeri bejelentkezés és delegáció megvalósítása, az erőforrások használatakor.

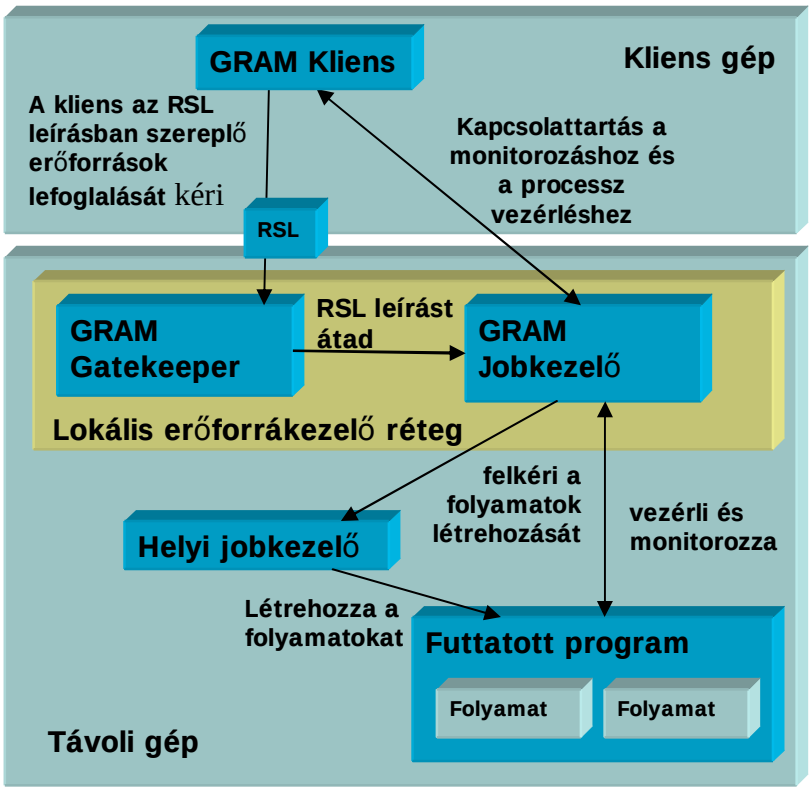
Ezek megvalósításához a GSI publikus kulcsú titkosítást [22], az SSL protokoll [21] feletti biztonságos kommunikációt és az X.509 tanúsítványokat [22] használja fel.
Mielőtt a felhasználó a grid bármely erőforrását használhatná, igényelnie kell egy X.509 tanúsítványt (*certificate*) a rendszertől. A tanúsítvány tartalmazza a felhasználó nevét, valamint a szervezetet, amelyhez tartozik. Az igényelt tanúsítványt ezután digitálisan alá kell íratni egy arra hivatott szolgáltatással, a hitelesítési szervezettel (*Cerificate Authority*). Az aláírt tanúsítványok hosszú lejáratúak – általában 5 évig érvényesek - és lejárat után meghosszabbíthatóak. Az aláírt tanúsítvány megléte azonban még nem elég a grid használatára, mert ehhez előbb a felhasználónak be kell jelentkeznie a gridbe. A bejelentkezéskor egy rövid ideig – tipikusan fél napig – érvényes átmeneti tanúsítvány generálódik, amelyet *grid-proxy*-nak neveznek. A Globus rendszerben levő programokhoz futtatáskor hozzárendelődik ez a *grid-proxy*, s azok ennek segítségével igazolni tudják magukat az erőforrások használatakor. Ezt a folyamatot nevezik delegációnak [23].

2.3.3. A Globus erőforrások lefoglalása

A Globus GRAM és DUROC komponense lehetőséget biztosít arra, hogy programokat futtassunk távoli erőforrásokon.
Az erőforrások kezelését két egymásra épülő réteg valósítja meg. A felső réteg tartalmazza a magas szintű globális erőforrás kezelő szolgáltatásokat, az alsó a lokális erőforrásokat lefoglaló szolgáltatásokat.

A számítási grid gépein a lokális erőforráskezelő rétegben (6. ábra) egy GRAM Gatekeeper nevű komponens várja az erőforrásfoglalási igényeket. A klienseknek egy erre kifejlesztett nyelven, az RSL (Resource Specification Language) [25] nyelven kell az igényeiket leírni, s ezt kell átadni a GRAM Gatekeeper szolgáltatásnak, ami továbbadja a helyi jobkezelő rendszerrel együttműködni képes GRAM jobkezelő

komponensnek a kérést. A GRAM jobkezelő feldolgozza az RSL kérést, majd a megfelelő paraméterekkel meghívja a helyi folyamatkezelőt (pl. a Condor feladatkezelőt), ami létrehozza a folyamatokat.



6. ábra A Globus lokális erőforráskezelő rétege

Az RSL leírás egy egyszerű *(attribútum=érték)* párokból álló fájl, amelynek legfontosabb attribútumai a következőek:

- *executable*: a futtatandó program neve és helye, amely lehet fájlnév vagy URL
- *directory*: a futtatás könyvtára
- *arguments*: a program argumentumai
- *stdin*: a futtatandó program standard bemenetét adó fájl vagy URL
- *stdout*: a standard kimenetet az itt megadott fájlba menti le (lehet URL is)
- *stderr*: a standard hibakimenetet ebbe a fájlba menti le (lehet URL is)
- *count*: az itt megadott peldányszámban indítja el a programot
- *jobtype*: itt a feladat típusát határozhatjuk meg

URL megadása esetén a Globus képes HTTP vagy FTP szerverről letölteni, illetve oda feltölteni a hivatkozott fájlt, de használhatjuk a Globus adatszolgáltató részéhez tartozó, ezen célra kifejlesztett GASS komponenst is.

Példa: a következő egyszerű RSL fájl segítségével fájlba mentünk egy könyvtárlistát

```
(count=1)
(directory=/home)
(arguments=" -l")
(stdout=/tmp/ls.out)
(executable=/bin/ls)
```

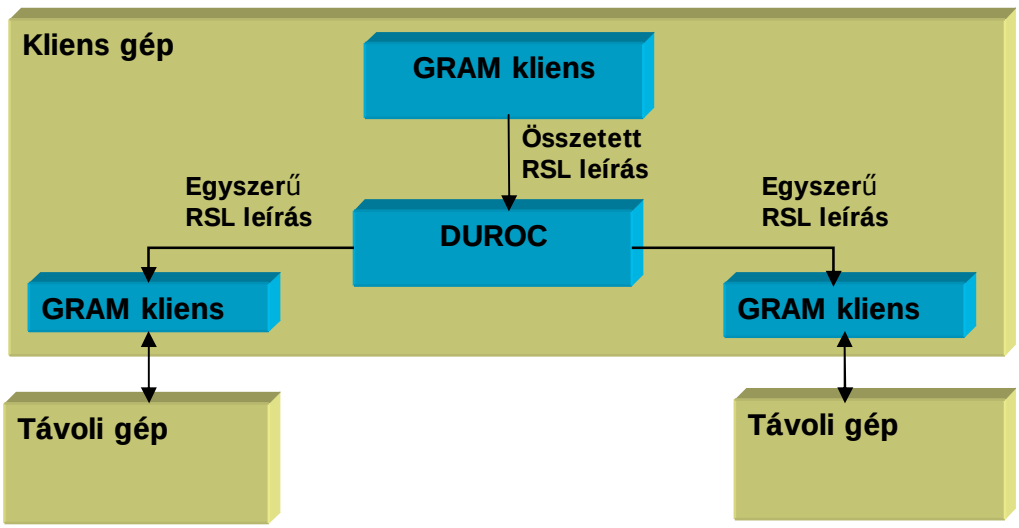
A kliensek képesek a GRAM jobkezelőn keresztül a létrehozott folyamatokkal kapcsolatot tartani. Például lekérdezhetik a folyamatok állapotát és megszakíthatják a folyamatok végrehajtását is.

A legegyszerűbb GRAM kliensek a *globusrun* és a *globus-job-run* programok, amelyek a Globus Toolkit részét képezik. Az elsővel egy RSL leírás alapján hajthatjuk végre a feladatot, a másodiknál RSL leírás nélkül, parancssori paraméterek segítségével adhatjuk meg a végrehajtandó feladatot.
Bonyolultabb GRAM kliensek írásához a Globus programozói interfészt (API-t) biztosít.

Példa GRAM kliens használatára:

```
repa> globusrun –r oldblackjoe.local –f elso.rsl
repa> globus-job-run oldblackjoe.local –s /bin/hostname –stdout –s
out.txt
```

A fenti példa egy egyszerű program távoli gépen történő végrehajtását mutatta. A grid rendszerek alapvető célja azonban összetett, több számítógépet használó alkalmazások futtatása. Ha egy kliens a számítási grid több gépén szeretné egyszerre futtatni programjait, akkor ezeket az erőforrásokat egyszerre kell lefoglalnia. Ennek a problémának a megoldásában nyújt segítséget a Globus globális erőforráskezelő rétege, amelynek felépítését a 7. ábra szemlélteti. A fő szerepet itt a Globus DUROC [26] komponense játssza, amely összetett RSL leírásokat képes több egyszerű RSL leírássá alakítani, majd az egyszerű leírások alapján az erőforrásokat egyszerre lefoglalni (6. ábra).
Párhuzamos programok, nevezetesen MPI programok számítási gridekben történő futtatásához elengedhetetlen az erőforrások szimultán lefoglalása.



7. ábra A Globus globális erőforráskezelő rétege

Példa összetett RSL leírásra:

```
+
( &(resourceManagerContact="repa.local")
   (count=1)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (LD_LIBRARY_PATH /usr/local/globus24gcc32//lib/))
   (directory= "/home/farkas")
   (executable= $(GLOBUSRUN_GASS_URL) # "/home/farkas/mandel_mpi")
)
( &(resourceManagerContact="oldblackjoe.local")
   (count=1)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (LD_LIBRARY_PATH /usr/local/globus24gcc32//lib/))
   (directory= "/home/farkas")
   (executable= $(GLOBUSRUN_GASS_URL) # "/home/farkas/mandel_mpi")
)
```

Az összetett RSL leírás + jellel kezdődik és több egyszerű RSL leírással megfogalmazott részfeladatot fog össze. A fenti példa egyszerre indítja el a két meghatározott gépen a megadott MPI programot.

2.3.4. A Globus információs szolgáltatása

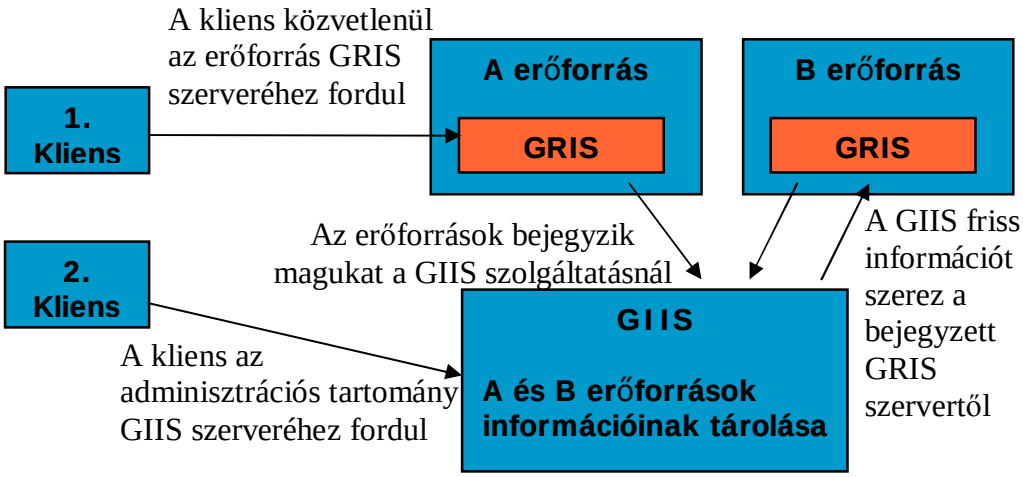
A grid rendszerek használatakor a következő alapvető kérdések merülnek fel. Hogyan döntheti el az alkalmazás, hogy milyen erőforrások elérhetőek el? Hol vannak ezek az erőforrások? Mi ezeknek az erőforrásoknak az aktuális állapota? Ezeknek a kérdéseknek a megválaszolásához egy általános információs rendszerre van szükség, amelyet a Globus Toolkitben az MDS (Meta-Directory Service) szolgáltatás képvisel.

A Globus első változataiban az MDS szerepét adminisztrációs tartományonként egy-egy központi LDAP [20] szerver látta el. Ennek azonban, mint minden központosított szolgáltatásnak, rossz hatása volt a rendszer teljesítményére, így szükségessé vált egy új változat tervezése. Az MDS második változatával egy elosztott, skálázható információs rendszert sikerült létrehozni.

Az MDS-2 [19] két szereplőből áll. Az első a GRIS (Grid Resource Information Service) [19], amely közvetlen az erőforrás mellett található, és magáról az erőforrásról ad információkat a klienseknek. A második a GIIS (Grid Index Information Service) [19], amely több erőforrásról tárol információt, temporális jelleggel (*cache*). A GRIS szolgáltatások saját információjuk terjesztéséhez a GRRP (Grid Resource Registration Protocol) protokoll segítségével bejegyezhetik magukat egy vagy több GIIS szolgáltatásánál.

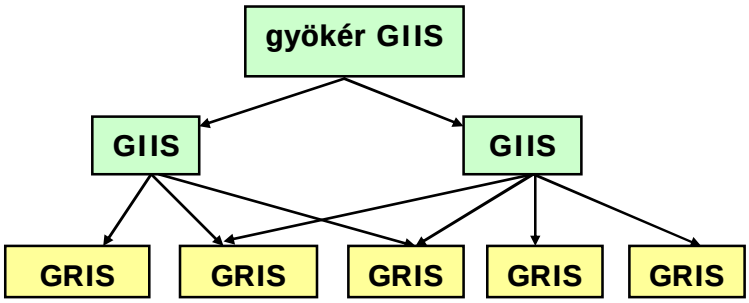
Az információra szoruló kliensek a GRIP (Grid Resource Inquiry Protocol) protokoll segítségével feltehetik kérdéseiket a GIIS szervereknek, amelyek a tárolt információ alapján megadják a lehetséges erőforrások halmazát. Ha a kliens ismeri az erőforrás elérhetőségét, akkor közvetlenül annak GRIS szolgáltatásától kérdezheti le az állapotot ugyancsak a GRIP protokoll segítségével (8. ábra).

Ez nagyon hasonló a Webes keresőrendszerekhez, ahol a beírt kifejezésre vonatkozó keresést a honlapok keresőrendszer által tárolt változatain végzik el, majd a találatokat – amelyek lehetnek elavultak is – visszaadják a keresést kezdeményezőnek, aki ellenőrizheti azokat.



8. ábra Az MDS működése

A GIIS szolgáltatások hierarchikus felépítésével a kliensek a DNS rendszerhez hasonló módon kérdezhetik le az információkat a névszolgáltatástól (9. ábra). Ha a kliens ismeri a gyökér GIIS szerver címét, akkor az összes bejegyzett szolgáltatást megtalálhatja.



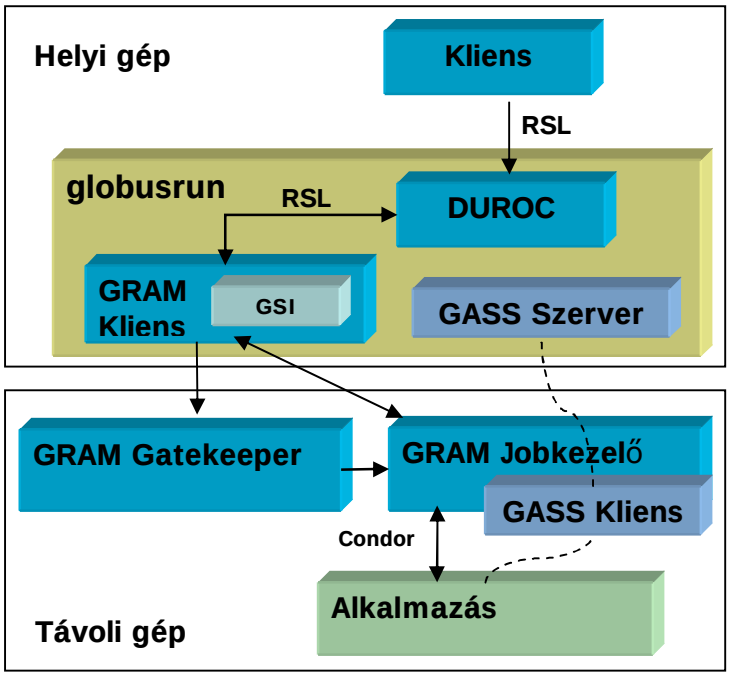
9. ábra Hierarchikus erőforráskereső rendszer

2.3.5. A Globus adatkezelése

A számítási grideket alkotó gépek erősen heterogén környezetében megvalósíthatatlan az, hogy minden gép ugyanazt a fájlrendszert lássa. A heterogenitáson kívül a felhasználók jogai is gondot okoznak Ez ahhoz a problémához vezet, hogy nem minden gépen van jelen, vagy nem ugyanott található a kliens által futtatandó program vagy a program által használt bemeneti és kimeneti fájl. A probléma megoldásában a Globus GASS komponense [35,27] segít.

A GASS működésének lényege (10. ábra), hogy ha az RSL leírásban URL fájlhivatkozások vannak, akkor a hivatkozásokat átmásolja a távoli gép GASS

kliensének átmeneti tárolójába. A távoli gépen futó program ekkor használni tudja az átmeneti tárolóban levő fájlokat. Az URL hivatkozások mutathatnak HTTP, FTP és GASS szerverekre. Ez utóbbi esetén a helyi gépen egy GASS szerver komponens szükséges, amelyen keresztül a hivatkozott fájlok letölthetők a távoli gépre, illetve a kimeneti fájlok feltölthetők a helyi gépre. GASS szerverre való hivatkozás esetén az URL *x-gass://szerver-név:szerver-port/fájl-elérési-út* alakú.



10. ábra A Globus adatkezelése

A 2.3.3. részben példaként szereplő összetett RSL leírásban a *$(GLOBUSRUN_GASS_URL)* bejegyzés hatására a futtatandó program átmozgatásra kerül a lefoglalt erőforrása. Ebben az esetben a rendszer biztosítja a GASS szervert a feladat végrehajtásának idejére

A Globus Toolkit használata során felmerült az igény egy gyors, biztonságos, hatékony és megbízható adatátviteli lehetőségre. Ezt az igény elégíti ki a Globus második változatához készített GridFTP protokoll [30].

A protokoll tervezésekor olyan tulajdonságok biztosítását írták elő, amelyeket egy létező rendszer sem biztosít egyszerre. Ezek közül a legfontosabbak a következők:

- A GSI lehetőségeivel biztosítja a biztonságos adatátvitelt.
- Képes az adatok párhuzamos TCP adatcsatornákon történő adatátvitelre.
- Nagy fájlok esetén, ha a kliensnél nem az egész fájlra van szükség, akkor képes a fájl részletének átvitelére is.
- Hatalmas adatok esetén elképzelhető az is, hogy egyetlen adatfájl nem egy, hanem több szerveren, szétosztva helyezkedik el. A GridFTP a sávos adatátvitel biztosításával képes ezeket a szervereket egyszerre használva elérni a szétosztott adatfájlt.
- Megbízható és újraindítható adatátvitelt biztosít.
- Monitorozási lehetőséget biztosít az átviteli sebesség méréséhez.

Ezek a tulajdonságok a GridFTP protokollt alkalmassá teszik a grides környezetbeli használatra. Létezése különösen fontos az adatintenzív alkalmazások [34] esetében, hiszen itt nagy méretű adathalmazok, gyors mozgatására van szükség. Ezeket az adatokat rendszerint párhuzamos programok segítségével dolgozzák fel.

3. fejezet

Az MPI használata az ismertetett rendszerekben

3.1. Hibatűrő MPI implementáció a Harness rendszerhez

A mai, nagy teljesítményű, osztott rendszerek világában egyre inkább előtérbe kerül a rendszer hibatűrő tulajdonsága. Ez két okra vezethető vissza.
Az első ok az, hogy a hardverek egyre bonyolultabbá válnak. A hardver összetettségével nem csak a rendszer teljesítménye növekszik, hanem a hardverhiba lehetősége is. Például manapság nem ritkaság az olyan számítógép, amelyben több ezer, vagy akár több tízezer processzor található, s ezeknél a gépeknél teljesen megszokott dolog, hogy időnként egy-egy processzor meghibásodik. A klaszterek esetében is fennáll ez a probléma, mert azokat költségkímélő okokból gyakran általános használatra szánt számítógépekből és hardverelemekből építik fel. Míg otthonainkban nem okoz nagy gondot egy hiba felbukkanása után a rendszer újraindítása, addig a klaszterekben ugyanezektől a hardverektől hibamentes környezetet várnánk el.
A másik ok a programok jellegének változása. A mai tudományos programok (például részecskefizikai problémák vagy akár géntérképek készítése) nem órákig, hanem hetekig, vagy akár hónapokig futnak, míg végül eredményt szolgáltatnak. Ez a futási idő gyakran meghaladja a futtató hardver átlagos meghibásodási értékét (MTBF – Mean Time Between Failure), ezért a program lefuttatása nem sikerülhet. Hibatűrés nélkül ezeket a programokat nem, vagy csak nagy ráfordítással lehetne végrehajtani. Ráadásul a megszakadt programfutások pazarolják az amúgy is szűkös és drága erőforrásokat.

Szekvenciális programok esetében a hibatűrést gyakran a program számára átlátszó módon biztosítják. Ilyenkor a programról futás közben állapotmentéseket végeznek, és hiba esetén az előző állapotmentés alapján folytatni tudják a program végrehajtását. A programnak nem kell tudnia az állapotmentésekről, mert azt a futtató rendszer automatikusan elvégzi.
Hiba esetén csak az előző állapotmentés óta eltelt idő alatt végzett munka veszik kárba. Ilyen állapotmentést végez a 2. fejezetben bemutatott Condor rendszer a standard univerzum által biztosított környezetben.
Párhuzamos programok esetén az ehhez hasonló hibatűrés biztosítása ennél bonyolultabb dolog. Nem egyszerű dolog ugyanis a párhuzamos program globális állapotát elmenteni, az éppen úton levő üzenetekkel és a részfolyamatok állapotával. Adódik azonban másféle lehetőség is. Ezeket a következő rész ismerteti.

3.1.1. Hibatűrés MPI programok esetében

A legtöbb MPI implementáció nem támogatja hibatűrő programok készítését. Ennek legfőbb oka, hogy a szabvány szerint minden MPI rutin végrehajtása csak

sikeres, illetve sikertelen jelzéssel tér vissza. Ez utóbbi a legtöbb esetben a program végrehajtásának azonnali megszakadásához vezet.

A ma használatos MPI implementációk hibatűrő képességeit három csoportba oszthatjuk. Mindhárom csoport különböző szinten támogatja hibatűrő MPI programok készítését. Lássunk egy-egy példát ezekre.

A legmagasabb szinten, az alkalmazás szintjén történő hibatűrésen az előbbiekben leírt állapotmentéseket értjük. Ilyen állapotmentéseket támogat az Indiana University LAM/MPI eszköze [31], mellyel képes az MPI programok globális állapotának elmentésére és visszaállítására. A programok állapota mellett biztosítja az állapotmentés alatt éppen úton lévő üzenetek célbajuttatását is.

A legalacsonyabb szinten, a hálózat szinjén történő hibatűrést biztosít a Los Alamos-Message Passing Interface (LA-MPI) [32]. Létrehozásának célja hibatűrő üzenetátadás megvalósítása volt nagy kiterjedésű klaszterek esetén. Bár az Internet esetében létezik hibamentes átvitelt biztosító kommunikációs protokoll (pl. TCP), de ennek sok folyamat esetén történő használata a kapcsolatorientált jellege miatt nem lehetséges. Képzeljük el, hogy milyen terhelést jelentene az operációs rendszernek néhány olyan MPI folyamat, amely több ezer folyamattal kommunikál. Ekkor mindegyik MPI folyamat felé biztosítani kellene egy kapcsolatot, azaz egy kapcsolódási végpontot a számítógépen, ami hamar az erőforrások kimerítéséhez vezet.
A hibatűrés érdekében a LA-MPI rendszer számos különböző hálózati protokoll (pl. UDP, HIPPI-800 stb.) felett képes kommunikálni, ráadásul mindezt több hálózati interfész szimultán használatával. A hálózat redundáns kiépítésével így elérhető, hogy az esetleges átviteli rendszerekben történő meghibásodás esetén is célbaérjen az elküldött adatcsomag.

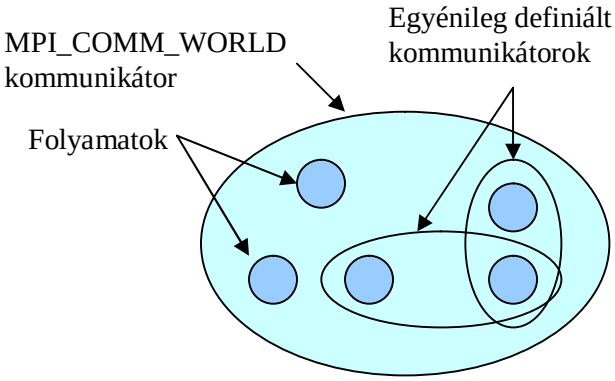
Az előző két szint között helyezkedik el a Harness metaszámítási rendszerhez készített FT-MPI [37, 38], amely az MPI kommunikációs szintjén kezeli a hibákat. A következőekben ezt ismertetem, de ehhez alapszinten ismerni kell az MPI folyamatmodelljét, ezért ahhoz egy rövid áttekintést is adok.

3.1.2. Az MPI folyamatmodellje

Minden MPI folyamat egy vagy több úgynevezett kommunikátorhoz tartozik (11. ábra). Ezek a kommunikátorok a folyamatok csoportba foglalása mellett egy kommunikációs környezetet is meghatároznak. A kommunikátorokban, mint csoportokban a folyamatok sorszámozva vannak (0-tól kezdve, folytonosan).

MPI folyamatok közötti kommunikáció esetén a küldő folyamatnak meg kell adnia a célfolyamat kommunikátorát és a célfolyamat megadott kommunikátorban értelmezett sorszámát. Az MPI kollektív kommunikációs lehetőségei ([5]/4. fejezet) egy paraméterként adott kommunikátor összes folyamatára vonatkozó kollektív műveletet valósítanak meg.

Minden MPI programban az *MPI_Init* meghívása után létrejön egy *MPI_COMM_WORLD* nevű kommunikátor, melyben az MPI program minden folyamata benne lesz. Ennek a kommunikátornak a segítségével a program minden folyamata kommunikálni tud egymással. Az MPI-1 szabvány szerint nincs lehetőség ennek a kommunikátornak a bővítésére, mert a szabványban nincs lehetőség folyamatok indítására.



11. ábra Az MPI folyamatmodell

A függvénykönyvtár lehetőséget biztosít saját kommunikátorok definiálására, kommunikátorok szétosztására és összefűzésére is.

3.1.3. Az FT-MPI modell

Az MPI szabvány szemantikája szerint, ha egy folyamat vagy egy üzenet elveszik, akkor minden ebben érintett kommunikátor *invalid* állapotúvá válik. Ilyen állapotú kommunikátorban a további MPI hívások eredménye nem definiált, azok akár a program abortálásához is vezethetnek. Ezzel szemben az FT-MPI modell biztosítja, hogy egy a modellt használó, *n* folyamatból álló MPI program túlélje *n-1* folyamatának elvesztését.

Az FT-MPI kiterjeszti a kommunikátorok állapotát a {*valid, invalid*} halmazról, a több állapot leírását biztosító {*FT_OK, FT_DETECTED, FT_RECOVER, FT_RECOVERED, FT_MPI_ABORT, FT_FAILED*} halmazra. A két szélső érték megfelel a korábbi állapotoknak. Az *FT_DETECTED* értéket akkor kapjuk vissza, ha a rendszer észlelte a hibát, de még nem kezdte el a kijavítását. Az *FT_RECOVER* állapot a hiba javítása közben, míg az *FT_RECOVERED* a hiba javítása után észlelhető. Ekkor a felhasználónak lehetősége van az *FT_OK* állapotba történő visszatéréshez, de ehhez újra kell építenie a kommunikátort. Az *FT_MPI_ABORT* állapot akkor figyelhető meg, ha a kommunikátorra, annak valamelyik folyamatában meghívták az *MPI_Abort()* függvényt.

3.1.3.1. A hiba felfedezése

Egy adott kommunikátor valamely folyamatának elvesztésekor a többi folyamat a kommunikátor állapotának lekérdezésével jöhet rá a hibára. Ha azonban valamelyik folyamat a kiesett folyamatnak akar üzenetet küldeni, akkor az *MPI_ERR_OTHER* hibaüzenettel tér vissza.
Hiba felfedezése MPI kommunikációs primitív használatakor:

```
rc = MPI_Send (&next, 1, MPI_INT, i, WORK_TAG, comm);
if ( rc == MPI_ERR_OTHER )
{
      /* hiba történt a kommunikátorban
         a hibás kommunikátort újra kell építeni */
}
```

A kommunikátor helyreállítása előtt lekérdezhető a rendszertől a hiba oka:

```
/* Hány folyamat veszett el? */
MPI_Comm_get_attr ( comm, FTMPI_NUM_FAILED_PROCS, &valp, &flag );
int numfailedprocs = (int) *valp;

/* Kik vesztek el? */
MPI_Comm_get_attr ( comm, FTMPI_ERROR_FAILURE, &valp, &flag );
int errorcode = (int) *valp;

MPI_Error_get_string ( errcode, errstring, &flag );
printf ( "A hiba oka: %s\n",errstring );
```

A hibaüzenet tartalmazza az *MPI_COMM_WORLD* kommunikátor elveszett folyamatainak sorszámát. Ezt ügyes szövegfeldolgozással ki lehet venni a hibaüzenetből. Erre példát az FT-MPI példaprogramjai [33] között találhatunk.

3.1.3.2. A kommunikátor újraépítése

Hiba esetén egy kommunikátor újraépítése egyszerűen annak lemásolásával történik, azzal a különbséggel, hogy a másoló eljárás hívásakor az új kommunikátort tartalmazó kimeneti változónak az *FT_MPI_CHECK_RECOVER* konstans értéket kell tartalmaznia.

```
oldcomm = MPI_COMM_WORLD;
newcomm = FT_MPI_CHECK_RECOVER;
rc = MPI_Comm_dup (oldcomm, &newcomm);
```

Az alkalmazás indításakor a felhasználó maga döntheti el, hogy milyen módon történjen a hiba javítása. Meghatározható az, hogy mi történjen az elveszett folyamatokkal, valamint az is, hogy mi történjen az úton levő üzenetekkel. A folyamatok lehetőségei a következők.
Az *FT_MODE_REBUILD* módban a rendszer minden hiba miatt elvesztett folyamatot újból elindít. A folyamatokban az *MPI_Init()* hívásakor lehetőség van annak ellenőrzésére, hogy az aktuális folyamat egy újraindított folyamat-e. Ennek köszönhetően lehetőség van olyan programrész hozzáadására, amely csak újraindítás esetén fut le.

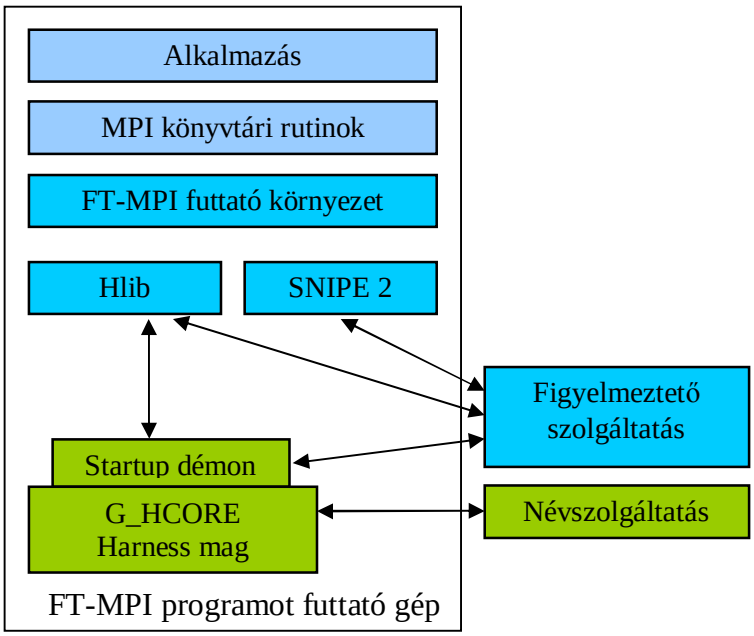
```
rc = MPI_Init ( &argc, &argv );
if ( rc == MPI_INIT_RESTARTED_NODE)
{
    /* ez egy újraindított folyamat:
       ide kerülnek az újraindítás után
       lefuttatandó programrészletek */
}
```

Az *FT_MODE_SHRINK* módban, folyamatok elvesztése esetén a kommunikátor újraépítése a kommunikátor folyamatainak újraszámozását jelenti. A végeredmény egy rövidebb, folytonos kommunikátor lesz. Ennek a módnak a használatakor az MPI programot alkotó folyamatoknak újra le kell kérdezniük a kommunikátorbeli sorszámukat, mert az megváltozhat.

Az *FT_MODE_BLANK* módban a rendszer nem tesz semmit az elveszett folyamatok pótlására, ezért az újraépített kommunikátor sem lesz folytonos, s benne a kiesett folyamatok hibás hivatkozások lesznek.
Az *FT_MODE_ABORT* módban hiba fellépésekor a program abortálni fog.
A kommunikátor újraépítésének módja mellett definiálható a hibás kommunikátorban történő üzenetek kezelésének módja is.
Az *FT_MODE_NOP* módban a hibás kommunikátorban meghívott minden üzenetküldési próbálkozás hibaüzenettel tér vissza, azaz a kommunikátor újraépítéséig nem lehetséges üzenetküldés.
Az *FT_MODE_CONT* módban a hibás kommunikátorban továbbra is lehetséges az üzenetküldés. A hibás folyamatot nem érintő üzenetek rendben célbaérnek, de a kommunikátor újraépítéséig minden hibás folyamat felé irányuló üzenetküldés hibaüzenettel tér vissza, még akkor is ha a folyamatot a rendszer már újraindította.

3.1.4. Az FT-MPI felépítése

Az FT-MPI a Harness rendszer architektúrájára épített MPI implementáció. A Harness rendszernél használt Java implementáció azonban a készítők szerint nem volt elég hatékony, ezért elkészítették annak C nyelvű implementációját, amelyet General Harness Core-nak (*G_HCORE*) neveztek el. Ez az implementáció távoli eljáráshívásokon (RPC) [43] alapul, ezért nem képes együttműködni az RMI-t [42] használó Java implementációval. Az új Harness mag megtartotta a Harness architektúrát, továbbra is használ névszolgáltatást, és képes *plug-in*-ek kezelésére (pl. *ft-mpi plug-in*).



12. ábra Az FT-MPI felépítése

Az MPI rendszerben fellépő hibák információjának terjesztéséhez egy új figyelmeztető szolgáltatásra (*ftmpi_notifier*) volt szükség. Ezt a szolgáltatást a névszolgáltatáshoz hasonló módon, egy távoli gépen lehet elindítani.

Az új Harness magra egy ***startup démon*** program épül, amely az FT-MPI használatát biztosító ***ft-mpi plug-in*** betöltéséért felelős.

Az FT-MPI a kommunikációhoz a SNIPE 2 könyvtárat használja. Ennek segítségével hatékony, gyors kommunikációt sikerült megvalósítani.

3.1.4. Az FT-MPI további tulajdonságai

Az FT-MPI jelenlegi állapotában teljes implementációja az MPI-1.2 szabványnak. Ezeken a rutinokon kívül tartalmaz néhány MPI-2 szabvány által előírt dolgot is, mint például a C++ nyelv támogatása, az MPI-IO rutinok használatának lehetősége nem hibatűrő programok esetén.

Sajnos a jelenlegi C implementáció nem képes együttműködni az eredeti Harness Java implementációjával, cserébe viszont majdnem olyan tág heterogenitást biztosít, mert a forráskód lefordítható AIX, IRIX-6, Tru64, Linux-Alpha, Solaris, Linux és Win32 rendszerekre.

3.2. Az MPI és a Globus Toolkit

Az MPICH függvénykönyvtár [28] egy szabadon elérhető, hordozható implementációja az MPI-1.1 szabványnak, amelyet az Argonne National Laboratory munkatársai készítettek. Az MPICH fejlesztésének menete során az MPI-1.1 szabvány funkciói mellett helyet kapott az MPI-2 szabvány 9. fejezetében leírt MPI I/O is.
Az MPICH függvénykönyvtárnak különböző hardverarchitektúrákhoz optimalizált változatai érhetők el, maga az MPICH is innen kapta a nevét: az MPI és a CHameleon szavak összepárosításából.

A homogén számítógépekből álló klaszterek népszerűségének növekedtével az elosztott alkalmazásokat fejlesztők megpróbálkoztak egymástól távol eső klaszterek összekötésével. A próbálkozások nehezen megoldható problémákhoz vezettek. Ilyen probléma például, hogy különböző adminisztrációs körzetekben levő klaszterek különböző módon azonosítják a felhasználókat, s azoknak a klaszteren belüli jogai is különböznek. Problémát jelentett a klaszterek közötti nagy távolság, ami lassú hálózati kommunikációban nyilvánult meg. Probléma volt az erőforrások egyszerre történő lefoglalása, valamint az, hogy a klasztereken különbözött a fájlrendszer.

A nagy teljesítményű számítási gridek megjelenésével számos probléma kiküszöbölhetővé vált, ezért az alkalmazásfejlesztők oldalán felmerült az igény olyan eszközök iránt, amelyek népszerű programozási modelleken alapulnak és képesek a grid lehetőségeit kihasználni. Így volt ez az MPI esetében is. Az igényeket kielégítve megjelent az MPICH-G, amely az MPICH függvénykönyvtár Globus gridet használni képes (grid enabled) változata.

Az MPICH-G megjelenésekor a Globus Toolkitben a kommunikációért a Nexus kommunikációs könyvtár volt a felelős, amelynek teljesítménye azonban nem bizonyult elegendőnek. A Globus Toolkit 2.0 megjelenésével eljött a lehetőség az MPICH-G új alapokra helyezéséhez. Az előző verzióhoz hasonlóan használja a Globus szolgáltatásait, de megváltoztatott kommunikációs infrastruktúrájának köszönhetően sokkal hatékonyabb elődjénél.
Az MPICH-G2 az MPICH jelenleg elérhető 1.2.5.2 verziójában megtalálható, és már néhány grides környezet esetén hasznos MPI-2 rutint is tartalmaz. Ezek részletesebb ismertetése a fejezet további részeiben található.

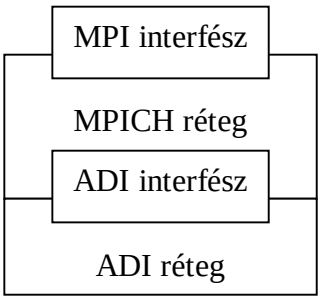
Az MPICH-G2 nem összetévesztendő az MPICH2-vel, amely egy új, fejlesztés alatt levő változata (béta változata) az MPICH függvénykönyvtárnak. Ennek célja az MPI-2 szabvány teljes implementációja. Az MPICH2 azonban nem képes a Grid használatára. Részletesebb ismertető elérhető az MPICH-2 honlapján [36].

Lássunk egy példát, ahol az MPICH-G2 használata nagy segítséget jelenthet. Ehhez térjünk vissza egy picit a dolgozat bevezetőjében említett, elosztott programokkal megvalósított feladatok csoportosítására. Eszerint a második csoportba azok a feladatok tartoznak, amelyek természetükből adódóan elosztottak. A nagy kiterjedésű hálózattal összekötött számítógépek egy számítási gridbe való összefogásával ezeknek a feladatoknak a megoldására is lehetőség nyílik.

3.2.1 Átlátszóság és heterogenitás az MPICH esetében

Az MPICH nagymértékű hordozhatósága a réteges felépítésének köszönhető. Ennek a felépítésnek köszönhetően a rendszer képes elrejteni a használója elöl az MPI folyamatok közötti kommunikáció tényleges menetét.

A rendszer rétegei interfészeken keresztül kapcsolódnak egymáshoz. Legfelül található az MPICH réteg, amely az alkalmazások felé az MPI interfészt szolgáltatja. Ebben a rétegben van a rendszer kódjának legnagyobb része, nevezetesen az, amely nem függ az alatta levő hálózat kiépítésétől és a folyamatok kezelésének módjától. Az alsó réteg az Abstract Device Interface, amely a hálózatfüggő részeket tartalmazza. Ez a réteg az MPI interfésznél lényegesen kisebb ADI interfészen keresztül kommunikál az MPICH réteggel. Az ADI réteg cseréjével egyszerűen a hardver architektúrájához igazítható a folyamatok közötti kommunikáció. Például a megosztott memóriás gépek esetén érdemes kihasználni az egy gépen futó folyamatok közötti kommunikációkor a közös memóriát.
Az MPICH függvénykönyvtár több ilyen ADI implementációt tartalmaz, amelyeket MPICH eszközöknek (device) nevezünk. A függvénykönyvtár forráskódjának lefordításakor ki kell választani a használni kívánt MPICH eszközt.
Az MPICH-G2 valójában a Globus 2 infrastruktúrát használó implementációja az ADI rétegnek. Ezt az implementációt a globus2 eszköz testesíti meg.



13. ábra Az MPICH réteges felépítése

3.2.2. Az MPICH-G2 használata

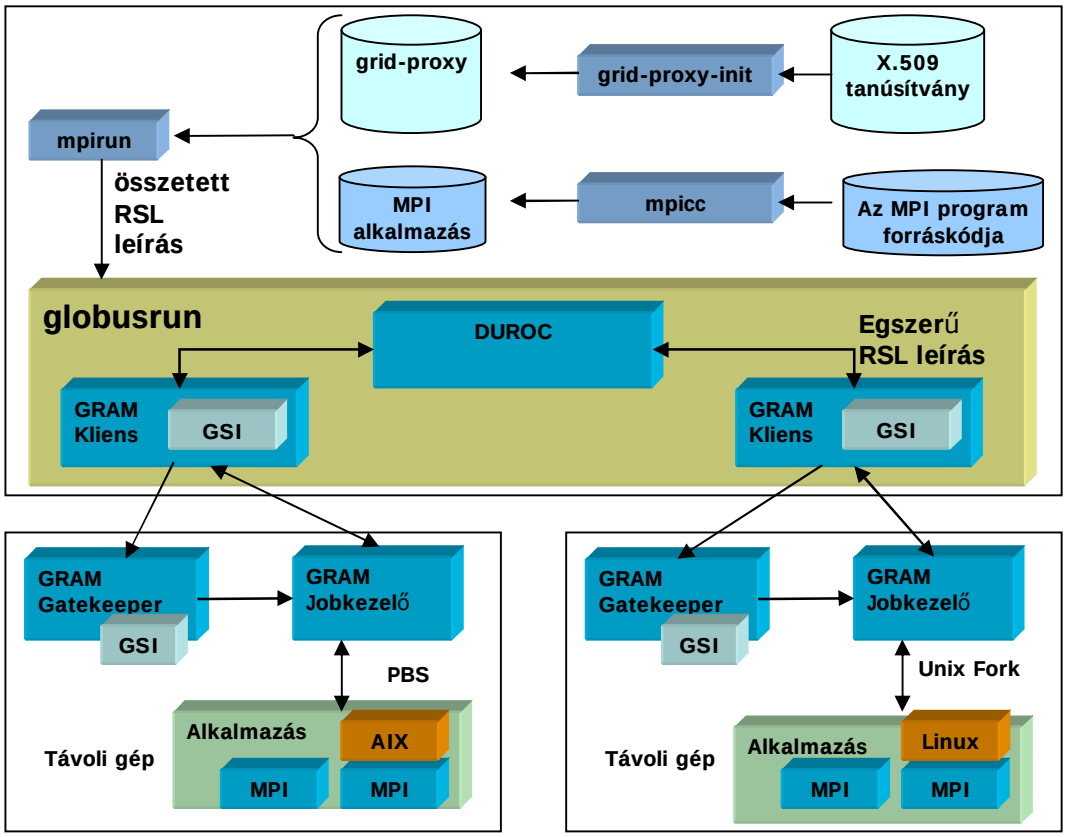
Mint már említettem, az MPICH-G2 a dolgozat 2.3. részében áttekintett Globus Toolkit nyújtotta infrastruktúrát használja ki MPI programok futtatására. Tekintsük át, hogy hogyan is történik ez a gyakorlatban (14. ábra).

A program lefordításához az MPICH-G2 esetében is használhatjuk az MPICH rendszer nyújtotta eszközöket. A C nyelvű programok fordításához az ***mpicc***, a C++ nyelvű programok fordításához az ***mpiCC***, Fortran77 és Fortran90 programok fordításához az ***mpif77*** illetve ***mpif90*** használható. A Globus Toolkit használatából eredően az MPICH-G2 programok lefordításához szükség van a Globus néhány könyvtárfájljára (library). Ezeknek a könyvtáraknak program kódjával való összeszerkesztését (link) is megoldják a fent említett eszközök.

A program futtatásánál, mint minden Globus infrastuktúrát használó alkalmazás esetében, elengedhetetlen az erőforrások használatához szükséges ***grid-proxy***, amelyet a ***grid-proxy-init*** paranccsal hozhatunk létre. Ezek után a programot az MPICH-ből ismert ***mpirun*** eszköz segítségével futtathatjuk a számítási gridben. Az mpirun parancsnak a ***-np*** paraméter után megadható, hogy hány példányban szeretnénk elindítani a programot a számítási gridben. Általában a számítási gridben levő gépek nem használnak azonos fájlrendszert, ezért nem minden gépeken található meg a futtatni kívánt program. A

probléma kiküszöböléséhez az mpirun *-stage* paraméterével megkérhetjük a Globust, hogy minden távoli gépre szállítsa át a futtatandó fájlt.

Az MPICH-G2 *mpirun* parancsa valójában egy RSL leírást készít a programhoz, amelyet átad a Globus Toolkit *globusrun* programjának. Az elosztott programokat a számítási grid több gépén célszerű futtatni, ezért több erőforrás, szimultán lefoglalására van szükség. Az mpirun által készített RSL leírás ezen okból egy összetett RSL leírás lesz, amelynek feldolgozásához a Globusnak a DUROC komponensre van szüksége. A DUROC a kapott leírást több egyszerű RSL leírássá alakítja, majd átadja őket az igényelt erőforrásokhoz létrehozott GRAM klienseknek. A GRAM kliensek a 2.3.3 részben leírt módon elindítják az MPI programokat a távoli gépeken.



14. ábra Az MPICH-G2 működése

Az *mpirun* parancs által készített RSL leírást megtekinthetjük, ha a *–dumprsl* paraméterrel indítjuk a programot. Ekkor az mpirun nem a *globusrun* parancsnak adja az RSL leírást, hanem kiírja a képernyőre. A kimentett leírásba szükség esetén mi is belenyúlhatunk, vagy akár mi is átadhatjuk a *globusrun* programnak.

3.2.3 Az MPICH-G2 teljesítményét javító lehetőségek

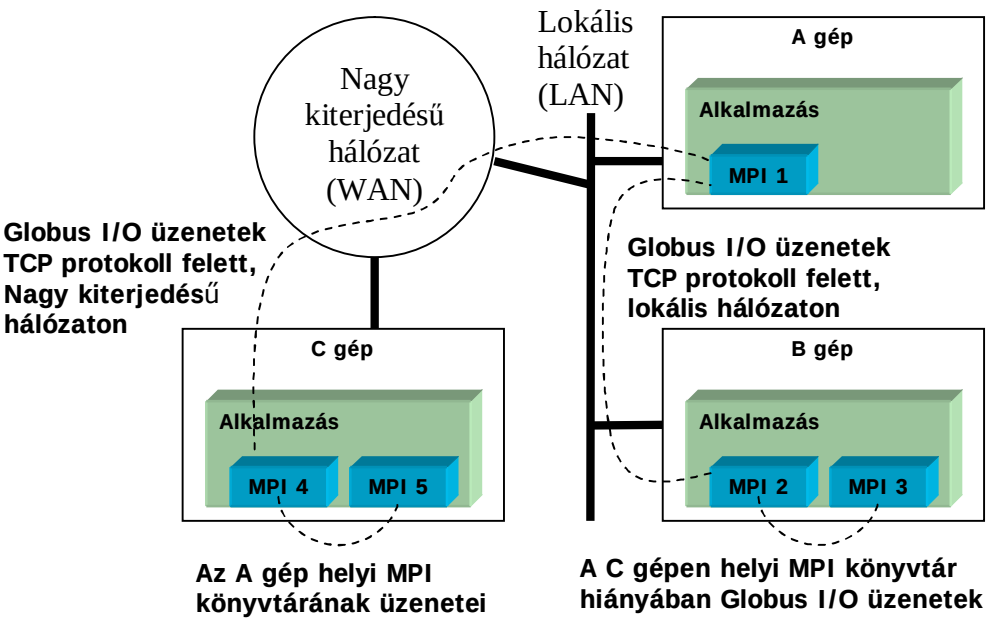
Az MPICH-G2 grides környezethez igazított MPI implementációja tartalmaz néhány olyan lehetőséget, amelyeket kihasználva javíthatunk a programunk teljesítményén. Ezek egy része az MPI szabvány rutinjainak grides környezethez igazított változata, míg mások új, az MPI szabványban nem rögzített műveleteket definiálnak.

Ez utóbbiak használatakor, a program forráskódja kizárólag az MPICH-G2 implementációval fordítható le, azaz a program forrás szinten sem lesz hordozható különböző MPI implementációk között.

3.2.3.1. MPICH-G2 MPI felett

Az MPICH-G2 a Globus Toolkit Globus I/O programozói interfészét (API) használva, TCP hálózaton keresztül bonyolítja le az MPI folyamatok közötti kommunikációt. Képzeljük el, mennyire nem hatékony ez például közös memóriás gépen (pl. SGI Origin 2000) futó MPI folyamatok számára. Ilyen esetekben az adott architektúra gyártójának MPI implementációja ki tudja aknázni a hardverben rejlő lehetőségeket, s hasznos lenne, ha erre az MPICH-G2 is képes lenne. Bár önmaga erre nem képes, azért ad egy megoldást a problémára.

Az előbbi hatékonysági problémára megoldást jelent az, hogy az MPICH-G2 képes a gyártók által szállított MPI implementációk használatára. Az 1.2.5.2 verziótól kezdve már nemcsak a gyártók által szállított implementációkra, hanem magára az MPICH csomagra is lehet építeni az MPICH-G2-t. Ehhez azonban újra kell fordítani a Globus Toolkit-et úgy, hogy az MPI használatára is képes legyen. Ezt úgynevezett MPI *flavor* módban tehetjük meg, de erről részletesen információt ad az MPICH-G2 honlapja [36].



15. ábra Az MPICH-G2 Globus feletti kommunikációja

A gyártó által szállított MPI implementáció használatakor az azonos gépen lévő MPI folyamatok egymás között ezt az optimalizált implementációt használják a kommunikációra. A távoli gépeken futó folyamatokkal viszont továbbra is csak a TCP feletti Globus I/O üzenetekkel kommunikálhatnak.

Az MPICH-G2 programok futtatásakor, ha egy erőforrás az MPI használatára felkészített Globus verzióval rendelkezik, akkor azt az RSL leírásban jelezni kell a *(jobtype=mpi)* attribútummal. Azokon az erőforrásokon, ahol ez jelezve van, a Globus

GRAM komponense a gyártó által szállított MPI implementáció ***mpirun*** parancsával indítja el a programot.

Ennek a lehetőségnek a kihasználása nem jelent változást a program forráskódjában.

3.2.3.2. Párhuzamos TCP adatfolyamok pont-pont TCP csatornák felett

Az MPICH-G2 következő lehetőségével olyan alkalmazások esetében érhető el szignifikáns teljesítménynövekedés, amelyek nagy mennyiségű adatot mozgatnak két pont között. Az adatmozgatáshoz az MPICH-G2 a Globus GridFTP komponense által nyújtott párhuzamos TCP adatfolyamokat használja fel. A küldő fél a küldendő adatokat transzparens módon több csomagra bontja, majd ezeket a párhuzamos adatcsatornákon egyszerre továbbítja a fogadóknak. A fogadó oldalon szintén transzparens módon történik a csomagok összeillesztése.

A lehetőség kihasználásához a forráskódban kérni kell az MPICH-G2-t a kapcsolat kiépítésére. A következő példán keresztül elmagyarázom, hogy hogyan tudjuk ezt használni. A kapcsolat kiépítéséhez egy ***gridftp_params*** nevű struktúra három paraméterét kell beállítanunk. Az első a kommunikációra használt MPI kommunikátorban a célfolyamatnak a száma (***partner_rank***), a második a párhuzamos adatfolyamok száma (***nsocket_pairs***), végül a harmadik a csomagok mérete (***tcp_buffsize***). Ha beállítottuk a paramétereket, akkor már csak a kommunikátornak kell jelezni szándékunkat. Ezt egy attribútum beállítás formájában tehetjük meg az ***MPI_Attr_put()*** rutinnal. Paraméterként át kell adni a feltöltött gridftp_params struktúrát és a MPICHX_PARALLELSOCKETS_PARAMETERS konstanst. A küldő és fogadó közötti kommunikáció a felépített párhuzamos adatfolyamon transzparens módon zajlik az MPI kommunikációs primitívjeivel.

Példaprogram párhuzamos adatfolyamok használatára:

```
#include <mpi.h>

int main(int argc, char **argv)
{
   int numprocs, my_id;
   struct gridftp_params gfp;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

   if (my_id == 0 || my_id == 1)
   {
       gfp.partner_rank  = (my_id ? 0 : 1);
       gfp.nsocket_pairs = 64;
       gfp.tcp_buffsize  = 256*1024;
       MPI_Attr_put(MPI_COMM_WORLD,
           MPICHX_PARALLELSOCKETS_PARAMETERS, &gfp);
   }
```

```
        /* Mostantól az MPI_COMM_WORLD kommunikátor 0 és 1 számú
         * folyamata között a kommunikáció párhuzamos adatcsatornákon
         * folyik, az üzenetek automatikus feldarabolásával és
         * összeillesztésével */

        MPI_Finalize();
    }
```

Mivel a párhuzamos adatfolyamok használatához a forráskódba MPICH-G2 specifikus részek kerültek (pl. *gridftp_params*), ezért ez a program nem lesz hordozható a különböző MPI implementációk között.
Ennek a lehetőségnek másik hátránya, hogy nem használható a Globus Toolkit többszálú (*threaded*) változataival.

3.2.3.3. A kollektív műveletek topológiafüggő működése

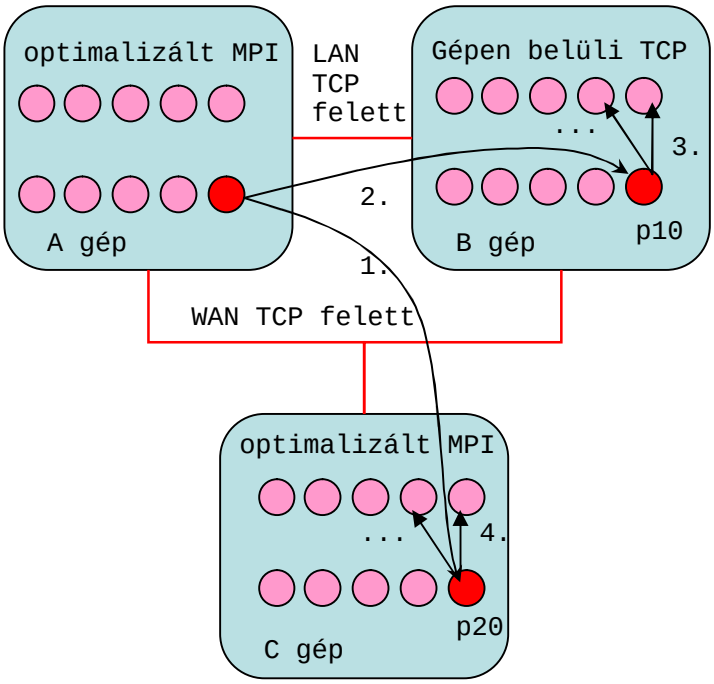
Az MPI kollektív műveletei lehetőséget adnak folyamatok közötti szimultán kommunikációra. A kommunikációban érintett folyamatok a műveletnek paraméterként átadott kommunikátor tagjai. A kollektív kommunikációs lehetőségek az általuk végzett művelet szerint három csoportba sorolhatóak. Az első csoportban azok a műveletek vannak, amelyeknél egy gyökér folyamat adatot küld a többi folyamatnak. Ide tartozik például az adatszóró *MPI_Bcast*, és az adatok szétosztását végző *MPI_Scatter*. A második csoportba olyan műveletek tartoznak, ahol egy gyökér folyamat a csoport többi tagjától adatot gyűjt össze. Ilyen művelet például az *MPI_Gather*. A harmadik csoportba azok a műveletek tartoznak, ahol mindegyik folyamat minden más folyamattal kommunikál. Erre példa az *MPI_Allgather* és az *MPI_Alltoall* művelet.

A kollektív kommunikációs műveletek a csoport minden tagjára vonatkozó szinkronizációt hajtanak végre. A program futási idejének szempontjából nem mindegy, hogy a szinkronizációt a rendszer mennyi idő alatt végzi el, azaz a folyamatoknak mennyi időt kell várnia a szinronizációra. Tegyük fel, hogy a rendszer a szinkronizációs kérést egyszerre küldi el az összes folyamatnak. Ekkor egy távoli hálózat gépén lévő folyamat sokkal később kapja meg a kérést, mint a gyökérfolyamattal azonos gépen lévő folyamat. Ez azt jelenti, hogy a gyökérfolyamathoz közel levő folyamatoknak sokat kell várakozniuk, azzal pedig nem javítják a rendszer teljesítményét.
Valójában az MPICH esetében a folyamatokból épített binomiális fák mentén történik a szinkronizációs kérések továbbítása. Ez a módszer grides környezetben nem túl hatékony, hiszen a folyamatok között nagy távolságok lehetnek.

Az MPICH-G2 a fenti problémára topológiafüggő kollektív műveleteket használ. A jelenleg támogatott műveletek a következők:
MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan, MPI_Barrier, MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Reduce.

Mint az a 14. ábrán is látszik, az MPICH-G2 folyamatok közötti kommunikáció négy csoportba sorolható. Az első a távoli gépek közötti, nagy kiterjedésű hálózatokon (WAN) folyó TCP kommunikáció. A második csoport a helyi hálózaton, TCP protokoll felett folyó kommunikáció, a harmadik az egy gépen belüli TCP protokollal történő kommunikáció. A negyedik csoport az egy gépen lévő folyamatok közötti, optimalizált MPI rutinkönyvtárak által biztosított kommunikáció. Az első csoport a leglassabb, míg az utolsó a leggyorsabb lehetőség.

Ezeket a csoportokat figyelembe véve topológiafüggő kollektív kommunikáció valósítható meg MPICH-G2 folyamatok között.
Ennek működésének megértéséhez vegyük a következő példát.
Tegyük fel, hogy van három gépünk A, B és C, gépenként 10 MPI folyamattal, rendre P_0,...,P_9, majd P_{10},...,P_{19}, végül P_{20},...,P_{29}.
Tegyük fel, hogy az A és C gépeken van optimalizált MPI változat. Legyen az A és B gép egy lokális hálózaton, a C gép pedig egy távoli hálózaton.
A rendszert a 16. ábra szemlélteti. Most lássuk hogyan is folyik le egy üzenetszóró *MPI_Bcast* által kezdeményezett szinkronizáció, ha az üzenetszórás gyökérfolyamata a P_0 folyamat.



16. ábra Kollektív kommunikáció az MPICH-G2-ben

Az üzenetszórás több a folyamatok egy részhalmazán értelmezett üzenetszórás végeredményeként fog megvalósulni.
Az első lépés szerint el kell juttatni az üzenetet a távoli hálózatok gépeire. Azért először ide, mert ezek a leglassabbak. Ennek végrehajtásához a gyökér folyamat és a távoli hálózatokon lévő gépek egy-egy folyamata közötti üzenetszórás indul meg. A példában ezt a {P_0, P_{20}} halmaz feletti P_0 gyökerű üzenetszórás valósítja meg.
A következő lépés a lokális hálózatokban történő üzenetszórás. A példában ezt a {P_0, P_{10}} halmazon értelmezett üzenetszórás végzi el.
A lokális hálózat szintje után a helyi gépen, TCP feletti kommunikációval történő adattovábbítás következik. A példában a {P_{10},...,P_{19}} halmaz feletti, P_{10} gyökerű üzenetszórás csinálja. A P_{10} folyamat a lokális hálózaton történt üzenetszóráskor már megkapta az adatokat, így tovább tudja küldeni őket a többieknek.
A legutoljára a gyors, optimalizált MPI hívások segítségével történő üzenetszórás következik. Ezt a példa alapján a {P_0,...,P_9} és {P_{20},...,P_{29}} üzenetszórások valósítják meg.

Az MPICH-G2 a topológia kihasználásához szükséges információt az RSL leírásból szedi össze. Ilyen információ, hogy egy adott gépen van-e optimalizált MPI változat, vagy hogy két gép azonos lokális hálózaton van-e.
Az elsőt az RSL leírás (jobtype=mpi) attribútuma alapján dönti el az MPICH-G2. A másodiknál pedig egy környezeti változó, a *GLOBUS_LAN_ID* definiálásával adhatjuk meg ugyanazon a lokális hálózaton levő gépeket. A módszer segítségével több LAN is leírható, csak annyira kell ügyelni, hogy az azonos lokális hálózaton levő gépeknél a *GLOBUS_LAN_ID* környezeti változót azonos értékkel legyen definiálva.
Példaként egy olyan RSL fájlt mutatok, amelyben két gép lokális hálózattal van összekötve, és az egyik tartalmaz optimalizált MPI változatot.

```
+
( &(resourceManagerContact="repa.local")
    (count=1)
    (label="subjob 0")
    (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
        (GLOBUS_LAN_ID lan1))
    (directory=/home/farkas)
    (executable=/home/farkas/mandel_mpi)
)
( &(resourceManagerContact="oldblackjoe.local")
    (count=10)
    (jobtype=mpi)
    (label="subjob 1")
    (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
        (GLOBUS_LAN_ID lan1))
    (directory=/home/farkas)
    (executable=/home/farkas/mandel_mpi)
)
```

3.2.3.4. A topológia felderítésének lehetősége

A topológiafüggő kollektív kommunikációs lehetőségek kihasználják az RSL leírás ilyen jellegű információit. Néha azonban az alkalmazásnak is szüksége lehet a topológia ismeretére. Például egy sok folyamatból álló, mester – szolga típusú MPI program esetén megvalósítható, hogy az indulás pillanatában minden folyamat azonos, s ezek a folyamatok csak a közöttük lévő hálózat topológiáját lekérdezve, egy közös döntés alapján jelölnek ki mestert. Értelemszerűen mesternek azt a folyamatot célszerű választani, amelyik a lehető legtöbb folyamathoz van közel (azonos gépen, esetleg lokális hálózattal összekötött szomszédos gépeken vannak). Ha egy olyan folyamatot választanánk, amelyik a többitől nagyon távol helyezkedik el, akkor a mester és a szolgák közötti kommunikáció lelassítaná a program futását.

Az MPICH-G2 bevezeti a topológia mélységének (*topology depth*) fogalmát, amelyet az MPI folyamatok közötti kommunikáció típusa alapján definiál. Mint azt már a topológiafüggő kollektív kommunikációs lehetőségeknél említettem, az MPICH-G2 esetében négyféle módon kommunikálhatnak az MPI folyamatok egymás között. Ennek a csoportosításnak megfelelően az MPICH-G2 bevezeti a lvl_0*=WAN-TCP,* lvl_1*=LAN-TCP,* lvl_2*=gépen belüli TCP* és lvl_3*=optimalizált MPI* kommunikációs szinteket.
Bármely két folyamat kommunikálni tud egymással lvl_0*=WAN-TCP* szinten. Két folyamat kommunikálni tud lvl_1*=LAN-TCP* szinten pontosan akkor, ha ugyanabban a lokális hálózaban vannak, és ez jelezve van az RSL leírásban a *GLOBUS_LAN_ID*

környezeti változó azonos értékkel való definiálásával. Kommunikálni tudnak *lvl_2=gépen belüli TCP* szinten pontosan akkor, ha ugyanabban az RSL részfeladatban vannak, *lvl_3=optimalizált MPI* szintjén akkor és csak akkor, ha ugyanabban az RSL részfeladatban vannak és a részfeladatban fel van tüntetve a *(jobtype=mpi)* attribútum.
Azoknak az MPI folyamatoknál, amelyek csak TCP protokoll felett képesek kommunikálni, a topológia mélysége 3, mert ezt háromféleképpen tehetik (*lvl_0*, *lvl_1*, *lvl_2* és *lvl_3*). Azoknál a folyamatoknál, amelyek képesek a gyártók által szállított optimalizált MPI használatára, a topológia mélysége 4, mert ezek az előző három lehetőségen kívül *lvl_4* szinten is képesek kommunikálni.
A topológia mélységét egy a kommunikátorhoz rendelt új attribútum, az *MPICHX_TOPOLOGY_DEPTHS* lekérdezésével kaphatjuk meg. Az attribútum lekérdezése a kommunikátor minden folyamatában azonos vektorral tér vissza, melynek hossza pontosan a kommunikátor mérete. A vektor i-edik eleme a kommunikátor i rangú folyamatához tartozik.
Példa a topológia mélységének lekérdezésére:

```
int *depths;
int flag, rv;

   …

rv = MPI_Attr_get(MPI_COMM_WORLD,
                  MPICHX_TOPOLOGY_DEPTHS,
                  &depths,   &flag);
if ( rv != MPI_SUCCESS )
{
      /* A topológia mélysége nem lekérdezhető */
      /* Valószínüleg nem MPICH-G2 változatot használ */
      MPI_Abort(MPI_COMM_WORLD, 1);
}
if ( flag == 0 )
{
      /* A topológia mélység információja nem elérhető */
      MPI_Abort(MPI_COMM_WORLD, 1);
}
```

A topológia mélysége azonban csak a folyamatok kommunikációs lehetőségeit jelzi. Néha azonban szükség lenne arra az információra is, hogy két folyamat legfeljebb milyen szinten tud egymással kommunikálni. Ez ugye mindkét folyamat topológia mélységétől függ. Az MPICH-G2 ennek lekérdezésére is lehetőséget biztosít, ugyancsak egy kommunikátor attribútum formájában. Ehhez bevezeti a topológia szín fogalmát, amelyet egy nem negatív egész szám reprezentál.
Két folyamat bármely kommunikációs szinten ugyanolyan színnel rendelkezik pontosan akkor, ha kommunikálni tudnak az adott kommunikációs szinten. Ebből már látszik, hogy a topológia szín magától a folyamattól és a kommunikációs szinttől is függ.
Az MPI folyamatok a topológia színét az *MPICHX_TOPOLOGY_COLORS* attribútum lekérdezésével kaphatják meg. Ez egy vektort ad vissza, amelynek hossza a mélységhez hasonlóan a kommunikátor méretével egyezik. Az i-edik elem itt is az i rangú folyamathoz tartozó információt jelent, itt azonban ezek az információk is vektorokkal vannak reprezentálva. Ezek a vektorok rendre a folyamat kommunikációs szintjeihez tartozó színeket tartalmazzák, ezért hosszuk értelemszerűen az adott folyamat topológia mélységével egyezik.
A topológia szín lekérdezése, a kommunikátor bármely folyamatában is történik, mindig ugyanazt az információt adja vissza.

Példa a topológia színének lekérdezésére:

```
int **colors;
int flag, rv;

   …

rv = MPI_Attr_get(MPI_COMM_WORLD,
                  MPICHX_TOPOLOGY_COLORS,
                  &colors, &flag);
if ( rv != MPI_SUCCESS )
{
      /* A topológia színe nem lekérdezhető */
      /* Valószínüleg nem MPICH-G2 változatot használ */
      MPI_Abort(MPI_COMM_WORLD, 1);
}
if ( flag == 0 )
{
      /* A topológia szín információja nem elérhető */
      MPI_Abort(MPI_COMM_WORLD, 1);
}
```

A fenti két lehetőség kihasználásával olyan MPI programokat készíthetünk, amelyek figyelembe vehetik a folyamatok közötti kommunikációs szinteket, és ezzel gyorsabbá tehetik a kommunikációt és a program futását.
Felépíthetnek például egy új kommunikátort úgy, hogy abba csak egymáshoz viszonylag közeli, lokális hálózattal összekötött gépek folyamatai legyenek, s ebben az új kommunikátorban végzik el a kommunikációigényes feladatokat.
Az alábbi kódrészletet a program minden folyamatában meghívva olyan kommunikátort hozhatunk létre, amelyben az adott folyamattal egy lokális hálózatba eső folyamatok tartoznak.

```
int me;
MPI_Comm LANcomm;
MPI_Comm_rank(MPI_COMM_WORLD, &me);
      …
MPI_Comm_split(MPI_COMM_WORLD,
            colors[me][1],
            0,
            &LANcomm);
```

A fenti csak egy egyszerű példa volt a lehetőség kihasználására. A grid hatékony használatához azonban ennél bonyolultabb módszerek szükségesek. A program futási idejének növekedése azonban megéri a fáradtságot.

A topológia felderítési lehetőség csak C és C++ nyelvek használata esetén érhető el, az MPICH-G2 Fortran nyelvű interfésze egyelőre ezt nem támogatja.

3.2.3.5. A kliens – szerver alkalmazások támogatása

Az MPICH-G2-ben a fejlesztők implementálták az MPI-2 szabvány kliens–szerver alkalmazások készítését támogató utasításait. Ezekkel lehetőség nyílik két, egymástól független MPI program között a kommunikációra. Ehhez a szerven az *MPI_Open_port(), MPI_Close_por()t, és MPI_Comm_accep()t* rutinok használatára, a kliensnél pedig az *MPI_Comm_connet()* rutin használatára van lehetőség.

A kapcsolat felépítéséhez a szerver oldalon meg kell nyitni egy kommunikációs portot, az *MPI_Open_port()* rutinnal. Eredményként a lefoglalt portnevet (*portname*) kapjuk vissza. A kliensek erre a portnévre hivatkozva találhatják meg a szervert.

```
char port_name[MPI_MAX_PORT_NAME];
MPI_Open_port(MPI_INFO_NULL, portname);
```

A megnyitott porton keresztül csak a kliens és a szerver szinkronizációja után lehetséges a kommunikáció. A szinkronizációt a szerver oldalon az *MPI_Comm_accept()* meghívásával kezdeményezhetjük.

```
MPI_Comm newcomm;
MPI_Comm_accept(portname, MPI_INFO_NULL, 0,
                    MPI_COMM_WORLD, &newcomm);
```

Ugyanezt a kliens oldalon az *MPI_Comm_connect()* meghívásával tehetjük meg. Itt paraméterként át kell adni a szerver oldalon, a port megnyitásakor kapott portnevet.

```
MPI_Comm newcomm;
MPI_Comm_connect(portname,   MPI_INFO_NULL, 0,
                  MPI_COMM_WORLD, &newcomm);
```

Mind a szerver, mind a kliens oldalán ezek a szinkronizációs rutinok kollektívek a hívó kommunikátoron (a példában ez az MPI_COMM_WORLD paraméter), azaz a hívó kommunikátor minden folyamata blokkolódik a szinkronizációs utasításnál, amíg azt minden folyamat meg nem hívja. Ha a klienseket és a szervereket sikerült szinkronizálni, akkor a rutinok egy új kommunikátort hoznak létre (*newcomm*). Az új interkommunikátor segítségével lehetséges a két MPI program folyamatai között a kommunikáció.
A szerver oldalon a port lezárását az *MPI_Close_port()* rutin végzi el.

```
MPI_Close_port(portname);
```

Az itt felsorolt lehetőségekkel olyan kliens–szerver alkalmazásokat készíthetünk, amelyeknél a kliens és a szerver is párhuzamos program.

3.2.3.6. További finomhangolási lehetőségek

Itt két olyan beállítási lehetőséget mutatok, amelyek a használata a legtöbb esetben nem indokolt, viszont néha elengedhetetlen, vagy csak szimplán hasznos.

Ha az MPI folyamatot futtató gép több hálózati interfésszel is rendelkezik, akkor az RSL leírásban lehetőség van a kommunikációhoz használt hálózati interfész kiválasztására. Ezt az *MPICH_GLOBUS2_USE_NETWORK_INTERFACE* környezeti változó definiálásával tehetjük meg. Például a következő alakokban:

```
(MPICH_GLOBUS2_USE_NETWORK_INTERFACE 140.221.8.120)
(MPICH_GLOBUS2_USE_NETWORK_INTERFACE 140.221.8.0/255.255.255.0)
```

Az RSL leírásban lehetőség van a TCP protokoll feletti kommunikációnál, a TCP puffer méretének beállítására is, az *(MPICH_GLOBUS2_TCP_BUFFER_SIZE nbytes)* környezeti változót megfelelő értékkel való definiálásával.

Ezt a funkciót olyan esetekben érdemes használni, ahol az MPI program futási sebessége nagy mértékben függ a kommunikációnál használt csomagmérettől.

3.2.4. Probléma a tűzfalakkal

A Globus Toolkit használatánál kisebb gondot okoznak a tűzfalak, mert megnehezítik a mögöttük lévő erőforrások elérését. Ilyen esetekben a rendszergazdáknak a tűzfal bizonyos tartományait meg kell nyitniuk. Ennek részletes leírása a Globus Toolkit tűzfalak használatát leíró dokumentációjában [44] található.

Ha az MPICH-G2 programunkat olyan gépeken szeretnénk futtatni, amelyek között egy tűzfal található, akkor az az MPI programok közötti kommunikációt is megnehezíti.

A Globus miatt a tűzfalon található rést az MPI programok is ki tudják használni, ha ismerik a pontos tartományt.

Az RSL leírás szerencsére a környezeti változók beállításával lehetőséget biztosít ennek az információknak az átadására. Semmi mást nem kell tenni, mint a *(GLOBUS_TCP_PORT_RANGE "min max")* attribútumot a környezeti változók közé beírni. Az MPI program ennek hatására ebbe a tartományba eső portokat használ a folyamatok közötti kommunikációra.

3.3. MPI programok futtatása a Condor segítségével

A Condor rendszerben lehetőség van MPI programok futtatására az MPI univerzumban. Az ilyen programoknak a futtatását a Condor a pool külön erre a célra dedikált gépein végzi el. A jelenlegi Condor verzió kizárólag az MPICH implementáció [28] ch_p4 eszközét támogatja, csak ez lehet a dedikált gépek MPI futtatókörnyezete.

Ezen megszorítások ellenére, van néhány eset, amelynél érdemes Condort használni. Ilyen például az, amikor a programot több bemeneti paraméterrel kell lefuttatni. Hasznos lehet a Condornak azon tulajdonsága is, hogy képes a fájlokat mozgatni a Condor pool-on bellül. Ezzel megoldhatjuk a különböző gépeken futó MPI program kimeneti fájljainak összegyűjtését.
A következő példa MPI program futtatásához használt feladatleíró fájlra:

```
universe = MPI
executable = calculatesum_mpi
log = logfile.out
input = stdin.$(NODE)
output = stdout.$(NODE)
error = stderr.$(NODE)
machine_count = 3
should_transfer_files = yes
when_to_transfer_output = on_exit
queue
```

A fenti példa 3 dedikált gépen futtatja le a megadott programot egy-egy példányban. Minden példány különböző standard bemenetet kap (stdin.0, stdin.1, stdin.2) és különböző standard kimenetet (stdout.[0,1,2]) illetve hibát generál (stderr.[0,1,2]). A bemeneti és kimeneti állományokat a Condor rendszer mozgatja a feladatot benyújtó gép és a futtató gépek között.

4. fejezet

A PVM használata az ismertetett rendszerekben

4.1. PVM programok futtatása a Condor rendszerben

A Condor rendszer a PVM univerzummal támogatja a PVM programok végrehajtását. Minden PVM program számára új virtuális gépet hoz létre, majd azt igény szerint bővíteni képes. Bővítéskor az aktuális terheltségtől függően választja ki az új gépet.

A Condor sok tekintetben magasabb szintű szolgáltatást nyújt, mint a PVM, viszont cserébe fel kell áldozni a programok interaktivitását. Condoron keresztül ugyanis nem lehet interaktív alkalmazásokat futtatni. A felhasználónak már a program elindítása előtt tudnia kell, hogy mi lesz annak bemenete, ugyanis az adatokat a futtatás előtt egy állományba kell írnia, majd az állományra a feladatleíró fájlban hivatkoznia kell. A Condor garantálja, hogy a futó program a hivatkozott fájlt fogja standard bemenetének tekinteni, és nem a billentyűzetet. A PVM démonhoz képest ez jelentősnek tűnő megszorítás, de szerencsére nem jelent gondot, ugyanis a PVM alkalmazások által végzett számítások bemenő adatai általában előre ismertek, vagy futás közben generálódnak.

A Condor és a PVM binárisan kompatibilisek, azaz a lefordított programokat a Condor képes lefuttatni, de a Condor nyújtotta lehetőségek teljes kihasználásához módosítani kell az eredeti PVM csomagon és újra lefordítani azt. Ezeknek a programoknak eltérő módon kell létrehozniuk a PVM folyamatokat, mint PVM démon esetén. Ez az eltérés a Condor erőforrás-menedzselési képességéből adódik.

A PVM módosítandó részeinek leírása megtalálható a Condor kézikönyvében, itt csak röviden a PVM programban használt utasítások új szemantikáját ismertetem.
A Condor a PVM univerzumban bevezeti a géposztály (*Machine Class*) fogalmát. A feladatleíró fájlban felsorolt minden architektúra egy-egy új feladatosztályt jelent.
Példa PVM univerzumot használó feladatleíró fájlra

```
# mandelbrot szamolo PVM programot futtatunk
universe = PVM
executable = mandelbrot_srv
output = out_mandel
error = err_mandel


# legalabb 2, legfeljebb 4 Intel alapu Linuxos gep kell
requirements = (OpSys == "LINUX") && (Arch == "INTEL")
machine_count = 2..4
queue

# legfeljebb 4 Intel alapu Solaris 2.6 op.rendszeres gep kell
requirements = (OpSys == "SOLARIS26") && (Arch == "INTEL")
machine_count = 0..4
queue
```

A fenti példában a Linux rendszeres gépek a 0. feladatosztály gépei, míg a Solaris rendszeres gépek az 1. feladatosztályba tartoznak. Minden feladatosztályhoz megadható a feladat lefuttatásához szükséges gépek száma.
A Condoros PVM programok *pvm_addhost()* utasításának első argumentuma a virtuális géphez hozzácsatolandó gép osztályát definiálja. A PVM démonokkal ellentétben ez a hívás itt nem blokkolódik, hanem értesítést küld, amelyet a *pvm_notify()* segítségével kérdezhetünk le (*PvmHostAdd* üzenet).
A csatlakoztatás után az új gépen a *pvm_spawn()* függvénnyel elindíthatunk **egy** új PVM folyamatot. Sajnos csak egy folyamatot indíthatunk, mert a Condor csak így képes figyelni a gépek terheltségét. Ha a *pvm_spawn()* függvényt *PvmTaskArch* paraméterrel hívjuk, akkor gépnév helyett a kívánt géposztályt kell megadni.

4.1.1 A Master-Worker modell

A Condor fejlesztői a rendszerhez készítettek egy PVM-en alapuló absztrakt C++ osztályokból álló modellt [41], amely segítségével lehetőség van hibatűrő elosztott programok készítésére. Habár a modell a PVM-re épül, azonban annak szemléletétől már távol esik. A benne biztosított hibatűrés átlátszó módon, állapotképek (*checkpoint*) létrehozásával valósul meg. A módszer hátránya, hogy kizárólag a mester-szolga típusú elosztott programokat támogatja.

Egy program megírásához az absztrakt osztályokból származtatással létre kell hozni három osztályt. Ezek a mester folyamatot leíró *MWDriver*, a szolga folyamatot leíró *MWWorker* és a feladat egy szolga által elvégzendő egységét leíró *MWTask* osztályok. Az osztályoknál csak néhány virtuális (*C++ pure virtual*) függvényt kell megvalósítani. Részletesebb információ a modell honlapján [35] található.

A hibatűrés biztosításához a Condor rendszer gondoskodik a mester folyamat állapotmentéseiről, így annak elvesztése esetén képes az előző állapot alapján az újraindításra. Ekkor csak az előző állapot óta elvégzett munka veszik el. A szolgafolyamatok elvesztését a modellben a mester folyamat automatikusan pótolja.

A Master-Worker modell csak a Condor rendszerrel működik együtt, így az ilyen programok egyáltalán nem lesznek hordozhatóak.

4.2. PVM emuláció a Harness rendszerben

A Harness metaszámítógépes rendszerben néhány *plug-in* szolgáltatás betöltése után lehetőség van PVM programok futtatására is. A rendszer lényegének megértéséhez röviden tekintsük át, hogy hogyan is épül fel a PVM rendszer.

4.2.1. A PVM démon működése

A PVM rendszer két részből áll. Az első rész egy démon program, amely a virtuális gép minden számítógépén fut, a második rész egy függvénykönyvtár, amelyben azok a függvények találhatóak, melyek a program PVM démonnal való kapcsolattartását hivatottak elvégezni. A virtuális gép démonjai között van egy kitüntetett szerepű mester démon. Ennek a feladata, hogy a virtuális gép bővítésekor az új gépen egy démont indítson. A démonok a PVM programok üzeneteinek célbajuttatásában is szerepet játszanak. Ha egy PVM folyamat üzenetet küld egy másik folyamatnak, akkor az üzenet a PVM függvénykönyvtár segítségével a folyamatot futtató gép PVM démonjához kerül. A PVM démon a célfolyamatot tartalmazó gép démonjának adja tovább az üzenetet, ahonnan az – szintén a függvénykönyvtár segítségével – a célfolyamathoz továbbítódik. Az üzenetek átadása mellett a démonnak a feladata a PVM folyamatok elindítása is.

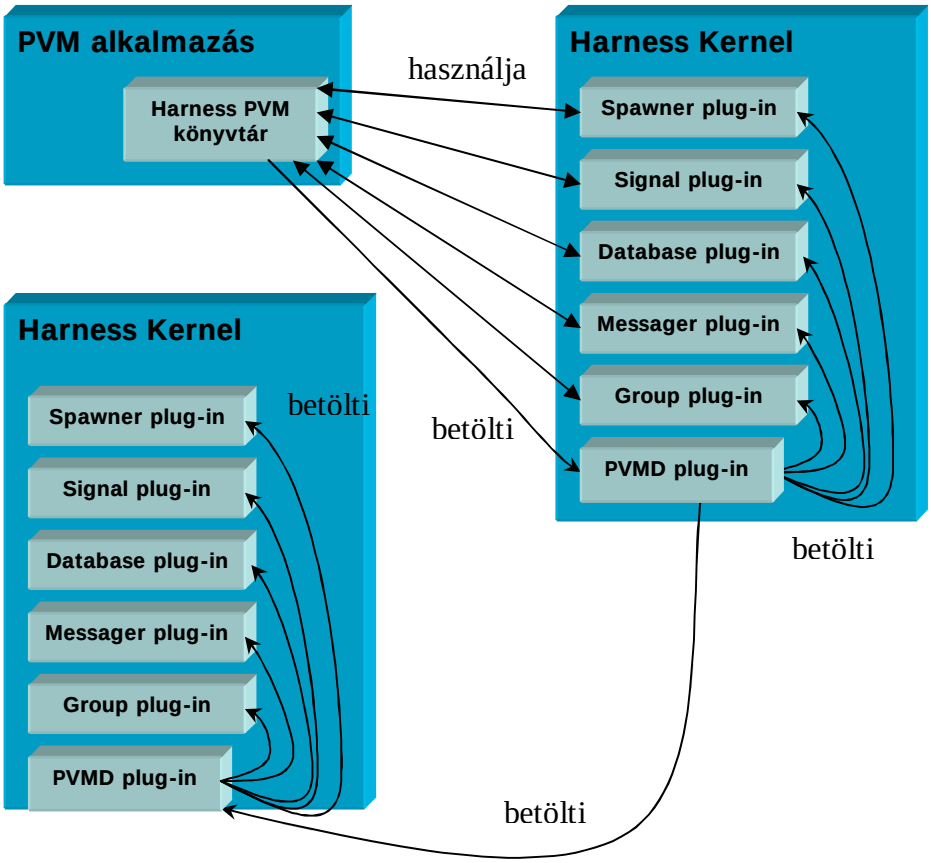
4.2.2. A Harness PVM démonja

A Harness PVM kompatibilitásának tervezésekor ügyeltek arra, hogy meglévő PVM programok Harness környezetben történő futtatása ne követeljen változtatást a programok forráskódjában. Ellentétben a programokkal, magában a PVM rendszerben viszont történt változás. Itt ugyanis a PVM démon szerepének emulációját több Harness szolgáltatás vette át.
A tervezők az eredeti PVM démon által nyújtott szolgáltatásokat öt csoportba oszották: folyamatvezérlés, információkezelés, jelzések kezelése, üzenettovábbító rendszer és csoportműveletek kezelése. A Harnessben ezeket a feladatokat csoportonként egy-egy szolgáltatás látja el. Ezek rendre a *spawner plug-in*, a *database plug-in*, a *signal plug-in*, a *messager plug-in* és a *group plug-in*. Ezeknek a *plug-in*-eknek a betöltéséért egy másik szolgáltatás, a *pvmd plug-in* a felelős (17. ábra). A PVM virtuális gép bővítésekor, *pvm_addhost()* hívás esetén a *pvmd plug-in* elvégzi a távoli gépen a *pvmd plug-in* betöltését is (17. ábra). A távoli gépen betöltött *pvmd plug-in* ezután betölti a többi plug-in-t.
A fenti *plug-in*-ek építenek a Harness architektúra nyújtotta lehetőségekre, kihasználják például a Harness üzenetküldő mechanizmusát.

A rendszer rugalmasságát mutatja, hogy a moduláris felépítés miatt bármelyik feladatcsoporthoz tartozó szolgáltatás lecserélhető egy saját, új lehetőségekkel kiegészített szolgáltatásra. Megoldható például az, hogy menet közben lecseréljük az üzenetátadó rendszert egy titkos kapcsolaton kommunikálni képes változatra.

A Harness jelenlegi verziója a PVM 3.4.3. legtöbb lehetőségét támogatja, a következők kivételével. Még nem képes levelesládák és szignálok kezelésére. Nem lehet vele nyomkövetési információkat összegyűjteni, ezért a Harness-szel nem képes

együttműködni a népszerű XPVM monitorozó rendszer sem. Nem támogatja a pvm_tickle() függvényt. Nincs lehetőség úgynevezett *hoster* és *tasker* folyamatok definiálására. Az előbbi a mester PVM démontól veszi át a démonok indításának feladatát, az utóbbi a folyamatok indítását kezeli. Használatuk elkerülhető saját folyamatvezérlő plug-in definiálásával.
Mindezek ellenére az implementált függvénykészlet elegendő normál tudományos számításokhoz készített alkalmazások futtatására.



17. ábra A Harness PVM démonja

5. fejezet

Összehasonlítás

Ez a fejezet a dolgozatban bemutatott eszközök, nevezetesen a Harness PVM emulációja, a Condor PVM univerzuma, az FT-MPI, a Condor MPI és az MPICH-G2 néhány elosztott rendszerekben fontos szempont szerinti vizsgálatát és összevetését tartalmazza.

Ezek a szempontok a következők. Az első a heterogenitás támogatása, a második a hordozhatóság és az eredeti rendszerrel való kompatibilitás. A harmadik a teljesítmény összevetése, a negyedik a biztonság kérdése, végül az ötödik a hibatűrés támogatása.

5.1. Heterogenitás

A nagyméretű, heterogén rendszerekben nincs meg az egységes rendszerszemlélet, azaz a futó alkalmazások nem feltételezik, hogy mindenütt ugyanolyan erőforrások vannak. Az ilyen heterogén rendszereken alapuló metaszámítógépek és grid rendszerek akkor igazán hasznosak, ha az alkalmazások számára olyan szoftverréteget biztosítanak, amelyek ezeket a különbségeket elrejtik.

A Harness rendszer ilyen szoftverréteget biztosít a hozzá készített alkalmazások számára. Valójában a heterogenitása a Java technológiának köszönhető, ugyanis egy szinte mindenhol elérhető Java virtuális gép (manapság már a mobiltelefonokban is van egy korlátozott változat) elég a Harness futtatásához. Ezt a heterogenitást a Harness rendszer Java nyelvű programozásával ki lehet használni.

A PVM rendszer támogatásához a Harness felhasználja az eredeti PVM függvénykönyvtárat, ezért azoknak a gépek, amelyek a Harnessel PVM programok futtatására képesek, pontosan azok a gépek, amelyeken az eredeti PVM könyvtár is elérhető, hiszen a program lefordításához az eredeti függvénykönyvtár szükséges. Szerencsére ezen a téren maga a PVM is elég nagy lehetőségekkel bír.

A Condor rendszer PVM futtatási lehetősége sem hoz újdonságot a heterogenitás terén, hiszen az is az eredeti PVM függvénykönyvtártól függ.

Az MPI programok futtatása a Condorral elég kötött, hiszen csak arra dedikált gépeken, kizárólag az MPICH implementáció *chp4* eszközével lehetséges. Az MPICH ezen eszközénél, az eredeti MPI szabvánnyal ellentétes módon, lehetőség van MPMD (Multiple Procedure Multiple Data) alkalmazások futtatására is, egy úgynevezett *procgroup* fájl megadásával. Ezzel a lehetőséggel élve egy program, különböző architektúrákhoz lefordított változatait egyszerre futtathatjuk, azaz egyszerre több architektúrán futtathatjuk a programot. A *procgroup* fájl használatának módját az MPICH chp4 eszközének felhasználói könyve [45] részletezi.

A *procgoup* fájlt az mpirun parancs megfelelő paraméterezésével adhatjuk át az MPICH rendszernek. Sajnos erre a Condor nem képes, ezért ezt a lehetőséget a Condorban mellőzni kell.

Az FT-MPI-ról is elmondható, hogy csak olyan környezetben használható, ahol a C nyelvű forráskódja lefordítható. Ezek a környezetek tipikusan a Unix jellegű operációs rendszerek és a Win32 környezet.

A heterogén környezet használatakor minden gépen a futtatandó programfájlnak a gép architektúrájára lefordított változatnak kell lennie.

A Globus Toolkit a Harnesshez hasonlóan programozói interfészek (API-k) formájában olyan szoftverréteget biztosít, amely elfedi az alatta elhelyezkedő rendszerek heterogenitását. Ezért az MPICH-G2, köszönhetően a Globus Toolkit nyújtotta alapnak, nagyfokú heterogenitást biztosít.

Összegezve elmondhatjuk, hogy heterogenitás terén a vizsgált PVM és az MPI eszközök nem hoztak semmi újat, az MPICH-G2-t kivéve.

5.2. Hordozhatóság és kompatibilitás

A hordozhatóság és kompatibilitás címszó alatt azt vizsgálom, hogy mennyire képesek a rendszerek a korábban írt PVM illetve MPI programok futtatására.

A Harness PVM emulációja esetén a PVM függvénykönyvtár *–NOUNIXDOM* kapcsolóval fordított változatára van szükség, mert a Harness virtuális gépe csak ilyen PVM-mel lefordított programok futtatására képes. Ezen kívül van néhány megkötés a programokban használható utasításokra nézve. Ezeket a Harness PVM-et leíró részben részleteztem. Az ott felsorolt dolgokat nélkülöző PVM programok lefordításuk után, gond nélkül futtathatóak a Harness PVM-mel.

A Condor esetében van egy nagyon fontos megkötés a PVM programok esetében. Eszerint egy gépen csak egy PVM folyamat indítható. Ha biztosítunk elég gépet a virtuális gép összeállításához (annyit, ahány folyamatot el szeretnénk indítani a programban), akkor bármilyen korábban írt PVM programot futtathatunk.

Az FT-MPI és az MPICH-G2 esetében az MPI szabványa miatt a korábbi szabványnak megfelelő programok gond nélkül futtathatóak. Értelemszerűen az MPICH-G2 és az FT-MPI nyújtotta lehetőségeket kihasználó programok más MPI implementációval nem használhatók.

Összegzésképpen elmondható, hogy a Harness PVM kivételével a korábbi programok megkötés nélkül lefordíthatók és futtathatók.

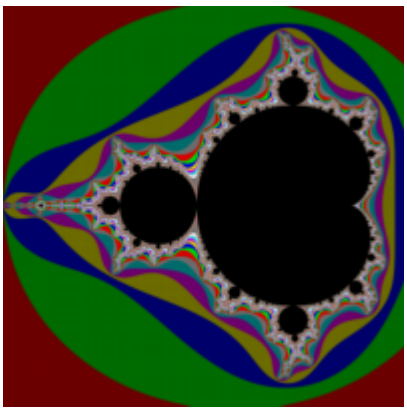
5.3. Teljesítmény

Az ismertetett új PVM és MPI futtatási lehetőségek kiterjesztették az eredeti csomagokhoz készített programok használhatósági körét. A nagyobb rendszereken való futtathatóságot biztosító architektúrák persze a teljesítmény romlását is okozhatják. Ebben a pontban ezért egy példaprogram segítségével megkísérlem összehasonlítani az ismertetett PVM és MPI eszközök teljesítményét.

5.3.2. A teljesítmény összehasonlításához használt program

Az összehasonlításhoz egy Mandelbrot fraktál [46] (Mandelbrot halmaz) számolását végző programot készítettem. A program egy Mandelbrot halmaz 256 színnel ábrázolt képét számolja ki. A tesztelés során a programmal két méretben számoltattam ki a képeket. A nagyobbik kép mérete 16000x16000 képpont, a kisebbiké 8000x8000 képpont volt. A program által generált képfájl PGM formátumú, amely az

első esetben 732 Mb méretű, a második esetben ennek negyede. A képek nagy mérete miatt eltekintettem azok fájlba írásától, így az a futási időbe sem számított bele.



18. ábra Mandelbrot halmaz képe - példaprogram kimenet

A program implementációjakor a mester-szolga elv mellett döntöttem, mert az egyes képpontok kiszámításához nem szükséges más képpontoktól függő információ, ezért a feladat egyszerűen osztható több részre. A feladat felosztásakor a mester folyamat a szolga folyamatoknak egy vagy több sor képpontjainak kiszámítását írja elő. Több sor esetén a sorok egymás alatt vannak, azaz a szolga folyamatnak a kimeneti kép egy téglalap alakú részét kell kiszámolnia. Ha lehetséges, akkor a mester a feladatot azonos méretű blokkokra bontja, ha nem tudja egyenlő részre bontani, akkor az utolsó blokknak (a kép alsó részének) a mérete különbözik a többitől. A feladat felosztása után a mester szétküldi a szolga folyamatoknak a kiszámolandó blokk méretét és pozícióját, akik a számolás végeztével visszaküldik az eredményt.
A szolgák által visszaküldött képet ábrázoló adatcsomag a kiszámolandó kép teljes méretétől, valamint a szolgák számától függően lehet egészen nagy és nagyon pici is. Például 16000x16000 képpontos kép esetén 16 szolgafolyamatnál mindegyik folyamat 15Mb információt küld vissza a mesternek, míg ugyanennél a képnél 512 folyamat esetében ½Mb-nál kisebb adatcsomagok vannak.

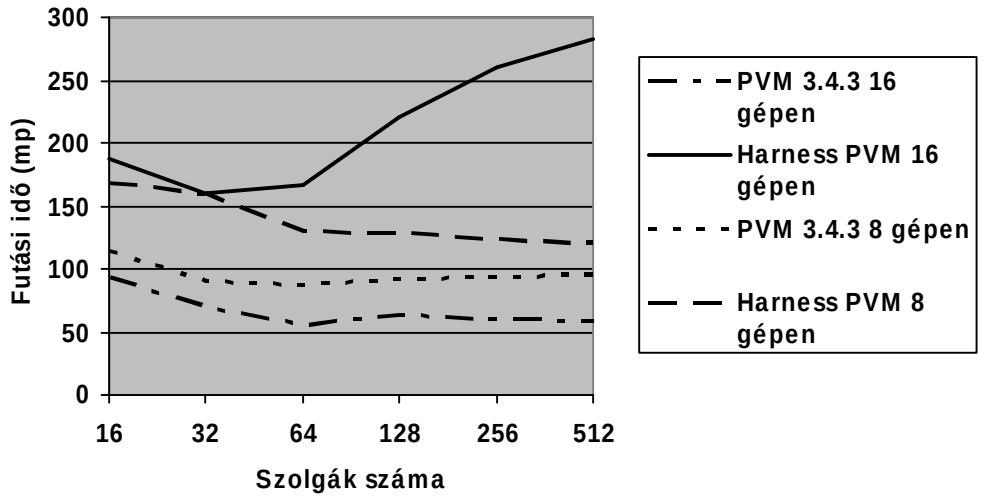
A program futtatását 16, 32, 64, 128, 256 és 512 szolga folyamattal végeztem. Voltak azonban esetek, amikor különböző okok miatt a nagy folyamatszámú tesztfuttatásokat nem sikerült végrehajtani. A tesztfuttatásokat az ELTE IK Programozási Nyelvek és Fordítóprogramok Tanszékének Nyelvi laborjában hajtottam végre.

5.3.2. A PVM implementáció tesztfuttatásai

A PVM implementáció tesztfuttatásait a PVM 3.4.3 és a Harness PVM rendszerében végeztem el. Habár a Condor PVM a nyelvi laborban fel van installálva, annak használatát mégis mellőztem a tesztben. Ennek egyik oka, hogy maga a rendszer nincs pontosan felkonfigurálva, s csak pár Linuxos gépen képes a programok futtatására. Ráadásul előfordul, hogy pont ezeket a gépeket nappal Windows operációs rendszerrel használják, és a használat befejezése után nem indítják újra. Ennél nagyobb probléma, hogy a Condor egy gépen csak egy folyamatot képes elindítani.

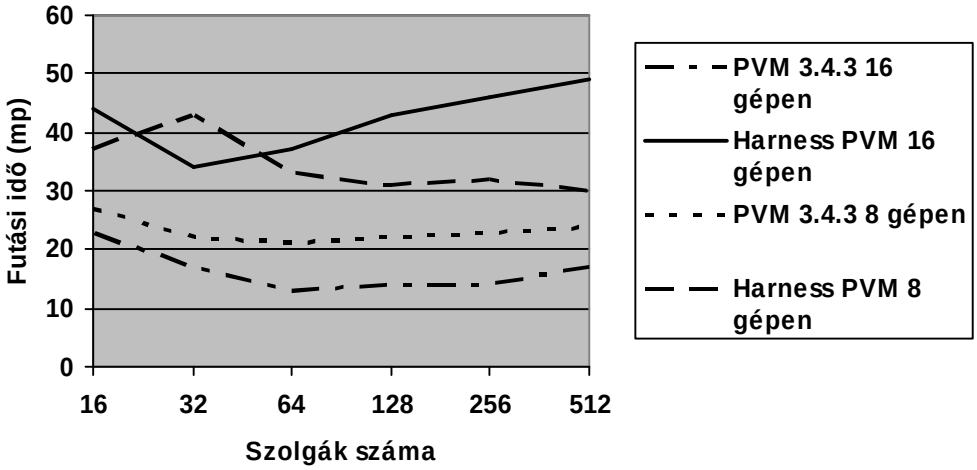
Próbálkozásaim során, a Condor rendszerben legfeljebb 4 gép segítségével sikerült lefuttatni a programot.
Hangsúlyozom, mielőtt az olvasó a Condort egy használhatatlan rendszernek hinné, hogy a fenti problémák egy jól felkonfigurált dedikált klaszter esetében nem jelentkeznek. Ilyenhez azonban nekem sajnos nincs hozzáférési jogom.

A PVM implementáció futási eredményeit a 19. ábra és a 20. ábra tartalmazza. Az első a 16000x16000 képpont méretű kimeneti képhez, a második a 8000x8000 képpont méretű kimeneti képhez tartozik.
Mint várható volt, a Harness architektúra felett futtatott PVM implementáció rosszabb teljesítményt mutatott. Ez szerintem a Java alapú Harness implementációnak köszönhető.
A diagrammokról leolvasható, hogy a kis folyamatszámú, 16 illetve 32 szolgát futtató esetekben a PVM, és a Harness sem volt az optimális futási idő közelében. Ilyen kevés számú szolga esetén egy-egy futtató gépre legfeljebb 4 PVM folyamat kerül, ami nem jelenti a gép erőforrásainak optimális kihasználását, ugyanis a szolga és a mester folyamatok között átmozgatandó nagy adatmennyiség küldésekor a gép processzora kihasználatlanul állhat.
A 16. ábrán feltűnhet a Harness 8 gépes futtatási eredményében, a 32 szolgánál bekövetkező teljesítményromlás. Tapasztalataim szerint ez a PVM folyamatok egyenletes elosztásának hiányából következik. Egyes gépekre több folyamat, míg másokra kevesebb kerül, s ez kis folyamatszám esetében több számítási munkát is jelent. A végeredmény pedig mindig a leglassabb gép számítási ideje lesz.



19. ábra Mandelbrot program futási eredménye 16000x16000 képpont méretű képre

A másik, ami feltűnhet, hogy a Harness-nél nagy folyamatszám esetén, a 16 gépen történő futtatás nem gyorsulást, hanem lassulást okozott. Ennek okai a Harness architektúrájában vannak, ugyanis a Harness architektúrájának elemei Java RMI hívásokat hajtanak végre, amelyek távoli gépek esetén lassabban zajlanak le, mint egy adott gépen belül. Ha a sok folyamatot kevesebb gépre osztjuk szét, akkor nagyobb lesz a gépen belüli RMI hívások aránya a távoli hívásokkal szemben. Ezzel szemben, ha több gépre osztjuk szét a folyamatokat, akkor sok távoli RMI hívás lesz a rendszerben a folyamatok közötti kommunikáció lebonyolításakor.



20. ábra Mandelbrot program futási eredménye 8000x8000 képpont méretű képre

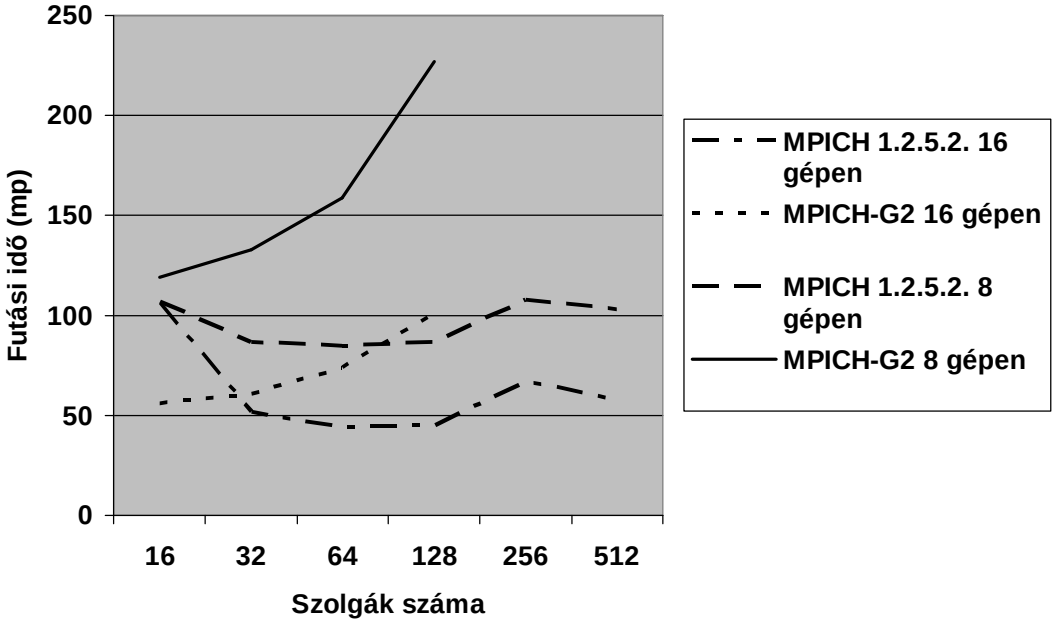
5.3.3. Az MPI implementáció tesztfuttatásai

Az MPI implementáció tesztfuttatásainál az MPICH-G2 és az FT-MPI futási eredményeit vetettem össze az MPICH 1.2.5.2 verziójával.

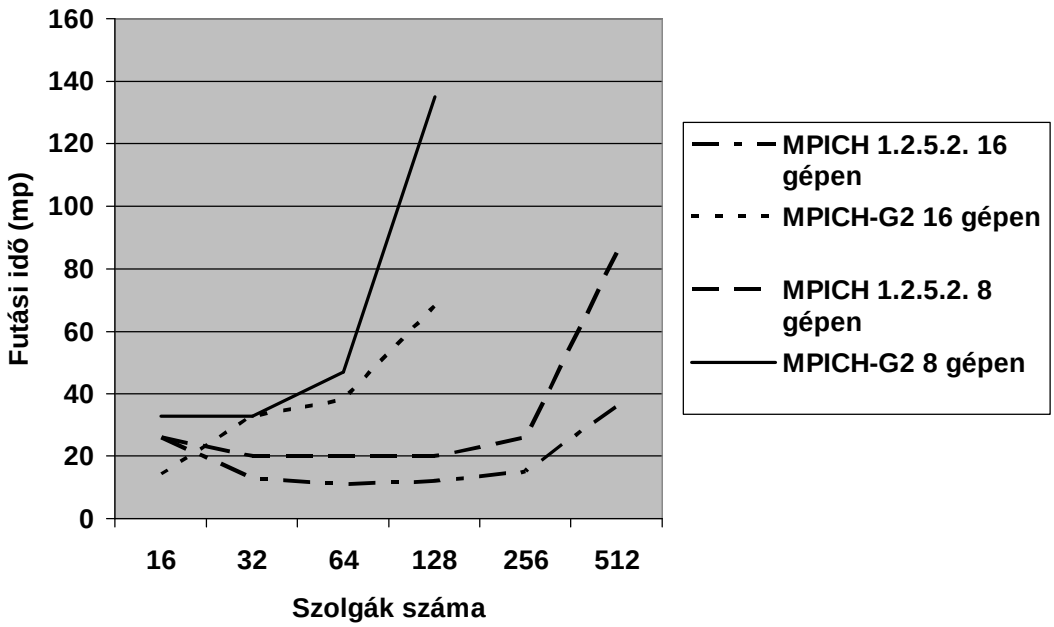
Az MPI implementáció futási eredményeit a 21. ábra, 23. ábra, 22. ábra és 24. ábra tartalmazza. Az első kettő 16000x16000 képpont méretű kimeneti képhez, a második kettő a 8000x8000 képpont méretű kimeneti képhez tartozik.

Mind az FT-MPI, mind az MPICH-G2 rendszernek gondot okozott 128-nál több szolga folyamat elindítása. Ez az MPICH-G2 esetében a következőre vezethető vissza.
A nyelvi laborban a Globus Toolkit 2.4 nincs feltelepítve, ezért a saját jogaimmal voltam kénytelen futtatni a Globus GRAM komponensét. Ebben az esetben a GRAM komponens alatt nem helyezkedett el az egész nyelvi laborban futó folyamatkezelő rendszer, aminek a folyamatok indítása lenne a szerepe. A Globus ilyen esetekben a saját egyszerű feladatindító programját használja, amelynek egyenként kell átadni az elindítandó feladatokat. Ez azt jelentette, hogy az például 128 MPI folyamat elindítása olyan összetett RSL leírást generál, amely 128 részfeladatból áll, s minden részfeladat ugyanarra a programkódra hivatkozik. Ezzel a módszerrel egy MPI program 128 folyamatának elindítása több percet vett igénybe. Ennél több folyamat elindításához egy helyi feladatkezelő rendszer lenne szükséges. Ilyen feladatkezelő rendszert és magát a Globus Toolkitet persze olyan klasztereken célszerű üzemeltetni, ahol a felhasználók időnként nem indítgatják újra a gépeket, ezzel a rajta futó feladatok elvesztését okozva. A felhasználók okozta rendszerterhelés a program lefutását nem veszélyezteti, csak a futási időt növeli.

Az MPICH-G2 esetében a program futási ideje növekszik a szolgák számának növekedésével. Ebből arra következtethetünk, hogy a nagyobb folyamatszám okozta nagyobb kommunikációs terhelés hosszabb futási időhöz vezet a Globus I/O alapú MPICH-G2 programok esetében. Az MPICH-G2 optimalizált MPI feletti használatának lehetőségét sajnos nem tudtam kipróbálni a Nyelvi labor gépein.

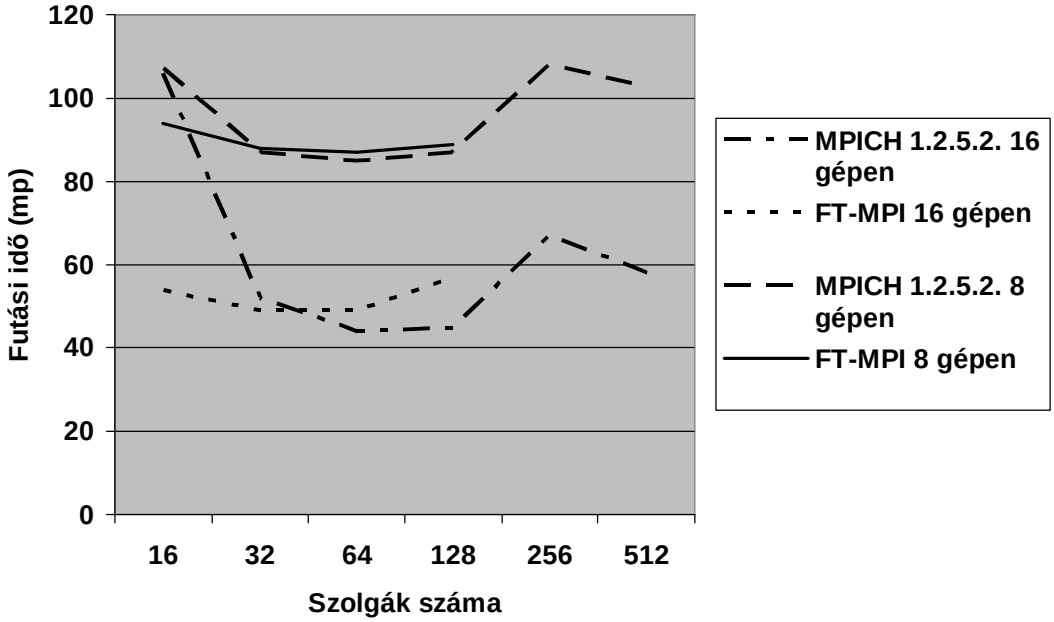


21. ábra Az MPICH és az MPICH-G2 összevetése 16000x16000 méretű kép esetén

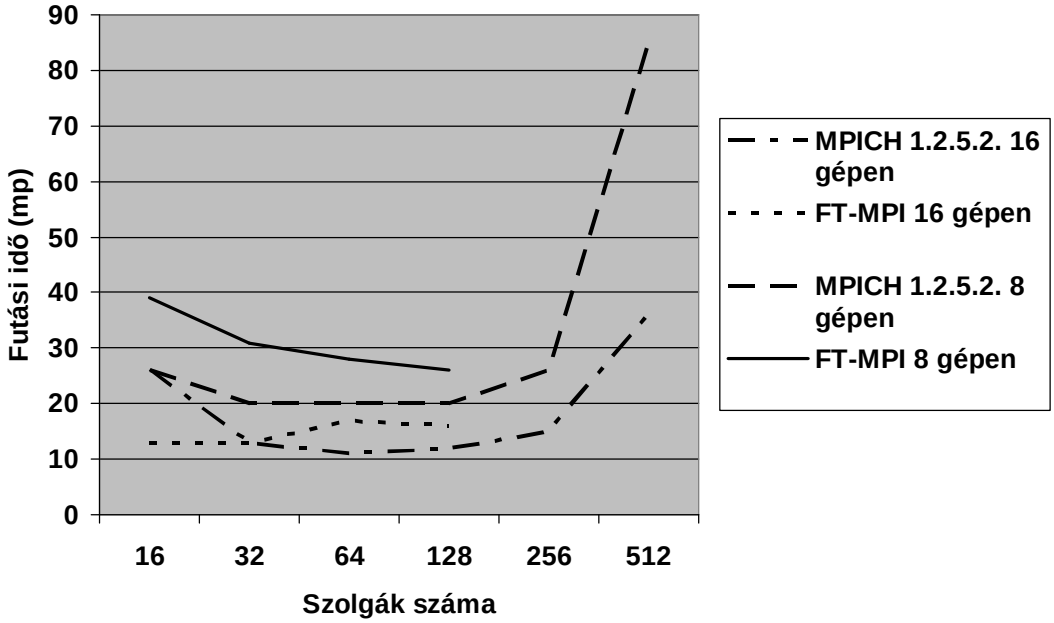


22. ábra Az MPICH és az MPICH-G2 összevetése 8000x8000 méretű kép esetén

Az FT-MPI C alapú Harness architektúrájának sebességét összevetve a Java változattal érthető, hogy a fejlesztők miért határozták el magukat az elkészítésére.
A nagyobbik kép kiszámításánál az FT-MPI teljesítménye közel volt az MPICH 1.2.5.2 teljesítményéhez, kis folyamatszám esetén még gyorsabb is volt nála. Az FT-MPI fejlesztői is végeztek általános adatátviteli sebesség méréseket [40], s ők is igazolták a jó teljesítményt.



23. ábra Az MPICH és az FT-MPI futási eredménye 16000x16000 méretű kép esetén



24. ábra Az MPICH és az FT-MPI futási eredménye 8000x8000 méretű kép esetén

A futási eredményeket megvizsgálva elmondhatjuk, hogy a Harness PVM és az MPICH-G2 nem hozzák a PVM illetve az MPICH nyújtotta teljesítményt, helyette viszont a szélesebb körű használat lehetőségét nyújtják. Az FT-MPI a teljesítmény szempontjából nem marad el az MPICH implementációtól.

5.4. Biztonság

A biztonság az elosztott rendszerek egyik legbonyolultabb alapeleme. A rendszer által biztosított biztonság vizsgálatakor a biztonságos kommunikáció lehetőségét és a felhasználók jogainak biztosítását megvalósító mechanizmust kell megnézni.

A Condor tervezésekor nem volt cél a biztonság biztosítása, ezért az nem vethető össze a Globus GSI nyújtotta lehetőségeket kihasználó MPICH-G2-vel, és a Harness által biztosított lehetőségekkel.

Az MPICH-G2 segítségével készített programok, hála a Globus GSI-nek, biztonságos kapcsolat felett kommunikálnak, ráadásul a grid infrastruktúrának köszönhetően a futtatásukhoz sem kell a lefoglalt gépeken felhasználói szintű jogosultság. Egyetlen aláírt grid bizonyítvány (*grid-certificate*) elegendő, ha az erőforrást biztosító szervezet jogot biztosít nekünk az erőforrás használatára.

A Harnessben is lehetőség van az elosztott virtuális gép felhasználóinak korlátozására. Ezeket a jogokat a virtuális gépek üzemeltetője minden gépen külön-külön állíthatja be. Egy Harness virtuális gép az erőforrások használata szempontjából jobban hasonlít a grid rendszerekhez, mint a metaszámítógépekhez, ugyanis a virtuális gép felhasználóinak csak a virtuális gépbe kell bejelentkezniük, és nem kell minden a virtuális gépet alkotó gépen felhasználói jogosultságokkal rendelkezniük. A futtatandó programot, ami egy Java alkalmazás, a Harness képes a virtuális gép elemei között mozgatni. A Harness PVM esetében a virtuális gépbe történő bejelentkezés azonban nem elegendő, ugyanis a PVM programnak, ami nem Java alkalmazás, a virtuális gép minden gépén helyben elérhetőnek kell lennie, azaz a felhasználónak oda kell másolnia.
A Harness PVM esetében a plug-in mechanizmusnak köszönhetően biztosítható a biztonságos adatkapcsolat iránti igény is. Egyszerűen meg kell írni a Harness kommunikációs plug-in titkosított kommunikációs lehetőségekkel kibővített változatát, és azt elindítani a virtuális gép minden tagján.

Elvileg az FT-MPI esetében is lehetségesek a fent leírtak, de ott a jól bevált SNIPE kommunikációs könyvtárat kellene lecserélni, ami valószínűleg teljesítményromláshoz vezetne. Az FT-MPI nem foglalkozik a felhasználók jogaival sem.

Összegzésképpen azt mondhatjuk el, hogy az MPICH-G2 a Globus Toolkitnek köszönhetően mindkét vizsgált szempontból támogatja a biztonságot, míg a Harness alapú PVM és MPI alapvetően nem támogatja, de megvan a lehetőség annak támogatására.

5.5. Hibatűrés

Ebben a részben azt vizsgálom, hogy az általam ismertetett rendszerek milyen mértékben támogatják hibatűrő programok készítését.

A Condor PVM univerzuma a hagyományos PVM programok, alapvető hibatűrő képességeit támogatja. Ezek szerint a programnak lehetősége van egy folyamat, vagy a virtuális gép egyik hosztjának elvesztésére figyelmeztető üzenet kérésére. Ezt és a PVM dinamikus folyamatindító tulajdonságát kihasználva lehetőség van hibatűrő programok készítésére.

A Harness architektúrájának tervezésekor elsődleges szempont volt a virtuális gép hibatűrő képessége, ezért azt úgy valósították meg, hogy nincs olyan pont a rendszerben, amelynek kiesése esetén a virtuális gép is elveszik.
Ez az architektúra jó alapot biztosít a PVM és MPI programok számára, de ezt eddig a PVM emuláció nem használja ki, mert nem képes figyelmeztető jelzések küldésére, s a PVM függvénykönyvtárban az alapvető hibatűrési funkciók ilyen figyelmeztető jelzések segítségével működnek. Mivel a Harness kísérleti rendszer, ezért előfordulhat, hogy ezeket a funkciókat nem is implementálják.
Az FT-MPI kifejlesztésének célja a hibatűrés volt, ezért ezzel hatékony, hibatűrő programokat készíthetünk.

A Condor által használt MPICH implementáció, és az MPICH-G2 sem alkalmas hibatűrő programok készítésére.

Mindkét hibatűrést támogató rendszerben, a PVM-ben és az FT-MPI-ban a programozó feladata a hibatűrés megvalósítása. Az utóbbi esetén sokkal szélesebb körű hibatűrés-támogatás érhető el.

6. fejezet

Összefoglalás

A dolgozatban bemutattam három elosztott rendszert, amellyel lehetőség nyílik elosztott PVM és MPI programok futtatására. Az elosztott rendszerek ismertetése után megmutattam, hogyan lehetséges ezekben a rendszerekben a programok futtatása. Részleteztem, hogy milyen új lehetőségek jelentek meg, és mik ezeknek az eszközöknek a korlátai.

Az eszközök ismertetése után néhány elosztott rendszerekben fontos szempont szerint összehasonlítottam őket. Ehhez szükség volt egy elosztott program MPI és PVM implementációjának elkészítésére is. Én egy Mandelbrot halmaz képét számoló programot implementáltam mindkét nyelven, majd elvégeztem velük a 6. fejezet méréseit.

Az ilyen rendszerek használata lassan elkerülhetetlenné válik. Miután a Globus Toolkit a grid rendszerek legjelentősebb képviselőjévé fejlődött, egyre több olyan tudományos területen használják, ahol nagy jelentőségű problémákat, párhuzamos programok segítségével vizsgálnak. Példaként az Európai Unió DataGrid projektjét [40] hoznám fel, amely eredetileg a CERN részecskegyorsítójában szerzett temérdek adat elemzésére irányul. Az ehhez hasonló tudományos számításokra irányuló igények azt mutatják, hogy nem hiábavaló erőfeszítések a dolgozatban részletezett, párhuzamos programok grides környezetben történő futtatását biztosító eszközök kifejlesztésére irányuló kísérletek.

Ezeknek az eszközöknek a segítségével, talán már a közeljövőben ugyanazokon az erőforrásokon fogják az ELTE diákjai letesztelni párhuzamos programjaikat, mint amelyeken a fontos tudományos számítások folynak. Ezek az erőforrások a legkülönbözőbb szervezetek gépei – például az ELTE, a SZTAKI és a CERN gépei – valamint az otthonainkban kihasználatlanul álló számítógépek lehetnek. Én remélem, hogy **lesznek** is!

Ábrajegyzék

Irodalomjegyzék

[1] Andrew S. Tanenbaum és Marteen van Steen: Elosztott rendszerek, Panem, 2004, [872], ISBN 963-545-387-6.
[2] OpenMP Architecture Review Board: OpenMP: Simple, Portable, Scalable SMP Programming, 1999, http://www.openmp.org/ (2004-06-10)
[3] Oak Ridge National Lab: PVM: Parallel Virtual Machine, 1994, http://www.epm.ornl.gov/pvm/ (2004-06-10)
[4] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek és Vaidy Sunderam. PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994, [298], http://www.netlib.org/pvm3/book/pvm-book.html (2004-06-10)
[5] MPI Forum: A Message-Passing Interface Standard, 1994, [236], http://www.mpi-forum.org/docs/mpi-10.ps (2004-06-10) http://www.mpi-forum.org/docs/mpi-11.ps (2004-06-10)
[6] MPI Forum: MPI-2: Extensions to the Message-Passing Interface, 1997, [376], http://www.mpi-forum.org/docs/mpi-20.ps (2004-06-10)
[7] Mauro Migliardi, Vaidy Sunderam: The Harness Metacomputing Framework, In Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999. március., [8], http://www.mathcs.emory.edu/harness/pub/general/pp99.ps.gz (2004-06-10)
[8] Harness Metacomputing Framework, http://www.mathcs.emory.edu/harness/ (2004-06-10)
[9] Mauro Migliardi és Vaidy Sunderam: Plug-ins, Layered Services and Behavioral Objects: Application Programming Styles in the Harness Metacomputing System, [15], http://www.mathcs.emory.edu/harness/pub/general/fgcs_shorter.ps.gz (2004-06-10)
[10] Tomasz Tyrakowski, Vaidy Sunderam és Mauro Migliardi: Distributed Name Service in Harness, 2001, [10], http://www.mathcs.emory.edu/harness/pub/hdns/iccs2001e.ps.gz (2004-06-10)
[11] Mauro Migliardi, Vaidy Sunderam és Arrigo Frisiani: A Simple, Fault Tolerant Naming Space for the HARNESS Metacomputing System, [8], http://www.mathcs.emory.edu/harness/pub/general/europvm_cr2.ps.gz (2004-06-10)
[12] Mauro Migliardi és Vaidy Sunderam: Heterogeneous Distributed Virtual Machines in the Harness Metacomputing Framework, [13], http://www.mathcs.emory.edu/harness/pub/general/hcw_cr3.ps.gz (2004-06-10)
[13] Kacsuk Péter: A magyar grid rendszerek és fejlesztési irányaik, 2003, [13], www.neumann-centenarium.hu/kongresszus/eloadas/ea20.pdf (2004-06-10)
[14] Condor Team, Univerity of Wisconsin-Madison: Condor Version 6.6.5 Manual, 2004, [576], http://www.cs.wisc.edu/condor/manual/v6.6.5/condor-V6_6_5-Manual.pdf (2004-06-10)
[15] Globus Toolkit honlap, http://www.globus.org/ (2004-06-10)
[16] Web Services, http://www.w3.org/2002/ws/ (2004-06-10)

[17] Global Grid Forum: Open Grid Services Infrastructure, 2003, [86], https://forge.gridforum.org/projects/ogsi-wg/document/Final_OGSI_Specification_V1.0/en/1 (2004-06-10)
[18] Global Grid Forum: Open Grid Services Architecture https://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-spec/ (2004-06-10)
[19] Karl Czajkowski, Steven Fitzgerald, Ian Foster, Carl Kesselman: Grid Information Services for Distributed Resource Sharing, Proc. 10th IEEE International Symposium on High Performance Distributed Computing, IEEE Press, 2001, [14], http://www.globus.org/research/papers/MDS-HPDC.pdf (2004-06-11)
[20] M. Wahl, T. Howes és S. Kille: Lightweight Directory Access Protocol, 1997 http://www.ietf.org/rfc/rfc2251.txt (2004-06-11)
[21] T. Dierks és C. Allen: The TLS Protocol, 1999 http://www.ietf.org/rfc/rfc2246.txt (2004-06-11)
[22] Public-Key Infrastructure (X.509), http://www.ietf.org/html.charters/pkix-charter.html (2004-06-11)
[23] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder és F. Siebenlist: X.509 Proxy Certificates for Dynamic Delegation**,** 3rd Annual PKI R&D Workshop, 2004, [17], http://www.globus.org/Security/papers/pki04-welch-proxy-cert-final.pdf (2004-06-11)
[24] I. Foster, C. Kesselman, G. Tsudik és S. Tuecke: A Security Architecture for Computational Grids, Proc. 5th ACM Conference on Computer and Communications Security, 83.-92. oldal, 1998, [10], ftp://ftp.globus.org/pub/globus/papers/security.pdf (2004-06-11)
[25] Globus RSL specifikáció**,** http://www.globus.org/gram/rsl_spec1.html (2004-06-16)
[26] Globus DUROC honlap, http://www.globus.org/duroc/frames.html (2004-06-16)
[27] Globus GASS honlap, http://www.globus.org/gass/ (2004-06-16)
[28] MPICH honlap, http://www.mcs.anl.gov/mpi (2004-06-16)
[29] Condor honlap, http://www.cs.wisc.edu/condor (2004-06-10)
[30] GridFTP protokoll, http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf (2004-06-10)
[31] LAM-MPI honlap, http://www.lam-mpi.org/ (2004-06-10)
[32] LA-MPI honlap, http://www.ccs.lanl.gov/ccs1/projects/la-mpi/ (2004-06-16)
[33] Fault Tolerant MPI honlap, http://icl.cs.utk.edu/ftmpi/ (2004-06-10)
[34] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel és S. Tuecke: Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing, 2000, [19], http://www.globus.org/research/papers/msc01.pdf (2004-06-19)
[35] J. Bester, I. Foster, C. Kesselman és Jean Tedesco: GASS: A Data Movement and Access Service for Wide Area Computing Systems, [11] http://imc.konkuk.ac.kr/~Grid/papers/gass.pdf (2004-06-19)
[36] MPICH-G2 honlap, http://www3.niu.edu/mpi/ (2004-06-10)

[37] Graham E. Fagg és Jack J. Dongarra: FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world, Recent Advances in PVM and MPI 7th European PVM/MPI Users' Group Meeting, Balatonfüred, 2000, Lecture Notes in Computer Science vol. 1908, Springer Verlag, Berlin, 346. oldal, [8], http://www.cis.ohio-state.edu/~yuw/courses/788_SU/papers/ft-mpi.pdf (2004-06-10)
[38] Graham E. Fagg és Jack J. Dongarra: HARNESS Fault Tolerant MPI design, usage and performandce issues, 2002, [17] http://www.netlib.org/netlib/utk/people/JackDongarra/PAPERS/ft-mpi-fgcs-grid-se.pdf (2004-06-19)
[39] Edgar Gabriel, Graham E. Fagg, Antonin Bukovsky és Jack J. Dongarra: Extending MPI: Fault-tolerance and MPI-2 issues, 2003, [28] http://www.hlrs.de/news-events/events/2003/metacomputing/slides/gabriel.pdf (2004-06-10)
[40] Európai DataGrid kezdeményezés, http://eu-datagrid.web.cern.ch/eu-datagrid (2004-06-10)
[41] Condor Master-Worker modell, http://www.cs.wisc.edu/condor/mw/overview.html (2004-06-10)
[42] Java RMI Specifikáció, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html (2004-06-19)
[43] RPC Specifikáció, RFC 1831, 1995, http://www.freesoft.org/CIE/RFC/1831/index.htm (2004-06-19)
[44] Von Welch: Globus Toolkit Firewall Requirements, 2003, [15], http://www.globus.org/security/firewalls/Globus%20Firewall%20Requirements-5.pdf (2004-06-19)
[45] William Gropp és Ewing Lusk: MPICH chp4 eszköz felhasználói kézikönyve, http://www.cs.utep.edu/~bdauriol/courses/ParallelAndConcurrentProgramming/mpichman-chp4.pdf (2004-06-19)
[46] A Mandelbrot halmaz története, http://home.att.net/~fractalia/history.htm (2004-06-19)