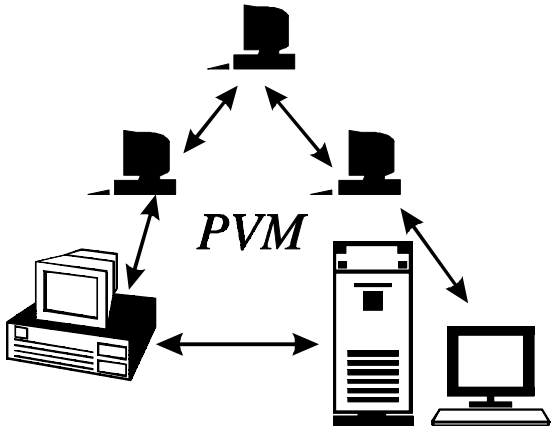
BUDAPESTI MŰSZAKI EGYETEM
Villamosmérnöki és Informatikai Kar
Folyamatszabályozási Tanszék

Parallel Virtual Machine

(3.3 verzió)

programozása



Összeállította:
Szigeti Szabolcs
Szeberényi Imre

© BME Folyamatszabályozási Tanszék

Budapest 1995.

TARTALOM

1. BEVEZETÉS 3
1.1 Mi a PVM ? 3
1.2 A PVM tulajdonságai: 3
1.3 A PVM felépítése: 4
1.4 Párhuzamos program fejlesztése 4
2. A PVM INSTALLÁLÁSA 5
2.1 A PVM installálása általános esetben 5
2.1.1 környezeti változók beállítása és egyéb beállítások 5
2.2 A PVM globális installációja 7
2.3 A PVM installálása a VSzK SUN10 gépén 9
2.4 A PVM installálása az ELTE SP1 gépén 9
3. A PVMD3 ÉS A PVM KONZOL 10
3.1 A pvmd3 közvetlen elindítása 10
3.2 A PVM konzol 12
4. PROGRAMFEJLESZTÉS 13
4.1 Rendszerállományok 13
4.2 Aimk 13
5. A LIBPVM HÍVÁSOK 14
5.1 Virtuális gép vezérlése 14
5.1.1 Hosztok felvétele a virtuális gépbe, hosztok törlése a virtuális gépbôl 14
5.1.2 A virtuális gép konfigurációjának lekérdezése 14
5.1.3 Virtuális gép felépítése és megszüntetése 15
5.1.4 Fizikai gép állapotának lekérdezése 15
5.1.5 Speciális funkciók 15
5.2 Taszk vezérlés 16
5.2.1 Belépés a virtuális gépbe, kilépés a virtuális gépbôl 16
5.2.2 Taszkot indítása és megállítása 16
5.2.3 Szülô taszk lekérdezése 17
5.2.4 Folyamatok állapotának lekérdezése 18
5.3 Üzenet küldés 18
5.3.1 Pufferkezelés 18
5.3.2 Üzenetek összeállítása és szétszedése 19
5.3.3 Üzenetek elküldése 20
5.3.4 Üzenetek vétele 22
5.3.5 Üzenet azonosító összehasonlítás 22
5.3.6 Várakozó azonosító 23
5.4 Csoportok 23
5.4.1 Belépés a csoportba, kilépés a csoportból 23
5.4.2 Csoport méretének megállapítása 23
5.4.3 Azonosító konverziók 24
5.4.4 Barrier szinkronizáció 24
5.4.5 Broadcast 24
5.4.6 Tömb műveletek 24
5.5 Egyéb funkciók 25
5.5.1 Adatbázis kezelés 25
5.5.2 Hoszt megállapítása 26
5.5.3 Nyomkövetési flag-ek beállítása 26
5.5.4 Adatábrázolás megállapítása 26
5.5.5 Kimenet átirányítás 26
5.5.6 Libpvm konfigurálása 27

© BME Folyamatszabályozási Tanszék

5.5.7 Valós idô szinkronizálása 27
5.5.8 Pvm események figyelése 28
5.5.9 Hibajelzés 28
5.6 Hibakódok 29
6. EGY PÉLDA 30

1. Bevezetés

Napjainkban egyre nagyobb és nagyobb számítási teljesítményre van szükség a tudományos technikai számításokhoz. A számítási teljesítmény bizonyos esetekben hatékonyan növelhetô párhuzamos feldolgozással. A párhuzamos feldolgozásra képes rendszerek általában célrendszerek, melyeknek a beszerzési és üzemeltetési költségei igen magasak lehetnek.

Az általános céllal gyártott processzorok és számítógépek között is kialakítható párhuzamos feldolgozás, ha létrehozzuk a megfelelô kapcsolatot. Természetesen ekkor párhuzamosítás csupán program (függvény) szinten valósulhat meg, ami általában nem olyan hatékony, mint utasítás szinten. Nagy elônye viszont, hogy a meglevô berendezések felhasználhatók. Ilyen típusú párhuzamosításra ad lehetôséget az *Oak Ridge National Laboratory* **PVM** (**P**arallel **V**irtual **M**achine) rendszere. A rendszer szinte tetszôleges UNIX operációs rendszerrel működô gépre telepíthetô, és segítségével könnyen kipróbálhatóak a különbözô párhuzamos algoritmusok.

1.1 Mi a PVM ?

Lényegében egy eljárásgyűjtemény, amely lehetôvé teszi, hogy az elsôsorban számításigényes, de párhuzamosan is végrehajtható feladatok valóban egymással párhuzamosan hajtódjanak végre. A párhuzamosan végrehajtható tevékenységek kijelölése a programozó feladata, a PVM csupán azt az eszközkészletet nyújtja, amellyel az egyes párhuzamos feladatokat megvalósító folyamatok egyszerűen kommunikálhatnak egymással. Az egyes folyamatok futhatnak ugyanolyan típusú processzoron, vagy akár teljesen más architektúrájú gépen is.

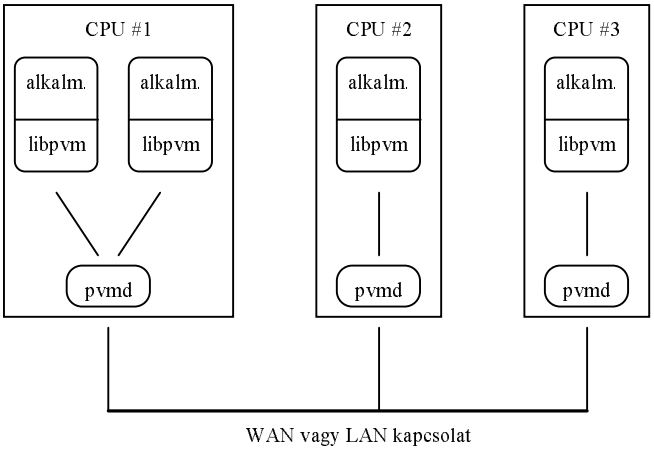
1.2 A PVM tulajdonságai:

- UNIX alapú operációs rendszer környezetet tételez fel.
- TCP/IP hálózati protokollra épül.
- Különbözô hardver platformokra épülô UNIX rendszerek is összekapcsolhatók segítségével egy virtuális géppé. Több mint 20 architektúrán tesztelték már.
- A program különbözô komponensei különbözô típusú gépeken futhatnak.
- Automatikusan biztosítja a különbözô számábrázolási formátumokhoz szükséges konverziókat (ha ez szükséges).
- C, C++ és FORTRAN nyelvi interfésszel rendelkezik.
- Kis méretű. (Forrása nem több mint 2 Mb.)
- Publikus rendszer.
- Forráskódja hozzáférhetô az Internet hálózaton keresztül anonymous ftp segítségével. Magyarországról a fix.fsz.bme.hu címrôl a /pub/pvm katalógusból tölthetô le.
- Egyszerű installálni.
- Egyetlen egy PVM daemon fut minden gépen (felhasználónként), és ez elvégzi az összes feladatot.

© BME Folyamatszabályozási Tanszék

1.3 A PVM felépítése:

A PVM két komponensbôl épül fel. Gépenként és felhasználónként létezik egy *pvm daemon*nak (**pvmd**) nevezett program, amely feladata az egyes taszkok közötti kapcsolattartás. A pvmd végzi a gép-gép kommunikációs és szükség esetén biztosítja az adatábrázolásbeli különbséget eltüntetô konverziókat.



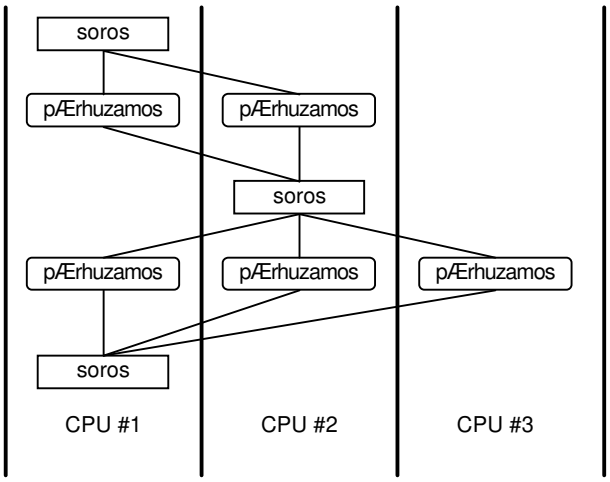
1. ábra: *a PVM felépítése*

Minden taszk csak a pvmd-on keresztül kommunikálhat egymással. A taszkok és a pvmd közötti kommunikációt a *libpvm*-nek nevezett eljárásgyűjtemény végzi. Ezt minden taszkhoz hozzá kell szerkeszteni, ez tartalmazza a kommunikációt megvalósító rutinokat, illetve az ennek megfelelô programozói felületet.

1.4 Párhuzamos program fejlesztése

A PVM rendszerben a párhuzamosítás alapegysége a folyamat[1]. Ennek ismeretében látható, hogy elsôsorban azok az alkalmazások párhuzamosíthatók sikeresen, ahol viszonylag nagy részekre tudjuk a feladatot szétszabdalni.

A PVM alatt a folyamatok közötti kommunikáció “drága”, ezért ezt kell minimalizálni. További tulajdonsága a PVM-nek, hogy mivel heterogén környezetben dolgozni, figyelembe kell venni a rendelkezésre álló számítógépeket. Elképzelhetô, hogy egyes típusú gépek lebegôpontos számítási teljesítménye, míg másokan I/O teljesítménye nagy.



2. ábra: *Egy párhuzamos program*

Ezeknek a feltételeknek ismeretében a megvalósítandó algoritmusban kiválaszthatjuk a párhuzamosan futtatható részeket, illetve a köztük szükséges kommunikációt megtervezhetjük. Ki kell választani a két alapvetô felépítés egyikét: a master-slave elrendezés a gyakoribb, de a PVM támogatja az egyenrangú modulokból álló programok kialakítását is.

[1] Azaz UNIX esetén egy önálló program.

© BME Folyamatszabályozási Tanszék

Ezután történhet a program megírása, a PVM megfelelô kommunikációs, szinkronizációs primitívjeinek felhasználásával. Az elkészült programon méréseket végezhetünk az esetleges szűk keresztmetszetek megtalálására és ezek alapján módosíthatjuk a felépítését.

2. A PVM installálása

2.1 A PVM installálása általános esetben

Az üzembe helyezés elsô lépéseként meg kell szerezni a PVM forrását. Ez elérhetô e-mailen keresztül a ***netlib@ornl.gov*** illetve *anonymous ftp* segítségével az ***ftp.netlib.gov:/netlib/pvm3*** vagy a ***fix.fsz.bme.hu:/pub/pvm/pvm3*** helyrôl. Magyarországról a gyorsabb kapcsolat miatt ez utóbbi javasolt.

A forrás általában egy UNIX **tar(1)** állományban van. Ezt a saját home könyvtárunkból kicsomagolva, létrehoz egy pvm3 nevű könyvtárat[2]. Ebbe az alkönyvtárba kerül a rendszer teljes forrása, és itt jönnek létre különbözô architektúráknak megfelelô futtatható programok is. A továbbiakban azt a könyvtárat, ahol a PVM van `PVM_ROOT` néven fogjuk hívni.

Következô lépésként lépjünk be ebbe a könyvtárba és adjuk ki a `make all` parancsot. Ezután a fordítás és az installálás automatikusan végrehajtódik. Sikeres fordítás után a PVM rendszer már használható. A rendszer használatba vétele elôtt azonban ajánlatos áttekinteni a rendszer komponenseinek kapcsolatát, és az esetleg szükséges beállításokat elvegezni. A következôkben ehhez nyújtunk némi segítséget.

2.1.1 környezeti változók beállítása és egyéb beállítások

A PVM_ROOT környezeti változó helyes beállítását a rendszer több komponense feltételezi. A kényelmes munkavégzés érdekében ajánlott ezt beállítani, amit a `setenv PVM_ROOT pvm_könyvtár` (**csh(1)**, **tcsh(L)** esetén) illetve `PVM_ROOT=pvm_könyvár; export PVM_ROOT` (**sh(1)**, **bash(L)** esetén) paranccsal érhetjük el. Amennyiben az általunk használt shell **csh(1)** vagy **tcsh(L)** akkor érdemes megvizsgálnunk a `pvm3/lib/cshrc.stub` nevű állományt és beilleszteni a saját `.cshrc` állományunkba, mivel így bejelentkezéskor automatikusan beállítódik a `PVM_ROOT`, `PVM_ARCH` és a `path` környezeti változó.

Fontos tudnunk, hogy a virtuális gépet alkotó gépek mindegyikén futnia kell a pvm daemonnak (pvmd). Ezeket automatikusan indítja el az elsôként elindított daemon. Az automatikus indítás *rsh* vagy *rexec* protokollal történik. Ehhez a távoli gépen be kell lépnie az indító processznek. A belépéshez megfelelô jogosultság vagy jelszó szükséges.

---

[2]A PVM máshol is elhelyezhetô, azonban javasolt a $HOME/pvm3, mert több program is arra számít, hogy itt található. Ha máshová tesszük, érdemes egy szimbolikus linket csinálni rá $HOME/pvm3 néven. Hasonlóan érdemes eljárni akkor is, ha globálisan installáljuk a PVM rendszert.

A jelszó szükség esetén a virtuális gép konfigurációs állományában (hostfile) adható meg ami természetesen nem túl elegáns megoldás. Ettôl jobb megoldás, ha használjuk távoli gépeken a *.rhosts* állományt, amiben engedélyezhetô a jelszó nélküli belépés és távoli végrehajtás. A *.rhosts* állomány egy szöveges állomány, amely tartalmazza azoknak a gépeknek a nevét és a felhasználók azonosítóját, melyekrôl a távoli végrehajtás és belépés jelszó nélkül lehetséges. Fontos tudni, hogy biztonsági okokból egyes UNIX rendszerek csak akkor veszik figyelembe a *.rhosts* állományt, ha annak a védelmi attribútuma 400 vagy 600.

Mivel a daemont PVM_ROOT környezeti változó alapján találja meg a távoli végrehajtáskor a rendszer, ezért alapvetô fontosságú ennek a változónak a helyes beállítása még akkor is, ha nem interaktív üzemmódban lépünk be (*rsh*). Ezt azért fontos tudni, mert egyes shell programok nem hajtanak végre semmilyen inicializáló állomány távoli shell használatakor. Ilyen pl. a Korn Shell (ksh). Más shell-ek viszont végrehajtják ugyan pl. a *.cshrc* állományt, de nem az egészet. Ha olyan shell-t használunk, amely nem futtat le a távoli shell indításakor semmilyen inicializáló script-et, akkor a problémát úgy tudjuk megoldani, hogy a virtuális gép konfigurációs állományába explicit módon be kell írni a daemon elérési útvonalát.

Az eredeti PVM csomag nem ajánl megoldást arra a problémára, hogy a PVM rendszert használó felhasználók gyakran felejtik a virtuális gépet "bekapcsolva", azaz a deamon programok az egyes gépeken üzemelnek annak ellenére, hogy semmilyen alkalmazást nem futtatnak. Ezen egyszerűen segíthetünk, ha olyan shell-t használunk, ami kilépéskor képes egy script-et lefuttatni. Ilyen pl. a csh *.logout* script-je. Ebbe egyszerűen betehetjük, hogy kilépés elôtt ellenôrizze, hogy fut-e a virtuális gép. Ha igen, akkor kérdezzen rá, hogy le kell-e állítani. Ehhez nyújt segítséget az itt ismertetett *ispvmrun* és *.pvmlogout* script:

```
#!/bin/sh
# ispvmrun script (Sz.I., 1995.09.25.)
# 1 ertekkel ter vissza, ha a PVM fut
TMPFIL=/tmp/pvmd
UID=`id | sed -e 's/uid=\(.*\)(.*) gid=.*/\1/'`
if [ -r $TMPFIL.$UID ]; then
   echo 'The PVM daemon is running'
   exit 1       # visszateresi ertek = false
fi
exit 0          # visszteresi ertek = true
```

```
# .pvmlogout script (Sz.I., 1995.09.25)
# Fuzze be ezt a scriptet a .logout file-ba! Ha a PVM fut a
rendszerbol
# valo kilepeskor, akkor automatiikusan behivodik a pvm
konzol.
# Ekkor eldontheti, hogy a PVM megalljon vagy ne.
#
ispvmrun || pvm
```

© BME Folyamatszabályozási Tanszék

2.2 A PVM globális installációja

Az olyan rendszerekben, ahol több felhasználó is használja a PVM rendszert, célszerű a PVM egyes komponenseit globálisan a rendszerben mindenki számára elérhetôen installálni. Sajnálatos módon erre nem igen ad tanácsot az eredeti programcsomag README állománya. Az általunk javasolt megoldás az, hogy a teljes rendszert (a források kivételével) célszerű a */usr/local/pvm* elhelyezni. Ezután helyezzünk el a /usr/local/bin katalógusban egy egyszerű installáló shell script-et, amelyet a PVM rendszert használni kívánó felhasználók lefutathatnak. Ez a shell script egyszerű szimbolikus linkekkel létrehozza minden felhasználó számára a szükséges felhasználói állományokat és katalógusokat. Ezután már csak arról kell gondoskodnia a PVM felhasználóknak, hogy a a PVM_ROOT környezeti változót a /usr/local/pvm katalógusra állítsák be. Ez megfelelô shell script-ekkel automatikusan a belépéskor biztosítható. Az installáló shell script-re a következôkben adunk egy példát:

```
#!/bin/sh
# instpvm script (Sz.I., 1995.09.25)
# PVM rendszer felhasznalonkenti installacioja globalis
installaciohoz
pvm_home=/usr/local/pvm
linked_bins='pvm_gstat pvmgroups pvmgs'

echo -n 'Akarja installani a PVM rendszert a login
katalogusaban (I/N) ?'
read answ
case $answ in
   'i' | 'i') ;;
   *) exit;;
esac

echo
pvm_arch=`$pvm_home/lib/pvmgetarch`
cd
if [ -d pvm3 ]; then
     echo "PVM mar installalva"
     exit
fi
mkdir pvm3
cd pvm3
mkdir bin
mkdir bin/$pvm_arch
for i in $linked_bins
do
ln -s $pvm_home/bin/$pvm_arch/$i bin/$pvm_arch
done
cd
cp $pvm_home/lib/pvmcshrc .pvmcshrc
cp $pvm_home/lib/pvmlogout .pvmlogout
```

© BME Folyamatszabályozási Tanszék

```
echo 'A PVM rendszer installalva van a login
katalogusaban.'
echo 'A rendszer futtatasahoz eloszor allitsa be a PVM_ROOT
kornyezeti valtozot'
echo 'Ezt pl. a ~/.pvmcshrc script hasznalataval erheti
el:'
echo '  A ~/.cshrc file-ba irja be az altalanos beallitasok
utan a'
echo '  "source ~/.pvmcshrc" paranccsot. Vigyazzon, hogy a
fenti parancs mind'
echo '  interaktiv, mind nem interaktiv modban lefusson!
Elofordulhat, hogy a'
echo '  fenti script megzavarja a .login vagy a .cshrc
mukodeset!'
echo
echo 'Ne felejtse el a virtualis gep leallitasat a munka
befejeztevel!'
echo 'Az automatikus leallitashoz hasznalhatja a
~/.pvmlogout shell scriptet'
echo 'a ~/.logout file-bol!'
echo
echo 'Ha nem csh-t, vagy tcsh-t hasznal, akkor
elofordulhat, hogy nem tudja'
echo 'automatikusan beallitani a PVM_ROOT kornyezeti
valtozot, mivel pl. a ksh'
echo 'nem futtaja le nem interaktiv modban a .kshrc file-t
ilyen esteben hasznalja'
echo "a '* dx=$pvm_home/lib/pvmd' sort a hostfile-ban!"
echo
echo 'Sok sikert!'
```

```
# .pvmcshrc script (Sz.I., 1995.09.25)
# Fuzze be ezt a scriptet a .cshrc file-ba!
#

set MANP = /usr/man # the default man directory
if (! $?PVM_ROOT) then
     if (-d ~/pvm3) then
          setenv PVM_ROOT /home/vsz/szebi/pvmsrc
          setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
          if ($?MANPATH) then
               setenv MANPATH $MANPATH':'$PVM_ROOT/man
          else
               setenv MANPATH $MANP':'$PVM_ROOT/man
          endif
     else
          echo "Error - PVM not installed on yor directory"
          echo "To use PVM, run instpvm script."
```

© BME Folyamatszabályozási Tanszék

```
     endif
endif

if ($?PVM_ROOT) then
          set path=($path $PVM_ROOT/lib)            #
generic
          set path=($path $PVM_ROOT/lib/$PVM_ARCH)  # arch-
specific
          set path=($path ~/pvm3/bin/$PVM_ARCH)     # local
arch-specific
endif
```

2.3 A PVM installálása a VSzK SUN10 gépén

A VSzK-ban a globális PVM installációhoz hasonló installációt alkalmaztunk, azzal a különbséggel, hogy adminisztratív okokból a PVM rendszer nem a /usr/local/pvm katalógusba van installálva, hanem a */home/vsz/szebi/pvm* katalógusba. A felhasználói katalógusokat, és a szükséges link-eket a */home/vsz/szebi/instpvm* script segítségével lehet létrehozni. Ez a script-et futtató felhasználó login katalógusában létrehoz egy pvm3 könyvtárat, és azon belül néhány linket. A script a login katalógusba másolja továbbá a .pvmschrc, .pvmbashrc, .pvmchslogout, .pvmbashlogout állományokat is, amelyeket a használt shell-nek megfelelôen célszerű a .cshrc ill. .bashrc és a .logout ill. .bash_logout állományokból meghívni.

2.4 A PVM installálása az ELTE SP1 gépén

Az ELTE számítóközpontjában üzemelô SP1 gépen a publikus pvm a /usr/local/pvm katalógusba lett installálva. Az egyes felhasználók a /usr/local/bin script segítségével hozhatják létre a szükséges lokális állományokat. Az elôzôekben leírtakhoz hasonlóan itt is célszerű a megfelelô shell inicializáló script-eket alkalmazni.

3. A pvmd3 és a PVM konzol

Minden fizikai gépen felhasználónként kell egy pvmd3-nak futni. A daemon elindítható közvetlenül, a PVM konzolból és PVM alkalmazásból is.

3.1 A pvmd3 közvetlen elindítása

A daemont a `pvmd [-opciók] [hostfile]` paranccsal indíthatjuk el. Az opciók a következôk lehetnek:

`-d`***mask*** A debug maszkot ***mask*** hexadecimális értekre állítja. A mask a következô bitekböl állhat:

1	Packet továbbítás
2	Üzenet továbbítás
4	Taszk kezelés
8	Slave pvmd-ok indítása
10	Hoszt táblázat figyelése
20	Alsószintű üzenetkezelés
40	IP hálózatkezelés
80	Multiprocesszor nyomkövetés
100	Erôforrás kezelés
200	Alkalmazási hibák

`-n`***name*** A fô pvmd számára a ***name*** hálózati nevet írjuk elô. Ez az opció akkor használatos, ha a **gethostname()** hibás értéket adna egyébként.

Léteznek még további pvmd opciók is, azonban ezeket csak a master pvmd használja slave pvmd-ok indításakor. Akkor kell használni, ha egyéni hoszter alkalmazást írunk (lásd: `pvm_reg_hoster` libpvm hívás), vagy kézzel építjük fel a virtuális gépet[3].

A `hostfile` a virtuális gépbe felveendô hosztokat és az erre vonatkozó egyéb információkat tartalmazza.

A hostfile egyes sorai egy hosztra vonatkozó információt tartalmaznak a következô formában:

hálózati-név [erre a gépre vonatkozó opciók]

Ha névként *-ot adunk, akkor az ebben a sorban szereplô opciók az összes gépre vonatkoznak. Késôbb ezeket gépenként felülbírálhatjuk.

A # karakterrel kezdôdô sorok megjegyzések.

Az opciók lehetnek:

`lo=NAME` A login név `NAME` legyen. Akkor kell használni, ha az adott gépen más a login nevünk, mint ahonnan a pvm-et indítjuk.

[3]Lásd a PVM users'guide-ot.

`so=pw` Ha az adott gépre csak jelszó megadásával lehet belépni, akkor a master pvmd kérni fogja a jelszót.

`so=ms` Manuális indítás. Ha IP kapcsolat igen, de távoli végrehajtás nincs a két gép között, akkor kell alkalmazni.

`dx=FILE` Ha az adott gépen a pvmd nem a `$PVM_ROOT/lib/pvmd`, akkor itt megadhatjuk az elérési útvonalát.

`ep=PATH` Hol keresse a pvmd az elindítandó taszkokat.

`wd=PATH` Az elindítandó taszkok aktuális munkakatalógusa `PATH` legyen.

`sp=SPEED` Az adott hoszt relatív sebességét specifikálja. Értéke 1..1000000.

`bx=PATH` A debugger elérési útvonala.

A $ jellel kezdôdô szavak a megfelelô környezeti változó értékével helyettesítôdnek.

Az opciók alapértékei:

`lo` A login név a master hoszton.

`so` semmi

`dx` `$PVM_ROOT/lib/pvmd` vagy `$PVM_DPATH`.

`ep` `pvm3/bin/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH`

`wd` `$HOME`

`sp` `1000`

`bx` `$PVM_ROOT/lib/debugger`

Az alapértelmezések megváltoztathatók a hostfile hátralévô részére, ha hosztnév helyett *-ot adunk meg.

Ha egy sor & jellel kezdôdik, akkor az információ eltárolódik, de a hoszt csak a pvm_addhost hatására lép be a virtuális gépbe.

Példa hostfile:

```
# Az elsô két géppel nincs probléma
eniac.fsz.bme.hu
odra.fsz.bme.hu
# Itt máshol van a pvmd.
zx80.asd.bme.hu dx=/usr/local/pvm3/lib/pvmd
# Erre a kettôre pvmuser néven kell belépni, és jelszót kérnek.
# Itt megváltoztatjuk a lo és so paraméterek alapértelmezését is.
* lo=pvmuser so=pw
beavis.vbn.bme.hu
butthead.ert.bme.hu
# Nehéz eset. Kézzel kell indítani, más login néven. Máshol
# vannak az alkalmazások is.
&          secure.kgb.su          lo=nikolai          so=ms
ep=/home/beria/pvm3/bin/$PVM_ARCH
# hostfile vége
```

© BME Folyamatszabályozási Tanszék

3.2 A PVM konzol

A virtuális gép felépítésének másik módja a PVM konzol. Segítségével a virtuális gépet interaktívan vezérelhetjük, és bizonyos nyomkövetési és hibakeresési funkciókat is ellát.

A PVM konzol a `pvm[-opciók][hostfile]` paranccsal indítható. Elindulása után, ha a virtuális gép még nincs felépítve, elindítja a helyi daemont, átadva neki a parancssori opciókat. Így a konzol tetszés szerint elindítható vagy megállítható a virtuális gép befolyásolása nélkül. A konzol végrehajtja a `$HOME/.pvmrc` állományban található parancsokat.

A PVM konzolban a következô parancsokat használhatjuk:

`add` ***host*** felveszi a ***host*** nevű gépet a virtuális gépbe.

`alias [`***alias***`] [`***cmd***`]` aliasokat hozhatunk létre. Argumentum nélkül: aktuális aliasok.

`conf` a virtuális gép konfigurációját kérdezi le.

`delete` ***host*** törli a ***host*** nevű gépet a virtuális gépbôl.

`echo [`***arg***`]` Kiírja az argumentumot.

`halt` Megszünteti a virtuális gépet

`help [`***cmd***`]` Bôvebb információt ad a ***cmd*** parancsról.

`id` Megadja a konzol TID-jét.

`jobs` A futó Pvm alkalmazások listáját adja.

`kill` ***task*** Megöli a ***task*** taszkot.

`mstat` ***host-TID*** Az adott hoszt statusát adja.

`ps [-a]` Kiírja a virtuális gépben futó folyamatokat

`pstat` ***task*** Az adott taszk statusát adja.

`quit` Kilép a PVM konzol, de a virtuális gép megmarad.

`reset` Alapállapotba hozza a virtuális gépet.

`setenv [`***var*** `[`***value***`]]` Környezeti változó beállítása/lekérdezése.

`sig` ***signal task*** ***Signal*** jelet elküldi a ***task***-nak.

`spawn [`***opt***`]` ***task*** Elindítja a ***task*** nevű futtatható állományt a virtuális gépben.

opt:

- `-(`***count***`)` Elindítandó taszkok száma, alapérték: 1
- `-(`***host***`)` Cél hoszt
- `-(`***ARCH***`)` Cél architektúra
- `-?` Nyomkereséssel indít
- `->` A taszk kimenete a konzol lesz
- `->`***file*** A taszk kimenetét a ***file***-ba irányítja
- `->>`***file*** A taszk kimenetét a ***file***-hoz fűzi

© BME Folyamatszabályozási Tanszék

`trace [`***`mask`***`]`	A nyomkövetési maszkot állítja be.
`unalias`	*Alias* definíciót megszüntet
`version`	Megadja a *libpvm* verziószámát.

Ha a taszkok kimenetét nem irányítjuk át, akkor minden távoli folyamat kimenete az elsônek indított (mester) pvmd-ra kerül. A daemon ezeket - az adott task tid-jével megjelölve - a `/tmp/pvml.xxx` állományba írja, ahol **xxx** a felhasználó UID-je.

4. Programfejlesztés

4.1 Rendszerállományok

A PVM heterogén környezetben működik, ezért szükség van a különbözô architektúrákhoz tartozó bináris állományok elkülönítésére. Gyakori eset, hogy a felhasználó bejelentkezési könyvtára (`$HOME`) a hálózati fájlrendszer (NFS) segítségével úgy van szervezve, hogy több különbözô géprôl is ugyanazt a könyvtárat látja. Ebben az esetben elôforduló névütközéseket a PVM úgy oldja fel, hogy a különbözô architektúrák alá lefordított bináris állományokat más-más könyvtárban helyezi el. Jelölje a `PVM_ROOT` változó a pvm gyökérkönyvtárát, a `PVM_ARCH` pedig az adott architektúrának megfelelô azonosítót (pl.: `RS6K` - IBM risc/6000, `SUN4` - Sun Sparc stb.). Ebben az esetben a futtatható állományok a `$PVM_ROOT/bin/$PVM_ARCH/` könyvtárban, a PVM library pedig a `$PVM_ROOT/lib/$PVM_ARCH/` könyvtárban lesznek. A `pvm.h` include állomány rendszerfüggetlen, helye a `$PVM_ROOT/include/` könyvtár. Ezek alapján, ha a `elso.c` állományt futtatható PVM programmá kívánjuk alakítani a következô parancsot kell kiadni:

```
cc -o$PVM_ROOT/bin/$PVM_ARCH/elso -I$PVM_ROOT/include
   elso.c -L$PVM_ROOT/lib/$PVM_ARCH -lpvm
```

Feltéve természetesen, hogy az adott változókat helyesen beállítottuk.

4.2 Aimk

Programfejlesztésre a PVM alatt is használható a **make(1)** parancs. Azért azonban, hogy ne kelljen mindig beállítani a szükséges környezeti változókat, érdemes használni az **aimk** parancsot. Az **aimk** nem tesz mást, mint beállítja a `PVM_ARCH` és a `PVM_ROOT` változókat az adott gépnek megfelelôen, majd meghívja a **make(1)** programot bemeneti állományként átadva a `Makefile.aimk` make állományt. A makefile-ban ezekre a változókra hivatkozva architektúra független makefile-t alakíthatunk ki.

© BME Folyamatszabályozási Tanszék

5. A libpvm hívások

Ez a fejezet a pvm library hívásait ismerteti. A pvm jelenleg a C, C++ és a Fortran nyelveket támogatja. Bizonyos függvények csak a C interfészben állnak rendelkezésre, Fortran megfelelôjük nincs.

5.1 Virtuális gép vezérlése

5.1.1 *Hosztok felvétele a virtuális gépbe, hosztok törlése a virtuális gépbôl*

A virtuális gép több fizikai gépbôl (hoszt) állhat. A virtuális gépet dinamikusan konfigurálhatjuk a következô két hívással:

```
int info = pvm_addhost(char **hosts, int nhost, int
         *infos)
CALL PVMFADDHOST(HOST, INFO)
```

Az addhost újabb fizikai gép(ek)et ad a virtuális géphez. A gépek hálózati neveit a `hosts` tömb, számát az `nhost` paraméter tartalmazza. Az `infos` tömb siker esetén az adott hoszt pvmd tid-jét, hiba esetén a hibakódot tartalmazza. Az `info` visszatérési érték a virtuális gépbe sikeresen felvett hosztok számát adja. A fortran hívás egyszerre csak egy gép felvételét teszi lehetôvé.

```
int info = pvm_delhost(char **hosts, int nhost, int
         *infos)
CALL PVMFDELHOST(HOST, INFO)
```

A delhost eltávolít egy vagy több fizikai gépet a virtuális gépbôl. Paraméterezése megegyezik az addhost-éval. Az adott gépeken minden pvm folyamat megáll.

5.1.2 *A virtuális gép konfigurációjának lekérdezése*

```
int info = pvm_config( int *nhost, int *narch,
         struct pvmhostinfo **hostp )
struct pvmhostinfo {
         int hi_tid;
         char *hi_name;
         char *hi_arch;
         int hi_speed;
         };
CALL PVMFCONFIG(NHOST, NARCH, DTID, NAME, ARCH,
         SPEED, INFO)
```

A config segítségével információt szerezhetünk a virtuális géprôl. A C nyelvű hívás egyszerre visszaadja a virtuális gép konfigurációját, míg a fortran függvényt ciklikusan meghívva egymás után egy-egy gép adatait adja meg. A virtuális gépben szereplô fizikai gépek számát az `nhost`, az architektúrák számát pedig az `narch` paraméter adja. Az egyes gépekrôl további információt ad a `hostp` struktúratömb. A `hi_tid` az adott pvmd tid-jét, a `hi_name` a hoszt nevét, a `hi_arch` az hoszt architektúrájának nevét végül a `hi_speed` a sebességét tartalmazza. Ez utóbbi paramétert megegyezik a

virtuális gép indításakor megadott értékkel. A fortran hívásban a hasonló nevű paramétereket használjuk.

5.1.3 *Virtuális gép felépítése és megszüntetése*

Virtuális gépet két módon hozhatunk létre: kézzel indítva a pvmd-t, vagy a start_pvmd hívással, amely elindítja a helyi pvmd-t és a hívót belépteti a virtuális gépbe.

```
int info = pvm_start_pvmd(int argc, char **argv,
         int block)
FORTRANBAN NEM HASZNÁLHATÓ
```

Az `argc` ill. `argv` specifikálja a helyi pvmd-nak átadandó parancssori paramétereket a C nyelv konvenciói szerint. A `block` 0-tól különbözô értéke esetén megvárja a pvmd elindulását. Ha `info==PvmOk`, akkor nem történt hiba, ha `info==PvmDupHost`, akkor a pvmd már létezett, minden egyéb érték pedig hibát jelez. A hívásnak nincsen fortran megfelelôje.

```
int info = pvm_halt(void)
CALL PVMFHALT(INFO)
```

A halt függvény megszünteti a virtuális gépet, leállítja az összes pvm folyamatot (beleértve a hívót is) és pvmd-t. Bemenô paramétere nincs. Ha nem szeretnénk a hívó folyamat megállítását, akkor a SIGTERM szignált vagy elfogni, vagy ignorálni kell.

5.1.4 *Fizikai gép állapotának lekérdezése*

```
int mstat = pvm_mstat(char *host)
CALL PVMFMSTAT(HOST, MSTAT)
```

Az mstat a virtuális gép host nevű tagjáról ad felvilágosítást. Az mstat visszatérési értéke lehet:

`PvmOk`	A hoszt tagja a virtuális gépnek, és működik.
`PvmNoHost`	A hoszt nem tagja a virtuális gépnek
`PvmHostFail`	A hoszt nem elérhetô

5.1.5 *Speciális funkciók*

A következô három hívás a pvm mélyebb ismeretét feltételezi. Segítségükkel olyan alkalmazásokat írhatunk (pl.: debugger, monitor), amelyek más módon nem lehetségesek.

```
#include <pvmsdpro.h>
int cc = pvm_reg_hoster(void)
FORTRANBAN NEM HASZNÁLHATÓ
```

A hívó taszk pvmd indító taszkká válik. Ha késôbb további hosztot veszünk fel a virtuális gépbe, akkor az újabb pvmd indítása nem történik automatikusan, hanem ez a taszk kap egy speciális üzenetet, és elvégezheti az új pvmd indítását. Fortran megfelelôje nincs.

© BME Folyamatszabályozási Tanszék

```
#include <pvmsdpro.h>
INT CC = PVM_REG_TASKER(VOID)
FORTRANBAN NEM HASZNÁLHATÓ
```

A hívó taszk pvm taszk indítóvá válik. Ha újabb taszkot indítunk, akkor ezt nem történik meg automatikusan, hanem ez a taszk egy speciális üzenetet kap, és elvégezheti az új folyamat indítását. Fortran megfelelôje nincs.

```
#include <pvmsdpro.h>
int cc = pvm_reg_rm(struct pvmhostinfo **hip)
STRUCT PVMHOSTINFO {
         INT HI_TID;
         CHAR *HI_NAME;
         CHAR *HI_ARCH;
         INT HI_SPEED;};
FORTRANBAN NEM HASZNÁLHATÓ
```

A hívó taszk erôforrás menedzserré válik. Lehetôséget ad beavatkozásra az ütemezésbe és terheléselosztásba. Fortran megfelelôje nincs.

5.2 **Taszk vezérlés**

5.2.1 *Belépés a virtuális gépbe, kilépés a virtuális gépbôl*

Minden virtuális gépben részt vevô folyamat egy 32 bites folyamatazonosítót (tid) kap, amely az egész virtuális gépben egyedi. Bármely pvm hívás belépteti a hívót a virtuális gépbe.

```
int tid = pvm_mytid(void)
CALL PVMFMYTID(TID)
```

A hívó folyamatot lekérdezheti saját tid-jét a `tid` visszatérési értékben.

```
int info = pvm_exit(void)
CALL PVMFEXIT(INFO)
```

A hívó taszkot kilépteti a virtuális gépbôl. A UNIX folyamat futása tovább folytatódik.

5.2.2 *Taszkot indítása és megállítása*

A pvm folyamatok szülô-gyermek viszonyban vannak egymással. Bármely folyamat újabb folyamatokat indíthat, illetve állíthat meg.

```
int numt = pvm_spawn(char *task, char **argv, int
         flag, char *where, int ntask, int *tids)
CALL PVMFSPAWN(TASK, FLAG, WHERE, NTASK, TIDS,
         NUMT)
```

A task nevű futtatható állományt elindítja és belépteti a virtuális gépbe. Ha a nevet nem abszolút elérési útvonalként specifikáljuk, akkor a pvm a `$HOME/pvm3/bin/$PVM_ARC/` könyvtárban keresi. Az `argv` paraméterben átadhatunk parancssori paramétereket. A folyamat `ntask` példányban indul el. A `flag`

paraméterrel befolyásolhatjuk az indítás körülményeit. A következô értékeknek összegeit veheti fel a C nyelvű hívásban:

`PvmTaskDefault`	a pvm választja ki, hogy hol indul a taszk.
`PvmTaskHost`	a `where` paraméter specifikálja a hosztot.
`PvmTaskArch`	a `where` paraméter specifikálja az architektúrát.
`PvmTaskDebug`	a taszk debugger alatt indul
`PvmTaskTrace`	a taszk nyomkövetési információt generál
`PvmMppFront`	multiprocesszoros gép front-endjén indul
`PvmHostCompl`	komplemens hosztokon indul

Természetesen az elsô három érték egymást kizárja.

A fortran hívásban a következô értékeket veheti fel:

`PVMDEFAULT`	mint a PvmTaskDefault
`PVMHOST`	mint a PvmTaskHost
`PVMDEBUG`	mint a PvmTaskDebug
`PVMTRACE`	mint a PvmTaskTrace

A `numt` visszatérési érték az elindított taszkok számát adja. Ezeknek a tid-jét a `tids` tömbben találjuk. Hiba esetén ez a tömb a hibakódokat tartalmazza.

Az elindítandó folyamatoknak a `PVM_EXPORT` környezeti változóban felsorolt environment átadódik.

```
int info = pvm_kill(int tid)
CALL PVMFKILL(TID, INFO)
```

A `tid` által azonosított folyamatnak egy SIGTERM szignált küld, és ezzel megöli az adott UNIX folyamatot.

5.2.3 *Szülô taszk lekérdezése*

```
int tid = pvm_parent(void)
CALL PVMFPARENT(TID)
```

A hívó szülôfolyamatának tid-jét adja meg a `tid` visszatérési értékben. Ha az adott folyamat nem spawn által jött létre, akkor `tid==PvmNoParent`.

© BME Folyamatszabályozási Tanszék

5.2.4 *Folyamatok állapotának lekérdezése*

```
int info = pvm_tasks(int where, int *ntask, struct
        pvmtaskinfo **taskp)
struct pvmtaskinfo {
        int ti_tid;
        int ti_ptid;
        int ti_host;
        int ti_flag;
        char *ti_a_out; };
CALL PVMFTASKS(WHERE, NTASK, TID, PTID, DTID, FLAG,
        AOUT, INFO)
```

A virtuális gép folyamatairól kapunk információt. A `where` paraméter 0 értéke esetén az összes, pvmdtid esetén az adott hoszton futó, tid esetén pedig az adott folyamatról. A kérdéses taszkok számát az `ntask` paraméterben kapjuk meg, a taszkok állapotát pedig a `taskp` struktúra tömbben:

`ti_tid`	a taszk tid-je
`ti_ptid`	a szülô tid-je
`ti_dtid`	az adott hoszt pvmd tid-je
`ti_flag`	a taszk státusa
`ti_a_out`	a végrehajtható állomány neve

A C hívás egyszerre adja meg a kért információt, a fortran formában ciklikusan kell `ntask`-szor hívni.

```
int status = pvm_pstat(tid)
CALL PVMFPSTAT(TID, STATUS)
```

A `tid` által azonosított taszk állapotáról ad felvilágosítást. Status a következô értékeket veheti fel:

`PvmOk`	a taszk fut
`PvmNoTask`	a taszk nem fut
`PvmBadParam`	a tid nem létezik

5.3 **Üzenet küldés**

5.3.1 *Pufferkezelés*

A pvm adatátviteli eszközei a pufferek. Egy üzenetet elküldés elôtt egy pufferben állítunk össze, illetve vételnél ide érkezik be. Alapállapotban egy folyamat egy vételi és egy adó pufferrel rendelkezi, de további puffereket hozhatunk létre és szüntethetünk meg.

```
int bufid = pvm_mkbuf(int encode)
CALL PVMFMKBUF(ENCODE, BUFID)
```

Létrehoz egy új üzenet puffert. A puffer azonosítója `bufid`. Az adatok pufferbe tételekor alkalmazott kódolást írja elô az `encode` paraméter:

© BME Folyamatszabályozási Tanszék

PvmDataDefault	XDR kódolás, ha kell
PvmDataRaw	nincs kódolás
PvmDataInPlace	nincsen adatmásolás

illetve a fortran megfelelôk:

PVMDEFAULT, PVMRAW, PVMINPLACE.

```
int info = pvm_freebuf(int bufid)
CALL PVMFFREEBUF(BUFID, INFO)
```

A freebuf megszünteti a bufid által azonosított puffert.

Akármennyi puffert hozott létre egy folyamat, ezek közül mindig csak egy lehet aktív adó, illetve vevô.

```
int oldbuf = pvm_setsbuf(int bufid)
CALL PVMFSETSBUF(BUFID, OLDBUF)
int oldbuf = pvm_setrbuf(int bufid)
CALL PVMFSETSBUF(BUFID,OLDBUF)
```

Az aktuális adó (setsbuf) ill. vevô (setrbuf) puffert állíthatjuk be a bufid által megjelölt pufferre. A régi puffer azonosítóját oldbuf-ban kapjuk vissza.

```
int bufid = pvm_getsbuf(void)
CALL PVMFGETSBUF(BUFID)
int bufid = pvm_getrbuf(void)
CALL PVMFGETRBUF(BUFID)
```

A bufid az aktuális adó (getsbuf) ill. vevô (getrbuf) puffer azonosítóját adja vissza.

Üzenetek küldése során a régi adó puffert meg kell szüntetni, létre kell hozni egy újat és ezt végül aktív adóvá kell tenni. Ezt a gyakran használt freebuf-mkbuf-setsbuf hármast helyettesíti a következô hívás:

```
int bufid = pvm_initsend(int encode)
CALL PVMFINITSEND(ENCODE, BUFID)
```

Paraméterezése megegyezik az mkbuf-éval.

A pufferek állapotáról felvilágosítást nyerhetünk:

```
int info = pvm_bufinfo(int bufid, int *nbytes, int
         *msgtag, int *from)
CALL PVMFBUFINFO(BUFID, NBYTES, MSGTAG, FROM, INFO)
```

A bufid által meghatározott pufferben található byte-ok számát az nbytes, az üzenethez rendelt azonosítót az msgtag és az üzenet forrását a from értéke adja meg.

5.3.2 *Üzenetek összeállítása és szétszedése*

Egy üzenet elkészítésekor különbözô adattípusokat teszünk a pufferbe. Ha az architektúra független konverzió elô volt írva az adott pufferre akkor az is megtörténik.

```
int info = pvm_packf(const char *format, ...)
int info = pvm_pkstr(char *string)
int info = pvm_pkxxx(xxx *data, int nitem, int
         stride)
CALL PVMFPACK(WHAT, DATA, NITEM, STRIDE)
```

A C nyelvű forma minden adattípusra külön hívást biztosít. Kivétel a packf, amely egy **printf(3)** szerű stringgel írja le az adatok típusát. NULL-lal lezárt string továbbítására elônyösen használható a pkstr. A pk*xxx* `nitem` számú adat bepakolását végzi, `stride` lépésközzel. *xxx* helyén a következô állhat:

xxx	C típus
`float`	float
`double`	double
`cplx`	2 darab float (komplex)
`dcplx`	2 darab double (komplex)
`byte`	char
`short`	short
`int`	int
`long`	long
`ushort`	unsigned short
`uint`	unsigned int
`ulong`	unsigned long

A fortran formában a `what` paraméter definiálja az adattípust:

`STRING, BYTE1, INTEGER2, INTEGER4, REAL4, COMPLEX8, REAL8, COMPLEX16.`

Egy vett üzenetet tartalmazó puffert ki kell csomagolni. Ezt a következô hívásokkal tehetjük meg:

```
int info = pvm_unpackf(const char *format, ...)
int info = pvm_upkstr(char *string)
int info = pvm_upkxxx(xxx *data, int nitem, int
         stride)
CALL PVMFUPACK(WHAT, DATA, NITEM, STRIDE)
```

Paraméterezése megegyezik a pack hívásokéval.

5.3.3 *Üzenetek elküldése*

A pufferben létrehozott üzentet elküldhetjük egy megadott folyamatnak, vagy folyamatok egy csoportjának. Minden üzenethez tartozik egy üzenetazonosító, amelyet a küldô folyamat specifikál. Ezen azonosító alapján tudja a vevô megkülönböztetni az üzeneteket.

© BME Folyamatszabályozási Tanszék

```
int info = pvm_send(int tid, int msgtag)
CALL PVMFSEND(TID, MSGTAG, INFO)
```

Az aktív adó pufferben tárolt üzenetet elküldi a `tid` által azonosított folyamathoz, `msgtag` üzenetazonosítóval.

```
int info = pvm_mcast(int *tids, int ntasks, msgtag)
CALL PVMFMCAST(TIDS, NTASKS, MSGTAG, INFO)
```

Az aktív adó pufferben tárolt üzenetet elküldi a `tids` tömben tárolt, `ntasks` számú folyamathoz, `msgtag` azonosítóval.

Gyakran van szükség nagyobb mennyiségű, azonos típusú adat (tömb) elküldésére. Ebben az esetben használható a következô hívás:

```
int info = pvm_psend(int tid, int msgtag, char
         *buf, int len, int datatype)
CALL PVMFSEND(TID, MSGTAG, BUF, LEN, DATATYPE,
         INFO)
```

A `tid` és `msgtag` paraméter megegyezik a send-nél használtakkal. A `buf` egy mutató arra a memória területre, ahol a `len` mennyiségű és `datatype` típusú adat található. Az adattípusok a következôk lehetnek:

datatype	C típus
PVM_STR	string
PVM_BYTE	byte
PVM_SHORT	short
PVM_INT	int
PVM_FLOAT	real
PVM_CPLX	2 float (komplex)
PVM_DOUBLE	double
PVM_DCPLX	2 double (komplex)
PVM_LONG	long integer
PVM_USHORT	unsigned short int
PVM_UINT	unsigned int
PVM_ULONG	unsigned long int

A fortran formában a típus értékei megegyeznek a pack-nál használtakkal.

Lehetôség van nem csak üzenetek, hanem UNIX szignálok küldésére is:

```
int info = pvm_sendsig(int tid, int sig)
CALL PVMFSENDSIG(TID, SIG)
```

A `tid` folyamatnak küld egy `sig` szignált. Az esetlegesen különbözô szignálok között nem történik konverzió.

© BME Folyamatszabályozási Tanszék

5.3.4 *Üzenetek vétele*

Üzenetek vétele során meghatározott folyamattól meghatározott azonosítóval ellátott üzenetre várhatunk. A vétel megtörténtekor az aktív vételi pufferba kerül az üzenet, ahonnal kicsomagolhatjuk.

```
int bufid = pvm_recv(int tid, int msgtag)
CALL PVMFRECV(TID, MSGTAG, BUFID)
```

A `tid` folyamattól `msgtag` azonosítójú üzenetet várunk, a vételig a hívó folyamat blokkolódik. Ezen paraméterek -1 értéke esetén bármely folyamattól bármely üzenetet vesz. `Bufid` a vett üzenetet tartalmazó puffer azonosítója.

```
int bufid = pvm_nrecv(int tid, int msgtag)
CALL PVMFNRECV(TID, MSGTAG, BUFID)
```

Az nrecv alkalmazása esetén nem blokkoló vételt hajthatunk végre. Ha nincsen megfelelô üzenet, akkor `bufid`-ben 0-t ad vissza. Paraméterezése megegyezik a recv-évél.

Üzenetre várhatunk meghatározott ideig is:

```
#include <sys/time.h>
int bufid = pvm_trecv(int tid, int msgtag, struct
        timeval *tmout)
CALL PVMFRECV(TID, MSGTAG, SEC, USEC, BUFID)
```

A trecv működése hasonlít az recv-éhez, de az üzenetre csak a `tmout` ill. `SEC`, `USEC` által meghatározott ideig vár. 0 idô esetén nrecv-ként viselkedik.

Lehetôség van vétel nélkül megvizsgálni, hogy van-e olyan üzenet amire várunk:

```
int bufid = pvm_probe(int tid, int msgtag)
CALL PVMFPROBE(TID, MSGTAG, BUFID)
```

A probe paraméterezése és visszatérési értékei megegyeznek az nrecv-évél. Különbség, hogy vétel nem történik meg, vagyis az aktív vételi pufferbe nem kerül üzenet. Ehhez még egy recv kiadására is szükség van.

A psend-hez hasonlóan közvetlenül is végezhetünk üzenetküldést:

```
int info = pvm_precv(int tid, int msgtag, char
        *buf, int len, int datatype, int *atid,
        int *atag, int *alen)
CALL PVMFPRECV(TID, MSGTAG, BUF, LEN, DATATYPE,
        ATID, ATAG, ALEN, INFO)
```

Paraméterezése hasonló a psend-éhez. `atid`-ben a küldô azonosítóját, `atag`-ban az üzenet azonosítóját, `alen`-ben pedig az üzenet hosszát kapjuk meg.

5.3.5 *Üzenet azonosító összehasonlítás*

A pvm a recv hívásokban a kívánt üzenetazonosítót összehasonlítja az üzenetével. Ha megegyezik, akkor a vétel megtörténik. Az egyszerű egyenlôség vizsgálaton kívül lehetôség van összetettebb vizsgálatokra is.

```
int (*old)() = pvm_recvf(int(*new)(int bufid, int
         tid, int tag))
FORTRANBAN NEM HASZNÁLHATÓ
```

A hívás lecseréli a pvm által használt összehasonlító függvényt. `Old` a régi, `new` az új függvény mutatója. New `bufid`-ben kap egy pufferazonosítót a venni kívánt üzenettel, `tid`-et a forrás taszk azonosítóját és `tag`-et egy üzenetazonosítót.

5.3.6 *Várakozó azonosító*

A várakozó azonosító további megkülönböztetést tesz lehetôvé az üzenet azonosítón kívül. Segítségével azonosíthatunk pl.: egy kérésre adott választ. A következô két hívás a várakozó azonosítót kérdezi le, ill. állítja be:

```
int waitid = pvm_getmwid(int bufid)
int info = pvm_setmwid(int bufid, waitid)
FORTRANBAN NEM HASZNÁLHATÓ
```

A `bufid` puffer tartalmazza a kérdéses üzenetet, `waitid` pedig a várakozó azonosítót. Csak C nyelvű formája létezik.

5.4 Csoportok

A virtuális gépen belül további felosztást jelenthet a taszkok csoportokba (group) szervezése. Minden csoportot egy név jellemez. Minden, a csoportba belépett taszk kap egy példány azonosítót (instance number), amely a csoporton belül egyedi. Egy taszk több csoporthoz is tartozhat.

5.4.1 *Belépés a csoportba, kilépés a csoportból*

```
int inum = pvm_joingroup(char *group)
CALL PVMFJOINGROUP(GROUP, INUM)
```

A hívó belép a `group` nevű csoportba, példányazonosítója `inum` lesz. Ha a csoport még nem létezett, akkor létrejön.

```
int info = pvm_lvgroup(char *group)
CALL PVMFLVGROUP(GOUP, INFO)
```

A taszk kilép a `group` nevű csoportból. Késôbbiekben visszaléphet, de példányazonosítója nem lesz szükségszerűen ugyanaz.

5.4.2 *Csoport méretének megállapítása*

```
int size = pvm_gsize(char *group)
CALL PVMFGSIZE(GROUP, SIZE)
```

`Size`-ban vissza adja a `group` nevű csoport tagjainak számát a hívás pillanatában.

5.4.3 *Azonosító konverziók*

Egy pvm taszk egyértelműen azonosítható tid-je, vagy csoportnév és példányazonosítója által. A következô két hívás ezek közötti megfeleltetést végzi el.

```
int inum = pvm_getinst(char *group, int tid)
CALL PVMFGETINST(GROUP, TID, INUM)
```

A `tid` taszkazonosítójú folyamat `group` csoport beli példányazonosítóját adja vissza `inum`-ban.

```
int tid = pvm_gettid(char *group, int inum)
CALL PVMFGETTID(GROUP, INUM, TID)
```

A `group` csoportba tartozó, `inum` példányazonosítójú folyamat tid-jét adja vissza. Mivel egy folyamat egyszerre több csoportba is tartozhat, ezért a tid és instance alapján a csoportnév nem lenne egyértelműen megállapítható, így erre nem létezik pvm hívás.

5.4.4 *Barrier szinkronizáció*

A barrier szinkronizáció több folyamatnak egy ponton való szinkronizálására használható.

```
int info = pvm_barrier(char *group, int count)
CALL PVMFBARRIER(GROUP, COUNT, INFO)
```

A folyamat futása felfüggesztôdik addig, amíg a `group` csoportba tartozó folyamatok közül `count` számú taszk végre nem hajtja ugyanezt a hívást. Az adott csoportnak már léteznie kell, és a hívónak a csoport tagjának kell lennie.

5.4.5 *Broadcast*

```
int info = pvm_bcast(char *group, int msgtag)
CALL PVMFBCAST(GROUP, MSGTAG, INFO)
```

Az aktív adó puffer tartalmát elküldi a `group` csoport összes tagjának, `msgtag` üzenetazonosítóval. A hívónak nem kell a csoport tagjának lenni, de ha az, akkor saját üzenetét nem kapja meg.

5.4.6 *Tömb műveletek*

Gyakran van szükség egy adattömb darabokra bontására, és a darabok szétküldésére több folyamathoz, illetve több helyrôl összegyűjtött adatok egy tömbbé alakítására. Ezeket valósítja meg a következô két hívás:

```
int info = pvm_scatter(void *result, void *data,
         int count, int datatype, int msgtag, char
         *group, int rootginst)
CALL PVMFSCATTER(RESULT, DATA, COUNT, DATATYPE,
         MSGTAG, GROUP, ROOTGINST, INFO)
```

A `rootginst` példányazonosítójú taszk, a `data` tömbben lévô, `datatype` típusú adatot szétosztja a `group` csoport tagjai (beleértve saját magát) között. Minden folyamat `count` mennyiségű adatot kap. A `msgtag` az üzeneteket azonosítja. Minden

folyamatnak meg kell hívni a scatter-t, ekkor a vett adat a `result` tömbbe kerül. A `data` paraméter csak a `rootginst` folyamatnál értelmezett.

```
int info = pvm_gather(void *result, void *data, int
        count, int datatype, int msgtag, char
        *group, int rootginst)
CALL PVMFGATHER(RESULT, DATA, COUNT, DATATYPE,
        MSGTAG, GROUP, ROOTGINST, INFO)
```

A gather működése és paraméterezése hasonló a scatter-hez. Itt a `rootginst` folyamat gyűjti össze a `result` tömbbe a többi folyamattól a `data` által mutatott adatokat. Minden érdekelt folyamatnak meg kell hívni a gather-t, a `result` paraméter csak `rootginst` esetén értelmezett.

```
int info = pvm_reduce(void(*func)(), void *data,
        int count, int datatype, int msgtag, char
        *group, int rootginst)
void func(int *datatype, void *x, void *y, int
        *num, int *info)
CALL PVMFREDUCE(FUNC, DATA, COUNT, DATATYPE,
        MSGTAG, GROUP, ROOTGINST, INFO)
CALL FUNC(DATATYPE, X, Y, NUM, INFO)
```

A `group` csoport minden tagja elvégzi a `func` műveletet a `data` által mutatott, `count` számú `datatype` típusú adatot tartalmazó tömbön. A végeredmény a `rootginst` példányazonosítójú folyamatnál gyűlik össze. Ez a hívás kényelmes lehetôséget biztosít például tömbök összegének elôállítására. Func elôredefiniált értékei: `PvmMax`, `PvmMin`, `PvmSum`, `PvmProduct`.

Az elôzô három hívásban a `datatype` paraméter lehetséges értékeit lásd a psend hívásnál.

5.5 Egyéb funkciók

5.5.1 *Adatbázis kezelés*

A pvmd nyilvántart egy globálisan elérhetô adatbázist amelyben egyszerű adatok, pl. tidek tárolhatók. Minden adat egy név-index párral érhetô el. Az adatok a név alapján osztályokba vannak sorolva, ezen belül érvényes az index.

```
int cc = pvm_insert(char *name, int index, int
        data)
FORTRANBAN NEM HASZNÁLHATÓ
```

A `name` osztályba az `index` helyre eltárolja data adatot. Ha `index==-1` akkor az elsô helyre.

```
int cc = pvm_delete(char *name, int index)
FORTRANBAN NEM HASZNÁLHATÓ
```

A `name` osztályból kitörli a `index` helyen lévô adatot. Ha `index`==-1 akkor az elsô helyen levôt.

```
int cc = pvm_lookup(char *name, int index, int
         *data)
FORTRANBAN NEM HASZNÁLHATÓ
```

A name osztályból az index helyen levô adatot data-ba tölti. Ha index==-1 akkor az elsô adatot

5.5.2 *Hoszt megállapítása*

```
int dtid = pvm_tidtohost(tid)
FORTRANBAN NEM HASZNÁLHATÓ
```

`Dtid`-ben visszaadja a `tid` folyamathoz tartozó pvmd azonosítóját.

5.5.3 *Nyomkövetési flag-ek beállítása*

```
#include <pvmtev.h>
int info = pvm_gettmask(int who, Pvmtmask mask)
int info = pvm_settmask(int who, Pvmtmask mask)
FORTRANBAN NEM HASZNÁLHATÓ
```

A nyomkövetési maszkot (`mask`) kérdezi le, ill. állítja be saját magára (`who==PvmTaskSelf`), vagy gyermekeire (`who==PvmTaskChild`).

5.5.4 *Adatábrázolás megállapítása*

```
int cod = pvm_archcode(char *arch)
CALL PVMFARCHCODE(ARCH, COD)
```

Az `arch` nevű architektúra adatábrázolására jellemzô állandót ad vissza `cod`-ban. Ha két különbözô architektúrára ez a szám egyezik, akkor a pk***xxx*** hívásokban a `PvmDataRaw` biztonsággal használható.

5.5.5 *Kimenet átirányítás*

```
#include <stdio.h>
int info = pvm_catchout(FILE *ff)
CALL PVMFCATCHOUT(ONOFF, INFO)
```

A catchout hívás után spawn-nal indított folyamatok standard outputra és errorra írt adatai az `ff` stdio fileba kerülnek. `Ff` ill. `ONOFF` NULL értéke az átirányítást megszünteti.

© BME Folyamatszabályozási Tanszék

5.5.6 *Libpvm konfigurálása*

```
int oldval = pvm_setopt(int what, int val)
CALL PVMFSETOPT(WHAT, VAL, OLDVAL)
```

A `what` által meghatározott libpvm értéket `val`-ra állítja. A régi értéket `oldval` adja. `What` a következô lehet:

`PvmRoute`	üzenet routolási mód
`PvmDebugMask`	libpvm debug
`PvmAutoErr`	hibák automatikus jelzése
`PvmOutputTid`	stdout cél folyamata gyermeknél
`PvmTraceTid`	nyomkövetési adatok cél folyamata gyermeknél
`PvmTraceCode`	nyomkövetési üzenetazonosító
`PvmFragSize`	üzenet fragment méret
`PvmResvTids`	engedélyezi fenntartott üzenetek küldését
`PvmSelfOutputTid`	stdout cél folyamat
`PvmSelfOutputCode`	stdout üzenetazonosító
`PvmSelfTraceTid`	nyomkövetési adatok cél folyamata
`PvmSelfTraceCode`	nyomkövetési adatok üzenetazonosítója

`Val` értékeire elôre definiált konstansok:

`PvmDontRoute`	ne engedélyezzen direkt kapcsolatot
`PvmAllowDirect`	ne kérjen, de engedélyezzen
`PvmRouteDirect`	kérjen és engedélyezzen

```
int val = pvm_getopt(int what)
CALL PVMFGETOPT( WHAT, VAL)
```

A `what` által meghatározott libpvm értéket adja vissza `val`-ban. `What` értékei a setopt-nál leírtak lehetnek.

5.5.7 *Valós idô szinkronizálása*

```
#include <sys/time.h>
int info = pvm_hostsync(int host, struct timeval
         *clk, struct timeval *delta)
CALL PVMFHOSTSYNC(HOST, CLKSEC, CLKUSEC, DELTASEC,
         DELTAUSEC, INFO)
```

Lekérdezi az adott `host` valósidejű óráját és visszaadja `clk`-ban ill `CLKSEC`, `CLKUSEC`-ban. Saját és a távoli hoszt idejének különbségét `delta` ill `DELTASEC`, `DELTAUSEC` adja.

© BME Folyamatszabályozási Tanszék

5.5.8 *Pvm események figyelése*

```
int info = pvm_notify(int what, int msgtag, int
         cnt, int *tids)
CALL PVMFNOTIFY(WHAT, MSGTAG, CNT, TIDS, INFO)
```

A `what` által meghatározott esemény bekövetkeztekor msgtag azonosítójú üzenetet kap a hívó a rendszertôl. What a következô értékeket veheti fel:

`PvmTaskExit`	taszk befejezi a futását
`PvmHostDelete`	hoszt kilép a virtuális gépbôl
`PvmHostAdd`	új hoszt kapcsolódik a virtuális gépbe

`Cnt` Exit és Delete esetén a `tids` tömb méretét, Add esetén a kért üzenetek számát tartalmazza. `Tids` Exit és Delete esetén a figyelendô taszkok vagy hosztok azonosítói, Add esetén nem használt.

5.5.9 *Hibajelzés*

```
int info = pvm_perror(char *msg)
CALL PVMFPERROR(MSG, INFO)
```

A **perror(3)** stdio függvényhez hasonló módon a legutoljára bekövetkezett hibára utaló szöveget ír ki `msg` után.

© BME Folyamatszabályozási Tanszék

5.6 Hibakódok

A pvm hívások sikertelenség esetén automatikusan megszakíthatják a program futását (lásd: `pvm_setopt()`). Ez nem mindig célravezetô, ezért ezt a lehetôséget kikapcsolva a program maga is figyelheti a hívások által visszaadott hibakódókat, és ezeknek megfelelôen dönthet a folyatatásról. A következô táblázat az `info` paraméterben visszadott hibakódókat tartalmazza.

PvmOk	*nem történt hiba*
PvmBadParam	*hibás paraméter*
PvmMismatch	*a count paraméterben eltérés van*
PvmNoData	*unpack során elfogytak az adatok*
PvmNoHost	*nem létezik ilyen hoszt*
PvmNoMem	*malloc nem sikerült*
PvmBadMsg	*a vett üzenet formátuma más architektúráé*
PvmSysErr	*libpvm nem tudta felvenni a kapcsolatot a pvmd-nal*
PvmNoBuf	*nincsen aktív puffer*
PvmNoSuchBuf	*a hivatkozott puffer nem létezik*
PvmNullGroup	*group név NULL volt*
PvmDupGroup	*a taszk már tagja a csoportnak*
PvmNoGroup	*nincsen ilyen nevű csoport*
PvmNotInGroup	*a csoportnak nem tagja az adott taszk*
PvmNoInst	*nincsen ilyen példányazonosítojú taszk*
PvmHostFail	*a hoszt elérhetetlen*
PvmNoParent	*ennek a taszknak nincsen szülôje*
PvmNotImpl	*nem implementált hívás*
PvmDSysErr	*pvmd belsô hiba*
PvmBadVersion	*verziószám eltérés*
PvmOutOfRes	*nincsen megfelelô erôforrás*
PvmDupHost	*az adott hoszt már tagja a virtuális gépnek*
PvmCantStart	*nem sikerült a pvmd indítása*
PvmAlready	*az adott hívás kizárólagos elérést kíván, és egy már folyamatban van*
PvmNoTask	*nincsen taszk ilyen tid-del*
PvmNoEntry	*az osztály szervernek nincsen ilyen bejegyzése*
PvmDupEntry	*az osztály szervernek már van ilyen bejegyzése*

© BME Folyamatszabályozási Tanszék

6. Egy példa

Az alábbi egyszerű példa[4] egy master-slave felépítésű programot mutat. A mester (1. lista) megkérdezi, hogy hány darab szolgát indítson el, vagy ha ezt nem tudja megtenni[5] akkor a processzorok számát veszi alapul, és elindítja az alfolyamatokat. Ha nem történik hiba, kiosztja a kiindulási adatokat és vár az eredményre.

A szolga folyamatok (2. lista) elindulásuk után megvárják a mester által küldött adatokat, azokat feldolgozzák és visszajuttatják a mesterhez. A feldolgozás közben szolga-szolga kommunikáció is történik.

1. lista: *master.c*

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
     int mytid;                /* saját TID    */
     int tids[32];        /* slave TID-ek     */
     int n, nproc, numt, i, who, msgtype, nhost, narch;
     float data[100], result[32];
     struct pvmhostinfo *hostp[32];

     mytid = pvm_mytid();      /* belép a PVM-be*/

     /* Lekérdezi az indítandó taszkok számát     */
     /* Ha a konzol alól indítottuk, akkor a */
     /* processzorok számát használja        */

     if( pvm_parent() == PvmNoParent ){
          puts("Hány darab szolga kell (1-32)?");
          scanf("%d", &nproc);
     }
     else{
          pvm_config( &nhost, &narch, hostp );
          nproc = nhost;
          if( nproc > 32 ) nproc = 32 ;
     }
```

[4] éd ó f pvm3/examples lö y b .

[5] éd k ó d l é d db .

© BME Folyamatszabályozási Tanszék

```
     /* Elindítja a szolgákat*/
     numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc,
tids);
     if( numt < nproc ){
          printf("Hiba a szolgák indításában.
Hibakódok:\n");
          for( i=numt ; i<nproc ; i++ ) {
               printf("TID %d %d\n",i,tids[i]);
          }
          for( i=0 ; i<numt ; i++ ){
               pvm_kill( tids[i] );
          }
     pvm_exit();
     exit();
     }

     n = 100;

     for( i=0 ; i<n ; i++ ){
          data[i] = 1;
     }

     /*Kezdeti paramétereket szétküldjük     */
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&nproc, 1, 1);
     pvm_pkint(tids, nproc, 1);
     pvm_pkint(&n, 1, 1);
     pvm_pkfloat(data, n, 1);
     pvm_mcast(tids, nproc, 0);

     /* És várunk az eredményre          */
     msgtype = 5;
     for( i=0 ; i<nproc ; i++ ){
          pvm_recv( -1, msgtype );
          pvm_upkint( &who, 1, 1 );
          pvm_upkfloat( &result[who], 1, 1 );
          printf("%f-t kaptam %d-tól\n",result[who],who);
     }
     /* A vége elôtt lépjünk ki a PVM-bôl */
     pvm_exit();
}
```

2. lista: *slave.c*

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

main()
{
     int mytid;/* my task id */
     int tids[32];/* task ids   */
     int n, me, i, nproc, master, msgtype;
     float data[100], result;
     float work();

     mytid = pvm_mytid();

     /* kiindulási adatok a metertôl */
     msgtype = 0;
     pvm_recv( -1, msgtype );
     pvm_upkint(&nproc, 1, 1);
     pvm_upkint(tids, nproc, 1);
     pvm_upkint(&n, 1, 1);
     pvm_upkfloat(data, n, 1);

     /* Megkeresi melyik szolga */
     for( i=0; i<nproc ; i++ )
     if( mytid == tids[i] ){ me = i; break; }

     /* Feldolgozza az adatokat */
     result = work( me, n, data, tids, nproc );

     /* A mesternek visszaküldi az eredményt */
     pvm_initsend( PvmDataDefault );
     pvm_pkint( &me, 1, 1 );
     pvm_pkfloat( &result, 1, 1 );
     msgtype = 5;
     master = pvm_parent();
     pvm_send( master, msgtype );

     pvm_exit();
}
```

```
float
work(me, n, data, tids, nproc )
     int me, n, *tids, nproc;
     float *data;
{
     int i, dest;
     float psum = 0.0;
     float sum = 0.0;
     for( i=0 ; i<n ; i++ ){
     sum += me * data[i];
     }
     /* Itt két szolga fog kommunikálni*/

     pvm_initsend( PvmDataDefault );
     pvm_pkfloat( &sum, 1, 1 );
     dest = me+1;
     if( dest == nproc ) dest = 0;
     pvm_send( tids[dest], 22 );
     pvm_recv( -1, 22 );
     pvm_upkfloat( &psum, 1, 1 );

     return( sum+psum );
}
```