

Párhuzamos és Grid rendszerek *(1. ea)*

alapfogalmak

Szeberényi Imre

BME IIT

<szebi@iit.bme.hu>



MŰEGYETEM 1782

Parallel programozás áttekintése

Miért fontos a párhuzamos feldolgozás ?

- Nagyobb teljesítmény elérése miatt ? Csak a teljesítményért ?
 - a párhuzamosítás ebben az esetben csak technológia
 - nem fontos az alkalmazás tervezője számára
 - el kell fedni (mint pl. a hardware regisztereket, cache-t,...)
- Egy lehetséges eszköz a valóság modellezésére
- Egyszerűsítheti a tervezési munkát

Párhuzamos feldolgozás problémái

- Gyakran nehezebb kivitelezni, mint a soros feldolgozást
- törékeny (nem determinisztikus viselkedés)
- deadlock problémák léphetnek fel
- erőforrás allokációs problémák lehetnek
- nem egyszerű hibát keresni

Ezek valós tapasztalatok, de nem szükségszerűek!

Történelmi áttekintés

- Neumann korában felmerült az ötlet (Daniel Slotnick), de a csöves technológia nem tette ezt lehetővé.

Első szupergép:

- 1967-ben ILLIAC IV (256 proc, 200MFlop)
- Thinking Machines CM-1, CM-2 (1980)
- Cray-1

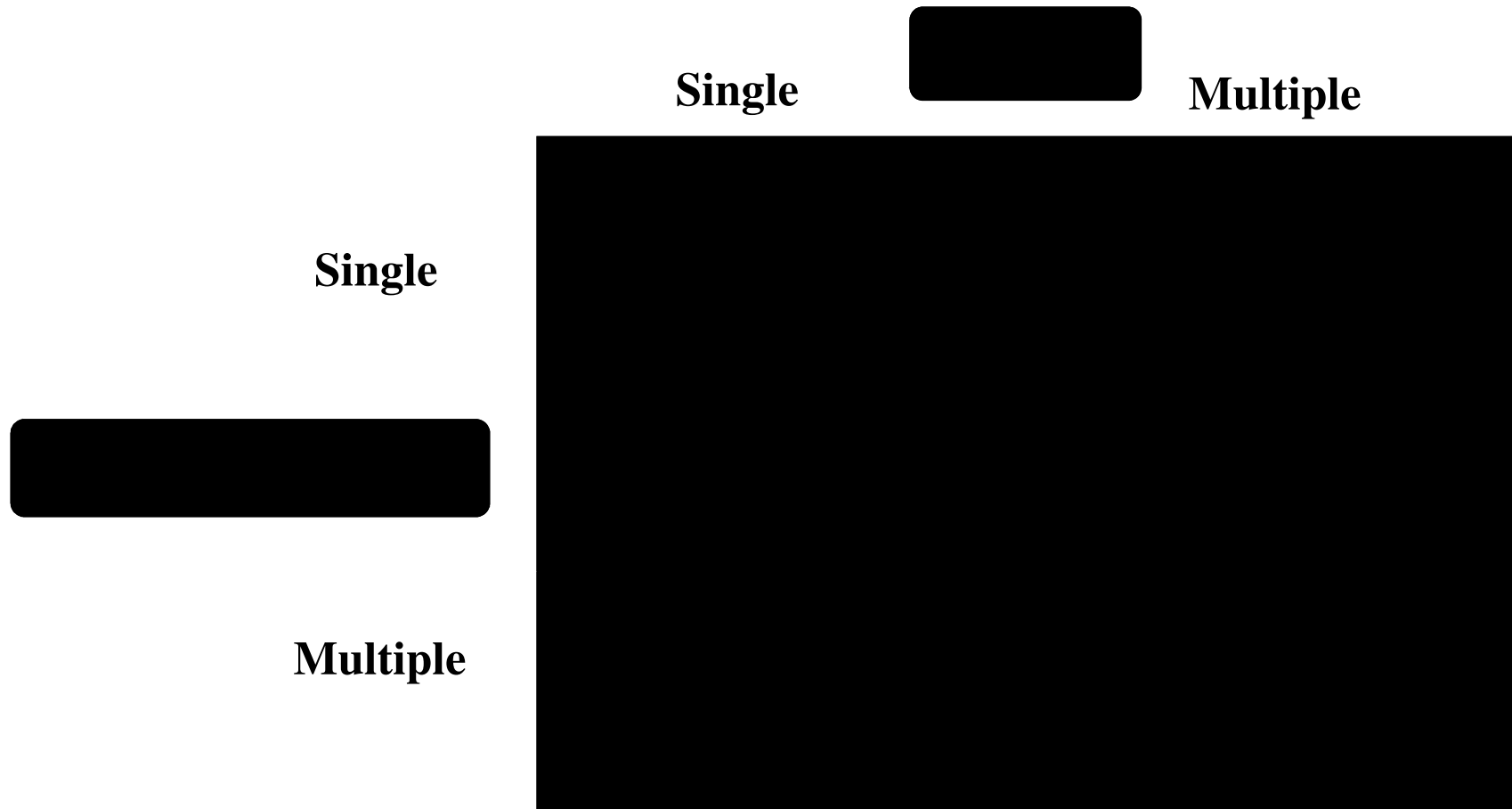
Jellemző szupersz.gép típusok

- Vektorprocesszoros rendszerek
 - Gyors műveletvégzés vektor jellegű adatokon
- Masszívan párhuzamos rendszerek (MPP)
 - Üzenetküldéses elosztott memóriás (MDM)
 - Szimmetrikus multiprocesszoros (SMP)
 - Elosztott közös memória (DSM)
- Elosztott számítási rendszerek
 - Homogén rendszerek
 - Heterogén rendszerek
- Metaszámítógépek és Grid rendszerek

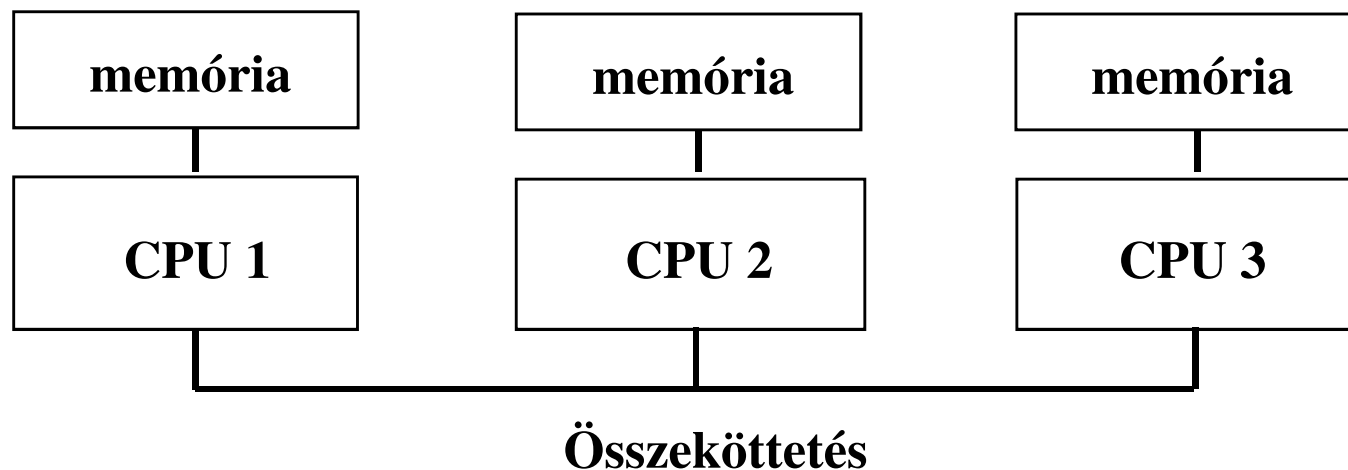
Párhuzamos gép modellje

- Több modell alakult ki.
- A legegyszerűbb a Flynn-féle modell, mely a Neumann modell kiterjesztésének tekinti a párhuzamos gépet.
- A másik gyakran alkalmazott modell az idealizált párhuzamos számítógép modell

Flynn-féle architektúra modell



Idealizált párhuzamos számítógép

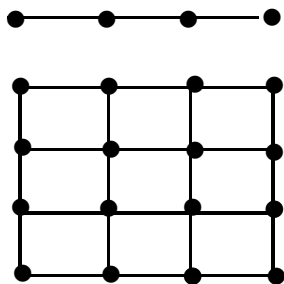


- Több processzor egyazon problémán dolgozik.
- Minden processzornak saját memóriája és címtartománya van.
- Üzenetekkel koordinálnak és adatokat is tudnak átadni.
- A lokális memória elérése gyorsabb.
- Az átviteli sebesség független a csatorna forgalmától.

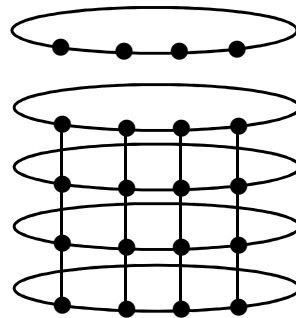
Architektúrák jellemzői

- Processzorok eloszlása
- Homogén vagy heterogén
- A kapcsolat késleltetése és sávszélessége
- Topológia

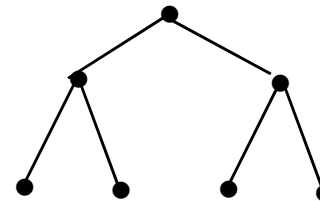
Hálók



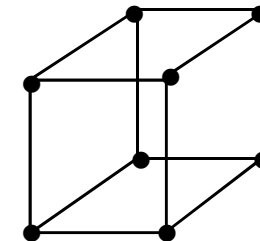
Gyűrűk



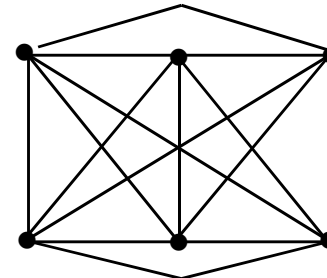
Fák



Hiperkockák



Teljesen összekötött



Programozási modell

- Közös memóriás
- Elosztott közös memóriás
- Üzenet küldéses

Valójában egyik modell sem kötődik szorosan a tényleges architektúrához

Közös memória elv jellemzői

Közös memória használatának előnyei

- Egységes hozzáférés
- Egyszerűbb programozás
- A hw adottságaitól függően jó speed-up értékel érhetőek el

Közös memória használatának hátrányai

- Memória-hozzáférés szűk keresztmetszetet jelenthet
- Nem jól skálázható
- Cache problémák megoldása külön hardvert igényel
- Nehéz nyomkövethetőség

Elosztott memória elv jellemzői

Elosztott memória használatának előnyei

- Skálázható
- Költségkímélő
- A redundancia növelésével növekedhet a megbízhatóság
- Speciális feldolgozó eszközökkel is együttműködik

Elosztott memória használatának hátrányai

- Kommunikáció igényes
- Nem minden algoritmus párhuzamosítható így
- A meglevő soros programokat és a közös memóriát használó alkalmazásokat át kell dolgozni
- Jó speed up értékeket nehéz elérni
- Nehéz nyomkövethetőség

Párhuzamos gépek osztályai

- Szimmetrikus multiprocesszoros (SMP)
 - sok azonos processzor közös memóriával
 - egy operációs rendszerrel
 - NUMA, ccNUMA
- Masszívan párhuzamos (MPP)
 - sok processzor gyors belső hálózattal
 - elosztott memória
 - sok példányban fut az operációs rendszer
- Klaszter
 - sok gép gyors hálózattal összekötve
 - elosztott memória
 - sok példányban esetleg heterogén operációs rendszer

Teljesítménymérés

Sebességnövekedés (Speed Up): $S_n = T_s/T_n$

ahol: S_n N processzonnal elért sebességnövekedés

T_s futási idő soros végrehajtás esetén

T_n futási idő N processzor esetén

Hatékonyság (Efficiency): $E_n = S_n/N$

ahol: E_n N processzonnal elért hatékonyság

S_n N processzonnal elért sebességnövekedés

N processzorok száma

Redundancia (Redundancy): $r = C_p/C_s$

ahol: r párhuzamos program redundanciája

C_p párhuzamos program műveleteinek száma

C_s soros program műveleteinek száma

Speed Up határa

Amdahl féle felső határ:

$$S_a = 1/(s+(1-s)/N)$$

ahol: S_a N processzorral elértetô sebességnövekedés felsô határa

s a feladat nem párhuzamosítható része

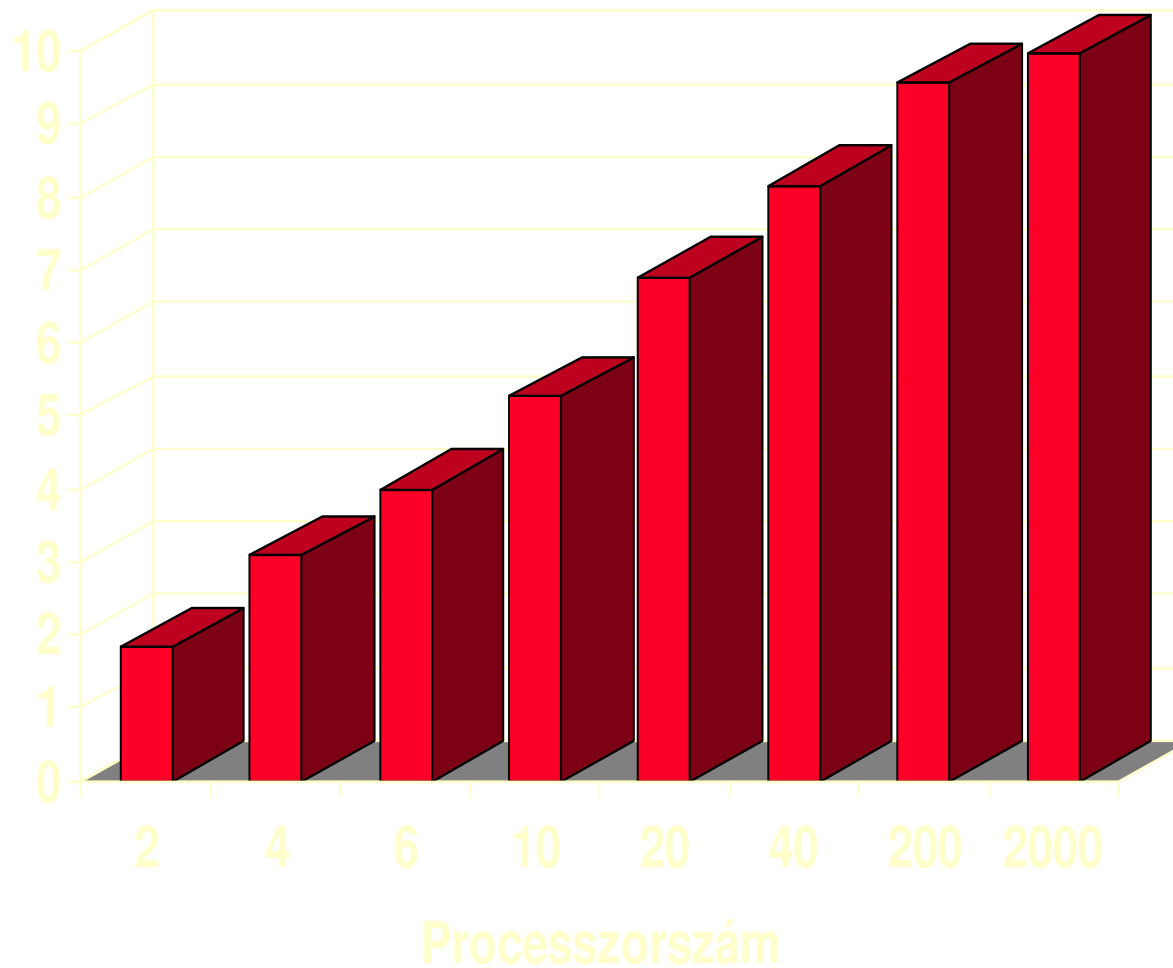
N processzorok száma

Az $(1-s)/N$ tagot elhagyva:

$$S_a < 1/s$$

összefüggést kapjuk, ami egy felső korlátot ad. Ez azt jelenti, hogy pl. 10% nem párhuzamosítható rész mellett $1/0.1 = 10$ adódik a speed up felső korlátjaként.

Nem gyorsítható korlátlanul



Programozási nyelvek

- Linda – közös memória modell, Tuple Space
 - out – kimásol egy adathalmazt a közös területre
 - in, inp – behoz egy adathalmazt (megszűnik)
 - rd, rdp – behoz egy adathalmazt
 - eval – végrehajt egy függvényt processzként

Egyszerű modell, de implementációs nehézségek vannak, főleg az üzenetküldéses architektúrákon.

Programozási nyelvek/2

- Express – elosztott memória modell
160 C-ből és fortranból hívható rutin:
 - programindítás, leállítás
 - logikai kommunikációs topológia felépítése
 - programok közötti kommunikáció, szinkronizáció
 - fájlműveletek
 - grafikai műveletek
 - teljesítmény analízis, debug

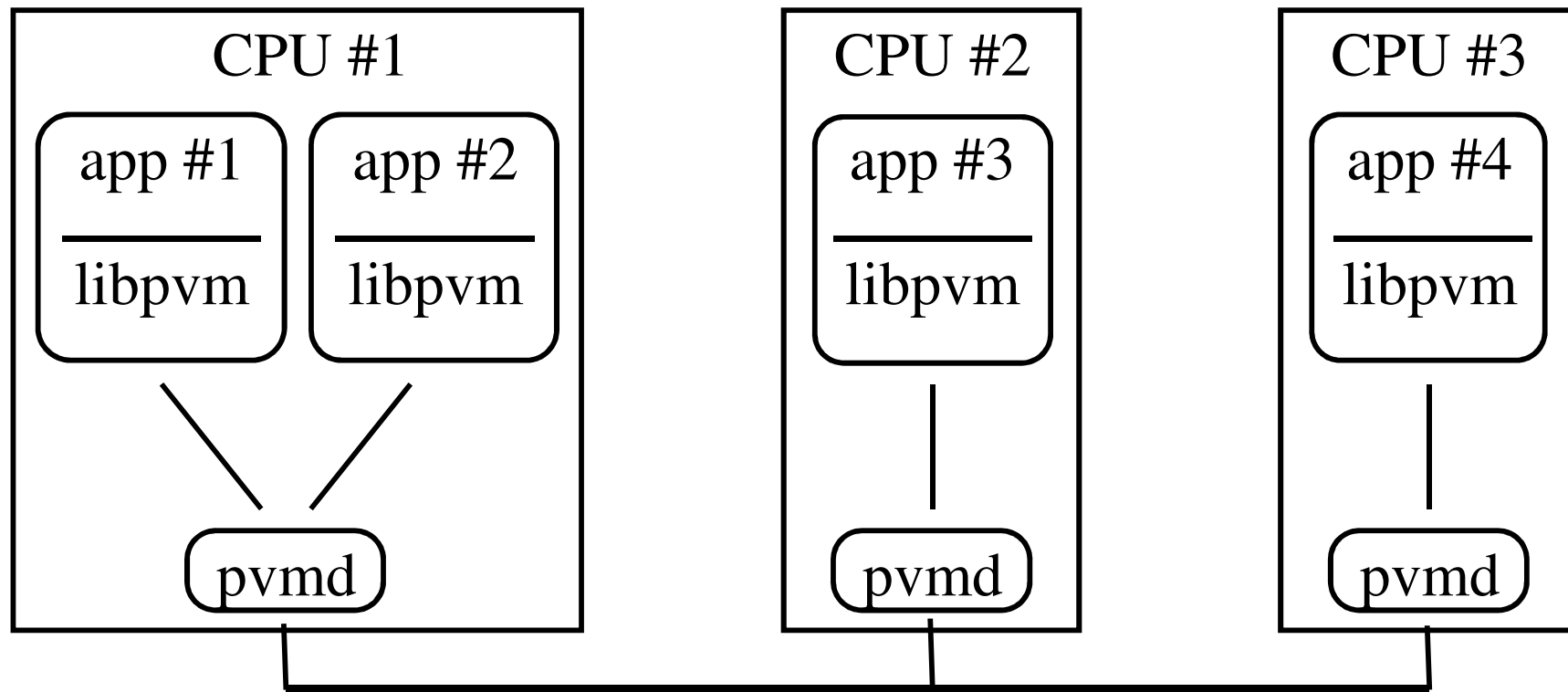
Programozási nyelvek/3

- PVM – elosztott memória modell
70 C-ből és fortranból hívható rutin:
 - programindítás, leállítás
 - programok közötti kommunikáció, szinkronizáció

Mire használják a PVM-et ?

- "Szegények" szuperkomputere
 - a szabad CPU kapacitások összegyűjthetők a munkaállomásokról és a PC-ről
- Több szuperkomputer összekapcsolásával hihetetlen számítási kapacitás állítható elő
- Oktatási eszköz
 - a párhuzamos programozás tanításához hatékony eszköz
- Kutatási eszköz
 - skálázható és költségkímélő

PVM alapkonceptiója



Összeköttetés

app # → Az alkalmazás része

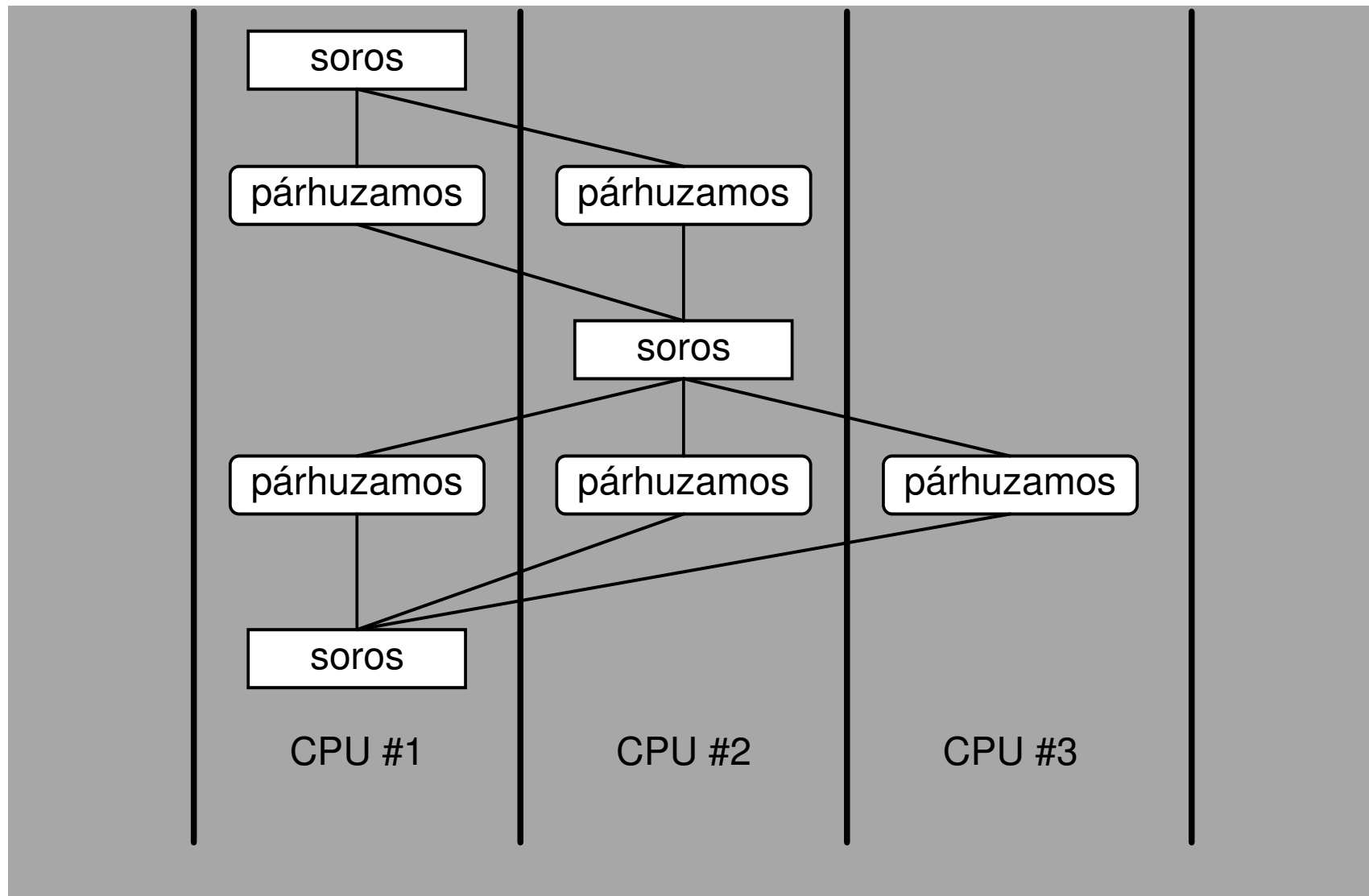
libpvm → Nyelvi interfész

pvmd → PVM daemon

A libpvm által nyújtott funkciók a négy csoportra oszthatók:

- Adminisztratív funkciók
Virtuális gép indítása, megállítása, állapotlekérdezés, új node felvétele
- Folyamatkezelő funkciók
Folyamatok indítása, megállítása
- Adatátviteli funkciók
Üzenetek összeállítása, elküldése, vétele, szétcsomagolás
- Szinkronizációs funkciók
Üzenetküldéssel, vagy barrier használatával

Párhuzamos program felépítése



Párhuzamosítási stratégiák

Kényszerű

- A program soros változatát futtatjuk párhuzamosan különböző adatokkal.
- Csak akkor kielégítő módszer, ha a soros változat elviselhető futási idejű.

Ciklusok párhuzamosítása

- Akkor alkalmazható, ha az egyes iterációk függetlenek egymástól

Felosztó párhuzamosítás (master / slave)

- Egy felügyelő taszk fut az egyik node-on
- Akkor alkalmazható, ha a felügyelő program feladatai egyszerűbbek mint a többi taszk feladatai.
- Ha a taszkok függetlenek egymástól, akkor jól skálázható a taszkok számának változtatásával.

Párhuzamosítási stratégiák /2

Egymást követő

- Minden node a következő node-nak adja tovább a részben feldolgozott adatot.
- Akkor használható, ha a soros része a feldolgozásnak lényegesen rövidebb, mint a párhuzamos rész.
- Rendszerint minden node azonos kódot futtat.
- Különösen alkalmas a gyűrű topológiához.

Régiók párhuzamosítása

- Az adatfüggőség régiókba lokalizálható.
- Akkor használható, soros végrehajtási idő nagyobb mint a párhuzamos.
- Rendszerint nagy kommunikációigényű.
- Legbonyolultabb.

Párhuzamosítási példa

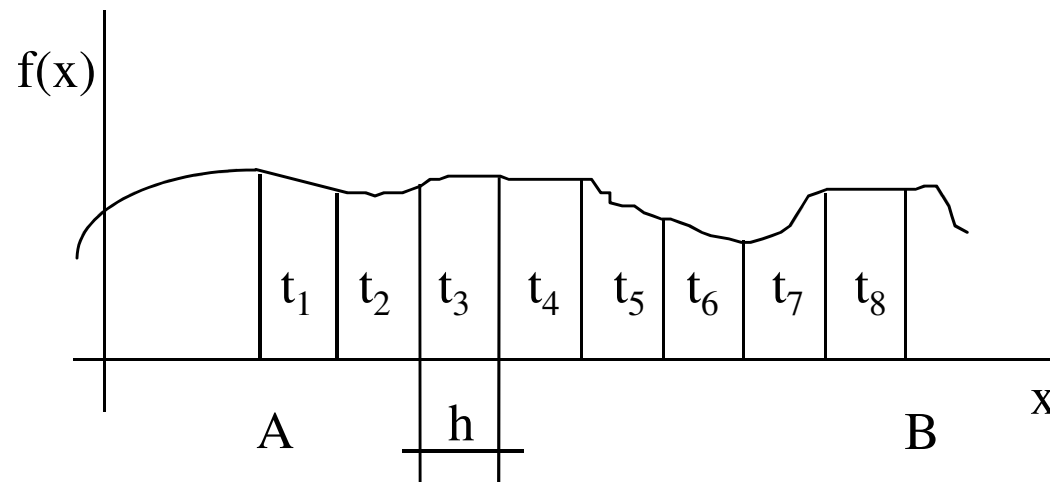
Számítsuk ki az

$$\int_A^B f(x) dx$$

integrál értékét egyszerű numerikus közelítéssel (téglány összegek)!

$$\int_A^B f(x) dx = h \cdot \sum_{i=1}^N f\left(A - \frac{h}{2} + i \cdot h\right) \quad \text{ahol } h = \frac{B-A}{N}$$

N=8 esetén pl:



Az egyes téglányok számítása egymástól függetlenül, párhuzamosan is elvégezhető.

Párhuzamosítási példa

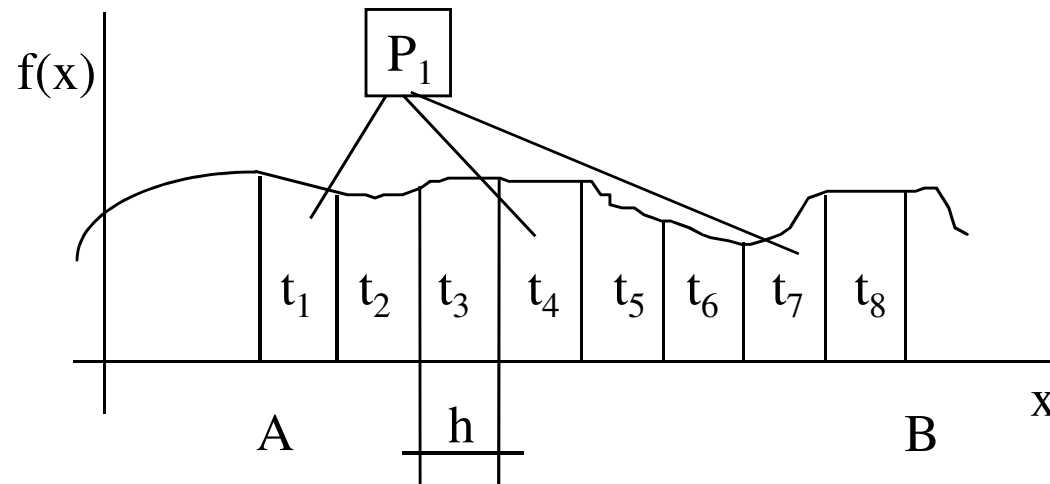
Számítsuk ki az

$$\int_A^B f(x) dx$$

integrál értékét egyszerű numerikus közelítéssel (téglány összegek)!

$$\int_A^B f(x) dx = h \cdot \sum_{i=1}^N f\left(A - \frac{h}{2} + i \cdot h\right) \quad \text{ahol } h = \frac{B-A}{N}$$

N=8 esetén pl:



Az egyes téglányok számítása egymástól függetlenül, párhuzamosan is elvégezhető. Pl. minden task csak minden M-edik téglányt számol ki, majd az összegezzük az eredményeket.

Párhuzamosítási példa

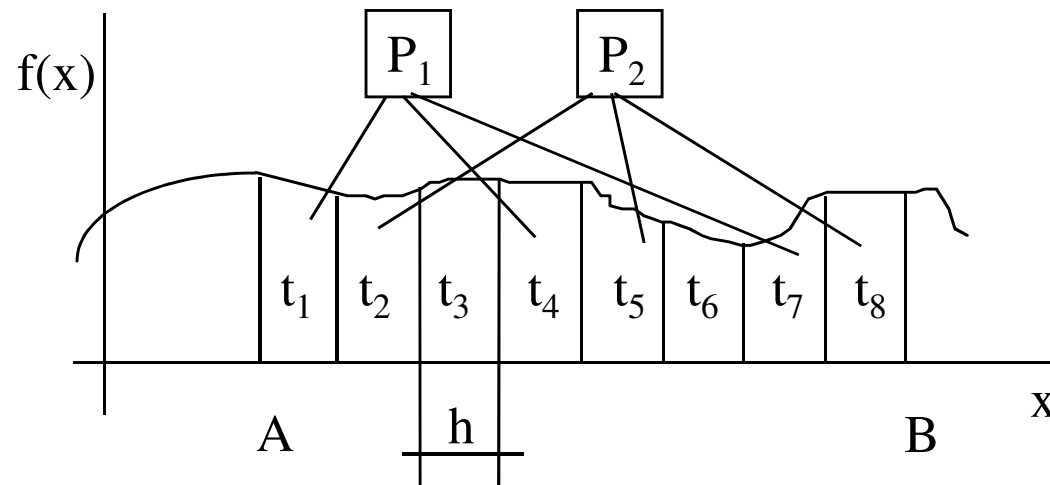
Számítsuk ki az

$$\int_A^B f(x) dx$$

integrál értékét egyszerű numerikus közelítéssel (téglány összegek)!

$$\int_A^B f(x) dx = h \cdot \sum_{i=1}^N f\left(A - \frac{h}{2} + i \cdot h\right) \quad \text{ahol } h = \frac{B-A}{N}$$

N=8 esetén pl:



Az egyes téglányok számítása egymástól függetlenül, párhuzamosan is elvégezhető. Pl. minden task csak minden M-edik téglányt számol ki, majd az összegezzük az eredményeket.

Párhuzamosítási példa

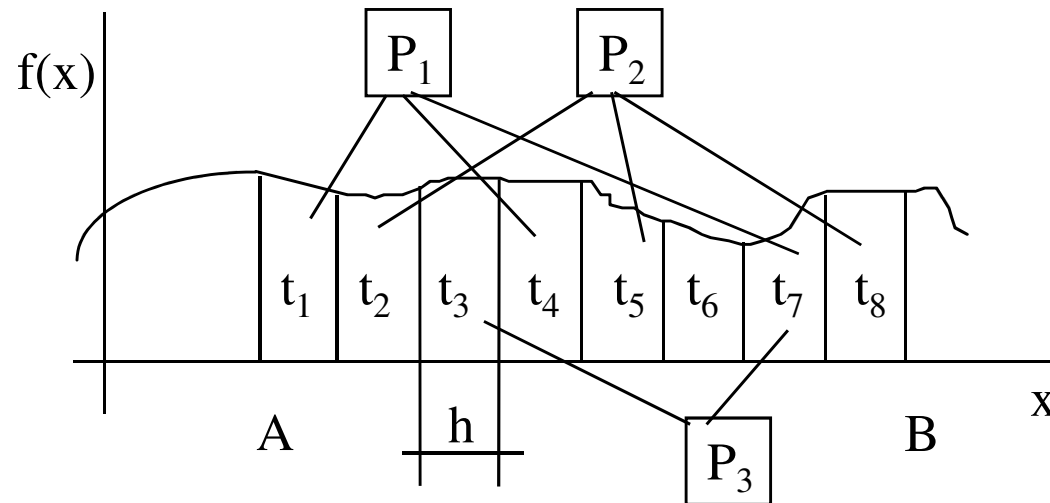
Számítsuk ki az

$$\int_A^B f(x) dx$$

integrál értékét egyszerű numerikus közelítéssel (téglány összegek)!

$$\int_A^B f(x) dx = h \cdot \sum_{i=1}^N f\left(A - \frac{h}{2} + i \cdot h\right) \quad \text{ahol } h = \frac{B-A}{N}$$

N=8 esetén pl:

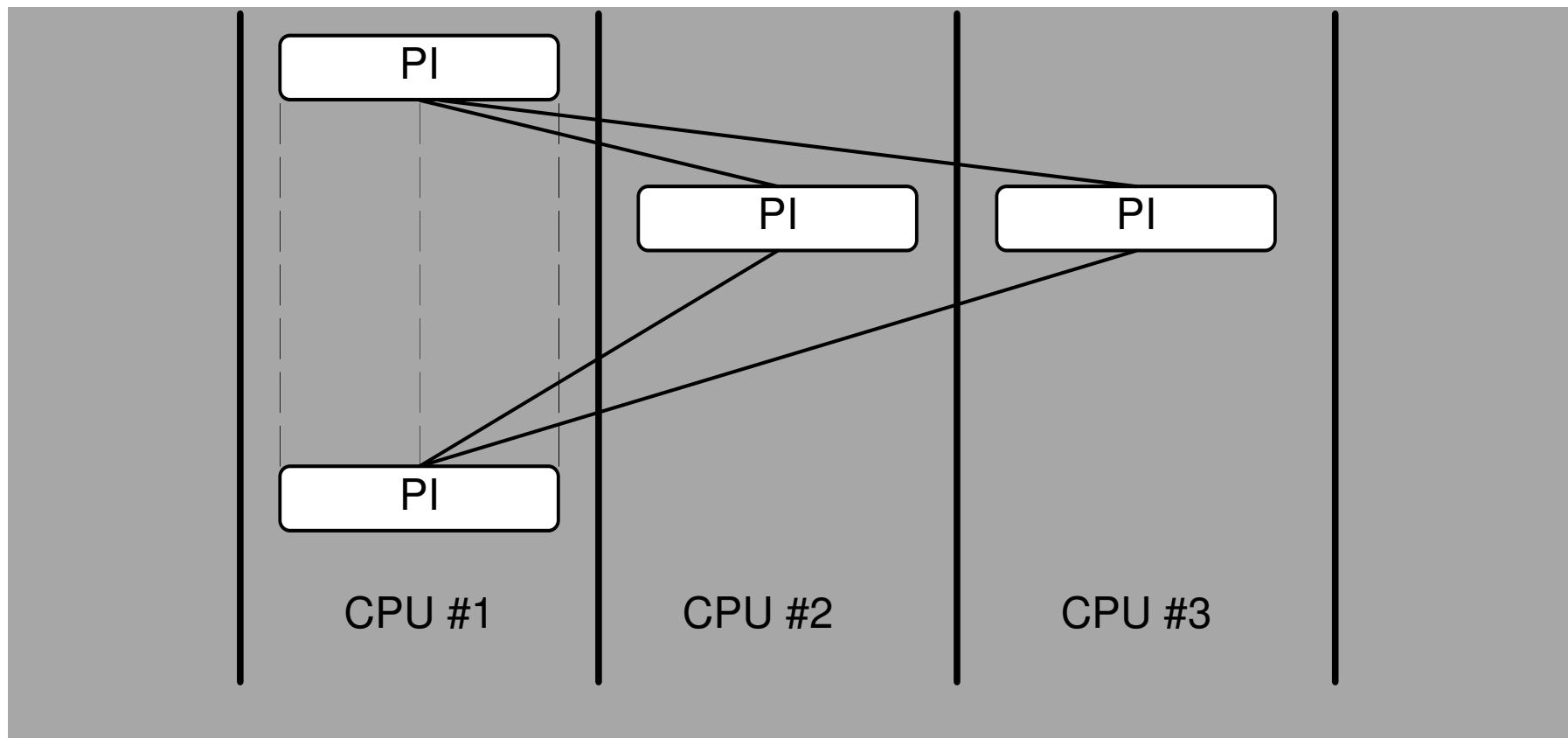


Az egyes téglányok számítása egymástól függetlenül, párhuzamosan is elvégezhető. Pl. minden task csak minden M-edik téglányt számol ki, majd az összegezzük az eredményeket.

Egy példa

Számítsuk ki PI értékét a $\int_0^1 \frac{4}{1+x^2} dx$ integrál numerikus integrálásával!

A fenti módszert alkalmazva SPMD programot írunk. A program első példánya bekéri a lépésközt és elindítja a többi példányt. Az egyes példányok csak minden M-edik téglányt számítanak, majd az eredményeket elküldik az indító taszk-nak.



pi.c program

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "pvm3.h"          /* PVM 3 include file */

#define f(x) ((float)(4.0/(1.0+x*x)))
#define PI ((float)(4.0*atan(1.0))) /* csak az ellenorzeshez */
#define MAXPROCS      32      /* node programok max szama */
#define TAG_N          111    /* N uzenettipus */
#define TAG_SUM        222    /* SUM uzenettipus */
#define TAG_TIDS       333    /* TIDS uzenettipus */
/*
 * Az elso peldany belep a PVM-be es elinditja
 * saját magát nproc peldanyban.
 */
void startup(int *pmynum, int *pnprocs, int tids[])
{
    int i, mynum, nprocs, mytid, numt, parent_tid;

    mytid = pvm_mytid();
    if (mytid < 0) {
        printf("sikertelen mytid\n"); exit(0);
    }
}
```

pi.c program /2

```
parent_tid = pvm_parent();
if (parent_tid == PvmNoParent) { /* ez az első példány */
    mynum = 0; tids[0] = mytid;
    printf ("Hány node példány(1-%d)?\n", MAXPROCS);
    scanf("%d", &nprocs);
    numt = pvm_spawn("pi", NULL, PvmTaskDefault, "", nprocs, &tids[1]);
    if (numt != nprocs) {
        printf ("Hibás taszk indítás numt= %d\n", numt); exit(0);
    }
    *pnprocs = nprocs; /* node példányok száma */
    pvm_initsend(PvmDataDefault); /* tid info az mindenkinek */
    pvm_pkint(&nprocs, 1, 1);
    pvm_pkint(tids, nprocs+1, 1);
    pvm_mcast(&tids[1], nprocs, TAG_TIDS);
} else { /* ez nem első példány */
    pvm_recv(parent_tid, TAG_TIDS);
    pvm_upkint(&nprocs, 1, 1);
    pvm_upkint(tids, nprocs+1, 1);
    for (i = 1; i <= nprocs; i++)
        if (mytid == tids[i]) mynum=i;
}
*mynum = mynum;
}
```

pi.c program /3

```
/*
 * N ertek eloallitasa. (Az elso peldany bekeri).
 */
void solicit(int *pN, int *pnprocs, int mynum, int tids[])
{
    if (mynum == 0) {                /* ez az elso taszk */
        printf("Kozelites lepesszama:(0 = vege)\n");
        if (scanf("%d", pN) != 1) *pN = 0;
        pvm_initsend(PvmDataDefault);
        pvm_pkint(pN, 1, 1);
        pvm_pkint(pnprocs, 1, 1);
        pvm_mcast(&tids[1], *pnprocs, TAG_N);
    } else {                          /* egyebkent a fonok node kuldi */
        pvm_recv(tids[0], TAG_N);
        pvm_upkint(pN, 1, 1);
        pvm_upkint(pnprocs, 1, 1);
    }
}
```


pi.c program /4

```
main()
{
  float sum, w, x;
  int    i, N, M, mynum, nprocs, tids[MAXPROCS+1];

  startup(&mynum, &nprocs, tids);
  for (;;) {
    solicit (&N, &nprocs, mynum, tids);
    if (N <= 0) {
      printf("A %d. peldany kilep a virtualis gepbol\n", mynum);
      pvm_exit(); exit(0);
    }
  }
  /*
  * Szamitas. M=nprocs+1 peldany van, igy csak minden M.
  * teglanyt szamolja egy processz. */
  M = nprocs+1;
  w = 1.0/(float)N;
  sum = 0.0;
  for (i = mynum+1; i <= N; i += M)
    sum = sum + f(((float)i-0.5)*w);
  sum = sum * w;
```

pi.c program /4

```
/* Eredmenyek feldolgozasa */
if (mynum == 0) { /* ha ez az elso peldany */
    printf ("Elso peldany szamitasa x = %7.5f\n", sum);
    for (i = 1; i <= nprocs; i++) {
        pvm_recv(-1, TAG_SUM);
        pvm_upkfloat(&x, 1, 1);
        printf ("Elso peldany x = %7.5f erteket kapott\n", x);
        sum = sum+x;
    }
    printf("sum=%12.8f\terr=%10e\n", sum, sum-PI);
    flush(stdout);
} else { /* tovabbi peldanyok elkuldik az eredmenyt. */
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(&sum, 1, 1);
    pvm_send(tids[0], TAG_SUM);
    printf("A %d. elkuldte a reszosszeget: %7.2f \n", mynum, sum);
    fflush(stdout);
}
}
}
```

Message Passing Interface (MPI)

- Alapvetően más célokkal fejlődött ki:
 - szabványos, a gyártók által elfogadott, speciális hw. környezetet is támogató fejl. környezet.
 - hosszú nyűgös fejlődés
 - kezdetben csak statikus processzkezelés
 - nem igényli a virtuális gép előzetes felépítését, mert a teljes kommunikációs séma az alkalmazáshoz szerkesztődik.
 - Ezzel szemben a PVM op.r. funkciókat nyújt. viszonylag gyorsan fejlesztették.

Cluster koncepció

- Gyors hálózattal összekapcsolt gépek
- Gyakran közös fájlrendszer
- CPU vagy tárolási kapacitás növelése
- Paraméter study, vagy párhuzamos alkalmazások

