Párhuzamos programok fejlesztése PVM könyvtárral (1. rész)

Bevezetés a PVM-be

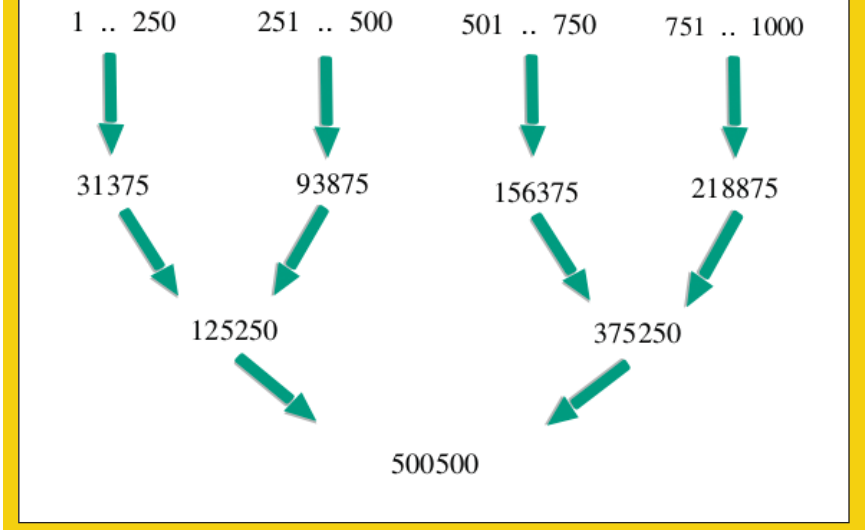
Egyszerű használata és robusztussága miatt párhuzamos programok fejlesztéshez a PVM könyvtár kezdők és profik számára egyaránt az egyik legjobb jelenleg fellelhető eszköz.

Párhuzamos architektúra és a párhuzamos programozás

Minden egyéb hardverismeretet mellőzve tegyük fel, hogy nem egy számítógéppel (processzorral) rendelkezünk, hanem többel. Legyen ezek száma *N*. Tegyük fel azt is, hogy ezek a számítógépek valamilyen összeköttetésben vannak egymással, legyen az akár hálózati vagy közvetlen kapcsolat (például többmagos processzorok, osztott memória), vagy esetleg valami egészen más. Természetesen szükség lesz még a megfelelő szoftverre is számítógépeink munkába fogásához. Egy ilyen hardver- és szoftverrendszert együtt párhuzamos architektúrának nevezünk. Párhuzamos architektúra valójában többféle létezik, de ebben a cikkben mi kizárólag hálózatba kötött számítógépekkel, úgynevezett klaszterekkel (cluster) fogunk foglalkozni.
A párhuzamos architektúrákra fejlesztett programokat párhuzamos programoknak, míg programírás illetve programtervezés folyamatát párhuzamos programozásnak nevezzük.
A példák kipróbálásához nem kell, hogy legyen otthon egy klaszterünk, mivel mindegyik működik egy processzoros, hálózatba nem kötött gépen is. Ám a dolog igazi előnyét akkor láthatjuk, ha igazi klaszteren futtatunk.

Ha egy tyúk egy nap alatt...

Kezdjük egy nagyon egyszerű példával. Összegezni szeretnénk az egész számokat 1-től M-ig. A tudománytörténetből ismert, hogy Gauss (1777-1885) már gyermekkorában kitalálta ehhez azt az algoritmust, amivel akár fejben is kiszámolhatja bárki az eredményt.
Mi most mégis essünk neki nyers erővel és adjuk össze a számokat N processzoron párhuzamosan. Tegyük fel, hogy egy összeadás egy processzoron 1 másodpercig tart. Legyen M=1000, a processzorok száma (N) pedig 4.
A számításhoz egy processzorral M-1=1000-1 azaz 999 másodpercre volna szükségünk, tehát ez lesz a viszonyítási alap. Osszuk most fel a számokat 4 csoportra: 1..250, 251..500, 501..750, 751..1000.



1. ábra Számok összeadásának egy lehetséges módja 4 processzoron

Ha minden processzorunkkal egy ilyen csoport összegét számoljuk ki, és a processzorok párhuzamosan dolgoznak, akkor a csoportösszegek kiszámítása 249 másodpercig tart.
A négy csoportösszeget összeadni a 4 processzoron (de egyen is) további 3 másodpercig tart, így a feladat teljes megoldása 252 másodpercig tartott. Ezt a fajta feladatmegoldást szemlélteti *1. ábránk*.
A lényeg ebben az esetben az volt, hogy a megoldandó feladatot felosztottuk részfeladatokra és az egyes részfeladatokra, és azok megoldását rábíztuk processzorainkra, amik

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

eztán a végső megoldást egymással összehangolva számolták ki jelentős időt takarítva meg ezzel. A megoldás 4 processzoron hozzávetőleg negyedannyi időt vett igénybe mint egy processzoron, hatékonyságunk tehát négyszeres a hagyományos módszerhez képest. Szinte minden párhuzamos algoritmus lényege ez. Az elvégzendő lépések felosztása, hogy egymástól függetlenül, több processzoron futhassanak.

A PVM telepítése

A *PVM* (*Parallel Virtual Machine*; *párhuzamos virtuális gép*) egy programkönyvtár, amellyel párhuzamos programokat írhatunk. Számos platformon elérhető (többek között *Linux*, *Unix* és *Windows* rendszereken), elterjedt, nagyon hatékony és viszonylag könnyen használható. Mi természetesen *Linux* alatt fogjuk kipróbálni, ám a programok más rendszereken minimális változtatással (vagy akár változtatás nélkül) szintén lefordíthatók.
A *PVM* telepítését az olvasóra bízom, erről itt csak annyit jegyeznék meg, hogy a *Debian* illetve *Ubuntu* felhasználóknak igen egyszerű dolguk van, hiszen csak ki kell adniuk az

```
apt-get install pvm
apt-get install pvm-dev
```

parancsokat.
A forráskódot a *PVM*-et honlapjáról (➲ *http://www.csm.ornl.gov/pvm/*) tölthetjük le.
Telepítés után a következő lépés a futtatási környezet elkészítése. Ez mindössze néhány könyvtár létrehozását jelenti:

```
mkdir ~/pvm3/
mkdir ~/pvm3/bin
mkdir ~/pvm3/bin/LINUX
```

Ez a PVM helyes működésének szükséges feltétele, mivel a rendszer párhuzamos programjainkat ezen az elérési úton fogja majd keresni. A gépek közti kommunikációt megvalósító PVM démont a `pvm` parancs kiadásával indíthatjuk el. Ha minden jól ment, akkor a következőt láthatjuk:

```
pvm>
```

1. Lista A master folyamat forrása

```
/* master.c */
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>

#include "tags.h"

/* az inditando slave folyamatok szama */
#defineNUM_SLAVES      8

int main ()
{
    intmytid, slave_tids[NUM_SLAVES];
    char       message[30];
    int        ;
    int        result;


    /* sajat tid lekerdezese, belepes a PVM-be */
    mytid = pvm_mytid();

    /* - az inditando allomany neve
       - a paranccsori parameterek
       - pvmnek info, hogy milyen modon futtassa
       - ide megadhato a gepnev ill. az architektura
       - hogy hany darab induljon
       - a tomb, ahova a tid -ek megerkeznek majd
    */
    result = pvm_spawn("slave1", (char **)0, PvmTaskDefault, "",
    ➥NUM_SLAVES, slave_tids);

    if (result != NUM_SLAVES) {
  fprintf(stderr, "HIBA: A futtatas nem lehetseges.
  ➥Keves bevonhato processzor\n");
  pvm_exit();
  exit(-1);
    }

    for (i = 0; i < NUM_SLAVES; i++) {
  /* az uzenetkuldes inicializalasa */
  pvm_initsend (PvmDataDefault); /* ez a default kuldesi mod */

  /* kuldendo uzenetet (ami egy egesz) becsomagolasa */
  pvm_pkint(&slave_tids[i], 1, 1);

  /* elkuldes MSG_DATA tag-gel */
  pvm_send(slave_tids[i], MSG_DATA);
    }


    for (i = 0; i < NUM_SLAVES; i++) {
  /* uzenet fogadasa az i. slave folyamattol */

/* -1 jelentese a feltetlen partatlansag, vagyis barkitol
➥johet uzenet */
```

1. Lista folytatás

```
  pvm_recv (-1, MSG_DATA);

  /* kicsomagolas */
  pvm_upkstr(message);

  /* kiiras */
  printf ("%s \n",message);
    }

    pvm_exit ();

    /* kilepes */
    return 0;
}
```

Adjuk ki a `conf` parancsot. Ekkor a konzolon a következőket láthatjuk:

```
pvm> conf
conf
1 host, 1 data format
 HOST   DTID   ARCH   SPEED   DSIG
 Lorien  40000  LINUX  1000
 ➥0x00408841
pvm>
```

A `conf` parancs hatására a *PVM* démon kiírta, hogy jelenleg milyen gépek tagjai a klaszterünknek. A fenti információ azt jelenti, hogy egyelőre csak egy gépünk van, ám a kezdéshez ez is elegendő lesz.

Első PVM programunk

Ha egy gépre írunk párhuzamos programot, akkor a *PVM* démon szimulálja a többi gépet méghozzá úgy, hogy az összes indított folyamat ezen az egy gépen fog futni.
Így a párhuzamosságot csak imitáljuk, időt nem nyerünk, de a teszteléshez ez egyelőre elég lesz.
Komolyabb programok tesztelése így csalóka lehet, hacsak nem írtunk bombabiztos programot. Egy párhuzamos program nyomkövetése – ha egyáltalán lehetséges – sokkal nehezebb mint egy hagyományos szekvenciális társáé.
Első programunk feladata az lesz, hogy bebizonyítsa párhuzamosságát, méghozzá úgy, hogy minden processzor (a klaszter egyes gépei) kiír majd a képernyőre egy rövid kis üzenetet. Valójában nem maguk a processzorok írnak a képernyőre, a példa a *PVM* folyamatainak üzenetküldési mechanizmusát demonstrálja.
Az implementáció két C nyelvű program – mondhatnánk a párhuzamos programunk egy-egy folyamatának is –, egy úgynevezett „master” és egy „slave”, magyarul mester és szolga, de talán szebb, ha úgy fogalmazunk, hogy szerver és kiszolgáló. Ez utóbbi elnevezés nem a hétköznapi értelemben vett szerver-kiszolgáló kapcsolatot jelenti. Jelen esetben a master feladata, hogy elindít adott számú slave folyamatot, a slave folyamatok pedig üzenetet küldenek a masternek, jelezvén, hogy ők futnak. Ezzel tulajdonképpen „bizonyítják” a párhuzamos futást.
Egy fontos fejléc fájlunk van, a ***pvm3.h*** ami a *PVM* könyvár függvényeinek deklarációit és előre definiált konstansait tartalmazza. A ***tags.h*** saját fejléc-fájl, itt konstansként definiáljuk a programjaink által küldött üzenetek típusait (itt csak egyetlen üzenettípust használok és legtöbbször ez elég is, később azonban szükség lehet az üzenetek megkülönböztetésére).
Lényeges változóink a `mytid` (egész) és `slave_tids` (egészek tömbje). Minden *PVM* programnak (folyamatnak) van egy egyedi azonosítója, úgynevezett `tid`-je. Ez az azonosító tulajdonképpen olyan mint az operációs rendszer futó folyamatainak azonosítója, azaz a `PID`, ám ez a *PVM* futtatókörnyezeten belüli azonosításra szolgál és nincsen kapcsolatban az esetleges operációs rendszerbeli `PID`-del.
A `tid` lekérdezésére szolgál a `pvm_mytid()` függvény. Ez a függvény egy egész számot ad vissza, ami a futó folyamatunk `tid`-je.
Minden egyes *PVM* program elején meg kell hívni ezt a függvényt, ezzel jelezzük a *PVM* számára programunk elindítását.
A `pvm_spawn()` függvény meghívásával újabb folyamatokat indítunk. Ezek lesznek az egyes slave-ek, akiktől majd üzeneteket fogadunk. Paramétereinek pontos magyarázatát, mint ahogy az összes *PVM* függvényét, man oldalának átböngészésével ismerhetjük meg. Dióhéjban a `pvm_spawn()` függvény paraméterei a következők:

- Az indítandó állomány neve (string)
- Az indítandó állomány parancssori paraméterei (string)
- Információ a futtatás módjáról (ez most számunkra kevésbé lényeges)
- Gépnév és architektúra (számunkra most és később sem lényeges)
- Az indítandó folyamatok száma
- Egy egész-tömb címe, ahová az elindult folyamatok `tid`-jei kerülnek

A függvény visszatérési értéke egy egész, ami a sikeresen indított folyamatok számát adja meg. Ezt összehasonlítottuk a `NUM_SLAVES` konstanssal amiben az indítani kívánt folyamatok számát definiáltuk. Ha nem sikerült az igényelt számú folyamatot elindítani, akkor kilépünk. Kilépéskor kötelezően meghívandó függvény a `pvm_exit()`.
Miután elindultak slave folyamatok, először egyenként üzenetet küldünk nekik, majd üzenetet fogadunk tőlük és azt kiírjuk a konzolra. Az egyes slave folyamatoknak elküldött üzenet a saját `tid`-jük, fogadott üzenet pedig egy karakterlánc lesz.
Az üzenetküldés folyamata három részre tagolódik:

- előkészítés
- a küldendő üzenet becsomagolása
- küldés

Az előkészítést a pvm_initsend() függvény végzi. Paramétere most legyen PvmDataDefault. Ez egy beépített konstans, jellemzően szinte mindig ezt használjuk adatküldésnél.
Az elküldendő adat becsomagolását a pvm_pkint() függvény végzi.
Ez a függvény csak int (azaz egész) típusú adatok becsomagolására használható, de a PVM lehetőséget nyújt más típusú adatok csomagolásához és átküldéséhez is. Az első paraméter a küldendő szám (számok) címe, második a darabszáma, a harmadik pedig a lépésközt adja meg (ezt hagyjuk most 1-nek).
Miután az adatokat előkészítettük, már csak el kell küldeni őket. A küldés *PVM* függvénye a `pvm_send()`, paraméterei a fogadó folyamat `tid`-je (a címzett) és az üzenet típusa, melyről már korábban szóltunk a `tags.h` fejlécfájl kapcsán. Az első paraméter nyilvánvaló, a második már némi magyarázatra szorul. Értéke gyakorlatilag teljesen ránk van bízva, leginkább magunk biztosítására használható.

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

2. Lista A slave folyamat

```
/* slave.c */
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>

#include "tags.h"

int main ()
{
    int  mytid;
    int  parent_tid;
    char message[30];
    int  number;


    /* a sajat tid
    ➥lekerdezese, belepes
    ➥a PVM-be */
    mytid = pvm_mytid();

    /* a szulo folyamat
    ➥tid-jenek lekerdezese */
    parent_tid = pvm_parent();

    /* az uzenetkuldes
    ➥elokeszitese */
    pvm_initsend
    ➥(PvmDataDefault);

    /* uzenetfogadas
    ➥a szulotol */
    pvm_recv(parent_tid,
    ➥MSG_DATA);

    /* tudjuk, hogy szamrol
    ➥van szo, kicsomagolas */
    pvm_upkint(&number, 1, 1);

    /* ha az uzenet helyesen
    ➥jott meg, azaz
    ➥megegyezik a sajat
    ➥tid-del */
    if (number == mytid) {

     /* akkor kuldjunk
     ➥stringet a master-nak */
  sprintf(message, "Hello from
  ➥%x", number);
  pvm_pkstr(message);
  pvm_send(parent_tid,
MSG_DATA);
    }

    pvm_exit ();
    exit (0);
}
```

3. Lista A tags.h

```
#ifndef TAGS_H
#define TAGS_H

#define MSG_DATA  1000

#endif /* TAGS_H */
```

Itt felhasználjuk a definiált `MSG_DATA` konstanst az üzenetek típusának megadására.
Adatokat elküldtük slave folyamatoknak, melyek valahogyan feldolgozzák azokat (jelen esetben nem tesznek semmi érdekeset). Következő feladatunk a visszaérkező üzenetek fogadása és kiírása a konzolra. Ez a küldéssel analóg módon történik, de némileg egyszerűbben. Az üzenetfogadás folyamata:

- várakozás megfelelő üzenetre és fogadás
- az üzenet kicsomagolása

A `pvm_recv()` függvénnyel addig várunk, míg megfelelő adat nem érkezik. Első paramétere a fogadás módja, jelen esetben jöhet az üzenet bárkitől, semmilyen sorrendet nem határoztunk meg (azaz értéke –1), a második a várt adat azonosítója, most `MSG_DATA`.
Példánkban vissza nem egészeket várunk, hanem karakterláncokat (string).
Ha megjött az üzenet (ami egy karakerlánc), kicsomagoljuk a `pvm_upkstr()` függvény segítségével. Természetesen más típusú adatokat is fogadhatunk, csak a példa miatt tárgyaljuk a stringet. Egyetlen paramétere a fogadott adat számára fenntartott memóriatömb címe.
Az üzenet megérkezése után kiírjuk a konzolra. Ha minden üzenetet fogadtunk és kiírtunk, akkor a program kilép.
Nézzük a slave folyamat megvalósítását.
Itt már nincs sok újdonság. Ami új, az a `pvm_parent()` függvény, mely annak a folyamatnak a `tid`-jét adja vissza, ami ezt a folyamatot létrehozta (spawn), vagyis most a master folyamatunk `tid`-jét.
Ez majd az üzenetküldéshez kell.
Másik új függvény a `pvm_pkstr()`, ami egy karakterlánc (string) becsomagolását végzi.
A teljesség kedvéért a fent említett ***tags.h*** forrását a listában olvashatjuk.
Programunk fordítását az alábbi parancsok kiadásával végezzük:

```
gcc -o ~/pvm3/bin/LINUX/master1
➥master.c -Wall -lpvm3
gcc -o ~/pvm3/bin/LINUX/slave1
➥slave.c -Wall -lpvm3
```

Ha már korábban elindítottuk a ***PVM*** démont, akkor lépjünk ki belőle a ***PVM*** konzolban kiadott `quit` paranccsal. Ennek hatására a ***PVM*** démon futása nem áll meg, a háttérben továbbra is fut.
Lépjünk be a ***~/pvm3/bin/LINUX/*** könyvtárba és futtassuk le a kapott master1 állományt.
A futtatás eredményeképpen valami hasonló kell, hogy kapjunk:

```
bha@Lorien:~/pvm3/bin/LINUX$
➥./master1
Hello from 40003
Hello from 40004
Hello from 40005
Hello from 40006
Hello from 40007
Hello from 40008
Hello from 4000a
Hello from 40009
bha@Lorien:~/pvm3/bin/LINUX$
```

Az első programunk, ami végül is semmi hasznosat nem művel lefutott és bebizonyította párhuzamos működését.
Legközelebb klasztert építünk majd, hogy programunk tényleg több gépen fusson és megismerkedünk egy igazi problémával és annak igazi megoldásával is.

Bánki Horváth András
(bha@elte.hu)
Végzős programtervező matematikus hallgató vagyok az ELTE-n. Minden érdekel ami az informatikával kapcsolatos. 1997 óta vagyok aktív Linux felhasználó. Ha nem dolgozom, legszívesebben a barátnőmmel és a barátaimmal vagyok.