Párhuzamos és Elosztott Rendszerek II

PVM, MPI

Készítette: Dr. Mileff Péter

1. Bevezetés

A hardver fejlődésének ellenére – illetve egyes vélemények szerint éppen ennek okán – nem mondhatunk le a párhuzamos feldolgozás által nyújtott teljesítménynövekedésről. Egy konkrét párhuzamosítása több szinten történhet (és a gyakorlatban általában történik is). Ez a skála az egyetlen processzoron belüli párhuzamosítástól egészen a szuper-számítógépek ill. komplett számítóközpontok szintjén történő párhuzamosításig terjed. A különböző szintű párhuzamosítások általában függetlenek egymástól, így végeredményben eredőjük jelentkezik. A teljesítménynövekedésnek köszönhetően (1) korábban számítógéppel megoldhatatlan feladatok megoldása válik lehetővé, (2) egyes problémák sokkal jobb „minőségben" oldhatók meg, (3) bizonyos – megoldási idő szempontjából kritikus – feladatok gyorsabban oldhatók meg, ez fontos például valós idejű alkalmazások esetén.

1.2 Az üzenetküldéses paradigma

Most kizárólag az operációs rendszer *folyamatainak* (egyes terminológiákban processz, processzus, taszk) szintjén végrehajtható párhuzamosításra – illetve annak egy részterületére – szorítkozunk. Ha a párhuzamos programot – a programfejlesztés valamilyen szintjén – egymástól független, egymással kommunikáló szekvenciális folyamatok halmazaként fogjuk fel, akkor a kommunikációt tekintve alapvetően kétfajta programozási paradigmáról beszélhetünk. Az első az *osztott memóriás* modell, amelyben a futó folyamatok ugyanahhoz a (nem feltétlenül fizikai) memóriához férnek hozzá. A második pedig az üzenetváltásos modell, amelyben minden egyes folyamat a többi által nem elérhető memóriát használ, a kommunikációt pedig a folyamatok közötti (szinkron, aszinkron, illetve a kettő közötti átmenetnek megfelelő) üzenetek biztosítják. Itt most kizárólag a második modellel foglalkozunk.

Az, hogy a programfejlesztési modellünk üzenetváltásos modell, nem jelenti feltétlenül azt, hogy végül az implementált programnak is feltétlenül „üzenetváltásos" hardveren kell futnia; de természetesen igaz az, hogy az üzenetváltásos modellnek megfelelő algoritmusok, programok egyszerűbben és általában hatékonyabban implementálhatók üzenetváltásos architektúrákon. (Valamint az osztott memóriás modell alapján készült programok egyszerűbben és hatékonyabban implementálhatók osztott memóriás hardveren.)

1.3 Üzenetküldő rendszerek szabványai

Az üzenetküldéses rendszerek fejlesztését alapvetően két fontos irány segítette. Az rendszerek szabványa hosszú ideig a **Parallel Virtual Machine** (**PVM**) könyvtárra épült. A PVM alapkoncepciója, hogy heterogén hardverrel és szoftverrel rendelkező, különböző

architektúrájú számítógépekből egy virtuális gépet épít fel, és a számítási feladatokat ezen a virtuális gépen futtatja.

A **Message Passing Interface** (**MPI**) szabvány amelynek első specifikációja 1991-ben készült el, egy üzenetküldést megvalósító függvénykönyvtár szintaxisának és szemantikájának specifikációjaként fogható fel. Jelenleg az üzenetküldéses paradigma szabványának tekinthető.

2. PVM: Parallell Virtual Machine

A PVM-et általában egy üzenetküldéses rendszer konkrét megvalósításának tekintik. Helyénvalóbb azonban, ha a PVM-et egy üzenetküldéses rendszer specifikációjaként fogjuk fel, amelyet a PVM kézikönyv definiál. Ezt a szemléletet indokolja, hogy az első implementáció óta a PVM-et többször is megvalósították. A PVM – specifikáció és implementáció első (nem nyilvános) változata 1989-ben készült el, az Oak Ridge National Laboratory-ban. Első publikus változata (2.0 verziószámmal) a University of Tenessee-n készült el 1991-ben. A teljesen újraírt 3-as sorozatszámú változat 1993-ban jelent meg. Jelenleg a 3.4.5 a legfrissebb verzió.

A PVM kifejlesztésének célja egy olyan szoftver-rendszer kifejlesztése volt, amely támogatja az elosztott alkalmazások készítését – leginkább UNIX környezetben. Vagyis maga a PVM csak kevés hagyományos értelemben vett operációs rendszer szolgáltatást biztosít, legfőbb profiljának az elosztott alkalmazás-komponensek kommunikációjának megszervezése tekinthető. A PVM lehetőséget ad több – például TCP/IP protokollal hálózatba kapcsolt – számítógép erőforrásainak összevonására, ezzel egy „virtuális számítógép” létrehozására, amelyben a futó programok egységes interfésszel kapcsolódhatnak egymáshoz.

A PVM alapkoncepciója a virtuális gép fogalma. A virtuális gépet erőforrások egy halmaza alkotja. A PVM tervezésekor az egyik legfőbb alapelv a heterogenitás volt, egyazon virtuális gép elemei különféle módon lehetnek heterogének. Ezek a következők:

a) **Architektúra:** A PVM által támogatott architektúrák a PC osztályú számítógépektől egészen a szuperszámítógépekig terjednek. A különböző architektúrájú gépek egyetlen virtuális gépet képesek alkotni.
b) **Adatformátum:** A különböző számítógépek által használt adatformátumok sokszor inkompatibilisek egymással. A heterogén környezetet támogató üzenetküldő rendszereknek biztosítaniuk kell, hogy egy üzenet különböző architektúrájú feladója és címzettje megértik egymást.
c) **Számítási sebesség:** A hatékonyság érdekében a PVM rendszernek gondoskodnia kell róla, illetve lehetőséget kell adnia a programozónak, hogy gondoskodhasson róla, hogy a virtuális gépet alkotó számítógépek számítási teljesítményüknek megfelelően részesednek a feladatokból.
d) **Processzorterheltség:** A virtuális gépet nem feltétlenül dedikált számítógépek alkotják, más felhasználók is használhatják őket, így a gépek terheltsége folyamatosan változhat.
e) **Hálózati terheltség:** A virtuális gép részei különböző hálózati architektúrákkal rendelkezhetnek, ami nagyon különböző hálózati teljesítményt eredményezhet az egyes gépek között.

Mindezen problémák ellenére a heterogén eloszlott számítás rendkívüli előnyöket rejt magában, amelyek eredményeképpen az alkalmazásfejlesztési idő rövidebb lesz, a költségek pedig jelentősen csökkenthetők. A PVM rendszer maga C nyelven készült, és az alaprendszer a C és a Fortran nyelvet támogatja. Majd minden manapság használt nyelvhez készült azonban PVM interfész.

Létezik a PVM-nek C++ nyelvhez kapcsolódó függvénykönyvtára is. Ez a PVM++. A továbbiakban áttekintjük a PVM felépítését és működését.

3. A PVM rendszer szerkezete

Mint korábban említettük a PVM rendszer központi fogalma a virtuális gép. A virtuális gép valódi (esetleg önmagukban is párhuzamos) gépekből épül fel. A rendszer **két** főbb komponensből áll: egyrészt egy eljáráskönyvtárból, amit a PVM-et használó alkalmazásokhoz hozzá kell linkelni (az ezekben levő eljárásokkal érhetjük el a PVM szolgáltatásait programjainkból), másrészt pedig van egy önállóan futtatható PVM-démon része (*pvmd*), amely gépenként és felhasználónként egy-egy példányban fut, és ez az, ami a PVM szolgáltatások tényleges implementációját tartalmazza.

A legelső gépen elindított pvmd démont *mester pvmd* (master pvmd) démonnak nevezzük, amelynek szerepe kitüntetett. A további pvmd-ket általában a mester pvmd indítja el. Kivételes esetekben a felhasználó kézzel is indíthat (nem mester=szolga) pvmd-t, illetve egy kitüntetett taszk, az úgynevezett hoster is átvállalhatja ezt a feladatot.

A pvmd folyamatok a felhasználó jogaival futnak (Unix terminológia szerint), így a különböző felhasználók virtuális gépei teljesen függetlenek egymástól, konfigurációjuk is különböző lehet. A PVM futási egysége a taszk. Minden egyes taszkot az adott gépen futó pvmd indít el. Minden egyes taszk és pvmd egy egyedi azonosítóval van ellátva, amely egy 32 bites egész szám.

A PVM elterjedtségének egyik oka a szabad elérhetősége, valamint az, hogy számos hardvergyártó a többprocesszoros/elosztott számítógépeihez ilyen interfészt (is) biztosít. Továbbá meg kell említeni vele kapcsolatban azt is, hogy hordozható program; valamint a UNIX operációs rendszert bővíti ki -- ezzel nyilvánvalóan lehetővé válik párhuzamos programok szabványos UNIX környezetbe történő beágyazására.

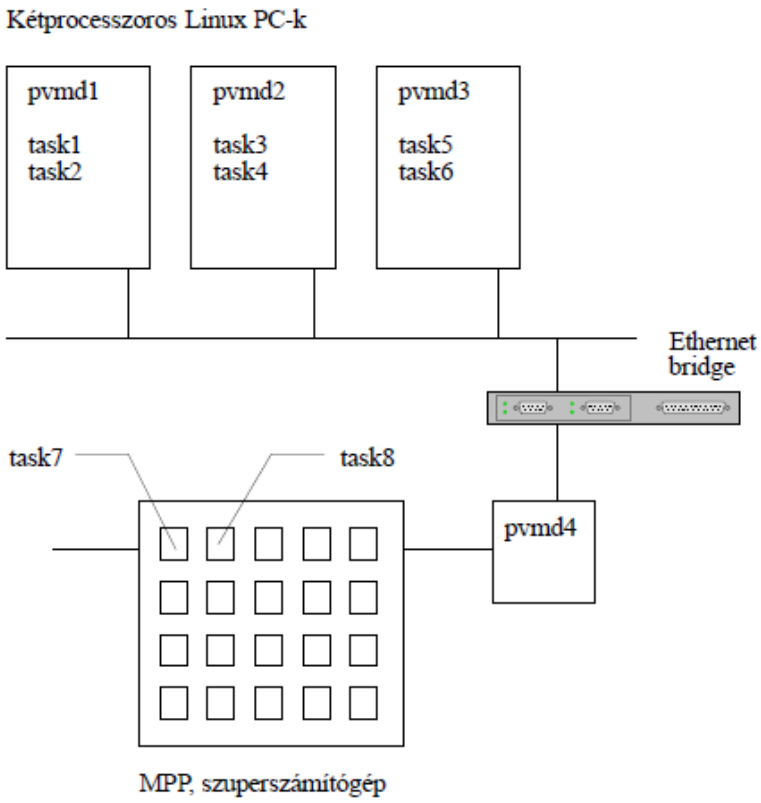
Ha például van két számítógépünk, amelyekből a PVM segítségével egyetlen virtuális gépet akarunk csinálni, azt megtehetjük úgy, hogy mindkét gépen elindítjuk a (megfelelően felkonfigurált) PVM-démont, majd el kell indítani a PVM-et használó alkalmazásokat. Ha egy gépet több különböző felhasználó akar egy virtuális géppé összekapcsolni, akkor mindegyik felhasználónak saját PVM-démont kell indítania, viszont egy felhasználónak elég csak egy PVM-démont futtatnia, ez megszervezi a felhasználó összes (akár több is!) elosztott alkalmazásának futtatását.

Elosztott alkalmazások készítésekor az egyes alkalmazás-komponenseket különálló programokként (mondjuk UNIX processzként) kell megírni, azok szinkronizációjára/adatcseréjére kell a PVM-et használnunk; vagyis a PVM nem biztosít eszközöket összetett alkalmazások komponenseinek automatikus párhuzamosításához. A PVM lehetőséget nyújt a következőkre:

- egy „mezei” UNIX processz bekapcsolódjon a virtuális gépbe,

- egy PVM-be kapcsolódott UNIX processz (a továbbiakban PVM-processz) kilépjen a PVM által biztosított virtuális gépből,
- lehetőség van PVM-processzek egymás közti kommunikációjára,
- PVM-processzek úgynevezett PVM-processzcsoportokba kapcsolására,
- biztosított egy-egy ilyen csoport összes tagja részére az üzenetküldés is,
- egy-egy csoportba új tag felvétele és tagok törlése a csoportból.

Az alábbi ábra egy PVM virtuális gép sematikus felépítését mutatja be.



1. ábra: PVM virtuális gép vázlatos felépítése

A virtuális gép tartalmaz három Linux operációs rendszerű PC-t, valamint egy szuperszámítógépet. Minden egyes PC-n fut egy pvmd, valamint a szuperszámítógép front-end gépén is fut egy pvmd. Utóbbi a gép minden egyes processzorán tud taszkot indítani, ezért elegendő tizenhat processzorhoz egyetlen pvmd. A PC-k és a szuperszámítógép egy ethernet bridgen keresztül vannak összekötve egymással.

3.1 A PVM üzenetek

A PVM kommunikáció egysége az *üzenet* (message). Noha a programozó által kezdeményezett üzenet címzettje egy másik taszk, az üzenet először mindig a küldő taszk pvmd folyamatához kerül, a pvmd fogja továbbítani azt a címzett pvmd folyamatához (hacsak ez nem önmaga lenne), amely végül eljuttatja a címzetthez. A két pvmd közötti kommunikáció minden esetben UDP protokollon keresztül történik. Az UDP protokoll

előnye, hogy egyetlen kapcsolódási végpont (azaz socket) segítségével lebonyolítható az összes partnerrel a kommunikáció, míg a TCP protokoll *N* gépből álló virtuális gép esetén minden egyes gépen N-1 darab kapcsolódási végpontot igényelne. Mivel az UDP protokoll nem biztosítja az UDP csomagok veszteségmentes és sorrendhelyes célba juttatását, erről a PVM rendszernek kell gondoskodnia. A PVM a taszkok és a lokális pvmd közötti kommunikációhoz TCP-t vagy UNIX domain socketeket használ. Az újabb PVM verziókban lehetőség van az egyes taszkok közötti közvetlen kommunikációra is, ehhez a PVM a TCP protokollt használja. Az üzenetek küldése és fogadása aszinkron módon történik.

Üzenet küldésére a *pvm_send* hívás szolgál, ez azonnal visszatér mihelyt az üzenetet tároló pufferből az üzenet rendszerpufferbe másolása befejeződött; a visszatéréskor nem biztos, hogy az üzenetet az adott gép már elküldte. Üzenetek fogadására a *pvm_recv* hívás, illetve ennek különféle variánsai (blokkoló, nem-blokkoló, stb.) szolgálnak. A *pvm_recv* hívás a lokális pvmd-től próbálja átvenni az üzenetet. Az üzenetekhez ún. üzenetcímkét (*message tag*) rendelhet a küldő taszk; a fogadó taszk pedig a küldő taszk azonosítója illetve az üzenetcímke alapján szelektálhat, hogy éppen milyen üzenetet szeretne fogadni.

A továbbiakban a PVM működését tekintjük át röviden.

3.2 A PVM elindítása

A PVM virtuális gép felélesztéséhez el kell indítanunk az ehhez szükséges programot: vagy a PVM-démont (neve pvmd3) vagy pedig a PVM konzolt (neve pvm). Az utóbbi a virtuális gép felélesztése mellett elindít egy PVM-konzolt, amivel interaktívan kapcsolhatunk és törölhetünk gépeket a virtuális gépből, interaktívan indíthatunk (és lőhetünk le) PVM-processzeket, és talán legfontosabb parancsa a *help* (amivel a további parancsok használatához kérhetünk segítséget a rendszertől). Például fontos parancsok a következők:

halt: lelövi az összes futó PVM-processzünket, és kilép a PVM-ből, leállítja a PVM démont.
spawn: új PVM-processzt indít.
kill: egy futó PVM-processzt lelő.
reset: lelövi az összes futó PVM-processzt a konzol kivételével, a PVM-démonokat pedig nem bántja.

3.3 A PVM processzek kezelése

Minden egyes PVM-processz rendelkezik egy egész típusú processz-azonosítóval (a UNIX processzek azonosítójához hasonló a szerepe, és fontos megemlíteni, hogy a PVM-processz azonosítónak semmi köze sincs a UNIX processzeinek a processz-azonosítójához). A PVM-processz azonosítókat a továbbiakban *tid*-del jelöljük.

pvm_spawn: Egy új PVM-processzt a *pvm_spawn* eljárással hozhatunk létre. Ennek alakja a következő:

```
int pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids);
```

Az első argumentum (task) az elindítandó PVM-processzt tartalmazó végrehajtható fájl nevét adja meg; a második (argv) az átadandó program-argumentumokat tartalmazza; a harmadik paraméter kijelöli, hogy hol kell elindítani az új PVM-processzünket (pl. CPU-architektúrára tartalmazhat utalást, de gyakran a processzt indító számítógép CPU-ján kell az új PVM-processzt futtatni, ezért gyakori, hogy az ezt kijelölő PvmTaskDefault (0 értékű) konstans adják itt meg). Amennyiben az alkalmazásnak speciális igényei vannak az új PVM-processz elindításával kapcsolatban, akkor azt a flag argumentumban jelezheti, és ilyenkor kell a where argumentumban megadni azt, hogy (fizikailag) hol is akarjuk a programunkat elindítani. Az ntask paraméterben adhatjuk meg, hogy hány példányban akarjuk a PVM-processzt elindítani, majd a rendszer a tids tömbben adja vissza az elindított PVM-processzek azonosítóit (tid-jeit). A függvény visszatérési értéke a ténylegesen elindított új PVM-processzek száma.

Használatára példa:

numt = pvm_spawn("progi1", NULL, PvmTaskHost, "macska", 1, &tids[0]);

Ami elindít a macska nevű hoston egy új PVM-processzt a progi1 nevű programból. Megjegyezzük, hogy a PvmTaskHost paraméter arra utal, hogy kijelöljük a negyedik argumentumban, hogy melyik hoston akarjuk elindítani az új PVM-processzt. Ha itt PvmTaskDefault értéket adtunk volna meg, akkor a PVM rendszer maga választhatott volna egy hostot, ahol a programot elindítja. A tids változó egy egészeket tartalmazó vektor.

pvm_exit: Egy PVM-processz a *pvm_exit* eljárás végrehajtásával léphet ki a PVM felügyelete alól.

pvm_mytid: Egy PVM-processz a saját tid-jét ezzel kérdezheti le. A függvény visszatérési értéke az új PVM-processz tid-je. Nincs paramétere.

pvm_parent: Egy PVM-processz a szülőjének a tid-jét kérdezheti le. Visszatérési értéke a szülőjének a tid-je. Nincs paramétere.

pvm_kill: Lelövi azt a PVM-processzt, amelynek a tid-jét megadtuk az argumentumában.

3.4 A PVM kommunikáció

A PVM eszközei közt vannak primitív, UNIX signal-okat használó kommunikációs eljárások valamint vannak összetettebb adatszerkezetek/adatterületek processzek közötti mozgatására is eszközök.

pvm_sendsig: A függvény segítségével egy adott signalt (UNIX signalt) küldhetünk valamelyik PVM-processznek.

pvm_notify: A függvény alakja a következő:

int pvm_notify(int about, int msgtag, int ntask, int *tids)

Hatása pedig az, hogy a későbbiekben az about argumentumban specifikált esemény bekövetkeztekor egy olyan üzenetet küld a tids vektorban megadott tid-del azonosított PVM-processzeknek, amely üzenet msgtag része megegyezik a függvény második argumentumában megadott értékkel.

3.5 Puffer allokálás/deallokálás

A PVM rendszerben üzenetküldés céljából több üzenet-puffert is allokálhatunk, és ezek közül kijelölhetjük, hogy melyik legyen az aktív küldési illetve aktív fogadási puffer: vagyis melyikbe akarunk adatokat bepakolni a bepakoló-függvényekkel (adatelküldés céljából) illetve melyikből akarunk adatokat kiolvasni (egy megkapott üzenetből). Megjegyezzük, hogy az adatbeolvasási és az adatküldési puffer megegyezhet.

Egy új üzenet-puffert a **pvm_mkbuf** függvénnyel hozhatunk létre. Ennek egyetlen argumentuma van (egy egész szám, amely a puffer tartalmának az összeállítási módját értelmezését/reprezentációját írja le), visszatérési értéke pedig (szintén egy egész szám) a létrehozott puffer egyértelmű azonosítóját tartalmazza.

pvm_freebuf: Egyetlen argumentuma az eldobandó üzenet-puffer egyedi azonosítója. A függvény eredményeként a paraméterben kijelölt puffer által lefoglalt memóriaterületeket a rendszer felszabadítja, visszarakja a memória szabad-listára.

pvm_getsbuf: Nincs argumentuma. Visszatérési értéke az aktuális (aktív) küldési puffer azonosítója.

pvm_getrbuf: Nincs argumentuma. Visszatérési értéke az aktuális (aktív) fogadási puffer azonosítója (ez az, amelyikből adatokat tudunk kiolvasni a később bemutatásra kerülő adatbeviteli függvényekkel).

pvm_setsbuf: Meg lehet változtatni vele az aktív küldési puffert. Egyetlen argumentuma van, amely megadja az új aktív küldési puffer egyedi azonosítóját kell, hogy tartalmazza, és visszaadja a korábban aktív küldési puffer azonosítóját.

pvm_setrbuf: Meg lehet változtatni vele az aktív fogadási puffert. Egyetlen argumentuma van, amely megadja az új aktív fogadási puffer egyedi azonosítóját kell, hogy tartalmazza, és visszaadja a korábban aktív fogadási puffer azonosítóját.

3.6 Processz csoportok

A PVM rendszer lehetőséget nyújt processz-csoportok szervezésére. Egy PVM processz bármikor beléphet egy csoportba -- ezzel csoporttaggá válik -- illetve bármikor kiléphet egy csoportból. A csoportoknak nevet adhatunk, a csoportnév ASCII karakterek sorozatából állhat. A csoportokkal kapcsolatban több művelet is van: üzenet küldése az összes csoporttag számára (az üzenet küldője nem kell, hogy tagja legyen a csoportnak) valamint lehetőség van a csoporttagok állapotának szinkronizálására.

Egy PVM-processz a **pvm_joingroup** függvényhívással lehet tagja egy processz-csoportnak. A függvény egyetlen paramétere a csoport neve. A függvény visszatérési értéke egy csoporton belüli azonosító szám.

Egy PVM-processz a **pvm_lvgroup** függvényhívással léphet ki egy csoportból. Egyetlen argumentuma a csoport neve, amiből a PVM-processz ki akar lépni.
Egy csoport résztvevőinek a számát a **pvm_gsize** függvénnyel kérdezhetjük le: egyetlen argumentuma a csoport neve.

Egy csoport összes tagjának küldhetünk egy művelettel üzenetet a **pvm_bcast** függvénnyel. Ennek alakja a következő: int pvm_bcast(char *group, int msgtag)

(itt az első argumentum a csoport neve; a második argumentum az üzenet típusának az azonosítója; visszatérési értéke negatív ha valami hiba történt.) Megjegyezzük, hogy a küldő nem kapja meg az így küldött üzenetet, ha tagja a megadott nevű csoportnak.

Egy csoport szinkronizálására szolgál a **pvm_barrier** függvény. Ennek az alakja a következő:

int pvm_barrier(char *group, int count)

Ahol az első a csoportnév; a második argumentum pedig egy szám. A függvény eredményeként a hívó PVM-processz blokkolni fog egészen addig, amíg (count) darab csoporttag meg nem hívja e függvényt. A többi processznek ugyanezekkel az argumentumokkal kell meghívnia e függvényt.

3.7 Összefoglalás

A PVM rendszert azért fejlesztették ki, hogy egy egységes felületet biztosítsanak párhuzamos programok fejlesztésére valóban párhuzamos architektúrákon, illetve hálózatba kapcsolt számítógépeken. A fejlesztés során a korábbi változatokkal való kompatibilitás megőrzése érdekében megtartottak olyan elemeket is, amelyeket a későbbiekben ismertetett MPI-ban már jobban oldottak meg (pl. üzenet összeállítása).

PVM-ben az MPI-hoz hasonlóan szinte minden párhuzamosságot igénylő feladat megoldható több-kevesebb ráfordítással. A rendszer dinamikus folyamat kezelése miatt kliens/szerver alkalmazások írására is alkalmas lehet.

A PVM rendszer csak programok párhuzamosságát támogatja. Nincsenek benne eszközök adat párhuzamosság, illetve folyamat-szál, utasítás szintű párhuzamosság támogatására.
Dinamikus folyamat modellje viszont nagy rugalmasságot ad a rendszernek. Bármikor lehet új folyamatokat létrehozni, és ezeket bármikor meg lehet állítani, akár egy másik folyamatból is. A folyamatok indításakor rábízhatja a felhasználó a rendszerre, hogy válassza ki a szerinte megfelelő processzort az új folyamat futtatására, de erre kikötéseket is tehet. A rendszer legújabb változataiban az erőforrás kezelő programrészeket le lehet cserélni, ezzel az adott környezet lehetőségeit maximálisan kihasználó erőforrás kezelési stratégiákat lehet implementálni.

PVM-ben nincsenek virtuális processzorok, de speciális változók beállításával az egyes folyamatok között csatornákat is létre lehet hozni, a gyorsabb kommunikáció érdekében.

PVM-ben csak aszinkron, pufferelt üzenetküldés áll a felhasználó rendelkezésére a folyamatok közötti kommunikáció megszervezése során. Ezek az üzenetküldések szimmetrikus címzést tesznek lehetővé, és az üzenetek sorrendjét két pont között megőrzik.

Az üzenetküldés mindig blokkoló eljárás, de ez az aszinkron tulajdonság miatt nem hátráltatja nagyon a folyamatok futását. Az üzenetek vételére vannak blokkoló, nem blokkoló és lejárati időhöz kötött eljárások is.

4. Az MPI - Message Passing Interface

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 hordozhatók.

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.

4.1 Áttekintés

A továbbiakban röviden áttekintjük a szabvány első változatában specifikált lehetőségeket. Ezek már minden MPI rutinkönyvtárban benne vannak. A felületet persze az MPI Forum tovább bővíti.

A szabványban C és Fortran 77 nyelvekhez adtak meg felületet. A rutinkönyvtárral a következő feladatokat lehet megoldani:

- Pont-pont közötti kommunikáció
- Üzenetekben lévő típusok megadása
- Kollektív kommunikáció
- Csoportok, környezetek kezelése

- Folyamatok topológiába rendezése
- Néhány kiegészítő rutin a program környezetéhez
- Segítség a teszteléshez

Az MPI nem definiál eljárásokat folyamatok indítására, illetve dinamikus kezelésére. Az MPI rendszerekben a folyamatok általában önálló programok, melyeket egyszerre indítanak el. A programok nem feltétlenül azonosak, tehát MIMD stílusban is lehet párhuzamos programokat írni ezzel a könyvtárral. A programok belső szerkezetét sem definiálja az MPI, tehát lehetnek szekvenciálisak, de akár többszálúak is, az MPI mégis egy önálló folyamatnak tekinti. (Többszálúságra nincs semmilyen támogatás, de az implementációk általában ügyelnek rá, hogy a rutinok ne okozzanak bajt ilyen környezetben sem.)

A szabvány következő változatába az alábbi dolgokat kívánják bevenni:

- I/O műveletek
- Aktív üzenetek
- Folyamatok indítása, dinamikus taszk kezelés
- Távoli elérés/tárolás (mintha osztott memória lenne)
- Fortran 90 és C++ nyelvhez illesztések
- Grafika
- Valós idejű programozás támogatása
- más "javítások"

Ezek egy részét az egyes implementációk már tartalmazzák.

4.2 Implementációk

Sok párhuzamos számítógépgyártó elkészítette a saját eszközére optimalizált változatot, de ezek mind kereskedelmi termékek. Ezeken kívül vannak publikus változatok is, amelyek a hatékonyságban sem maradnak el:

- Argonne National Laboratory / Missipi State University: **MPICH** (ftp://info.mcs.angl.gov/pub/mpi)
- Edinburgh Parallel Computer Centre: **CHIMP** (ftp://ftp.epcc.ed.ac.uk/pub/chimp/)
- Ohio Supercomputer Center: **LAM** (ftp://tbag.osc.edu/pub/lam/)
- Missisipi State University: **UNIFY** (ftp://ftp.erc.msstate.edu/unify/), ez csak a szabvány egy részét valósítja meg a PVM-re építve
- University of Nebraska at Omaha: **WinMPI** (ftp://csftp.unomaha.edu/pub/rewini/WinMPI)

Az MPICH implemsentációt az MPI szabvány megvalósítására írták, a CHIMP, LAM és UNIFY csomagok már meglévő rutinkönyvtárakra épülnek rá, ezért nem annyira hatékonyak.

4.3 Felhasználói felület

A felületben lévő rutinokra a következő fogalmak használatosak:

- **lokális** egy eljárás, ha csak egy folyamat belső adataitól függ, és befejezéséhez nem igényel kommunikációt más programokkal.
- **nem-lokális** egy eljárás, ha a benne lévő művelet más folyamatokkal való kommunikációt is igényel.
- **kollektív típusú** egy rutin, ha egy csoportban minden folyamatnak meg kell őt hívni, hogy befejeződhessen.

A szabvány a nyelvek közötti átjárhatóság biztosítása érdekében törekedett az egységes felület kialakítására, ezért nem használják ki mindig az adott nyelv által biztosított lehetőségeket maximálisan.

A szabványban leírt rutinok alakjai:

C nyelvben, ahol a visszatérési érték a rutin lefutásáról ad információt:

int MPI_Xxxxxxxx(...);

Fortran 77 nyelven, ahol az IERROR (utolsó) paraméternek van ilyen funkciója:

MPI_XXXXXXXX(..., IERROR);

4.3.1 Kezdet és vég

A program indulása után az MPI függvények használata előtt kell meghívni az MPI könyvtár inicializálását: **MPI_Init**. Ez az eljárás parancssori argumentumokkal is foglalkozik, ezért azokat is át kell adni.

A program normális befejezése esetén a könyvtárban definiált **(MPI_Finalize)** eljárás meghívásával lehet az által lefoglalt erőforrásokat felszabadítani.

Hiba esetén a párhuzamos program futása megszakítható az **MPI_Abort** eljárással. Ezen a módon nem csak az adott folyamat, hanem a hozzá tartozó folyamatok is leállíthatók. A leállítandó csoport leírására szolgál a **comm** paraméter.

A comm paraméter egy kommunikációs közeget ír le. Az **MPI_Init** meghívása után létrejön egy MPI_COMM_WORLD nevű MPI_Comm típusú *kommunikátor*, amelyben minden taszk benne lesz. Ezt is paraméterül lehet adni az MPI_Abort rutinnak, hogy az egész programrendszer futását megszakítsa.

A kommunikátorokról elöljáróban: egy kommunikátor folyamatok egy csoportját azonosítja, és egy üzenetküldési réteget definiál, mellyel ugyanazon folyamatok között küldött üzenetek is teljesen elkülöníthetők egymástól.

Ezek alapján a minimális MPI program:

```
#include mpi.h

void main(int argc, char *argv[])
{
        MPI_Init(&argc, &argv);
        /* a program utasításai ide kerülnek */
        MPI_Finalize();
}
```

4.3.2 Pont-pont kommunikáció

A pont-pont kommunikációkban mindig két taszk vesz részt. Az egyik küldi a másik pedig fogadja az üzenetet. A pont-pont kommunikációk mindig megőrzik az üzenetek sorrendjét.

A legegyszerűbb küldő rutin az **MPI_Send**. Ez egy adott területről az MPI könyvtárban definiált típusú adatvektor elküldésére alkalmas. A címzésnél meg kell adni a fogadó sorszámát, az üzenet típusát és az üzenet továbbítására szolgáló kommunikátort. Ez egy blokkoló rutin.

Ebben a könyvtárban az **MPI_Datatype** típus adhatja meg egy üzenet adattípusát. Új típusok is kialakíthatók struktúrák, illetve tömbrészletek leírására.

Egy üzenetben mindig benne van, hogy melyik folyamat küldte, mi az üzenet típusa, ki a címzett és mi a kommunikátor.

Blokkoló üzenet vételére szolgál az **MPI_Recv** eljárás. Az üzenet forrása és típusa előírható, de az MPI_ANY_SOURCE, illetve az MPI_ANY_TAG használatával bárkitől bármilyen üzenetet is lehet fogadni.

A vett üzenetről információkat az eljárás paramétereként adott status struktúrából lehet kinyerni. Ennek MPI_SOURCE, MPI_TAG és MPI_ERROR mezőiből egy ismeretlen üzenet küldőjét, típusát, illetve a vétel sikerességét lehet megállapítani.

MPI-ban a vett adatok egyszerű adattípusának meg kell egyeznie a küldött adatok egyszerű adattípusával. Ez azt jelenti, hogy egy bonyolult struktúrát, vagy tömböt az MPI_Send-nél felsorolt típusokra lebontva pont ugyanazokat és ugyanannyi ilyen egyszerű adatot kell venni, mint amennyit küldtek. (Pl.: két {*int, char*} struktúra vehető egyetlen {*int,char,int,char*} struktúraként) Az MPI nem gondoskodik típuskonverzióról - *int* típusú adatot nem lehet *double*-ként venni -, sőt ilyen esetben hibát jelez.

Az eljárásnak a használt kommunikátort is specifikálni kell. MPI-ban a kommunikátorból megállapítható, hogy a folyamatok azonos architektúrákon futnak-e, ha nem akkor a reprezentációk között automatikusan konvertálódnak az adatok.

A pont-pont közötti kommunikációknak még több fajtája elképzelhető:

- **Pufferelt vagy nem Pufferelt üzenetküldés:** lokális eljárás, mert az üzenet továbbítása nélkül is véget érhet, hiszen egy helyi tárolóban még megőrződhet annak tartalma.
- **Szinkron vagy aszinkron:** Szinkron kommunikáció nem lokális; csak akkor fejeződhet be, ha az üzenetet ténylegesen vették egy MPI_Recv rutinnal. A kommunikáció az MPI_Recv előtt is elkezdődhet.
- **Ready:** Ez a kommunikáció, csak akkor sikeres, ha az üzenet küldést a másik oldal már várja, azaz végrehajtott már egy MPI_Recv függvényhívást. Ha a vevő oldal ezt nem tette meg akkor a művelet eredménye definiálatlan.

A pufferelt, ready és szinkron üzenetküldések alakja csak a nevében tér el a MPI_Send-től:

- **MPI_Bsend** - pufferelt.
- **MPI_Ssend** - szinkron.
- **MPI_Rsend** – ready.

A fenti rutinoknak vannak nem blokkoló változataik is. Ez az MPI-ban azt jelenti, hogy egy műveletet két részre lehet bontani: egy kezdeti inicializáló részre, illetve a művelet befejeződését vizsgáló részre. Az ilyen szétbontásnak akkor van értelme, ha egy művelet hosszú ideig tart, és a várakozás helyett lehet hasznos számításokat is végezni. A nem blokkoló műveletek akkor lehetnek hatékonyak, ha azokat a program egy másik végrehajtási szála, vagy egy másik - esetleg kommunikációra specializált - processzor hajtja végre. A végrehajtás módszerének részletei implementáció függőek.

A nem blokkoló rutinok mind egy **MPI_Request** típusú kérelmet használnak a háttérben futó művelet jellemzőinek tárolására.

Nem blokkoló üzenetküldés indítására az **MPI_Isend** rutin szolgál. Az eljárás paraméterei a blokkoló változattal megegyeznek, eltekintve az utolsó argumentumtól.

Az eljárás lefutása során inicializálja az utolsó paraméterként megadott **MPI_Request** típusú struktúrát. Ez tárolja a függőben lévő, vagy háttérben futó eljárás jellemzőit. Ennek vizsgálatával megállapítható, hogy az üzenetküldés befejeződött-e, vagy előkerült-e valamilyen hiba.

Az eljárás az inicializálás után visszatér, és az üzenetküldés folyamata a program további futásával párhuzamosan fog megtörténni.

A küldendő adatokat a művelet befejezéséig nem illik bántani, mert nem definiált, hogy mikor olvassa ki azokat a rendszer.

Ennek persze megvannak a szokásos változatai:

- **MPI_Ibsend** - pufferelt
- **MPI_Issend** - szinkron
- **MPI_Irsend** - ready

A nem blokkoló üzenet küldéshez hasonló a nem blokkoló üzenet vétel. Ennek paraméterezése a normális üzenet vételhez hasonló, csak a **MPI_Request** típusú plusz paraméter jelent az **MPI_Irecv** eljárásban is eltérést.

Egy háttérben futó eljárás vizsgálatára a következő lehetőségek vannak:

Meg lehet várni a művelet végét, blokkolva a programot: **MPI_Wait**. Az **MPI_Test** eljárással meg lehet vizsgálni egy műveletet. Ez nem blokkolja a program futását, csak jelzi, hogy a vizsgált művelet befejeződött-e már. **MPI_Request_free** szolgál a háttérben futó művelethez tartozó jellemzőket leíró struktúra felszabadítására. A hozzárendelt művelet ettől függetlenül még befejeződhet, bár a sikerességet már nem lesz lehetőség megvizsgálni.

Ha egy programban több nem blokkoló művelet is van, akkor ezeket egyszerre is lehet vizsgálni:

- bármelyik befejeződését meg lehet várni az **MPI_Waitany** eljárással,
- illetve bármelyik lefutását lehet vizsgálni az **MPI_Testany** rutinnal.

Ezek az eljárások egyetlen nem blokkoló művelet befejeződését figyelik csak.

Ha mindegyik műveletet meg kell várni, akkor az **MPI_Waitall**, ha mindegyik állapotát meg kell vizsgálni, akkor **az MPI_Testall** eljárások használhatóak.

Egy nem blokkoló művelethalmaz néhány eleme vizsgálható az **MPI_Waitsome**, illetve az **MPI_Testsome** eljárásokkal. Ezek jelzik, hogy mely műveletek fejeződtek be.

Üzenetek megérkezését - az üzenet tényleges vétele nélkül - az **MPI_Probe** és **MPI_Iprobe** eljárásokkal lehet vizsgálni. Ezen rutinok segítségével a program az üzenet vétele nélkül tájékozódhat annak méretérő, típusáról, illetve feladójáról, de nem a tartalmáról. Az eljárások az **MPI_Recv és MPI_IRecv+MPI_Test** rutinokhoz hasonlóan viselkednek, csak nem veszik az üzenetet.

Egy futó művelet megszakítása való az **MPI_Cancel** rutin. Az **MPI_Cancel** után egy **MPI_Wait** vagy **MPI_Test** művelettel lehet a nem blokkoló művelethez kapcsolódó erőforrásokat felszabadítani.

Ez az eljárás lényegében azt teszi lehetővé, hogy valamilyen eseményhez kötött művelet-végrehajtás legyen a programban. Egy küldő, vagy fogadó műveletet ezen a módon egy meghatározott türelmi idő letelte után, vagy valamilyen más esemény hatására - pl. felhasználó megnyomja az ESC billentyűt - meg lehet szakítani.

A nem blokkoló művelet megszakításának sikerességét a kérelem befejezése után a **MPI_Test_cancelled** rutinnal lehet elvégezni.

Ha nagyon sokszor kell azonos paraméterekkel leírt kommunikációt végezni, akkor ezeket a paramétereket egy kérelemben el lehet tárolni. Ilyen kérelmek létrehozására szolgálnak az **MPI_Send_init és MPI_Recv_init** eljárások.

Ezekkel létrehozott kérelmet, illetve kérelmeket az **MPI_Start**, illetve **MPI_Startall** eljárásokkal lehet elindítani.

Az egyszerűsített híváson kívül ez a módszer lehetőséget ad az implementációnak, hogy egy csatorna megnyitásával gyorsabb üzenetküldést biztosítson a két fél között.

Az MPI még egy kényelmes függvényt biztosít egyszerű kommunikációra: az **MPI_Sendrecv** eljárással egyesíteni lehet egy üzenet elküldésének és egy üzenet vételének folyamatát. A rutin hatása ekvivalens nem blokkoló üzenet küldés és vétel elindításával, majd a műveletek befejeződésének megvárásával.

4.3.3 Az üzenetekben lévő típusok meghatározása

Az üzenetekbe csak az MPI által ismert adattípusokat lehet belerakni. Mivel az elképzelhető típusok száma végtelen, ezért csak az **MPI_Send**-nél említett alaptípusokat tartalmazza induláskor a rendszer. Ha a programozónak másra is szüksége van, akkor azt az alábbi típuskonstrukciós műveletekkel építheti fel.

Egy új MPI adattípust egy **MPI_Datatype** átlátszatlan típusban lehet létrehozni. A létrehozás után a típust engedélyeztetni kell az **MPI_Type_commit** eljárással, ami lényegében a típus elismertetése a teljes rendszerben. Sikeres engedélyeztetés után az új típust bárhol használható az eddigi egyszerű típusok helyett, akár egy újabb típus létrehozására is.

Egy MPI adattípus lényegében egy egyszerű típusokból álló típus térkép. Leírja, hogy egymás után milyen egyszerű adattípusok vannak, és az egyes elemek milyen eltolással

vannak eltárolva. Az adattípusok belső tárolására persze tömörebb formát használnak, de ez a szemléletes kép jól használható az MPI típus-kompatibilitás megértéséhez.

Egy adott adattípussal küldött MPI üzenet csak olyan MPI adattípussal vehető, amiben az egyszerű adattípusok sorrendje és száma megegyezik a küldött MPI típuséval. Ez azt jelenti, hogy az **MPI_Datatype** átlátszatlan típust az egyes folyamatokban különbözőképpen is lehet definiálni, mégis ekvivalensek, ha ugyan azt az egyszerű adattípus sorrendet írják le.
A továbbiakban néhány fontos rutin kerül felsorolásra:

MPI_Type_contiguous: Ezzel egy korábban definiált típusból kiindulva, adott hosszúságú - a memóriában folytonosan elhelyezkedő elemeket tartalmazó - vektor hozható létre.
MPI_Type_vector és MPI_Type_hvector: egy dimenziós, és azonos adattípusú elemekből álló - szerkezetek létrehozására alkalmasak.
MPI_Type_struct: Strukturált adattípus leírására szolgál. Az itt megadott típusok bármilyen korábban definiált, MPI számára ismert adattípus közül kikerülhetnek. Ezzel a rutinnal lényegében bármilyen adattípus leírható.
MPI_Type_free: Egy adattípus felszabadítása. Amennyiben egy kommunikáció éppen használja ezt, akkor az még rendben lefut, és csak utána kerül felszabadításra a hozzá tartozó memóriaterület.

Egy üzenet összeállítására az **MPI_Pack** eljárást lehet használni. Ez a rutin lényegében az **MPI_Send**-hez hasonló formában megadott adatokat másol be a felhasználó által előre lefoglalt puffer területre. Ezt a rutint többször meghívva tetszőleges adatokat lehet berakni egy üzenetbe. Az így előkészített üzenet puffert a szokásos **MPI_Send** rutinnal lehet elküldeni, csak **MPI_PACKED** paramétert kell megadni az üzenet adattípusaként.

Az **MPI_Unpack** eljárással egy **MPI_PACKED** típussal vett üzenetből lehet kipakolni az adatokat. Szintaxisa a bepakolásra hasonlít.

A pakolt üzenetek küldésének kétségtelen előnye, hogy több lépésben is össze lehet állítani az üzenetet, de így az MPI semmilyen típusellenőrzést nem tud végezni a küldött üzenetben.

Az MPI nem definiálja, hogy egy szokásos módon küldött üzenet vehető-e **MPI_PACKED** típussal, illetve egy így küldött üzenet vehető-e a szokásos módon. Egyes implementációkban ez működik, de erre nem mindenhol lehet számítani.

4.3.4 Kollektív kommunikáció

Az itt leírt műveletek több folyamat együttes részvételével jöhetnek létre. Ezeket e rutinokat a kollektív kommunikációban résztvevő összes folyamat egyszerre kell, hogy meghívja. A rutinok folyamatok közötti kommunikáció segítségével végzik el tevékenységüket, tehát nem lokálisak. Az MPI rendszer legfőbb előnye ezen rutinok használatában van.

Az eljárások nagy része egyszerű algoritmusokat valósít meg, melyek ezek nélkül is megoldhatóak lennének, de több okból is indokolt ezek használata:

- egyszerűbb írásmód, emiatt kisebb a tévesztés veszélye a program írásakor,

- a műveletek kommunikációja a program többi kommunikációjától teljesen el van választva, ezért a futás közbeni hibák esélye is csökken,
- egyes implementációkban ezek a rutinok kihasználhatják az adott gép minden lehetőségét, ami gyorsabbá teheti az algoritmusokat a pont-pont üzenetküldésekkel megírt változatuknál.

Az **MPI_Barrier** rutinnal több folyamatot lehet egymással szinkronizálni. A rutin addig vár, ameddig az összes - a kommunikátorban lévő - folyamat meg nem hívja ugyanezt az eljárást.

MPI_Bcast eljárás adatok lemásolására szolgál. Egy kijelölt folyamat adatterülete lemásolódik a műveletben résztvevő összes folyamat adatterületére. Ez a művelet közösen használt változók, paraméterek elterjesztésére szolgálhat, például egy folyamat feldolgozza a parancssori argumentumokat, és az ezekből kinyert információkat megosztja a többi folyamattal.

Az **MPI_Gather** eljárás adatdarabkák összegyűjtésére szolgál. Minden folyamat egy adott mennyiségű adatot küld el, ami egy kijelölt folyamatban összegyűlik. A fogadó folyamatban a különböző helyről érkező adatelemek egy vektorba kerülnek, melyben az adatelem helyét a küldő folyamat sorszáma határozza meg. A küldött és a vett adatok mennyiségének és típusának meg kell egyezni. (Ez a megkötés a további rutinokra is igaz.)

Az eljárás hatása ekvivalens azzal, hogy a kijelölt folyamat ciklusban fogad adatelemeket a többi folyamattól, és a vett adatokat elhelyezi egy vektorban.

Az **MPI_Scatter** és **az MPI_Scatterv** az összegyűjtő eljárások inverzei, adatok szétosztására szolgálnak. Egy vektorban megadott adathalmazt osztanak szét a kollektív kommunikációban résztvevő folyamatok között. Az egyes folyamatokhoz érkező adatelemek a kijelölt folyamatban lévő vektorból származnak. Az **MPI_Scatter**-ben egyszerűen indexek alapján történik a szétosztás, az **MPI_Scatterv**-ben az adott sorszámú folyamathoz kerülő adatelem kezdetét is meg lehet határozni.

MPI_Reduce asszociatív és kommutatív művelet végrehajtására használható a folyamatokban tárolt lokális adatokon. Az eredmény egy kijelölt folyamatba érkezik meg.

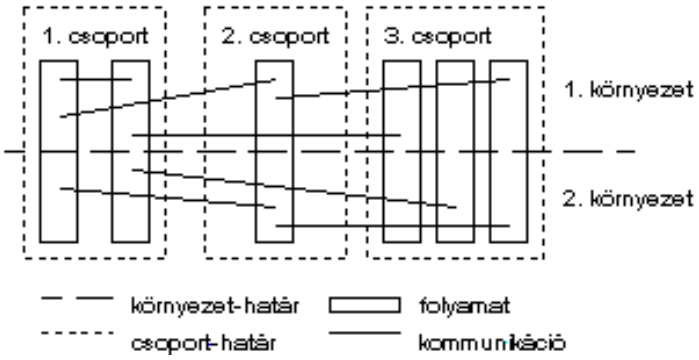
A rendszer a következő műveleteket definiálja előre:

- maximum, illetve minimum értékének keresése,
- összegzés, valamit produktum számítása,
- logikai ÉS, VAGY, illetve kizáró VAGY művelet,
- bináris ÉS, VAGY, illetve kizáró VAGY művelet.
- maximum, illetve minimum értékének és helyének keresése.

Természetesen az MPI további lehetőségeket biztosít az üzenetekkel való műveletek elvégzésére, de ezekre a továbbiakban nem térünk ki.

4.3.5 Csoportok, környezetek kezelése

Nagy programok fejlesztésénél, illetve rutinkönyvtárak kialakításánál nagyon jó lehetőség, ha az abban folyó kommunikációt el lehet választani a program többi részétől. Erre e problémára nagyon jól használhatóak a *folyamat csoportok*, illetve *kommunikációs környezetek* kialakításának módszerei.



2. ábra: Csoportok és környezetek

Folyamatok ***csoport***ba rendezésével egyes problémák egyszerűbben leírhatóak, nem kell a kód ismétlésével többször hivatkozni önálló folyamatokra. A csoportba rendezés emellett lehetőséget biztosít a rendszer számára magasabb szintű optimalizációra is, hiszen így még több információt kap a program működéséről.

A ***kommunikációs környezetek*** arra szolgálnak, hogy egymástól független kommunikációs rétegeket lehessen kialakítani egy programon belül, akár azonos folyamatok között. Ez a megosztás nem vertikális - mint a csoportosítás -, hanem horizontális. Üzeneteket csak egy kommunikációs környezetben belül lehet küldeni. Egy adott környezetben utazó üzenet más környezet számára nem látható és annak működését nem zavarja. Ez az eszköz ideális rutinkönyvtárak kialakításakor, hiszen annak felhasználója elől teljesen el lesz rejtve a benne zajló kommunikáció, még akkor is, ha az a program is MPI eljárásokat használ.

MPI-ban a csoportok és a környezetek összefogására szolgálnak a **kommunikátorok**.

<u>Csoportok</u>

Az MPI-ban folyamatokat csoportokba lehet összefogni, hogy ezeket együttesen lehessen kezelni a program hátralévő részében. A csoportok folyamatok rendezett halmazai, melyben minden folyamatnak egy sorszáma van. A sorszámozás mindig nullával kezdődik, és folyamatosan halad végig az összes folyamaton.

Ilyen csoportok az **MPI_Group** átlátszatlan típussal írhatóak le. Ez a leírás az adott folyamaton belül érvényes, nem lehet azt egy másik folyamatnak átadni.

A csoportok kialakítására szolgáló műveletek mind lokálisak, azaz nem igényelnek kommunikációt más folyamatokkal. Az így létrehozott csoportok folyamatonként eltérőek lehetnek.

Ha már vannak csoportjaink, akkor azokra többféle halmazműveletet el lehet végezni, és így kialakíthatóak új csoportok. A csoportokat össze lehet hasonlítani

(**MPI_Group_compare**), meg lehet tudni méretüket (**MPI_Group_size**), valamint le lehet kérdezni az adott folyamat sorszámát egy csoportban (**MPI_Group_rank**).

A nem használt csoportokat a **MPI_Group_free** művelettel lehet felszabadítani.

Kommunikátorok

Kommunikációs környezetek magukban nem hozhatóak létre, csak egy csoportra vonatkozóan lehet ilyen tulajdonságot definiálni. E két eszköz összefogásából alakulnak ki a kommunikátorok.

Minden kommunikációban meg kell adni egy ilyen kommunikátort. Új kommunikátort létre lehet hozni egy csoportból az **MPI_Comm_create** eljárással, vagy egy másik kommunikátorból. Az új kommunikátor létrejöttekor a benne lévő folyamatok egy kollektív művelet során kialakítanak egy új kommunikációs környezetet is, ezért az új kommunikátorban küldött üzenetek nem keveredhetnek a korábbiakkal.

Egy régi kommunikátor lemásolását végzi az **MPI_Comm_dup** eljárás. Ennek hatására a régi kommunikátor folyamat csoportja megmarad, de létrejön egy új kommunikációs környezet. Ez egy új - hosszabb ideig tartó, vagy többlépéses - eljárás indulásakor lehet hasznos, hogy a benne folyó kommunikáció ne keveredjen a program többi részével.

MPI_Comm_split művelettel lehet egy kommunikátort részekre bontani. Minden a műveletben résztvevő folyamat megmondhatja, hogy az új kommunikátorok közül melyikben szeretne benne lenni, és milyen sorszámot szeretne viselni.

Egy kommunikátort az **MPI_Comm_free** eljárással lehet felszabadítani.

4.3.5 MPI összefoglalás

Az MPI szabványon nagyon jól látszik, hogy egy jól átgondolt és sok szakértő közreműködésével létrehozott rendszer. A szabvány nevében megfogalmazott (üzenetküldés) területen szinte minden fontos gondolatot ötvöztek benne. A rendszerrel szinte mindenféle feladat megoldható. A könyvtár kifejezetten párhuzamos folyamatok közötti kommunikáció támogatására íródott, nem tartalmaz elemeket adat párhuzamosság kezelésére. A folyamatok közötti kommunikáció terén az összes üzenetküldés fajtára kínál megoldást, sőt ezek kollektív változataival sok problémára kínál előre elkészített, egyszerűsített megoldást.

Forrásmunkák:

Csárdi Gábor: PVM programok áthelyezése GRID környezetbe
PVM Home: http://www.csm.ornl.gov/pvm/pvm_home.html
PVM++: http://pvm-plus-plus.sourceforge.net/
PVM áttekintés: http://www.cab.u-szeged.hu/local/linux/pvm/pvm-guide.html
Frohner Ákos: Párhuzamos programozást támogató nyelvi eszközök összehasonlítása, http://www.geocities.com/madhousebbs/prog/parallel/e22.html