Párhuzamos programozást támogató nyelvi eszközök összehasonlítása

szerző: Frohner Ákos
témavezető: Horváth Zoltán

Bevezetés

Eme dolgozat témája párhuzamos programok írását támogató nyelvi eszközök összehasonlítása. A nyelvi eszközök itt konkrét programozási nyelvek, amelyek tartalmaznak párhuzamosságot támogató elemeket, vagy hagyományos nyelvekhez kapcsolódó rutinkönyvtárak.

A világon jelenleg számtalan olyan eszköz van, amely párhuzamosan futó programok írására használható – kezdve az I/O egységek alacsonyszintű programozásától, az egyes operációs rendszerek által biztosított megoldásokon és az itt bemutatandó eszközökön keresztül a grafikus felületű CASE eszközökig – de ezekből csak egy réteg kerül bemutatásra.

Az első fejezet ezen a széles palettán helyezi el a később részletesen ismertetésre kerülő eszközöket. Bemutatja a jelenleg elérhető párhuzamos hardware architektúrákat, absztrakt elméleti osztályokat egy-egy konkrét példával. Ezen architektúrákat elfedő absztraktabb lehetőségeket biztosító operációs rendszereket. Itt szó esik az általuk nyújtott alapvető szolgáltatásokról is, amelyek sokféleségük miatt nem terjedtek el széles körben, vagy nem használhatóak kényelmesen párhuzamos programok írására. E fölött helyezkednek el a programozási nyelvek és a rutinkönyvtárak, amelyekről a dolgozat fő témájául szolgálnak. Ezeknél magasabb absztrakciós szintet nyújtó eszközök a problémakör viszonylag új volta miatt nem nagyon vannak.

A következő fejezetben esik szó az egyes eszközökről. Az eszközök ismertetése előtt egy egységes osztályozási rendszer, illetve összehasonlítható tulajdonságok halmaza van definiálva, az eszközök elemzése végett. A fejezet hátramaradó részében kerülnek bemutatásra részletesebben ezek: rövid áttekintés a történetükről, a részletes leírás a lehetőségekről, egy-két példaprogrammal mutatva be a konkrét használatot.

A szakdolgozat végén konkrét teszteredmények találhatóak, amelyek az egyes eszközök hatékonyságáról és absztrakciós szintjéről nyújtanak tájékoztatást a különböző architektúrákon mutatott futási eredmények és a programlisták által.

1. ÁTTEKINTÉS 4

1.1. PÁRHUZAMOS MODELLEK, OSZTÁLYOZÁSI SZEMPONTOK 4
1.2. HARDWARE 4
1.2.1. SISD (Single Instruction Single Data) 5
1.2.2. SIMD (Single Instruction Multiple Data) 5
1.2.3. MISD (Multiple Instruction Single Data) 5
1.2.4. MIMD (Multiple Instruction Multiple Data) 5
1.3. OPERÁCIÓS RENDSZEREK LEHETŐSÉGEI 8
1.4. NYELVEK, ESZKÖZÖK 9

2. NYELVEK, ESZKÖZÖK 10

2.1. OSZTÁLYOZÁSI SZEMPONTOK, JELLEMZŐK 10
2.1.1. Feladatok osztályozása 10
2.1.2. Modellek, eszközök osztályai 11
2.1.3. Jellemzők 17
2.1.4. Példák 18
2.2. MPI 20
2.2.1. Bevezető az eszközről 20
2.2.2. Felhasználói felület 22
2.2.3. Példák 35
2.2.4. Összefoglalás 37
2.3. PVM 38
2.3.1. Bevezető az eszközről 38
2.3.2. Felhasználói felület 39
2.3.3. Példák 45
2.3.4. Összefoglalás 46
2.4. CC++ 47
2.4.1. Bevezető az eszközről 48
2.4.2. Felhasználói felület 48
2.4.3. Összefoglalás 56
2.5. HPF 57
2.5.1. Bevezető az eszközről 57
2.5.2. Felhasználói felület 58
2.5.3. Példák 64
2.5.4. Összefoglalás 65

3. FÜGGELÉK 67

3.1. IRODALOMJEGYZÉK 67

1. Áttekintés

1.1. Párhuzamos modellek, osztályozási szempontok

A hagyományos számítógépes feladat-megoldási módszerek sok területen elégtelennek bizonyultak, ezért próbálnak napjainkban új utakat keresni a problémák megoldására. Ezek egyik fajtája a párhuzamosság felhasználása.

A felhasznált megoldást két irányból is megvizsgálhatjuk: milyen feladatfajtát old meg, illetve milyen modellt használ a feladat megoldásához.

A párhuzamos megoldások igénye következő problématerületeken jelentkezett:

- Az első ezek közül a futási idő csökkentése volt. Ezen a területen nagyon sok fajta megoldás létezik a különböző párhuzamos módszerek felhasználása terén. Ezek a módszerek a problémák megoldásának szinte minden szintjén felfedezhetőek a mikroprocesszorok belső felépítésétől kezdve, a magas szintű programozási nyelvekig.
- Következő terület a munkamegosztás, a speciális feladatok elosztásai. Ez is tág terület hiszen az I/O processzorok és a CPU közötti munkamegosztástól a különböző kliens/szerver alkalmazásokig sok minden ide sorolható.
- A harmadik feladatosztály a hibatűrő képesség, illetve megbízhatóság növelése.

Ezek a feladatok a megoldásának keresése során hasonló problémákat vet fel, amelyek a különböző absztrakciós szinteken hasonló modellekkel írhatóak le.

A párhuzamos modellek fő osztályozási szempontjai az absztrakciós szintektől függetlenül:

- párhuzamosság típusa: taszk, vagy adat párhuzamosságot támogat
- szinkronizációs, kommunikációs eszközök (ezek inkább a kommunikációs eszközök típusai, hiszen ezek felhasználhatóak szinkronizációra is)

Ezek a szempontok persze az adott szinten más-más irányba finomíthatóak.

1.2. Hardware

A fentiekben feladatosztályok megoldására nagyon sok speciális hardware megoldás született.

A futásidő gyorsítására a párhuzamosságot először a hetvenes évek végén megjelent szuperszámítógépeket kell megemlíteni. Ezek vektorprocesszorokat használtak, amellyel azonos műveleteket lehetett elvégezni nagy mennyiségű adatokon. Ez a megoldás főleg speciális problémák megoldására, nagy mennyiségű adaton végzett műveletekre használható. (pl.: Cray-1, Los Alamos National Laboratory, 1976)

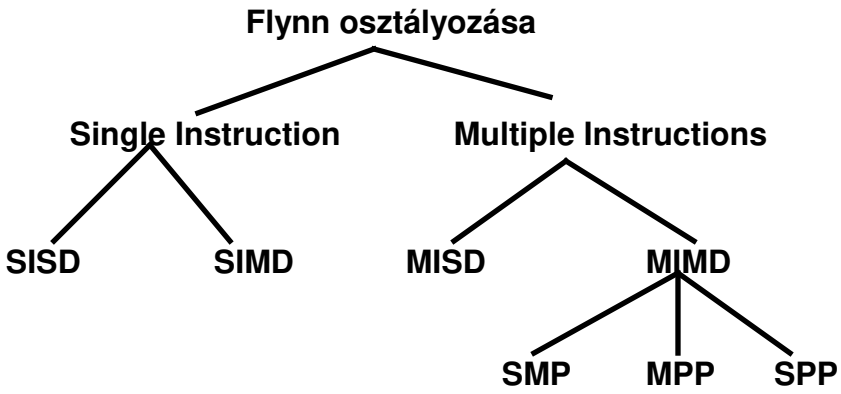
A következő lépést a RISC processzorok megjelenése jelentette, amelyekben az utasításonkénti fix hosszúsági végrehajtási idő nagyon megkönnyítette a processzorokon belüli elemi utasításonkénti párhuzamos végrehajtást.

Ez a módszer manapság már majdnem minden mikroprocesszorban jelentkezik. A processzorokban többszörözött feldolgozó egységek vannak, amelyek az elemi utasítások szintjén párhuzamosítják a szekvenciálisan íródott programokat.

Ennek a módszernek persze vannak korlátjai, hiszen egy hagyományos program utasításai nem párhuzamosíthatóak tetszőleges mértékben.

Hasonló fejlődés figyelhető meg a számítógépek makro-architektúrájában is.
Eleinte a futásidő csökkentésére egyre nagyobb sebességű processzorokat használtak, de egy idő után több hagyományos processzort kapcsoltak össze egyetlen számítógépben.

E számítógépek főbb osztályai:



1. Ábra: Flynn osztályozása

1.2.1. SISD (Single Instruction Single Data)

A hagyományos számítógépek ilyenek.

1.2.2. SIMD (Single Instruction Multiple Data)

Nagyon sok egyszerű processzorból álló gépek, amelyek mindegyike rendelkezik saját memóriával, de mindegyiken azonos program fut. Ezek főleg mátrixműveletek megoldására alkalmas rendszerek.

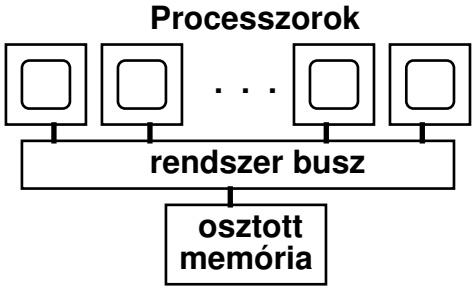
Példa:
Thinking Machine Corporation CM200, 16-tól 65536 darab processzort tartalmazó gép. Ebben a viszonylag egyszerű processzorokat a számuktól függően 4 vagy 16 dimenziós hiperkockába rendezik, ahol az élek mentén kapcsolják össze őket.

1.2.3. MISD (Multiple Instruction Single Data)

Nem nagyon létezik ilyen gép, csak az osztályozás teljessége miatt került bele az ábrába.

1.2.4. MIMD (Multiple Instruction Multiple Data)

SMP ***(Symmetric multiprocessing)***



2. Ábra: SMP számítógép

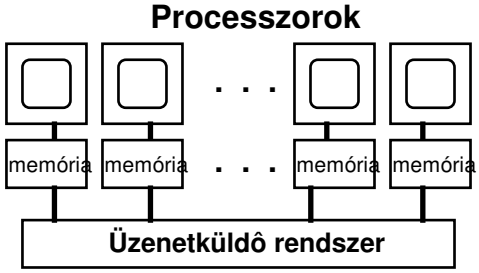
Az SMP rendszerek jellemzője, hogy a rendszerbuszra egyetlen fizikai memória csatlakozik, amelyet minden processzor lát. Ennek az architektúrának nagy előnye, hogy az osztott memórián keresztül nagyon egyszerű a programok közötti kommunikáció, illetve szinkronizáció megoldása, hagyományos programok nagyon könnyen átültethetőek ilyen környezetbe. Hátránya, hogy a memória elérésének fizikai korlátjai meghatározzák a lehetséges processzorok számát.

Ilyen rendszerek:

- Cray Research, CS6400, amely maximum 64 SuperSPARC processzor összefogására használható
- SGI Power Challange, amely maximum 18 MIPS10000 processzor beépítésére alkalmas
- SUN Sparc Server, amelybe 4 darab UltraSPARC processzor rakható

Ezen rendszerek szépsége, hogy hagyományos operációs rendszereket futtatnak, ezért kezelésük, programozásuk nem kíván nagy többletráfordítást, mégis nagy számítási teljesítményt képviselnek.

MPP ***(Massively parallel processing)***



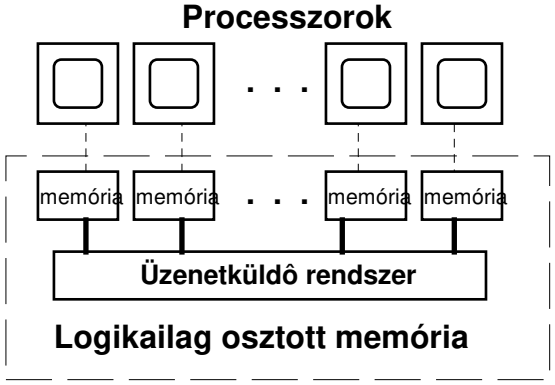
3. Ábra: MPP számítógép

Az MPP rendszerek lényegében önálló memóriával rendelkező processzorokból felépített nagy számítógépek. Egy egyszerű LAN is felfogható ilyen számítógépnek, a különbség a processzorok közötti kapcsolatban jelentkezik. Egyszeri LAN-okban az egyedi gépek közötti kapcsolat felépítése elég lassú, ezekben a gépekben erre a feladatra speciális hardware elemeket használnak amely mind a kapcsolat felépítését, mind az adatátvitelt nagyon meggyorsítja.

Az MPP rendszerek nagy előnye, hogy felépítésüknél fogva nagyon könnyen bővíthetőek. Hátrányuk, hogy a processzorok közötti kommunikációra általában csak üzenetküldés használható, amely a hagyományos programok használatát megnehezíti, és új - a párhuzamosságot már figyelembe vevő - programok írását igénylik.

- IBM SP2, 1-től 1024 processzorig bővíthető rendszer. A számító egységek lényegében önálló számítógépek, melyek mindegyikén egy-egy operációs rendszer fut. Az összeköttetést nagy teljesítményű kapcsoló dobozokkal biztosítják, amelyek 8 pontból álló teljes gráfokban tudják biztosítani a kapcsolatot.
- Intel Terra FLOP-os teljesítményű gépe. Hagyományos elemekből épített gép, ahol a több mint 9000 Pentium Pro processzort alapvetően 2 dimenziós hálókba szervezik, amelyek között nagy sebességű utakat építenek ki.
- Parsytec PowerXplorer, 4-től 512 darab PowerPC processzorból felépített gép. A processzorokat 2 dimenziós hálóba rendezik, amelyben T800-as transputerek biztosítják a kommunikációs kapcsolatokat.

*SPP **(Scalable parallel processing)***



4. Ábra: SPP számítógép

MPP rendszer amelyben a processzorok közötti kommunikációs eszköz szimulálni tudja, az egyetlen osztott memóriát. A processzorok számára nem saját memóriájuk elérése csak időbeni különbséget jelent. Lényegében ötvözik az SMP és MPP rendszerek előnyeit, viszont ehhez sokkal bonyolultabb hardware megoldásokra van szükség.

Példák:

- Convex Exemplar, amely maximum 128 PA RISC 7200-as processzorig bővíthető. Itt 8-8 processzor egy kockában van összefogva. Ezeket az egységeket egyirányú üzenettovábbításra alkalmas tóruszokba fogják össze.
- Cray Research T3D, T3E, amely maximum 1024 darab Alpha processzort tartalmazhat. A gyors kommunikáció érdekében ezeket 3 dimenzióban kétirányú kommunikációra alkalmas tóruszok kapcsolják össze. Jellemzője, hogy a nem saját memória elérésének ideje maximum hatszorosa a saját memória elérésének.

A fennmaradó két problémakör megoldására is sok megoldás született, amely kapcsolatba hozható a párhuzamossággal. Ilyen lehet egy egyszerű PC I/O processzorra, amely a CPU-tól függetlenül tud működni, vagy a repülőgépek többszörözött számítógépes rendszere a megbízhatóság növelésére.

1.3. Operációs rendszerek lehetőségei

Operációs rendszerek szintjén nyújtott támogatás a párhuzamos programok támogatására nehezen választható el a programozási nyelvektől és a bennük használható rutinkönyvtáraktól.

A határ – lehet, hogy ez elég szubjektív – a megoldható feladatok fajtája alapján húzható meg. Egy operációs rendszernek, illetve az általa nyújtott eszköznek megbízhatóan és biztonságosan kell működnie a feladatok széles skáláján.

A feladatok széles skálája miatt az operációs rendszer által nyújtott szolgáltatások nagyon egyszerűek lehetnek, tehát ezekre épülő eszközök használata szükséges lehet a problémák egyszerű megoldása érdekében.

A megbízhatóság természetes igényi bármely probléma megoldása során, a biztonság viszont inkább napjainkban került előtérbe a sokfelhasználós rendszerek megjelenésével. A biztonságot biztosító eszközöket csak az operációs rendszer szintjén lehet hatékonyan megvalósítani, tehát egy eszköz osztályozásakor ez a szempont irányadó lehet.

Operációs rendszerek általában a következő feladatokat oldják meg:

- folyamatok indítása, leállítása, ütemezése
- I/O igények kielégítése
- egyszerű eszközök a folyamatok közötti kommunikációra

Párhuzamosan dolgozó, illetve osztott operációs rendszerek ezen feladatok megoldására a következő lehetőségeket biztosítják:

- **UNIX fork():** új folyamat indítása. Operációs rendszertől függ, hogy ez valóban párhuzamos módon, azaz egy másik processzoron indul-e el.
- **System V. IPC:** programok közti kommunikáció, szinkronizációra szolgáló megoldások: szemaforok, osztott memóriakezelés, üzenetküldés támogatása. Hátránya az, hogy csak osztott memóriát használó gépeken működik, és elég nehezen kezelhető lehetőségeket biztosít.
- **Hálózati protokollok**: TCP/IP, SPX/IPX és DECnet protokollok önálló gépek közötti kapcsolatok megteremtésére.
- **thread-ek, Light Weight Process-ek**: osztott memóriával kommunikáló folyamatok kezelésének eszköze
- **Távoli eljáráshívás**: A legtöbb osztott operációs rendszer illetve osztott programozási környezet a távoli eljáráshívás módszerét veszi alapvető eszköznek a különböző feladatok megoldásában. Az egyszerűség érdekében ehhez nagyon hatékony implementációkat, és nagyon egyszerű felületet biztosítanak, hogy ne legyen látszólag különbség egy gépen belüli, illetve azok közötti eljáráshívásoknál. Ilyen eszközök vannak a Choros, illetve Mach alapú mikrokerneles rendszerekben, valamint az Amoeba osztott operációs rendszerben is. Ezt az eszközt a SUN is elkészítette, erre épül sok gépek közötti szolgáltatása (pl.: NFS, NIS+, Kerberos).

A fejezet elején szereplő feladat osztályozási szempontok szerint:

- **futásidő csökkentése**: több processzorra íródott, vagy osztott operációs rendszerek képesek a feladatokat a processzorok között terheltségüktől függően elosztani. Az adott problémáról nem rendelkezhetnek több információval, ezért más megoldást csak magasabb szinten lehet keresni.

- **funkcionális elosztás**: rendszerfüggő. Itt az operációs rendszernek valamilyen módon plusz információkat kell adni a problémáról.
- **megbízhatóság, biztonság**: megbízhatóságot egy operációs rendszer az alatta lévő hardware hibáinak elfedésével biztosíthat. Ha erre nem képes (például egy programot futtató processzor elromlik), akkor ezt jelenti az alkalmazónak. Biztonsági kérdésekben az operációs rendszeré a fő felelősség. Ez védi egymástól a rosszul viselkedő programokat, esetleg felhasználókat.

Párhuzamos modellek szempontjából egy operációs rendszer nagy mértékben az alatta lévő hardware-től függ. Néhány esetben az operációs rendszer a hardware néhány jellemzőjét módosíthatja, például az IBM SP2-es MPP rendszerén az operációs rendszer egy SPP architektúrát is szimulálhat.

1.4. Nyelvek, eszközök

A világon nagyon sok olyan programozási nyelv, illetve eszköz van, amely valamilyen fokon támogatja a párhuzamosságot. Ezek közül található itt néhány. Ezek osztályozási szempontjai, illetve egy-egy eszköz részletesebb elemzése a következő fejezetben található.

- Algol 68 szemaforai
- Concurrent Pascal monitorjai
- Occam, tarnsputerek assembly nyelve
- Linda osztott memóriája, szemaforjai és a tupple nevű egyedi eszköze
- P4 üzenetküldési könyvtár (MPI és PVM is örökölt tőle)
- Ada, Ada95
- Objective C distributed class nevű eszköze, mint az objektumok párhuzamosításának egy példája
- TCL/Tk distributed modulja
- C*, CC++, pC++, HPC++, uC++ – C és C++ nyelvek párhuzamosított változatai
- FORTRAN 90 variánsok: CMF (Connection Machine Fortran), Fortran M, HPF (High Performance Fortran). Itt kell megemlíteni, hogy ez generálja a párhuzamos kódot, itt nem is kell semmi explicit dolgot kiírni
- Java nyelv thread-jei

2. Nyelvek, eszközök

2.1. Osztályozási szempontok, jellemzők

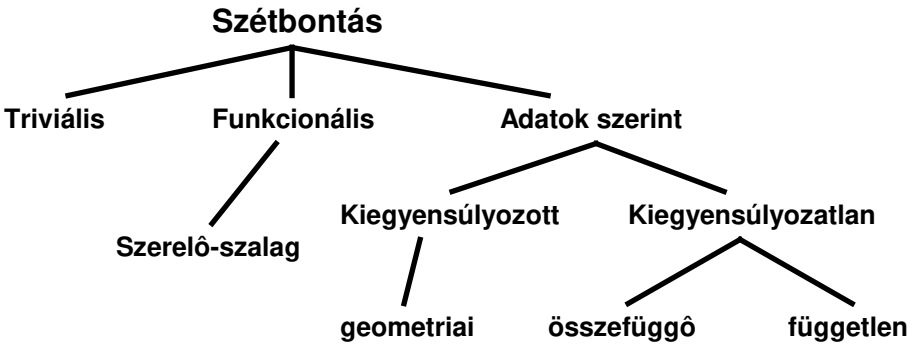
A különböző programozási nyelvek és eszközök osztályozását is két oldalról lehet megközelíteni: az általuk nyújtott lehetőségek, illetve a velük megoldható feladatok oldaláról.

2.1.1. Feladatok osztályozása

A párhuzamos programozási eszközöket nagyon sokszor a futásidő csökkentése érdekében alkalmazzák. A problémamegoldás sikeressége nagy mértékben függ a feladat fajtájától.

Ha van egy szekvenciális program, akkor azt többféle módon is fel lehet bontani kisebb részekre, amelyek már párhuzamosan is végrehajthatóak. A részekre bontás eredménye sok kis szekvenciális programrészlet lesz, amelyek között valamilyen eszközzel szinkronizálni, vagy kommunikálni kell. A szinkronizációs, illetve kommunikációs eszközök óhatatlanul valamilyen többletköltséggel járnak, amelyek korlátozzák a párhuzamosítási folyamatot abban, hogy minden határon túl gyorsítsa egy feladat megoldását.

A feladatok a részekre bontás szempontjából a következő kategóriákba eshetnek:



5. Ábra: Feladat szétbontása

- **Triviálisan** szétbontható feladatok azok, melyekben az egyes feladatrészeknek semmi közük sincs egymáshoz, egymástól teljesen függetlenül megoldhatóak. Az ilyen problémákhoz lényegében bármilyen eszközt használva nagyon jó gyorsítási eredményeket lehet elérni.
- **Funkcionálisan** szétbontható feladatok azok, melyek részfolyamatokra bonthatóak, de egy-egy rész függ egy, vagy több előtte lévő rész eredményétől. Az ilyen feladatoknál a párhuzamosítás módszere nagy mértékben függ a feladat algoritmusától. Ezen feladatok egy speciális osztályát képezik a **szerelő-szalaggal** megoldható feladatok, amelyeknél lényegében kis adatokkal kell sorban ugyanazt a műveletet-sorozatot elvégezni. Ilyen feladatokhoz a legegyszerűbb megoldás a UNIX operációs rendszer által biztosított *pipe* eszköz lehet.
- **Adatok szerint** felbontható feladatoknál lényegében ugyanazt a műveletet kell elvégezni egy adathalmaz minden elemén.
  - Ha az elvégzendő művelet mennyisége nem függ az adattól, akkor **kiegyensúlyozott** módon eloszthatóa a feladat a számító egységek között. Az elosztás legegyszerűbb módja, ha az adatokat valamilyen geometriai módon tudjuk szétválasztani, például egy képet vízszintes csíkokra vágva feldolgozni. Ilyen problémákat a párhuzamosító fordítóprogramok is nagyon jól tudnak kezelni.

- Ha az elvégzendő lépések száma az adatoktól függ, akkor **kiegyensúlyozatlan** problémákról lehet beszélni. Az ilyen feladatok megoldása során további problémákat vethet fel az, ha a kiszámolandó érték nem csak az adott adattól, hanem a szomszédosaktól is függ. Ha az adatok teljesen függetlenek, akkor dinamikus kiegyensúlyozással (pl. task farm) általában megoldható a probléma. Ezek a feladatok már párhuzamos algoritmusok írását, illetve folyamatok párhuzamosítását támogató rendszereket igényelnek.

A párhuzamosságot támogató eszközök másik nagy felhasználási területe a munkamegosztás támogatása.

Sok olyan feladat van, melyek nagyon jól kezelhető, illetve definiálható részekre bonthatóak. Ezen feladatok megoldásában is érdemes ezeket a részeket megtartani, mert így sokkal rugalmasabban alakítható át a rendszer egy másik feladat megoldására. Ha ezeket a részfeladatokat különálló programokban, illetve rendszerekben valósítják meg, akkor meg kell oldani az ezek közötti kommunikációt szinkronizációt.

Ilyen feladat lehet például egy kliens/server modellre épülő adatbázis-kezelő alkalmazás, melynél a kliens és a szerver közötti kommunikációt párhuzamosság támogatására létrehozott nyelvekkel, vagy eszközökkel lehet megszervezni.

Megbízható alkalmazások, illetve rendszerek igénye általában a fenti feladatokkal együtt szokott jelentkezni, de ezen a területen vannak olyan problémák, amelyek magukban is szükségessé teszik párhuzamosságot támogató eszközök használatát, pl.: repülés irányító számítógép működésének tükrözése több gépen egy repülőgépen, majd a kapott eredmények összevetése.

2.1.2. Modellek, eszközök osztályai

A vizsgálat másik iránya a lehetőségek felmérése, azaz milyen eszközöket biztosít az adott környezet a programozó számára a fenti feladatok megoldására.

Az eszközök vizsgálatánál az alábbi három szempont elsődleges:

- a párhuzamosság típusa
- kommunikációs eszközök
- szinkronizációs eszközök

Párhuzamosság típusa

A legfontosabb szempont, hogy az adott eszköz adat, vagy folyamat párhuzamosítását támogatja.

Az **adat-párhuzamosságot** támogató eszközök általában egy szekvenciális programnyelvre épülnek, melyet nagy mátrixokkal végzett műveletek párhuzamosítására szolgáló elemekkel bővítenek ki. Ezek párhuzamosító fordítók, ahol a programozónak a legegyszerűbb esetben semmi beleszólása sincs az elkészülő program processzorok közötti elosztásába.

A nyelveket általában kiegészítik olyan elemekkel is amelyekkel az adatok elosztása finomabban is szabályozható, de nem adnak eszközöket nem kiegyensúlyozott problémák megoldására. Általában nincs lehetőség arra sem, hogy teljesen független utasítássorozat hajtódjanak végre egyes adatelemeken.

Ilyen nyelvek a FORTRAN-ra épülő FORTRAN 90, illetve ennek variánsai, valamint a C++-ra épülő pC++ (parallel C++) nyelv.

A **folyamat-párhuzamosságot** támogató nyelvek – valamilyen szinten – egymással párhuzamosan futó folyamatokat valósítanak meg, melyek közötti kommunikációt és szinkronizációt a programozónak kell megszerveznie.

Ezek a nyelvek illetve eszközök párhuzamos algoritmusok megvalósítására használhatóak, nem kínálnak lehetőséget egy hagyományos szekvenciális program egyszerű párhuzamosítására.

Ebbe a csoportba sorolható lényegében az összes megmaradt eszköz.

Vannak törekvések a két fajta módszer összemosására egyetlen nyelvben, illetve eszközben. Ilyen a FORTRAN 90-ben párhuzamosan indítható függvények, amelyek viszonylag független kódok futtatását teszik lehetővé egyes adatelemeken. Ennek a törekvésnek a jegyében készül az HPC++ (High Performance C++) nyelv is a pC++ és a CC++ (Compositional C++) nyelvek kombinációjából.

A folyamatok párhuzamosságánál meg lehet vizsgálni, hogy milyen **folyamat-modelleket** támogatnak:

- A **statikus** modellben a folyamatok száma a program szövegéből, vagy a futtató környezetből kiderül. Az ilyen programoknál az összes folyamat a program megkezdésekor elindul, az eszköz csak ezen folyamatok közötti interakció megszervezésére szolgál. Ilyen például a Concurrent Pascal nyelv és az MPI.
- A **félig-statikus** modellben újabb folyamatokat a program bármely pontján létrehozhatunk, de ezek maximális száma – esetleg elhelyezése is – a program szövegéből egyértelműen kiderül. Ilyen nyelvek az OCCAM és a Pascal-Plus.
- Egy **dinamikus** modellben bármikor és bármennyi új folyamat létrejöhet. Ezek számára semmilyen korlátot nem lehet a program megkezdése előtt meghatározni. Ennek a modellnek az előnye a feladathoz, illetve a helyzethez való rugalmas alkalmazkodás és a hibatűrő képesség növelésének lehetősége. Ilyen lehetőségeket például az Ada nyelv, vagy a UNIX fork függvénye, vagy a PVM könyvtár biztosít.

A folyamat-modellek vizsgálatakor nem csak az indulás, hanem a megállás problémájával is lehet foglalkozni:

- folyamat lefutása után befejeződik, vagy várni fog, amíg az egész program megáll
- folyamat képes-e megállítani magát, vagy egy másik folyamatot
- kivételes helyzetet a folyamat lekezelhet-e, és/vagy erről egy másik folyamat értesül, amely kezelni tudja a helyzetet

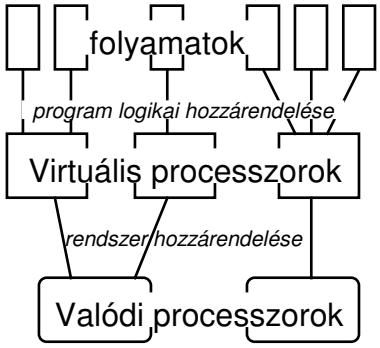
Egy program folyamatainak leképezése a meglévő erőforrásokra a nyelvek illetve eszközök egy másik fontos tulajdonsága.

A statikus és félig-statikus modellekben ez a feladat könnyebben megoldható, hiszen a program indítása előtt megbecsülhető, hogy egy-egy folyamata mennyi erőforrást fog felhasználni, és ennek megfelelően statikusan is eloszthatóa a feladat.

A dinamikus modellben szintén megadhatóak statikus elvek az erőforrások elosztására, azonban ezek a legritkább esetben vezetnek jó eredményre. Ezek a modellek dinamikus erőforrás elosztást, a feladat pillanatnyi helyzetéhez illeszkedő leképezést igényelnek.

Ennek legegyszerűbb módja az, hogy a program futása közben megvizsgálja a lehetőségeit, és a rendszernek ettől függően jelzi erőforrás igényét (pl. megmondja, hogy melyik processzoron kell a következő folyamatot elindítani).

Ez a folyamat egyszerűsíthető azzal, hogy a program csak azt definiálja, hogy milyen igényei vannak és a rendszer keres ehhez megfelelő lehetőségeket. Ennek egyik formája a folyamatok indításakor a PVM könyvtárban figyelhető meg.



6. Ábra: Virtuális processzorok

Magasabb absztrakciós szinten is leírhatóak egy algoritmus erőforrás igényei **virtuális processzorok** és **virtuális topológiák** használatával. Virtuális processzorokon szorosan összetartozó – közös részfeladaton, adatterületen dolgozó, sok kommunikációt igénylő – folyamatokat lehet elindítani, amellyel a rendszer számára információt adhatunk át, hogy ezeket lehetőleg ugyanazon, vagy minél közelebb lévő valódi processzorokon helyezze el. A virtuális topológiák a kommunikácó absztrakt leírására használhatóak. Segítségükkel megadható, hogy a program folyamatai egy logikai gyűrűben, vagy fában kommunikálnak. Ezzel az információval lehetőséget adnak a rendszernek, hogy a folyamatokat a topológia által definiált szomszédsági viszonyoknak megfelelően helyezze el. Ez az eszköz a programozó munkáját is megkönnyítheti az egyszerűbb jelölésmód miatt. Ilyen lehetőségek vannak a PARIX operációs rendszer és az MPI rutinkönyvtárában.

Futás közbeni dinamikus erőforrás elosztás egy érdekes eszköze lehetne még a folyamatok átvitele a processzorok között, de ezt a problémát még nem oldották meg hatékonyan.

A **párhuzamosság szintjét** egyes nyelvek, illetve eszközök különböző helyeken húzhatják meg:

- **utasítás**, ha lehetőség van utasítások szintjén lévő párhuzamosság kezelésére (pl. szemaforok a biztonság növelésére)
- **kifejezés** szintjén párhuzamos, ha lehetőség van párhuzamosan végrehajtódó kifejezések írására, de azok atomicitása biztosítható (pl. CC++ *par* blokkja)
- **folyamat** szintű ha lehetőség van LWP-k (Light Weight Process) vagy thread-ek definiálására, amelyek párhuzamosan futnak, de a program globális változóit megosztva használhatják. Ilyenek az Ada task-jai.
- **program**, ha a legkisebb egység, ami másokkal párhuzamosan futhat az csak egy program lehet. Ilyen például a UNIX rendszerek fork lehetősége.

Egy eszközben persze egyszerre több szint is szerepelhet, ami egy feladat optimális megoldását nagy mértékben segítheti (pl.: programok és thread-ek használatával virtuális processzorok nélkül is egy fizikai processzorra csoportosíthatóak az összetartozó feladatok).

Kommunikációs eszközök

A feladatok megoldása során – kivételes, triviálisan párhuzamosítható esetektől eltekintve – a folyamatoknak valamilyen módon kommunikálni kell egymással. A legtöbb kommunikációs eszköz az **osztott memória** vagy az **üzenetküldés** osztályába sorolható be. Vannak persze más eszközök is, – például a Linda nyelvben az osztott memória és az üzenetküldés kereszteződéséből létrehozott *tuple-space* fogalma – de ezek nem terjedtek el olyan széles körben.

Osztott memória használata programok fejlesztésénél nagyon nagy előny, mert a szekvenciális algoritmusok általában könnyedén párhuzamosíthatóak íly módon. Az osztott memóriának persze vannak nagy hátrányai is:

- más eszközöket is fel kell használni a folyamatok közötti szinkronizáció megoldására, az osztott memória konzisztenciájának megőrzése érdekében
- osztott memória használata általában csak kevés processzor esetén kifizetődő, nagy rendszereknél a fizikailag osztott memória nagy teljesítményveszteséget jelent szinkronizálási gondok miatt (erre az SPP rendszerek egy megoldást jelenthetnek)
- a folyamatok közötti interakció nehezen követhető, ami a programok vizsgálatánál, nyomkövetésénél kényelmetlen lehet

Az **üzenetküldés** folyamán egy küldő folyamat egy adat csomagot megcímez és valamilyen úton eljuttat egy vevő folyamathoz.

Ez a módszer nehezebben használható szekvenciális programok párhuzamosításakor, viszont minden más esetben nagyon jól illeszkedhet az kifejezetten párhuzamos algoritmusokhoz. Ez a kommunikációs forma osztott memóriát használó gépeken is nagyon könnyen megvalósítható, ezért az így elkészített algoritmusok minden párhuzamos architektúrán implementálhatóak. Az üzenetküldés hátránya – a szűk kapcsolat a folyamatok között – egy program nyomkövetése, vagy helyességének bizonyítása során nagy előnyt jelent.

Az üzenetküldés több szempontból is vizsgálható:

- A folyamatok közötti **együttműködés** alapján lehet
  - **szinkron** (amikor az üzenetküldés pontjánál szinkronizálódnak a folyamatok, azaz megvárják egymást)
  - **aszinkron** (valamilyen közbenső közeg tárolja az elküldött üzenetet, így a kommunikáló folyamatoknak nem kell találkozniuk)
- Az üzeneteket a folyamatok a tényleges elküldés, vagy az üzenet tényleges vétele előtt **pufferben** tárolhatják. A puffer helye rendszerenként változó lehet: PVM-ben a vevő memóriájában, MPI-ban a küldő memóriájában gyűlnek a pufferelt üzenetek. Utóbbi esetben a vevő folyamat memóriája lesz megmentve a nem kívánt – esetenként akár igen nagy – üzenetek tárolásától, és csak az üzenet fejléce lesz elküldve a vevő folyamathoz. A valódi üzenet eljuttatása a címzetthez még egy üzenetváltást igényel ebben az esetben, amiben a fogadó elkéri az adott üzenetet a küldő folyamat pufferéből. A vevő oldali pufferelés ezt az üzenetváltást elkerülheti, viszont ez esetleg nem várt üzenetek tárolásához vezethet.
- Az üzenet elküldése egyes architektúrákon hosszú folyamat lehet. Annak érdekében, hogy a folyamat eközben hasznos munkát végezhessen különböző fogásokat lehet alkalmazni:
  - **nem blokkoló** vevő, illetve küldő eljárások a szokásos **blokkoló** változatok helyett. Ez az eljárás azonnali visszatérését jelenti, egy háttérfolyamat beindításával, amely csak az adott eljárás elvégzéséért felelős. (A háttérfolyamat egy új végrehajtási szál

(thread), vagy a szekvenciális programba ágyazott periodikusan meghívott eljárás is lehet.)
    - Egyes nyelvek, illetve eszközök ezekhez a nem blokkoló eljárásokhoz **lejárati időt** is rendelhetnek, amely elteltével az eljárás sikertelenként értékelődik, és törlődik (ilyen az Ada *select* ágába rakott *delay* utasítás, vagy a PVM *pvm_trecv* függvénye).
    - Az üzenetküldés meggyorsítása érdekében virtuális, vagy valódi **csatornák** építhetőek ki a folyamatok között. Ez az eszköz a kapcsolatfelvétel és a címzés protokollját lényegében csak a csatorna kiépítésekor igényli, tehát nagyon sok üzenet küldésekor megtérülhet időben a többlet erőforrásigény.
- Az üzenetek **sorrendje** szempontjából lehet
    - sorrendet nem megőrző (ilyen az UDP protokoll)
    - sorrendet két pont között megőrző (lényegében minden rendszer)
    - sorrendet több pont közötti kommunikációban is megőrző (amikor például egy egységes idő szerint indexelik az üzeneteket, és ez adja a beérkező üzenetek sorrendjét)
- a **résztvevők** aktivitása alapján:
    - **szimmetrikus**, ahol a küldő definiálja, hogy kinek küldi az üzenetet, és a vevő is definiálhatja, hogy kitől hajlandó azt elfogadni (ilyen a PVM és az MPI eljárása)
    - **aszimmetrikus**, ahol a küldő megnevezheti a címzettet, de a vevő legfeljebb csak az üzenet fajtáját specifikálhatja, annak küldőjére semmilyen megszorítást nem tehet (ilyen az Ada *randevú* fogalma, illetve a távoli eljáráshívások)
    - küldő által kezdeményezett, vagy **generált** üzenetküldés, ahol az üzenet küldője rákényszerítheti a vevőt annak átvételére (ilyen az osztott memória használata is)
    - **szétsugárzott** üzenetküldés, ahol egy folyamattól egy specifikált csoport minden tagjához eljut egy üzenet, de a vevő folyamatnak lehetősége van, hogy kiválassza kitől fogad el üzenetet.

A kommunikációs eszközök közül – a párhuzamosság támogatásához hasonlóan – egy rendszerben több is előfordulhat, és ezeket kombinálva is lehet használni.

Szinkronizációs eszközök

Folyamatok összehangolása, szinkronizációja nehezen választható el a kommunikációtól. Egy feladat megoldása során persze akadhat olyan eset amikor nem kell információt közölniük egymásnak a feladatot megoldó folyamatoknak, „csak” szinkronizálni kell őket. Ilyen lehet valamilyen kritikus szakasz kezelése – például egy I/O művelet valamilyen háttértárolón –, amikor csak azt kell biztosítani, hogy az adott helyen egyszerre csak egy folyamat van.

A szinkronizálással kapcsolatos problémákra sok megoldás született, az egészen egyszerű – gépi kódú utasítás szintjén lévő megoldástól – a magas absztrakciós szinten lévő megoldásokig.

A legegyszerűbb eszköz a **szemafor**, melyet Dijkstra vezetett be a számítástudományba 1965-ben.

A szemafor egy atomi módon – egyszerre csak egy folyamat számára – hozzáférhető tároló egység. Ha egy folyamat nem tud hozzáférni a szemaforhoz, akkor általában egy várakozási sorba kerül, amelyből a szemafor állapotának megváltozása hozhatja ki.

Szemaforokat főleg közösen használt erőforrásoknál – lemez, memória, stb. – a kölcsönös kizárás megvalósítására használnak.

Legegyszerűbb változata a bináris szemafor, melyben csak 1, illetve 0 értéket lehet eltárolni. A tárolás P (probern) művelete lényegében az erőforrás lefoglalását, a felszabadítás V (verhogen) művelete, pedig annak felszabadítását jelenti ebben az esetben. Ezen kívül vannak több értékű szemaforok, sőt szemafor családok is, melyeket egy művelettel lehet kezelni.

Nagy előnye, hogy működése nagyon egyszerű, és nagyon egyszerűen implementálható, de inkább csak osztott memóriát használó rendszerekben lehet velük találkozni. A szemaforok hátránya az egyszerűségükben rejlik, mivel így nem egyszerű velük bonyolult kizárásos problémák – például az író-olvasó probléma – megoldása.

Ennél magasabb absztrakciós szinten áll a **monitor**, melyet Brinch-Hansen, illetve Hoare használt először a hetvenes évek közepén.

Ez az eszköz adatok és a rajtuk végezhető műveletek összefogására szolgál – így az objektum orientált törekvések elődjének is tekinthető –, de a megoldható feladatok szempontjából nem bonyolultabb. A monitorban lévő adatokat csak a monitor műveletein keresztül lehet elérni, melyekre az kölcsönös kizárást biztosít, azaz közülük egyszerre csak egy lehet aktív. Foglalt monitort meghívó folyamatok – a szemaforokhoz hasonlóan – egy várakozási sorba kerülnek.

A monitor egyszerű szerkezete miatt alapvetően osztott adattípusok, illetve egyszerű erőforrás szinkronizációs módszerek leírására használható.

Monitorokat például a konkurens Pascalban implementáltak.

A következő eszközökben a szinkronizáció és a kommunikáció eléggé összekeveredik. Ez a szétválaszthatatlanság bonyolultabb esetekben elkerülhetetlen, hiszen a szinkronizáció maga is valamilyen módon egy információ átadását jelenti, másrészt bármilyen kommunikáció használható szinkronizálási problémák megoldására.

A szinkronizálás és a kommunikáció legegyszerűbb esete a **szinkron üzenetküldés** – ami persze két aszinkron üzenetküldéssel is szimulálható, melyben két folyamat egy pontnál megvárja egymást, és az egyik valamilyen információt juttat el a másikhoz. Itt a szinkronizáció a kommunikációban implicit benne van, nem választható el attól.

A szinkron üzenetküldés módszerével egyszerű problémák oldhatóak meg. Egy osztott adattípus megvalósításához egy ilyen egyszerű hívás nem elegendő, ilyen kommunikációkból egy bonyolultabb kapcsolatot kell felépíteni.

Ennek egyik formája a **távoli eljáráshívás**, melyben lényegében két szinkron kommunikáció történik. Távoli eljáráshívás esetén – a monitorhoz hasonlóan – a hívó folyamat a fogadónak adja a vezérlés, és mindaddig várakozni fog, amíg az végre nem hajtja a műveletet, és vissza nem adja a vezérlést.

A távoli eljáráshívásnál a monitorhoz viszonyított különbséget az osztott adattípus hiánya jelenti. Ennél az eszköznél a fogadó oldalon nincs beburkolt adatrész, a fogadó oldal működése lényegében tetszőleges lehet.

A másik nagy különbséget a kölcsönös kizárás implementációfüggő megvalósítása jelenti. Ez azt takarja, hogy ennek az eszköznek az adott implementációjától függ, hogy az eljárások valamilyen módon kölcsönösen kizárhatják-e egymást.

Az szinkron üzenetküldés és a távoli eljáráshívás kombinációjának tekinthető az Ada **randevú** fogalma. Az Ada-ban két folyamat szinkronizálásának, illetve kommunikációjának eszköze (az osztott változóktól eltekintve) a randevú.

Az Ada randevújában is egy kliens-kiszolgáló kapcsolat alakul ki – a monitorhoz hasonlóan –, de itt a kiszolgáló egység egy önálló folyamat, nem csak egy adathalmaz műveleteit implementáló kód. A randevúk az Ada-ban is kölcsönösen kizárják egymást, és a nem szabad randevú pontokat hívó folyamatok itt is várakozási sorba kerülnek.

A nagy különbséget – a monitorhoz képest – a randevúpontokhoz kapcsolódó elkülönített várakozási sorok, és a kiszolgálásra adható bonyolultabb feltételek jelentik. A fogadó folyamat minden randevúponthoz a belső állapotából kiszámolható plusz feltételeket is köthet, amivel sokkal finomobban tudja szabályozni az adatrészeihez való hozzáférést.

Mivel a kiszolgáló rész is egy folyamat, ezért sokkal bonyolultabb műveletek írhatóak le ezzel az eszközzel, mint egy egyszerű osztott változóval.

Az Ada randevú eszközének a hátránya, hogy csak kliens-kiszolgáló kapcsolatok kezelhetőek vele, ami egymás mellé rendelt folyamatok szinkronizációjára nem igazán jó módszer.

Az osztott erőforrások kezelésének következő (számítástudomány történetében megelőző, de az Ada nyelv történetében következő) lépését az osztott adattípusok egy általánosabb formája jelenti.

Egy osztott adattípuson végezhető művelethalmaz végrehajtási sorrendjét azokra vonatkozó feltételekkel, útkifejezésekkel, vagy eseményorientált leírásokkal lehet megadni. Ezek a leírási módszerek általában nagyon bonyolult hozzáférési módok (prioritásos többszörös hozzáférésű író-olvasó probléma) leírására is alkalmasak, implementálásuk viszont nem minden esetben oldható meg egyszerűen.

Az Ada95 nyelv **osztott adattípusa** az ilyen útkifejezésekkel leírható szinkronizálási problémák egy részét tudja kezelni, ezeket viszont egy új folyamat létrehozásának költsége nélkül. Az osztott adattípushoz lehet definiálni műveleteket, amelyek a monitoroknak megfelelően működnek – azaz kölcsönös kizárás van köztük –, lehet megadni olyanokat is amelyek akár egyszerre is végrehajtódhatnak, valamint megadhatóak az osztott adattípus belső állapotából kiszámolható feltétellel őrzött belépési pontok is.

Általánosabb szinkronizálási problémák megoldására osztott adattípusok esetében még nem születtek széles körben elterjedt megoldások.

Az osztott adattípusok kliens/szerver modelljén kívül még előfordulhatnak egymás mellé rendelt folyamatok között is szinkronizálási problémák.

Egyik ilyen probléma, ha több folyamatot egyetlen pontnál szinkronizálni kell. Ekkor az összes folyamatnak meg kell várni az összes többit, anélkül, hogy egy külön kiszolgáló folyamat összehangolná tevékenységüket.

Az ilyen feladatok megoldására vannak példák a PVM és az MPI rendszerekben.

2.1.3. Jellemzők

Az egyes programozási eszközöket a fenti szempontokon kívül a hagyományos programozási nyelvekből merített elvek alapján is lehet értékelni:

- **helyesség:** Egy program helyességének belátása hagyományos nyelveknél könnyebben megoldható. Párhuzamos eszközöknél a megoldások bonyolultsága könnyen túllépheti a rendelkezésre álló bizonyítási módszerek hatókörét, ezért az ilyen programok helyes voltának belátása nagyobb nehézségekbe ütközik. A helyesség belátásának az alapvető feltétele az egyes eszközöknél, hogy pontosan és jól definiálják annak működését.

- **megbízhatóság:** Nem várt körülmények között is elfogadható eredményeket ad-e a rendszer (pl. egy processzor kiesése esetén nem áll minden le), valamint biztosít-e eszközöket az ilyen helyzetek kezelésére (esetleg a programon belül).
- **hatékonyság:** Párhuzamos eszközök egyik fő felhasználási területe a hatékonyság javítása, így érdemes az egyes eszközöket egymással összehasonlítani, hogy egy konkrét feladatot mennyire hatékonyan tudnak megoldani. Hatékonysági kérdéseknél érdekes megvizsgálni, hogy egy automatikusan generált párhuzamos kód milyen gyors egy igazi párhuzamos algoritmus megvalósításához képest. Ezt a gondolatot továbbfejlesztve meg lehet nézni, hogy különböző absztrakciós szinteken álló programokból milyen hatékony kód generálható.
- **hordozhatóság:** Párhuzamos programok fejlesztése közben nem áll mindíg rendelkezésre a futtató környezet, ezért a programozó egyszerűbb helyzetben kénytelen fejleszteni a programot. Ilyen esetekben lehet nagyon fontos kérdés, hogy az így elkészített kód mennyire hordozható a különböző gépek között, az adott eszköz mennyire nyújt egységes felületet más-más architektúrán. Rutinkönyvtáraknál a nyelvek közötti átjárhatóság is vizsgálati szempont lehet.
- **biztonság:** Nyílt rendszerekben lényeges szempont lehet, hogy egy eszköz milyen lehetőségeket nyújt biztonsági problémák kezelésére (pl. egy kliens folyamat jogainak definiálása).
- **modularitás:** Modul rendszerű programfejlesztés támogatása. Az adott eszköz milyen lehetőségeket nagy programok fejlesztésére, a környezetétől függetleníthető rutinkönyvtárak kialakítására.
- **tesztelés, optimalizálás támogatása:** Megmérethetnek a nyomkövetés beépített eszközei, illetve a külső segédprogramok is. A optimalizálás szempontjából fontos, hogy az eszköz ad-e lehetőséget profile készítésére (pl.: PMPI), valamint ezt értékelő program van-e hozzá.
- **felület a fejlesztésre:** Egy eszköz használatának a hatékonyságát növelheti, ha valamilyen kényelmes felületet biztosít a használatára, vagy a fenti problémák megoldásának valamelyikére (pl. grafikus felületű futtató rendszer).

Az egyes eszközök értékelése a fentiekben felsorolt modellek, osztályok és jellemzők alapján a fejezet hátralévő részének anyaga az *MPI, PVM, CC++, Ada95* és *FORTRAN 90* nyelvek vizsgálata.

2.1.4. Példák

Mandelbrot halmaz számítása

Nagyon jól párhuzamosítható, lényegében teljesen független lépésekből álló feladat a Mandelbrot halmaz számítása.

Szekvenciális megoldására a következő C nyelvi program adhat példát:

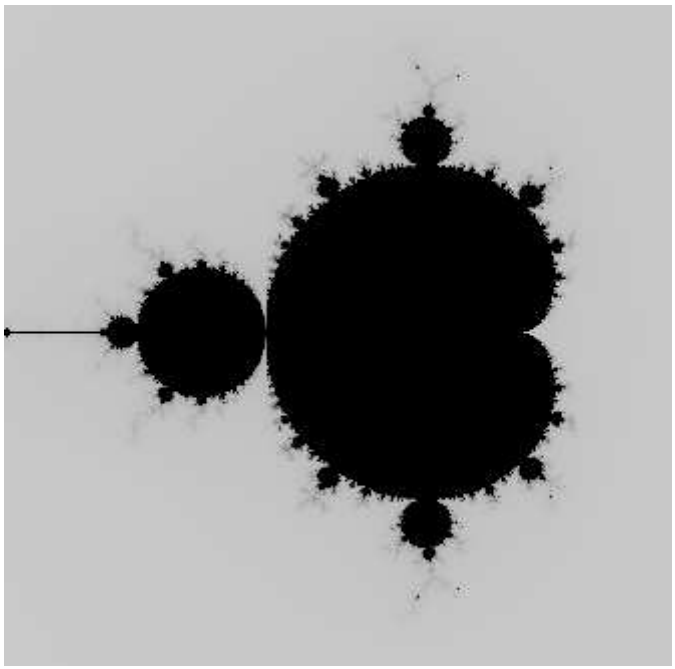
```
/*
 * Mandelbrot halmaz szamitasa (-2,-2)-(2,2)
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
      int         N = 512;          /* kepmeret       */
```

```
        char        *pix;       /* a kep                */
        int         ix, iy;           /* pixel koord.   */
        double            x, y; /* re, im koord.  */
        register double ar, ai;
        register double a1, a2;
        register int      ite;        /* iteracioszam   */
        double            x1,y1,x2,y2;      /* sarkok         */
        FILE        *f;               /* az eredmeny          */

        x1 = -2; y1 = -2; x2 = 2; y2 = 2;
        pix = (char*)malloc(N * N);

        /*
         * A fo szamitasi ciklus
         */
        x2 -= x1;
        y2 -= y1;
        for (iy = N; iy-- > 0; )
        {
              y = (iy * y2) / N + y1;
              for (ix = N; ix-- > 0; )
              {
                    x = (ix * x2) / N + x1;
                    ar = x;
                    ai = y;
                    for (ite = 0; ite < 255; ite++)
                    {
                          a1 = (ar * ar);
                          a2 = (ai * ai);
                          if (a1 + a2 > 4.0)
                                break;
                          ai = 2 * ai * ar + y;
                          ar = a1 - a2 + x;
                    }
                    pix[iy * N + ix] = ~ite;
              }
        }

        /*
         * az eredmeny kiirasa
         */
        f = fopen("mandel.pgm", "wb");
        fprintf(f, "P5\n# Mandelbrot set\n");
        fprintf(f, "%3d %3d\n%3d\n", N, N, 255);
        fwrite(pix, 1, N * N, f);
        fclose(f);

        return(0);
}
```

A feladat megvalósításainál az eszközök matematikai problémák kezelésére nyújtott lehetőségeit, illetve a kivételes esetek kezelésére nyújtott lehetőségeket lehet megvizsgálni.

A programok eredménye 8 bites PGM formátumú kép file lesz, melyet bármilyen képfeldolgozó/nézegető eszközzel meg lehet nézni.

A végeredmény a következő képre kell hogy hasonlítson:



7. Ábra: Mandelbrot halmaz

2.2. MPI

MPI (Message Passing Interface) egy rutinkönyvtár szabvány, amely a programok közötti üzenetek útján való kommunikáció biztosítását írja le.

2.2.1. Bevezető az eszközről

Történet

Sok párhuzamos számítógépen hasonló alapelveken nyugvó rutinkönyvtárak készültek a processzorok közötti kommunikációk megoldására. Noha ezek eléggé különböztek egymástól, az alapelvük lényegében azonos volt. Az utóbbi időben ezek mellett megjelent néhány szélesebb körben is elismert rendszer, amelyek bebizonyították, hogy hordozható kódok is készíthetőek.

Ilyen alapokból indulva alakult meg 1992-ben az MPI Forum, amely feladata az MPI szabvány fejlesztése. Ennek a testületnek sok hardware gyártó, software készítő cég és szervezet lett tagja. A szabványra a következő software-k készítői voltak nagy hatással: IBM T. J. Watson Research Center, NX/2 (Intel), Express, Vertex (nCUBE), p4, PARAMACS, Zipcode, Chimp, PVM, Chameleon és PICL.

A felület kifejlesztésének a célja az volt, hogy egy széles körben használható, hordozható szabványt készítsenek üzenetek küldésére. Jelenleg több publikus és több hardware gyártók által támogatott változata is van a szabványban specifikált rutinkönyvtárnak. Az MPI szabványt használó programok forrás szinten hordozhatóak a különböző változatok között.

Áttekintés

Ebben a dokumentumban a szabvány első változatában specifikált lehetőségekről lesz szó. Ezek már minden MPI rutinkönyvtárban benne vannak. A felületet persze az MPI Forum

tovább bővíti. 1996 végére várható a szabvány következő változata, amely sokkal bővebb lesz.

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
- Teszteléshez segítség

Az MPI nem definiál eljárásokat taszkok 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 taszknak 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
- Taszkok 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.

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* implementáció íródott teljesen az MPI szabvány megvalósítására, a *CHIMP*, *LAM* és *UNIFY* csomagok már meglévő rutinkönyvtárakra épülnek rá, ezért nem annyira hatékonyak.

2.2.2. Felhasználói felület

A felületben lévő rutinokra a következő elnevezések használatosak:

- **nem blokkoló** egy eljárás, ha a még az általa elindított folyamat befejeződése előtt visszatérhet. Az ilyen eljárások általában valamilyen átlátszatlan típussal leírt belső adatszerkezetben tárolják a be nem fejezett folyamathoz tartozó információkat.
- **blokkoló** egy eljárás, ha megvárja azt amíg az általa elindított folyamat befejeződik.
- **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 taszkokkal való kommunikációt is igényel.
- **kollektív** típusú egy rutin, ha egy csoportban minden taszknak 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 alakja:

```
int MPI_Xxxxxxxx( ... )
```

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

```
MPI_XXXXXXXX( ..., IERROR)
```

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

Kezdet és vég

Ezeknek az eljárásoknak a könyvtárt használó programokban szükségszerűen benne kell lenniük.

```
int MPI_Init(int *argc, char ***argv)
```

MPI könyvtár inicializálása. A program indulása után az MPI függvények használata előtt kell meghívni. A parancssori argumentumokat megváltoztathatja, ezért kell azokat cím szerint átadni.

```
int MPI_Finalize(void)
```

A program normális befejezése.

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

A program és a *comm* által definiált csoportban lévő programok abnormális leállítása *errorcode* hibakóddal.

A *comm* paraméter egy taszk csoportot illetve 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.

Az *MPI_Comm* típus egy kommunikátort specifikál, amely taszkok egy csoportját – **MPI_Group** típussal leírható – és egy kommunikációs közeget foglal magában.

Minden taszk legalább egy csoportban benne van (*MPI_COMM_WORLD*), amiben egy sorszámmal lehet őt azonosítani. Egy taszk más-más csoportokban más-más sorszámot is kaphat.

A kommunikációs közegek az egymástól független üzenetváltások elválasztására szolgálnak. Segítségükkel a normális üzenetváltásoktól teljesen független taszkok közötti kommunikációk is kialakíthatók új könyvtárak vagy modulok számára.

Ezek alapján a minimális *MPI* program:

```
#include ″mpi.h″

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

Pont-pont közötti 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.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
```

A *buf* című területről *count* darab *datatype* típusú adatot elküld *dest* sorszámú – a *comm* által definiált csoportban – taszknak *tag* fejléccel a *comm* kommunikátorban. Ez egy blokkoló rutin.

Az *MPI_Datatype* egy üzenetbeli adattípust határoz meg. Az *MPI_Init* végrehajtása után a felhasználó rendelkezésére állnak a következő típusok: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE és MPI_PACKED. Új típusok is kialakíthatók struktúrák, illetve tömbrészletek leírására.

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

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status *status)
```

Blokkoló üzenet-vétel.

Az üzenet forrását és fejlécét a *source* és *tag* paraméterek definiálják, de itt az MPI_ANY_SOURCE, illetve az MPI_ANY_TAG használatával bárkitől bármilyen üzenetet is lehet fogadni. Ilyen helyzetben lehet hasznos a *status* paraméter.

A *status* struktúrát a programozónak kell lefoglalnia, ezt az MPI nem teszi meg. Az *MPI_Status* struktúrának van MPI_SOURCE, MPI_TAG és MPI_ERROR mezője, amelyekkel információkat nyerhetünk egy ismeretlen üzenetről is. Ebből a struktúrából az *MPI_Get_count* rutin használatával az üzenet eredeti méretét is megkaphatjuk.

MPI-ban a vett adatok egyszerű típusának meg kell egyeznie a küldött adatok egyszerű típusával. 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. MPI-ban a kommunikátorból megállapítható, hogy a taszkok azonos architektúrákon futnak-e, ha nem akkor a reprezentációk között automatikusan konvertálódnak az adatok.

A pont-pont közötti kommunikációknak még több fajtája elképzelhető (* jelöli a fenti módszerre jellemzőt):

- **blokkoló* vagy nem** Az üzenetküldés blokkolása addig tart amíg az üzenet nem továbbítódik a címzett felé (ez nem jelenti azt, hogy a másik taszk vette az üzenetet). Vételnél az üzenet tényleges megérkezéséig tart a blokkolt állapot.
- **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() bufferelt
MPI_Ssend() szinkron
MPI_Rsend() ready
```

```
int MPI_Buffer_attach(void *buffer, int size)
int MPI_Buffer_detach(void **buffer, int *size)
```

Az MPI könyvtár rendelkezésére lehet bocsátani, illetve vissza lehet tőle venni egy memóriaterületet. Ezt a területet használja a könyvtár a pufferelt üzenetküldések végrehajtásakor. Egy programban egyszerre csak egy aktív puffer lehet.

A fenti rutinoknak vannak nem blokkoló változataik is. Ez 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 nem blokkoló rutinok mind egy **MPI_Request** típusú kérelmet használnak a párhuzamosan futó művelet adatainak tárolására.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request)
```

Nem blokkoló üzenetküldés. Az üzenetküldés sikerességét a *request* vizsgálatával tudhatjuk meg. A *request* struktúrát a programozónak kell lefoglalnia.

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:

```
int MPI_Ibsend()
int MPI_Issend()
int MPI_Irsend()
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Request *request)
```

Nem blokkoló üzenet-vétel.

Egy *request* vizsgálatára a következő függvények szolgálhatnak:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

A *reuest* által azonosított művelet befejeződéséig vár. A művelet eredményét a *status* paraméterből tudhatjuk meg.

A befejeződés után a *request* értékét MPI_REQUEST_NULL-ra állítja.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Megvizsgálja a *request* által azonosított műveletet. Ha már befejeződött, akkor a *flag* értékét igazra állítja, és a *MPI_Wait*-hez hasonlóan fejezi be futását. Ha a művelet még fut, akkor a *flag* értéke hamis lesz.

```
int MPI_Request_free(MPI_Request *request)
```

A *request* felszabadítása, a futás vizsgálata nélkül. A hozzárendelt művelet ettől függetlenül még befejeződhet, bár a sikerességet már nem lesz lehetőség megvizsgálni.

Egy tömbben több kérelem is összegyűjthető, amiket együttesen is lehet vizsgálni:

```
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
MPI_Status *status)
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index, int
*flag, MPI_Status *status)
```

Az *array_of_requests* tömbben lévő bármely – de csak egy – kérelem befejeződésének vizsgálata. Az *index* paraméter mutatja meg, hogy a tömb melyik kérelme futott le.

```
int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status
*array_of_statuses)
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
MPI_Status *array_of_statuses)
```

Az összes kérelem egyidejű vizsgálata.

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int
*outcount, int *array_of_indicies, MPI_Status *array_of_statuses)
int MPI_Testsome(int incount, MPI_Request *array_of_requests, , int
*outcount, int *array_of_indicies, MPI_Status *array_of_statuses)
```

Egy vagy több kérelem befejeződésének figyelése. Ha több kérelem is sikeresen végetért, akkor ezek sorszámai az *array_of_indicies* tömbbe kerülnek. A tömb méretét az *outcount* paraméter mondja meg.

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status
*status)
```

Üzenet vételének tesztje. A rutinok az *MPI_Recv* és *MPI_IRecv+MPI_Test* rutinoknak megfelelően viselkednek, csak nem veszik az üzenetet.

```
int MPI_Cancel(MPI_Request *request)
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

Egy futó művelet megszakítása. Az *MPI_Cancel* után egy *MPI_Wait* vagy *MPI_Test* művelettel lehet a kérelmet felszabadítani.

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

```
int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
```

Állandósított kommunikáció. Ha nagyon sokszor kell azonos paraméterekkel leírt kommunikációt végezni, akkor ezeket egy kérelembe el lehet tárolni. Az egyszerűsített

híváson kívül ez a módszer lehetőséget ad az implementációnak, hogy egy csatorna megnyitásával gyorsabb üzenetküldést biztosítson.

A küldésnek megvannak a szokásos változatai:

```
int MPI_Bsend_init()
int MPI_Ssend_init()
int MPI_Rsend_init()

int MPI_Start(MPI_Request *request)
int MPI_Startall(int count, MPI_Request *array_of_requests)
```

A fenti függvényekkel inicializált kérelemmel definiált üzenetküldés indítása.

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

Blokkoló üzenet küldés és vétel. A rutin hatása ekvivalens nem blokkoló üzenet küldés és vétel elindításával, majd a műveletek befejeződésének megvárásával.

Gyűrűbe kapcsolt folyamatok közötti eltolás — mindenki az előtte lévőnek ad át adatot — művelet végrehajtásakor lehet előnyös, mert így a holtpont elkerülhető.

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

Az előző művelet megfelelője, azzal a különbséggel, hogy a küldő és fogadó adatterületek egybeesnek.

Üzenetekben lévő típusok megadása

Az üzenetekbe csak az MPI által ismert típusokat 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 típust egy **MPI_Datatype** átlátszatlan típusban lehet létrehozni. A létrehozás után a típus engedélyeztetni kell, 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 típus lényegében egy egyszerű típusokból álló típus térkép. Leírja, hogy egymás után milyen egyszerű típusok vannak, és az egyes elemek milyen eltolással vannak eltárolva. A típusok belső tárolására persze tömörebb formát használnak, de ez a szemléletes kép jól használható az MPI típus-kompatibilitás megértéséhez. Egy adott típussal küldött MPI üzenet csak olyan MPI típussal vehető, amiben az egyszerű típusok sorrendje és száma megegyezik a küldött MPI típuséval.

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

A legegyszerűbb konstrukciós rutin az *MPI_Type_contiguous*. Ezzel *count* darab *odltype* típusú, a memóriában folytonosan elhelyezkedő típus kerül be az új típusba. Lényegében egy egy dimenziós tömböt írhatunk le.

```
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Az eredmény *count* darab, *blocklength* hosszú szakaszokból álló, *oldtype* alaptípusú új típust ad eredményül. A szakaszok kezdetei közötti eltolást a *stride* paraméterrel lehet

definiálni. Az *MPI_Type_vector-nál* ez az érték elemek típusának méretével, az *MPI_Type_hvector* rutinnál pedig byte-okban mérendő.

```
int MPI_Type_indexed(int count, int *array_of_blocklengths, int
*array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_hindexed(int count, int *array_of_blocklengths, MPI_Aint
*array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

A rutinok eredménye *count* darab szakaszból álló új típus lesz. Az egyes szakaszok hosszát az *array_of_blocklengths* tömbben, a szakaszok kezdetének eltolását az adatterület kezdetéhez mérten az *array_of_displacements* tömbben kell megadni. Az eltolás mértéke az első eljárásban az alaptípus méretével szorzódik.

```
int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint
*array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype
*newtype)
```

Strukturált adattípus leírására szolgál ez a rutin. A struktúra *count* szakaszból áll, amelyek hosszát, eltolását és alaptípusát az *array_of_blocklengths, array_of_displacements,* illetve az *array_of_types* tömbökben kell megadni. Az eltolást byte-okban mérve kell definiálni.

A típusok létrehozásában az alábbi rutinok segíthetnek:

```
int MPI_Address(void* location, MPI_Aint *address)
```

Egy adatterület címét adja meg. Ez a C-beli & operátorral ekvivalens hatású.

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

Adattípus mérete. Eltolások számításánál lehet hasznos.

```
int MPI_Type_size(MPI_Datatype datatype, MPI_Aint *size)
```

A típusban lévő elemek összmérete. Ekkora lesz az elküldött üzenet mérete.

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

Egyes számítógépeken bizonyos típusok — például a karakter — nem helyezhetők el akárhova a memóriában, csak szó, vagy dupla-szó határra. Ilyen esetekben az adattípus elejére, illetve végére lukakat kell tenni. Ezek lekérdezésére szolgálnak ezek a rutinok.

Egy típus létrehozásakor az MPI_LB, illetve MPI_UB egyszerű típusok használhatóak a lukak létrehozására.

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Az új típus engedélyeztetése. Csak ez után használható a kommunikációban.

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Egy adattípus felszabadítása. Ha egy kommunikáció éppen használja ezt, akkor az még rendben lefut, és csak utána kerül felszabadításra.

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
```

A *status* által definiált vett üzenetben lévő *datatype* típusú adatok számát adja vissza.

Az MPI lehetőséget biztosít egy üzenet pufferbe való többlépcsős pakolásra is. Ez a viselkedés leginkább a PVM üzenet-felépítéséhez hasonlít, de itt a programozónak magának kell kezelnie az üzenetpuffert.

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
int outsize, int *position, MPI_Comm comm)
```

Az *outbuf* által definiált helyen lévő üzenetpufferbe *incount* darab *datatype* típusú adatot rak be a *inbuf* című adatterületről. A lefoglalt üzenetpuffer méretét a *outsize* paraméterben kell megadni (nehogy többet pakoljon bele). A *position* paraméterben megadott helytől kezd el

pakolni a pufferbe, és ebben a paraméterben adja vissza a rutin azt is, hogy meddig jutott el benne.

Ezt a rutint többször meghívva tetszőleges adatokat berakhatunk egy üzenetbe. A kész üzenetpuffert a szokásos *MPI_Send* rutinnal lehet elküldeni, csak **MPI_PACKED** típusként kell leküldeni.

A módszer nagy hátránya, hogy így az MPI semmilyen típusellenőrzést nem tud végezni a küldött üzenetben.

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

Egy *MPI_PACKED* típussal vett üzenetből lehet kipakolni az adatokat.

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.

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
```

A puffer méretének előzetes becslésére szolgáló eljárás. Ha *incount* darab *datatype* típust szeretnénk egy üzenetpufferbe berakni, akkor annak mérete legfeljebb *size* byte-tal fog megnőni.

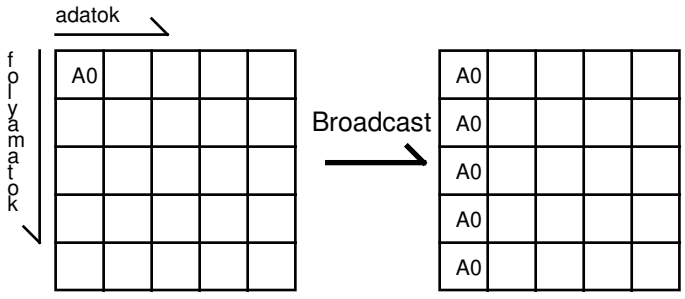
Kollektív kommunikáció

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.

Szerintem az MPI rendszer legfőbb előnye ezen rutinok használatában van.

```
int MPI_Barrier(MPI_Comm comm )
```

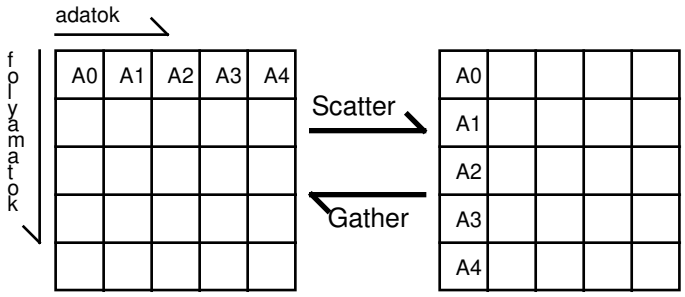
Egy találkozó a folyamatok között. A rutin addig vár, ameddig az összes – a kommunikátorban lévő – folyamat meg nem hívja ugyanezt az eljárást.



8. Ábra: Adatok lemásolása

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
```

A *root* sorszámú folyamat adatairól minden folyamatban készül egy másolat.



9. Ábra: Adatok szétszórása és összegyűjtése

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Minden folyamat a *sendbuf, sendcount* és *sendtype* adatokat küldi el a *root* sorszámú folyamatnak, aki azokat a *recvbuf, recvcount* és *recvtype* által meghatározott helyre gyűjti össze. A vett tömbben a folyamatok sorszámának megfelelően helyezkednek majd el az adatok.

A küldött és a vett adatok mennyiségének és típusának meg kell egyezni. Ez a megkötés a további rutinokra is igaz lesz.

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

Az előzővel megegyező működésű rutin, csak a *displs* tömbben meg lehet adni, hogy az egyes folyamatoktól érkező adatok milyen eltolással tárolódjanak el.

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

A *root* folyamatban a *sendbuf, sendcount* és *sendtype* paraméterekkel leírt adatok kerülnek szétosztásra a *comm* által definiált csoportban. A szétosztásra kerülő adatok egy-egy darabját a folyamatok a *recvbuf, recvcount* és *recvtype* paraméterekkel leírt helyre kerülnek (a *root* is kap egy darabot).

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

Az *MPI_Gatherv-hez* hasonlóan itt az egyes folyamatoknak szétküldendő adatrészek kezdete adható meg a *displs* tömbben.

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
MPI_Comm comm)
```

Ez a két rutin az *MPI_Gather,* illetve *MPI_Gatherv* eljárásokra hasonlít, csak az összegyűjtött adatok minden folyamatban el lesznek tárolva. (Hatása olyan mint egy *MPI_Gather* majd *MPI_Bcast* rutinok meghívása.

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls,
MPI_Datatype recvtype, MPI_Comm comm)
```

Eme rutinok hatása a *MPI_Allgather-re* hasonlít, csak minden folyamat minden másiknak más-más adatokat küld. Hatása olyan, mintha *MPI_Scatter* és *MPI_Gather* rutinokat egyszerre hajtanánk végre.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Asszociatív és kommutatív művelet végrehajtása a folyamatokban tárolt lokális adatokon. Az eredmény a *root* folyamatba érkezik meg. A műveletet az *op* paraméterben kell megadni. A felhasználó is létrehozhat ilyen műveleteket, de a rendszerben vannak előre definiáltak is:

- MPI_MAX, MPI_MIN maximum, illetve minimum értéke
- MPI_SUM összegzés
- MPI_PROD produktum számítás
- MPI_LAND, MPI_LOR, MPI_LXOR logikai ÉS, VAGY, illetve kizáró VAGY művelet
- MPI_BAND, MPI_BOR, MPI_BXOR bináris ÉS, VAGY, illetve kizáró VAGY művelet
- MPI_MAXLOC, MPI_MINLOCmaximum, illetve minimum értéke és helye; Ennél a műveletnél az adatokban érték-index párokat kell megadni. Ezek leírására létezik néhány előre definiált adattípus: MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT és MPI_LONG_DOUBLE_INT.

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
int MPI_Op_free( MPI_Op *op)
```

Ha a programozónak más műveletekre is szüksége van, akkor azt ezekkel a rutinokkal tudja regisztrálni, illetve felszabadítani.

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Az *MPI_Reduce* művelet végrehajtásával megegyező a rutin működése, de az eljárás végén minden folyamatban meglesz a végeredmény másolata, nem csak a *root*-ban.

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Működése az *MPI_Reduce* rutinhoz hasonlít, de a művelet befejezése után egy *MPI_Scatter* rutinhoz hasonlóan az eredmények szétosztásra kerülnek.

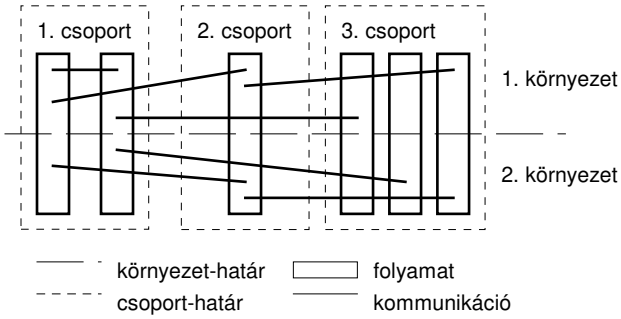
```
int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Asszociatív művelet kiszámítása a folyamatokban tárolt lokális adatokon. Az adott indexig kiszámolt részeredmény lesz eltárolva a folyamatban eredményképpen.

A 0 indexű folyamatban a bemenő értéket, a legnagyobb indexű folyamatban pedig az *MPI_Reduce* rutin eredményeképpen kapott értéket adja vissza.

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.



10. Ábra: Csoportok és környezetek

Folyamatok **csoportba** 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 **kommunikátor**ok. Ezek a fenti két fogalommal leírt kommunikációs közeget határozzák meg. Minden kommunikációban meg kell adni egy ilyen kommunikátort. Az indulásnál az *MPI_Init* eljárás létrehozza az MPI_COMM_WORLD kommunikátort, amiben minden folyamat benne van.

Csoportok

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 kihagyások nélkül halad végig az összes folyamaton.

Ilyen csoportok az *MPI_Group* átlátszatlan típussal írhatóak le, ezért nem lehet őket folyamatok között közvetlenül átadni. Két speciális csoport az *MPI_GROUP_EMPTY* az üres, és az *MPI_GROUP_NULL* a nem létező csoport jelölésére szolgál.

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

```
int MPI_Group_size(MPI_Group group, int *size)
```

Egy csoport méretének a lekérdezése.

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

A folyamat az adott csoportban meglévő sorszámának lekérdezése.

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
MPI_Group group2, int *ranks2)
```

Egy folyamat a *gorup1*-beli rangját a *group2*-beli sorszámmá konvertálhatja.

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
```

Az eljárás megmondja, hogy mi két csoport viszonya: egyenlőek-e, vagy az egyik tartalmazza-e a másikat, vagy nem összehasonlíthatóak.

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Egy kommunikációs környezethez tartozó csoportot lehet megkapni. Kezdetben csak az *MPI_COMM_WORLD* környezet létezik, és csak ennek a csoportját – azaz az összes folyamatot magában foglaló csoportot – lehet megkapni.

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group
*newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group
*newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group
*newgroup)
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group
*newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group
*newgroup)
```

Csoportokon, mint halmazokon végezhetőek műveletek, és így kialakíthatóak új csoportok. A lehetséges műveletek: unió, metszet, különbség, egyedi, illetve több egymásután következő folyamat bevétele egy halmazba, valamint kihagyása egy halmazból.

```
int MPI_Group_free(MPI_Group *group)
```

Egy csoport felszabadítása.

Kommunikátorok

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

Az *MPI_Comm* átlátszatlan típus egy kommunikátor leírására szolgál.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Egy kommunikátorhoz tartozó csoport mérete, illetve ebben a csoportban a folyamat sorszáma.

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

Két kommunikátor csoportokhoz hasonló összehasonlítása.

Az utóbbi három művelet csak lekérdezésre szolgál, ezért lokális jellegűek.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

Egy kommunikátor lemásolása. Egy új – hosszabb ideig tartó, vagy többlépéses – eljárás indulásakor lehet hasznos, hogy a benne folyó kommunikáció ne keveredjen a program többi részével.

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

Új kommunikátor létrehozása a csoport által meghatásozott folyamatokból.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

Egy kommunikátor felbontása részekre. Az azonos színű (*color*) folyamatok kerülnek egy kommunikátorba, és ebben a *key* paraméter által meghatározott sorszámot kapják.

A folyamatok két részre osztása ezzel könnyedén megoldható, ha az egyes folyamatok a szín és kulcs paraméterekben a *sorszám mod 2*, illetve a *sorszám div 2* értékeket használják.

```
int MPI_Comm_free(MPI_Comm *comm)
```

Egy kommunikátor felszabadítása.

Az utóbbi eljárások mind kollektívak, azaz más folyamatokkal való kommunikációra is szükség van bennük a sikeres befejeződéshez.

Távoli kommunikáció

Különálló folyamat csoportok közötti kommunikáció biztosítására szolgálnak az úgynevezett **inter kommunikátorok**. Ezzel az eszközzel egymástól teljesen független folyamtok között teremthető meg a kapcsolat. Az egyetlen feltétel az, hogy az egymással

kommunikálni akaró folyamatok között legyen egy-egy, amelyek azonos kommunikátorba tartoznak, így ismerhetik egymást.

Ez az eszköz az *MPI*-ban a kliens szerver típusú programok írásához nyújthat segítséget. Ez a típus előremutató jellegű, hiszen az *MPI* első változatának folyamat kezelése statikus, így minden folyamat ismerheti a másikat az MPI_COMM_WORLD kommunikátoron keresztül.

Ha az MPI könyvtár adott implementációja tartalmaz dinamikus folyamat kezelésre szolgáló kiegészítéseket – például új folyamat indítása, vagy MPI világok csatlakoztatása –, akkor valóban kihasználható ez az eszköz.

Az ilyen kommunikátorokban csak a pont-pont közötti üzenetküldés megengedett.

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

Teszteli, hogy az adott kommunikátor távoli kapcsolatot ír-e le.

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

Egy távoli elérésre szolgáló kommunikátor egy helyi és egy távoli csoportból áll. A fenti műveleteket alkalmazva egy ilyen kommunikátorra a helyi csoportról lehet információkat kinyerni. A távoli csoport méretének meghatározására szolgál ez az eljárás, a következő, pedig a távoli csoport leírásának elérésére.

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)
```

Egy új inter-kommunikátor készítése. A felépítéshez a helyi kommunikátoron kívül a helyi és a távoli folyamatcsoport vezetőjére, és az őket összefogó kommunikátorra van szükség.

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)
```

Kollektív műveletek elvégzéséhez nem alkalmas ez a távoli kapcsolat. Ez az eljárás egy inter-kommunikátort alakít át normálissá, így alkalmassá téve bármely művelet elvégzésére.

Tulajdonságok

Kommunikátorokhoz a felhasználó tulajdonságokat rendelhet, amelyek speciális függvények felhasználásával továbbörökíthetőek a kommunikátorral végzett műveletek során is.

Ez az eszköz alkalmas lehet arra, hogy egy könyvtár írója a saját könyvtárához szükséges információkat a kommunikátorhoz rendelje, és annak változásait rugalmasan követhesse.

A tulajdonságokat *(kulcs, érték)* párokkal lehet meghatározni.

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn, int *keyval, void* extra_state)
int MPI_Keyval_free(int *keyval)
```

Eme eljárásokkal lehet tulajdonságokhoz tartozó kulcsokat kezelni. Egy kulcs lényegében az adott tulajdonság neve lesz.

A tulajdonság létrehozásánál biztosítani kell azokat a függvényeket is, amelyek a hozzá tartozó adatok kezeléséről is gondoskodnak.

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

Tulajdonsághoz tartozó adat definiálása, lekérdezése, illetve törlése.

Ezek a tulajdonságok alkalmasak lehetnek a következőekben bemutatásra kerülő topológiák kezelésére is.

Folyamatok topológiába rendezése

Folyamatokat gráfokba rendezve le lehet írni a közöttük folyó kommunikációk absztrakt útjait. Az ilyen gráfok csúcsaiban a folyamatok helyezkednek el, és éleik a közöttük történő üzenetküldéseket reprezentálják. Bizonyos problémák megoldásánál ezek a gráfok jellegzetes alakzatokat vehetnek fel: gyűrű, fa, avagy hiperkocka.

Ilyen kommunikációs gráfok megadásával folyamatok egymáshoz viszonyított elhelyezkedését, topológiáját lehet megadni. Ez a két előnnyel is járhat:

- A programozó számára leegyszerűsítheti az üzenetküldésekkor a folyamatok elnevezését, így egyszerűbbé, világosabbá, és könnyebben kezelhetővé teszi a kódot.
- A rendszer számára információt ad a folyamatok közötti kommunikációról, melyet felhasználva az hatékonyabban képezheti le a folyamatokat a fizikai erőforrásokra (esetleg átrendezheti a folyamatokat, hogy optimálisabban használják ki környezetüket).

Az *MPI* lehetőséget biztosít teljesen általános topológiák leírására is, de sokszor használatos változatokhoz is biztosít kényelmi függvényeket.

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

Megvizsgálja, hogy az adott kommunikátorhoz tartozik-e valamilyen topológia.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
int reorder, MPI_Comm *comm_cart)
int MPI_Dims_create(int nnodes, int ndims, int *dims)
int MPI_Graph_create(MPI_Comm, int, int *, int *, int, MPI_Comm *)
```

Egy új topológia létrehozására szolgáló eszközök:

- *MPI_Cart_create* egy *ndims* dimenziós hálózat kialakítására szolgál, melyben megadható, hogy az egyes dimenziókban a hálózat periodikus legyen-e. A *reorder* paraméterrel engedélyezni lehet a folyamatok átsorszámozását a hatékonyabb kommunikáció érdekében.
- *MPI_Dims_create* kiegyensúlyozott elosztását biztosítja adott számú folyamatnak adott számú dimenzióban.
- *MPI_Graph_create* egy teljesen általános topológia kialakítására használható.

```
int MPI_Graphdims_Get(MPI_Comm comm, int *nnodes, int *nedges)
int MPI_Graph_get(MPI_Comm, int, int, int *, int *)
```

Általános gráf topológia lekérdezésére használatosak.

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int
*coords)
```

Hálózat topológiáról szerezhető információ a függvények használatával.

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

Egy hálózatban egy adott koordinátán elhelyezkedő folyamat rangja, illetve reng alapján annak koordinátája kérdezhető le a topológiából.

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int nneighbors)
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int
*neighbors)
```

Egy topológiában a szomszédokkal való összekötöttségi viszonyokról lehet ezekkel a függvényekkel információt szerezni.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int
*rank_source, int *rank_dest)
```

Egy hálózatban *direction* dimenzióban, *disp* távolságra lévő társ folyamatok rangját lehet megtudni.

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
```

Egy hálózat egy részét lehet kiemelni, és a további kommunikációkban önállóan felhasználni.

Néhány kiegészítő rutin a program környezetéhez

Az *MPI* ezen kívül definiál néhány eljárást a program környezetének kezelésére:

- a futtató processzor, vagy gép neve
- hibakezelő függvények definiálása
- helyi, vagy globális idő lekérdezése
- I/O lehetőségek az adott processzoron (van-e egyáltalán)
- program rendezett leállítása, vagy az egész rendszer futásának megszakítása

Teszteléshez segítség

Az *MPI* szabvány az optimalizálás, optimalizálás minimális egységesítését is megteszi. Egy szabványos függvényt definiál a programon belüli információ szolgáltatási szint beállítására. A szabvány előírja az statisztikai kóddal kiegészített függvények alakját is: minden eljárás nevét az *MPI_* helyett a *PMPI_* szóval kell kiegészíteni.

Ezek a lépések nyilván nem teremtenek egységes felületet az MPI programok tesztelése terén, de mégis definiálnak egy minimális követelményrendszert, melyet minden könyvtár implementálójának be kell tartani.

2.2.3. Példák

Mandelbrot halmaz számítása

Ez a példa csak a MPI-t használó program lényegét tartalmazza. A szakdolgozat végén megtalálható a példa teljes változata.

```
/*
 * Mandelbrot halmaz szamitasa (-2,-2)-(2,2)
 */
#include "mpi.h"

void main(int argc, char *argv[])
{
      int   rank;

      MPI_Init(&argc, &argv);       /* belepes az MPI-ba */
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      if(rank == 0)
            master(argc, argv);           /*    fo program  */
```

```
      else
            slave();                        /*    alprogramok */

      MPI_Finalize();
}

void master(int argc, char *argv[])
{
      ...
      read_args(argc, argv, &mandel, &fn);

      MPI_Comm_size(MPI_COMM_WORLD, &np); /* programok szama     */

      /* munka szetosztasa a programok kozott */
      Mandel_t = reg_Mandel();
      ...
      for(i=1, sy=0;
            i<np;
            i++, sy+=dy, work.y1+=delta_y, work.y2+=delta_y)
            MPI_Send(&work, 1, *Mandel_t, i,
                  MANDEL_WORK, MPI_COMM_WORLD);

      MPI_Type_free(Mandel_t);

      ...
      /* maradek szamitasa helyben  */
      slice = calc_mandel_tile(&work);

      /* eredmnyek osszegyujtese */
      for(i=1, sy=0; i<np; i++, sy+=dy)
            MPI_Recv(pix + (mandel.wd * sy), mandel.wd * dy,
                  MPI_BYTE,   i, MANDEL_RETURN, MPI_COMM_WORLD,
                  &status);
      bcopy(slice, pix + (mandel.wd * sy), mandel.wd * work.ht);

      /* eredmeny kiirasa file-ba */
      write_pix(fn, pix, &mandel);

}

void slave(void)
{
      ...
      /* varakozas a munkara */
      Mandel_t = reg_Mandel();
      MPI_Recv(&work, 1, *Mandel_t, master, MANDEL_WORK,
            MPI_COMM_WORLD, &status);
      MPI_Type_free(Mandel_t);

      /* a feladat elvegzese */
      slice = calc_mandel_tile(&work);

      /* az eredmeny visszakuldese */
      MPI_Send(slice, work.wd * work.ht, MPI_BYTE,
            master, MANDEL_RETURN, MPI_COMM_WORLD);
}

/* sajat adattipus a munka leirasara */
MPI_Datatype *reg_Mandel(void)
{
      MPI_Datatype      *Mandel_t;
      int         lengths[2];
      MPI_Aint    displs[2];
      MPI_Datatype      types[2];

      Mandel_t = (MPI_Datatype*)malloc(sizeof(MPI_Datatype));
```

```
        /* Mandel = 2*int, 4*double */
        lengths[0] = 2;
        displs[0]  = 0;
        types[0]   = MPI_INT;
        lengths[1] = 4;
        displs[1]  = sizeof(int) * 2;
        types[1]   = MPI_DOUBLE;

        MPI_Type_struct(2, lengths, displs, types, Mandel_t);

        MPI_Type_commit(Mandel_t);

        return(Mandel_t);
}
```

A példa a feladat Task-Farm típusú megoldását tartalmazza.

A párhuzamos programok közötti kommunikáció helyesség ellenőrzésére az MPI minden üzenet számára külön típust használhat. Ez a programok helyességének ellenőrzését csak futásidőben biztosítja, és némi többletköltséggel is jár.

2.2.4. 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, ezért fogalomrendszere az üzenetküldéses rendszerek követelményeinek leírásánál is nagyon jól használható.

A rendszerrel szinte mindenféle **feladat** megoldható, bár a szabvány jelenlegi változatában csak a statikus folyamatmodellt támogatja, ezért nem használható kliens/szerver megoldások, illetve hibatűrő alkalmazások írására.

A könyvtár kifejezetten **párhuzamos folyamatok** közötti kommunikáció támogatására íródott, nem tartalmaz elemeket adat párhuzamosság kezelésére (bár implementálhatóak vele ilyen könyvtárak is). Jelenlegi változata statikus folyamatmodellt használ, bár az egyes implementációk ezt a lehetőségeikhez mérten kiterjeszthetik (a következő változatban már lesznek eszközök a dinamikus modell kezelésére is). Ez a rendszer lényegében csak egy rutinkönyvtár, ezért a párhuzamosságnak a szintjét a futtató rendszer határozza meg: alapvetően programok lehetnek párhuzamosak, de – ha ezt a rendszer és az implementáció lehetővé teszi – akár párhuzamosan futó folyamat szálak közötti kommunikációra is használható.

A folyamatok közötti **kommunikáció** terén az összes üzenetküldés fajtára kínál megoldást (az időhöz kötött üzenetküldés is leírható nem blokkoló kommunikációval), sőt ezek kollektív változataival sok problémára kínál előre elkészített, egyszerűsített megoldást. A szabvány következő változatában az osztott memória használatára is vannak előterjesztések.

A **szinkronizáció** területén csak a szinkron kommunikáció, illetve a több folyamatot összefogó korlát áll a programozó rendelkezésére. Bármilyen osztott adattípus megvalósítása során a programozónak az üzenetküldést kell felhasználnia a szinkronizálási feladatok megoldására.

A szabvány egyéb jellemzői egy átgondolt könyvtár képét mutatják:

- helyesség: az eljárások viselkedése implementációktól és nyelvektől függetlenül jól definiált, és ez a helyesség belátásában nagy segítség
- megbízhatóság, hatékonyság: ezek implementációfüggő tulajdonságok, ezért nem értékelhetőek a szabvánnyal kapcsolatban.
- biztonság: alapvetően felhasználói programok írásához nyújt segítséget, osztott rendszer implementálásának nem ideális eszköze
- tesztelés, optimalizálás támogatása: a statisztikai elemekkel bővített könyvtárváltozatra tett kikötésekkel egy könyvtártól elvárható támogatás nagy részét teljesíti
- fejlesztéséhez elterjedésével egyre több segédeszköz elkészülte várható, jelenleg azonban ezek nincsenek nagy számban

Az *MPI* szabvány tehát az üzenetküldést használó rendszerek programozásának nagyon jó alapja.

2.3. PVM

PVM (Parallel Virtual Machine) egy software rendszer, amellyel hálózatba kapcsolt számítógépeket egyetlen nagy párhuzamos számítógépként lehet látni, és kezelni.

A rendszer fejlesztése 1989-ben kezdődött meg az Oak Ridge National Laboratory-nál. Azóta – több amerikai intézmény támogatásával – a rendszer kiteljesedett és a párhuzamos programok írásának egyik szabványává vált. Nagy előnye, hogy a publikus változata ingyen elérhető, emellett sok hardware gyártó biztosítja a saját gépére optimalizált, gyorsabb változatát is.

A rendszerben vannak programok, amelyek a felhasználó által írt párhuzamos program futtatását biztosítják, és C, illetve Fortran 77 könyvtárak ezek megírásához.

PVM rendszerbe kapcsolhatunk több – akár különböző típusú – számítógépet. A rendszer a rajta futó programok szempontjából ezek után egy nagy, elosztott memóriájú virtuális számítógépnek látszik: különböző processzorokon, különböző programokat indíthatunk el, melyek közötti szinkronizációt, és kommunikációt a PVM könyvtári függvényekkel oldhatjuk meg.

A virtuális gépben egy-egy taszk, vagy folyamat egy-egy UNIX programnak fog megfelelni (vannak más rendszerprogramokhoz írt változatok is). PVM három szinten támogat különböző felépítésű dolgokat:

- alkalmazásokat C és Fortran 77 nyelven is írhatunk
- a programokat különböző felépítésű gépeken is futtathatjuk, a rendszer biztosítja az adatok közötti értékhelyes konverziót
- a gépeket összekötő hálózatnak sem kell egységesnek lenni

A PVM rendszer két részből áll. Az első része egy démon program – amelyet *pvmd3*-nak hívnak – minden, a rendszerbe bekapcsolt számítógépen fut, hogy létrejöjjön a virtuális gép. A rendszer beindításakor minden gépen a felhasználó jogaival elindul egy-egy példánya ennek a programnak. A második rész a rutinkönyvtár, amelyben a felhasználó által hívható függvények vannak a kommunikáció, taszk indítás és kezelés támogatására.

2.3.1. Bevezető az eszközről

PVM-et bárki installálhat egy számítógépen, ha van rá azonosítója. Ha egy gépen többen is szeretnék használni a rendszert, akkor persze érdemes azt egy központi helyre rakni, hogy ne foglaljon el fölöslegesen területeket. A továbbiakban a PVM_ARCH név az adott gép

architektúrájának PVM-beli nevét fogja takarni. A publikus változat nagyjából 35 rendszeren fut, de léteznek a gépek gyártói által adott programcsomagok is.

2.3.2. Felhasználói felület

A felület függvényeiről részletes információ a **User's Guide** függelékében, illetve a rendszerhez adott online dokumentációkban található. Ez a fejezet csak áttekintést ad a lehetőségekről. A függvények legtöbbje Fortran 77 nyelvben is használható, de itt csak a C nyelvű felületről esik szó.

A taszkoknak a virtuális gépben egyértelmű azonosítójuk van, amelyet címként lehet használni a kommunikációban és a folyamatok kezelésében. A továbbiakban ezt a **tid** szó jelöli.

Folyamat vezérlés

```
int tid = pvm_mytid(void)
```

A rutin hatására a program egy PVM taszkká válik, bekerül a virtuális gépbe. A visszatérési érték a taszk PVM-beli azonosítója lesz. A rutin többször is meghívható, de csak egyszer hajtja végre az inicializáló lépéseket.

```
int info = pvm_exit(void)
```

E rutin közli a rendszerrel, hogy kilép belőle. A program ez után folytathatja a működését, csak PVM hívásokat nem hajthat már végre.

```
int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int
ntsask, int *tids)
```

Ez a rutin a *task* nevű programot *ntask* példányban *argv* parancssori argumentumokkal elindítja a virtuális gépen. A taszkkok azonosítóit a *tids* tömb fogja tartalmazni. Az indítás helye a *flag* és *where* paraméterekkel definiálható. Ha a *flag* értéke

- PvmTaskDefault PVM választja meg a helyet
- PvmTaskHost *where* által megadott gépen indul
- PvmTaskArch *where* által megadott architektúrán indul el
- PvmTaskDebug debugger alatt indul a taszk
- PvmTaskTrace a taszk nyomkövetési információkat is szolgáltat
- PvmTaskMppFront többprocesszoros gép kiszolgáló processzorán indul a taszk
- PvmHostCompl *where* komplemensén indulnak a programok A *where* paramáterben az aktuális gépet a „.” jelöli.

```
int info = pvm_kill(int tid)
```

A *tid* azonosítójú taszk leállítása.

Információk

```
int tid = pvm_parent(void)
```

Annak a taszknak az azonosítóját adja vissza, aki *pvm_spawn()*-nal elindította, vagy a *PvmNoParent* értéket, ha ilyen nem volt.

```
int pstat = pvm_pstat(int tid)
```

A taszk állapotát mutatja meg: *PvmOk* jelzi hogy fut, *PvmNoTask* ha nem, és *PvmBadParam* ha nincs is ilyen taszk.

```
int mstat = pvm_mstat(char *host)
```

Visszatérési értéke *PvmOk* ha a gép rendben van, *PvmHostFail* ha a gép nem elérhető, illetve *PvmNoHost* ha a gép nincs is benne a virtuális gépben. Hasznos lehet ez a rutin, ha hibatűrő alkalmazást írunk.

```
int info = pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)
```

A virtuális gép felépítését mondja el: gépek, architektúrák száma, valamint az egyes gépek nevei, rajtuk futó PVM démonok azonosítói, architektúrái, illetve a relatív sebességük.

```
int info = pvm_tasks(int which, int *ntask, struct pvmtaskinfo **taskp)
```

A *which* paraméterrel definiált taszkokról ad információkat (0 = minden taszk, démon tid = adott gép taszkjai, tid = adott taszk). Minden taszkról a következő adatokat adja meg: tid, szülő tid-je, állapot, démon tid-je és a taszk neve.

```
int dtid = pvm_tidtohost(int tid)
```

Adott taszkhoz tartozó démon azonosítóját adja meg.

Rendszer dinamikus változtatása

```
int info = pvm_addhosts(char **hosts, int nhost, int *infos)
int info = pvm_delhosts(char **hosts, int nhost, int *infos)
```

Gépek hozzáadása illetve törlése a virtuális gépből. Visszatérési érték a sikeresen végrehajtott műveletek száma, de az *infos* tömbből részletesebb információkat is szerezhetünk hiba esetén.

Signal küldés

```
int info = pvm_sendsig(int tid, int signum)
```

Egy *tid* azonosítójú taszknak a *signum* signal-t küldi el a rendszer. Ez a taszk más gépen is futhat, mint a küldő.

```
int info = pvm_notify(int what, int msgtag, int cnt, int *tids)
```

E rutin felkéri a rendszert, hogy bizonyos eseményekkor értesítse őt a rendszer. Az üzenet fejlécét a *msgtag* adja meg. A *what* paraméter értékétől függően üzenet érkezhet:

- PvmTaskExit ha a *tids* azonosítójú taszkok közül valamelyik megáll
- PvmHostDelete ha a *tids* azonosítójú gépek közül valamelyik leáll
- PvmHostAdd ha egy új gép kerül a rendszerbe (ez mindig meg kell újítani)

Paraméterek állítgatása

```
int oldval = pvm_setopt(int what, int val)
int val = pvm_getopt(int what)
```

Ezek a rutinok a PVM rendszer belső paramétereinek lekérdezésére és beállítására szolgálnak. A *what* paraméter értéke lehet:

- PvmRoute üzenetek irányítása
- PvmDebugMask debug kiírások aránya
- PvmAutoErr automatikus hibakiírás állapota
- PvmOutputTid az stdout ide van irányítva
- PvmOutputCode stdout ilyen fejléccel megy el
- PvmTraceTid a nyomkövetési üzenetek ide mennek
- PvmTraceCode ilyen fejléccel
- PvmFragSize üzenetek max. ekkora darabokban mennek el
- PvmResvTids üzeneteket a rendszer által használt címre is lehet-e küldeni

A lehetséges értékekről a függvény leírása bővebb tájékoztatást nyújt. Egy normál programban nincs szükség ezen paraméterek állítására, vagy lekérdezésére.

Üzenetküldés

Taszkok közötti kommunikáció egyetlen eszköze a PVM-ben az üzenetek küldése és vétele.

Az üzenet elküldése általában három részből áll. Először inicializálni kell egy küldő puffert a pvm_initsend() vagy pvm_mkbuf() rutinok meghívásával. Másodszor ebben a pufferben össze kell állítani az üzenetet a pvm_pk*() nevű függvények használatával. Harmadszor a kész üzenetet el kell küldeni a címzett(ek)hez a pvm_send() vagy pvm_mcast() rutinokkal.

A pvm_psend() rutinnal a három lépést egybe lehet fogni, de csak egy folyamatos tömböt lehet így elküldeni. A rutin előnye, hogy néhány speciális architektúrán sokkal gyorsabb, mint a fenti három lépéses üzenetküldés.

Üzenetek fogadására sokféle megoldás van. Lehet a programot blokkolva, bizonyos időkorláttal várakozva, vagy nem várva venni egy üzenetet. Specifikálhatjuk, hogy kitől, és mit szeretnénk venni. Ha a rendszer által biztosított üzenetvételi módszer nem megfelelő, akkor lecserélhetjük azt a pvm_recvf() függvény használatával. Az üzenetet a pvm_upk*() rutinokkal csomagolhatjuk ki.

Ez a két lépés a pvm_precv() rutinnal egybefogható. A pvm_psend() és pvm_precv() által nyújtott egyszerű módszerek – néhány speciális architektúrától eltekintve – ekvivalensek a többlépéses megoldásokkal.

pufferek: a következő rutinok akkor szükségesek, ha a felhasználó több különböző puffert szeretne használni az alkalmazásában. PVM-ben mindig egy *aktív* küldő puffer van, amibe egy elküldendő üzenetet lehet összeállítani, és egy *aktív* vevő puffer, amiben a legutóbb vett üzenet van benne.

```
int bufid = pvm_mkbuf(int encoding)
```

Új küldő puffer létrehozása. Az *encoding* paraméter az üzenet összeállításakor használt módszert adja meg:

- PvmDataDefault XDR kódolás szerint lesznek az üzenetben az adatok; akkor hasznos, ha különböző belső adatábrázolást használó gépek között küldünk üzenetet.
- PvmDataRaw semmilyen kódolás nem történik, az adatok az üzenetbe másolódnak
- PvmDataInPlace az adat a helyén marad, és csak az üzenet elküldésekor lesz kiolvasva; előnye, hogy az üzenet összeállítása után is módosíthatjuk az adatokat, így akár többször is elküldhetjük ugyanazt az üzenetet más-más adatokkal

```
int bufid = pvm_initsend(int encoding)
```

Törli az aktuális aktív küldő puffert, és egy újat inicializál a megadott kódolással. Ha csak egy puffert használunk, akkor mindig meg kell hívni egy üzenet összeállítása előtt.

```
int info = pvm_freebuf(int bufid)
```

Egy puffer felszabadítása. Csak akkor kell használni, ha külön készítettünk puffert, vagy átállítottuk az aktív puffert.

```
int bufid = pvm_getsbuf(void)
```

Az aktív küldő puffer azonosítóját adja meg.

```
int bufid = pvm_getrbuf(void)
```

Az aktív vevő puffer azonosítóját adja meg.

```
int oldbuf = pvm_setsbuf(int bufid)
```

Az aktív küldő puffert *bufid-re* állítja. Visszatérési értéke az előző azonosítója.

```
int oldbuf = pvm_setrbuf(int bufid)
```

Az aktív vevő puffert *bufid-re* állítja. Visszatérési értéke az előző azonosítója.

Az utóbbi két rutinban 0 értéket írva a *bufid* paraméterbe törölhetjük a puffereket. Ennek akkor lehet értelme, ha több PVM-et használó könyvtár van egy alkalmazásban, és ezek egymástól függetlenül szeretnének dolgozni. Ezzel a megoldással megőrizhetik a puffereik aktuális állapotát.

Üzenet összeállítása: a következő rutinokkal adatokat pakolhatunk be egy már létrehozott küldő pufferbe. Az összeállított üzenet nagyságának csak a memória szab határt, de vigyázni kell, mert semmilyen ellenőrzés nincs a vevő oldalon, hogy azt pakoljuk-e ki amit beleraktunk egy üzenetbe. Egyszerű változók és tömbök pakolása egyszerű, struktúrákat viszont elemenként kell belerakni egy üzenetbe.

```
int info = pvm_pkbyte(char *cp, int nitem, int stride)
```

```
int info = pvm_pkcplx(float *xp, int nitem, int stride)
int info = pvm_pkdcplx(double *zp, int nitem, int stride)
int info = pvm_pkdouble(double *dp, int nitem, int stride)
int info = pvm_pkfloat(float *fp, int nitem, int stride)
int info = pvm_pkint(int *np, int nitem, int stride)
int info = pvm_pklong(long *np, int nitem, int stride)
int info = pvm_pkshort(short *np, int nitem, int stride)
int info = pvm_pkuint(unsigned int *np, int nitem, int stride)
int info = pvm_pkushort(unsigned short *np, int nitem, int stride)
int info = pvm_pkulong(unsigned long *np, int nitem, int stride)
```

A függvények első paramétere a bepakolandó tömb első elemére mutat. Az *nitem* paraméter az elemek számát, a *stride* paraméter pedig a lépésközt határozza meg.

```
int info = pvm_pkstr(char *cp)
```

C-beli szöveg bepakolása (0-val terminált).

```
int info = pvm_packf(const char *fmt, ...)
```

printf formátum használata az üzenetek összeállítására. Egyedi változóknál értéket, tömböknél pedig az első elem címét kell átadni a további paraméterekben.

Üzenetküldés és vétel:

```
int info = pvm_send(int tid, int msgtag)
```

Üzenet elküldése a *tid* azonosítójú taszknak *msgtag* fejléccel.

```
int info = pvm_mcast(int *tids, int ntask, int msgtag)
```

Üzenet elküldése a *tids* tömbben felsorolt *ntask* darab taszknak *msgtag* fejléccel. Hatása ekvivalens *ntask* pvm_send() meghívásával, de egyes esetekben a rendszer így gyorsabban megoldhatja a feladatot.

```
int info = pvm_psend(int tid, int msgtag, void *vp, int count, int type)
```

Küldés három lépésének egybefogása. A *vp* címen található, *count* darab elemből álló, *type* típusú (értéke PVM_STR, PVM_BYTE, PVM_SHORT, PVM_INT, PVM_LONG, PVM_USHORT, PVM_UINT, PVM_ULONG, PVM_FLOAT, PVM_CPLX, PVM_DOUBLE, vagy PVM_DCPLX lehet) tömb elküldése a *tid* azonosítójú taszknak. Ez a rutin az aktív puffert nem változtatja meg.

```
int bufid = pvm_recv(int tid, int msgtag)
```

Üzenet fogadása a *tid* azonosítójú taszktól. Az üzenet fejléce *msgtag* kell hogy legyen. A -1-es érték – bármely paraméterbe írva – mindenhez illeszkedik, így *pvm_recv(-1,-1)* az első beérkező üzenetet fogadja.

A rutin addig vár amíg a megfelelő üzenet meg nem érkezik a taszkhoz. Ha az üzenet megjön, akkor az aktív vételi puffert törli, és azt az új üzenetre állítja.

```
int bufid = pvm_nrecv(int tid, int msgtag)
```

Üzenet fogadása a *pvm_recv()-hez* hasonlóan, de ha még nem érkezett meg az üzenet, akkor nem vár rá, és 0 értékkel tér vissza.

```
int bufid = pvm_probe(int tid, int msgtag)
```

E rutin megnézi, hogy érkezett-e a paraméterekben specifikált üzenet a taszkhoz. Ha igen, akkor 0-tól eltérő visszatérési értékkel fejeződik be. Ez nem törli az aktív vételi puffert.

```
int info = pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid)
```

Egy pufferben lévő üzenet hosszát (*bytes*), fejlécét (*msgtag*) és küldőjének azonosítóját (*tid*) adja meg.

```
int bufid = pvm_trecv(int tid, int msgtag, struct timeval *tmout)
```
Üzenet fogadása a *pvm_recv()-hez* hasonló módon, de a *tmout* paraméterben megadott ideig vár csak az üzenet megérkezésére.

```
int info = pvm_precv(int tid, int msgtag, void *vp, int count, int type,
int *rtid, int *rtag, int *rcount)
```
Üzenet fogadásának és kipakolásának kombinálása. A *tid, msgtag* és *count* valódi értékeit a *rtid, rtag* és *rcount* változókban adja vissza.

```
int (*old)(int,int,int) = pvm_recvf(int (*new)(int buf, int tid, int tag))
```
Az üzenetfogadó eljárások által használt szűrő függvény lecserélése.

Üzenetek kipakolása: az aktív vevő pufferből a következő rutinokkal nyerhetjük ki az adatokat:
```
int info = pvm_upkbyte(char *cp, int nitem, int stride)
int info = pvm_upkcplx(float *xp, int nitem, int stride)
int info = pvm_upkdcplx(double *zp, int nitem, int stride)
int info = pvm_upkdouble(double *dp, int nitem, int stride)
int info = pvm_upkfloat(float *fp, int nitem, int stride)
int info = pvm_upkint(int *np, int nitem, int stride)
int info = pvm_upklong(long *np, int nitem, int stride)
int info = pvm_upkshort(short *np, int nitem, int stride)
int info = pvm_upkuint(unsigned int *np, int nitem, int stride)
int info = pvm_upkushort(unsigned short *np, int nitem, int stride)
int info = pvm_upkulong(unsigned long *np, int nitem, int stride)
```
A függvények első paramétere a kipakolandó tömb első elemére mutat. Az *nitem* paraméter az elemek számát, a *stride* paraméter pedig a lépésközt határozza meg.

```
int info = pvm_upkstr(char *cp)
```
Szöveg kipakolása egy üzenetből.

```
int info = pvm_unpackf(const char *fmt, ...)
```
printf formátum használata az üzenet kipakolására.

Az összes rutinnál elegendő helyet kell biztosítani az adatok eltárolására.

A PVM rendszer nem ellenőrzi, hogy azt pakoljuk-e ki, amit beleraktunk egy üzenetbe. Erre a felhasználónak kell ügyelni!

Folyamatok csoportosítása

A taszkok dinamikus csoportosítása a PVM publikus változatában a meglévő rutinokra épül rá. Az ehhez szükséges adminisztrációt egy külön program (pvmgs) végzi el. Néhány hardware gyártó által adott változatban ezek is benne vannak a PVM rendszerben, ezért ott nem kell külön programot használni erre a célra (pl.: Cray T3D, Parsytec PowerXplorer).

A PVM rendszerben a csoportok használata nagyon egyszerű. Egy taszk bármikor csatlakozhat egy csoporthoz, vagy elhagyhatja azt anélkül, hogy a többieket értesítenie kéne. A legtöbb csoportokhoz tartozó függvényt bárki meghívhatja, még akkor is ha nem tagja a csoportnak.

```
int inum = pvm_joingroup(char *group)
int info = pvm_lvgroup(char *group)
```
A taszk a *group* nevű csoporthoz csatlakozik, illetve elhagyja azt. Egy taszk egyszerre több csoportnak lehet tagja, sőt egy csoportból kiléphet, majd újra beléphet futása során kedve szerint.

A *pvm_joingroup()* rutin visszatérési értéke a taszknak a csoportban kapott sorszáma.

```
int tid = pvm_gettid(char *group, int inum)
```

A *group* nevű csoportban az *inum* sorszámú taszk azonosítója.

```
int inum = pvm_getinst(char *group, int tid)
```

A *group* nevű csoportban a *tid* azonosítójú taszk sorszáma.

```
int size = pvm_gsize(char *group)
```

A *group* nevű csoport mérete.

```
int info = pvm_barrier(char *group, int count)
```

Taszkok szinkronizációjára szolgáló rutin. Addig vár amíg a *group* nevű csoportban *count* taszk meg nem hívja ezt a függvényt. A híváskor a taszknak benne kell lennie a csoportban. A *count* értéke kisebb is lehet, mint a csoport mérete.

```
int info = pvm_bcast(char *group, int msgtag)
```

Üzenet küldése a *group-ban* lévő összes taszknak – kivéve magának, ha ő is tag – *msgtag* fejléccel.

```
int info = pvm_reduce(void (*func)(), void *data, int nitem, int type, int
msgtag, char *group, int rootginst)
```

Asszociatív művelet végrehajtása egy csoporton belül. A művelet eredménye a *rootginst* sorszámú taszkon lesz eltárolva. A művelet adatait a *data, nitem* és *type* paraméterek adják meg (pvm_psend-hez hasonlóan). Lehetséges műveletek:

- PvmMax maximum keresés
- PvmMin minimum keresés
- PvmSum összegzés
- PvmProduct szorzat

A műveletek először az adat tömb első elemein lesznek elvégezve, és az eredmény a kimenet első elemébe kerül. A bemenet további elemei hasonló módón lesznek feldolgozva.

A felhasználó maga is írhat ilyen eljárásokat.

```
int info = pvm_gather( void *result, void *data, int count, int type, int
msgtag, char *group, int rootginst)
int info = pvm_scatter( void *result, void *data, int count, int type, int
msgtag, char *group, int rootginst)
```

Adatok szétterítése, illetve begyűjtése egy csoportban. Az eredmény mindig a *result* tömbbe kerül. A helyi adatokat a *data, count* és *type* paraméterek határozzák meg a *pvm_psend()* rutinhoz hasonlóan. A kommunikáció során a *msgtag* fejléccel fognak menni az üzenetek. A műveletet a *group* csoportban hajtódik végre úgy, hogy a *rootginst* sorszámú taszk lesz a vezető.

A *pvm_gather()* függvénnyel a csoportban lévő taszkoktól lehet adatokat összegyűjteni. Ha *n* taszk van, és mindegyik *count* adatot ad meg, akkor a vezető taszkban *n*count* adat fog összegyűlni a *result* paraméter által specifikált tömbben.

A *pvm_scatter()* rutinnal a vezető taszkon lévő *data* című *n*count* hosszú tömböt lehet *n* taszk között szétosztani. A tömb részletei a csoportban lévő taszkokban a *result* címre kerülnek.

2.3.3. Példák

Mandelbrot halmaz számítása

Ez a példa csak a PVM-et használó program lényegét tartalmazza. A szakdolgozat végén megtalálható a példa teljes változata.

```
/*
 * Mandelbrot halmaz samitasa
 */
#include "pvm3.h"

void main(int argc, char *argv[])
{
      pvm_mytid();            /* belepes a PVm rendszerbe   */
      if(pvm_parent() == PvmNoParent)
            master(argc, argv);     /* fo program                */
      else
            slave();          /* seged programok                */
      pvm_exit();       /* kilepes                    */
}

void master(int argc, char *argv[])
{
      ...
      /* parameterek beolvasasa */
      read_args(argc, argv, &mandel, &fn);

      np = /* processzorok szama (pl. pvm_conf ) */

      /* Segedprogramok inditasa    */
      pvm_spawn("mandel_pvm", NULL, PvmTaskDefault,
                  "", np, tids);

      ...

      /* munka szetosztasa a segedek kozott     */
      for(i=0, sy =0, step_y=mandel.y1;
            i<np;
            i++, sy+=dy, step_y+=delta_y)
      {
            pvm_initsend(PvmDataDefault);
            pvm_packf("%d %d %lf %lf %lf %lf", mandel.wd, dy,
                  mandel.x1, step_y, mandel.x2, step_y+delta_y);
            pvm_send(tids[i], MANDEL_WORK);
      }

      ...
      /* a foprogram is dolgozik */
      slice = calc_mandel_tile(&work);

      /* eredmenyek osszegyujtese */
      for(i=0, sy=0; i<np; i++, sy+=dy)
      {
            bufid = pvm_recv(tids[i], MANDEL_RETURN);
            pvm_upkbyte(pix + (mandel.wd * sy),
                              mandel.wd * dy, 1);
      }
      bcopy(slice, pix + (mandel.wd * sy), mandel.wd * work.ht);

      /* eredmeny kiirasa egy file-ba */
      write_pix(fn, pix, &mandel));
}

void slave(void)
```

```
{
      ...
      master = pvm_parent();

      /* varakozas az instrukciokra */
      pvm_recv(master, MANDEL_WORK);
      pvm_unpackf("%d %d %lf %lf %lf %lf", &work.wd, &work.ht,
            &work.x1, &work.y1, &work.x2, &work.y2);

      /* a kiadott feladat elvegzese */
      slice = calc_mandel_tile(&work);

      /* az eredmeny visszakuldese */
      pvm_initsend(PvmDataDefault);
      pvm_pkbyte(slice, work.wd * work.ht, 1);
      pvm_send(master, MANDEL_RETURN);
}
```

A példa egy egyszerű Task-Farm megoldással párhuzamosítja a feladatot.

A forráskódból látható, hogy az üzenetküldések között a fordító és a futtató rendszer csak a programozó által biztosított üzenettípus alapján fedezhetné fel a kapcsolatot, tehát a párhuzamos program egészére nem lehet semmilyen automatikus helyességvizsgálatot végezni.

A PVM-ben írt megoldás erőssége a rugalmasság (futás időben indít új folyamatokat), és a kivételes helyzetek kezelésének lehetősége (pvm_notify).

2.3.4. 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 fejezet elején 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 rendszerben találhatóak elemek a kivételes esetek kezelésére is (pvm_notify), amelyek nagy segítséget jelentenek hibatűrő alkalmazások írásakor.

A *PVM* rendszer csak programok **párhuzamosság**át 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 üzenet küldés 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.

A **szinkronizációra** csak több folyamat egy határnál való várakoztatása használható fel függetlenül. Szinkron üzenetküldés is szimulálható egy aszinkron üzenetküldés párral, de ennél magasabb szintű szinkronizációs módszerek megvalósítására már nincs egyszerű megoldás.

A *PVM* rendszer egyéb jellemzői:

- helyesség: az alap rendszer – publikus változat – jól definiált eljárásokat tartalmaz, és ezeket minden architektúrán azonos módon működnek. Sajnos a *PVM* számítógép gyártók által biztosított változatai több-kevesebb eltérést tartalmaznak a publikus változattól, ezért egyes esetekben egy másik architektúrán helyesen működő program meglepő eredményeket produkálhat (én az IBM SP1-es, Cray T3D és Parsytec PowerXplorer gépekhez adott PVM implementációk használata után voltam kénytelen leszűrni ezt a tapasztalatot).
- megbízhatóság: a PVM rendszer nagyon jól tűri a hibákat, akár még egy processzor kiesését is lehet vele kezelni.
- hatékonyság: a rendszer publikus változata is nagyon jó eredményeket mutat, de a számítógép gyártók által adott implementációk még gyorsabb programok írására adnak lehetőséget. Az ilyen *PVM* változatok időeredményei kis üzeneteknél maximum 10% többletidőt igényelnek alacsonyabb szintű kommunikációs módszerekhez képest, nagy üzeneteknél ez a többletköltség is eltűnik.
- hordozhatóság: a legtöbb *PVM*-ben írt program változtatás nélkül vihető át bármely architektúrára, csak a helyességnél említett kis különbségek igényelhetnek finom változtatásokat az adott implementáció igényeinek megfelelően
- tesztelésre már a rendszer korai változataitól volt lehetőség, nyomkövetés magasabb szintű támogatására a legújabb változatokban jelentek meg eszközök
- a *PVM* rendszer előnye, hogy viszonylag hosszú ideje létezik már, ezért nagyon kényelmes fejlesztő és futtató környezetek készültek már hozzá (pl. XPVM)

A *PVM* rendszer tehát a használat során fejlődött mai állapotáig, ezért sok nagyon hasznos, és számításigényes feladatok megoldása során felmerülő problémák megoldására alkalmas eljárást tartalmaz. Néhány – kompatibilitás miatt megőrzött – megoldástól eltekintve nagyon jól használható eszköz párhuzamos programok írására.

2.4. CC++

Compositional C++ a C++ nyelv egy kiegészítése párhuzamos elemekkel. A kiegészítésekkel a nyelv alkalmas lesz folyamat és utasítás szintű párhuzamosság kezelésére.

Osztott adattípusok implementálására, és párhuzamosan futó programrészek közötti kommunikáció és szinkronizáció lépéseinek leírására.

2.4.1. Bevezető az eszközről

A *CC++* nyelvet a kaliforniai California Institute of Technologie egyetemen fejlesztik. Ez a nyelv most is még a fejlesztés állapotában van, de megoldásait már más nyelvek is átvették (pC++ és HPC++).

A nyelvet jelenleg Solaris 2.3 és 2.4, AIX 3.2.5 valamint IBM SP1-es rendszereken lehet használni. Minden helyen szükség van C++ fordítóra és egy thread könyvtárra is (a C++ fordító a GNU C++ fordító is lehet).

A *CC++* nyelv a C++ nyelv kiterjesztése, azaz minden helyes C++ program helyes CC++ program is. Egy CC++ program lefordításához szükség van CC++ fordítóra és a hozzá tartozó könyvtárakra. Az így elkészített program futtatása a nyelvhez készített *Nexus* kommunikációs rendszerrel lehetséges.

A CC++ nyelv a következő kiegészítéseket adja a párhuzamosság támogatására:

- **processor object** egy osztály *virtuális processzorok* definiálására, amivel a program objektumait lehet több processzor között szétosztani. A virtuális processzorok elhelyezkedését is definiálni lehet a *proc_t* típus segítségével.
- **global** típusmódosító, mellyel virtuális processzorok között megosztott változókat lehet létrehozni.
- **folyamat-szál** kezelésére szolgáló megoldások: *par, parfor* ***és*** *spawn* kulcsszavakkal
- **sync** típusmódosító folyamatok közötti szinkronizálásra használható változók dekralásakor.
- **atomic** kulcsszó az atomicitás jelölésére
- új **adatfolyam** definiálása a kommunikáció objektumorientált leírására

2.4.2. Felhasználói felület

A CC++ nyelvben az összes C++ konstrukció használható. Az itt bemutatásra kerülő kiterjesztések a C++ nyelvben meglévő fogalmakhoz igazodva azok hatáskörével – nagyrészt – összhangban terjesztik ki a nyelvet, ezért a megfelelő C++ nyelvi elemek ismerete elengedhetetlen a kiterjesztés helyes használatához.

Itt csak a C++ és a CC++ nyelv közötti eltérésekről lesz szó, a C++ nyelvről nem.

Konkurencia

A CC++ program a C++ programhoz hasonlóan egy egyszerű egy szálon futó programként működik, ha nem használjuk a nyelv kiterjesztéseit.

Egy többprocesszoros, osztott memóriát használó rendszerben hatékonysági, vagy egy egyprocesszoros rendszerben a holtpont elkerülése végett szükségünk lehet több szálon futó programok írására. Erre e célra a CC++ nyelv a **par, parfor** és **spawn** kulcsszavakat biztosítja.

A **par** kulcsszóval a C nyelv szokásos blokk fogalmát lehet átalakítani párhuzamos blokká:

```
par {
      a = 2;
      b = 3 * 4;
}
```

Az így definiált blokk utasításai külön-külön lesznek végrehajtva egy-egy önálló szálként a főprogramban. Ha a futtató rendszerben több processzor is rendelkezésre áll, akkor ezek az utasítások valóban párhuzamosan is végrehajtódhatnak. Az utasítások végrehajtásának sorrendjére a nyelv semmilyen megkötést nem tesz, és a szálak közötti igazságos időmegosztást sem határozza meg, ezért a következő konstrukciók veszélyesek lehetnek:

```
a = 1;
par {
      while(a);
      a = 0;
}
```

Veszélyes, mert lehet, hogy csak a végtelen ciklus fog futni.

```
par {
     b = 3;
      a = b * 2;
}
```

Veszélyes, mert lehet, hogy előbb *a* kap értéket.

A *par* blokk végén az elindított szálak megvárják egymást, és a blokk lefutása után megszűnnek, tehát a blokk vége egy implicit szinkronizációt tartalmaz.

Párhuzamos blokkok egymásba ágyazhatóak, és bármilyen CC++-beli utasítást tartalmazhatnak. Az egyetlen kikötés az, hogy nem tartalmazhatnak változó deklarációt, hiszen ennek a szemantika szempontjából nem lenne semmi értelme.

A **parfor** kulcsszóval párhuzamosan végrehajtott ciklusokat lehet létrehozni. Az ilyen ciklusok fejrésze végigfut a ciklusváltozó lehetséges értékein, és az egyes értékekre a ciklusmagot több szálon elindítja:

```
int   A[N];
parfor(int i=0; i<N; i++)
      A[i] = work(i);
```

A *parfor* kulcsszó a *for* kulcsszó helyett használható. A párhuzamos ciklusokra a következő megkötések érvényesek:

- A ciklusváltozót a ciklus fejében kell deklarálni, és csak egyszerű típus lehet.
- A ciklusváltozót csak a ciklus fejében lehet módosítani.
- A ciklus magjában a ciklusváltozó *const* változóként látható, ezért nem lehet megváltoztatni.
- A ciklusmagban – a hagyományos ciklustól eltérően – nem lehet *goto, break* és *continue* utasítás.
- A ciklusmagban használt lokális változókat a ciklus magjában kell deklarálni, különben több szál közösen használná őket.

Ezekkel a megkötésekkel a CC++ parfor ciklusa már Ada cikluskénti is megírható, hiszen ott is ezek a szabályok érvényesek a ciklusváltozóra.

A ciklus lefutása után az újonnan indított szálak megvárják egymást, majd befejezik működésüket.

Nem strukturált elem az új **spawn** kulcsszó által bevezetett párhuzamosság:

```
void f(...);
spawn f();
```

A *spawn* kulcsszóval eljárásokat lehet a főprogramtól független végrehajtási szálon elindítani. Az ilyen függvények visszatérési típusa *void* kell, hogy legyen, hiszen nem térnek vissza a főprogramhoz. A *spawn*-nal elindított függvények futása teljesen független szálon történik, semmilyen implicit szinkronizáció nem jön létre, mint az előző két konstrukciónál.

A független futásnak megvannak a veszélyei is:

- a *spawn*-nal indított függvény hivatkozhat pointeren keresztül olyan lokális változóra, mely már régen felszabadult
- a főprogram – nem ismerve a párhuzamos függvény állapotát – befejezheti működését az így indított eljárás lefutása előtt

Ez a konstrukció sok veszélyt rejt magában, helyes használatához szinkronizációs módszereket is igénybe kell venni.

E három konstrukció tehát ugyanazt a lokális memóriaterületet használó folyamat szálak indítására szolgál. Az ilyen folyamatok indítása csak számolásigényes feladatoknál, vagy szemantikai szükségszerűségből indokolt az erőforrás-igény nagysága miatt. Egészekből álló tömb elemeinek inicializálására – ld. fent – ezek a konstrukciók nagy teljesítménycsökkenést eredményezhetnek.

Atomicitás

Párhuzamos szálak által közösen használt változók esetén a konkurens módosítás nem várt eredményeket hozhat a változó érétkeiben. Ilyen problémák elkerülésére A változót valamilyen szinkronizációs burokba érdemes berakni. Ennek legegyszerűbb formája az **atomic** kulcsszóval érhető el:

```
class Value {
      private:
            int   a;
      public:
            atomic void assign(int i)
                  { a = i; }
};

int main(int argc; char *argv[])
{
      Value v;
      par {
            v.assign(1);
            v.assign(2);
      }
}
```

Az *atomic* kulcsszót osztályok tagfüggvényeinek definiálásakor a *static* és a *virtual* kulcsszavakkal megegyező módon lehet használni. Az így deklarált tagfüggvények köré a fordító egy szinkronizációs kódot rak, amely gondoskodik róla, hogy a tagfüggvényből egyszerre csak egy példány futhasson. Ha a tagfüggvényt egy másik végrehajtási szálból is meghívják, akkor annak a futása az atomi függvény lefutásának idejére fel lesz függesztve.

A kulcsszó meggondolatlan használatával nagyon könnyen el lehet érni a holtpontot, ezért érdemes betartani a következő szabályokat:

- az atomi függvényeknek mindig terminálniuk kell (lehetőleg minél hamarabb, hiszen ezek jelenthetik a szűk keresztmetszetet egy párhuzamos program futásában)
- atomi függvényeknek nem szabad várakozniuk más folyamatokra

- atomi függvények ne tartalmazzanak párhuzamos blokkokat illetve ciklusokat
- atomi függvények ne hívjanak más függvényeket (főleg ne atomiakat)

Ezen szabályokat persze meg lehet sérteni, és még így is jól működhet a program, de a betartásuk garantálja a holtpont elkerülését.

Az egyetlen megkötés a nyelvben az *atomic* kulcsszó használatával kapcsolatban az, hogy nem szerepelhet a *static* kulcsszóval együtt, mert az összes párhuzamossággal kapcsolatos konstrukció csak objektumok szintjén érvényesíthető (nincs önállóan futó osztály-objektum, amiben ezek kezelhetőek lennének).

Szinkronizáció

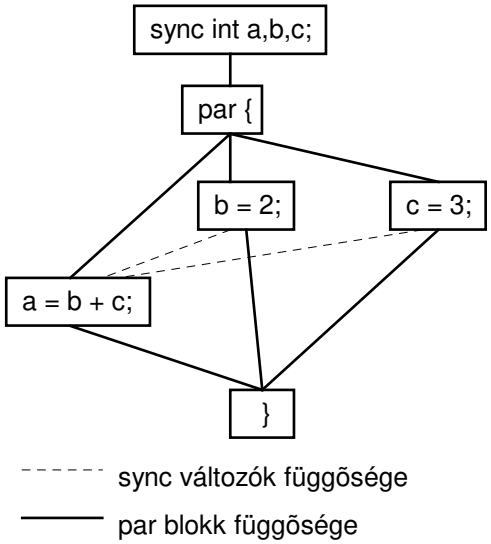
Az összes szinkronizációs feladat megoldására a CC++ nyelv a **sync** kulcsszóval meghatározott konstrukciókat használja. Ez a kulcsszó a *const* kulcsszóval megegyező helyeken használható, és hatása is majdnem ugyanaz:

```
{
      sync int a, b, c;
      par {
            a = b + c;
            b = 2;
            c = 3;
      }
      // a == 5, b ==2, c == 3
      return a;
}
```

A *sync* kulcsszó típusmódosításra használható. Az így definiált változó egy késleltetett inicializálású konstans, avagy egy egyszer meghatározható értékű változó lesz.

A *sync* változónak csak egyszer lehet értéket adni – nem feltétlenül a létrehozásakor, mint a *const* változónak –, és ezt az értéket csak ezután lehet belőle lekérdezni. Ha egy folyamat az érték meghatározása előtt szeretné megtudni egy ilyen változó értékét, akkor annál a pontnál a futása felfüggesztődik, és addig fog várakozni, amíg egy másik folyamat nem inicializálja az adott változót. Az inicializálás után a változó konstansként viselkedik, bármely újabb értékadási kísérlet futás idejű hibához vezet.

A fenti példa a *par* blokk ellenére meghatározott végeredménnyel fog lezárulni, mert az *a* változóhoz csak a *b* és *c* változók ismert értéke után rendelődhet érték. A *sync* változók hatására a program futása:



11. Ábra: Függőségi viszonyok sync változókkal

Sync típusmódosítót függvények argumentumlistájában és visszatérési értékénél is lehet használni, jelentése ott is hasonló lesz.

Az egyszeres értékadású változót nem lehet típuskényszerítéssel, vagy címének meghatározásával hagyományos változóvá konvertálni, és ezzel kikerülni a védő mechanizmusát. Ebben a viselkedésben teljesen megegyezik a *const* változók használatánál betartandó szabályokkal.

Egy ilyen változó élettartalma a szokásos láthatósági viszonyoknak megfelelő élettartammal egyezik meg. Ha a lokális *sync* változó nem elég, akkor persze lehet deklarálni dinamikusan lefoglalt változatát is:

```
{
      sync int    *a = new sync int;

      *a = 3;
}
```

Globális mutatók

Több processzor között elosztott párhuzamos programnál más-más processzorokon futó programrészek más-más memóriákat láthatnak. Az egymástól elválasztott memóriaterületeken belül – lokálisan – a *par, parfor* és *spawn* kulcsszavakkal létre lehet hozni önállóan futó végrehajtási szálakat, ezek azonban csak a saját memóriájukon belüli változókat érhetik el.

Ahhoz hogy több processzoron futó program külön memóriákban futó részei között valamilyen kapcsolat alakuljon ki a *CC++* nyelv a ***global** típusmódosítót vezette be. Ez a kulcsszó a *volatile*, és *register* típusmódosítókkal egyezően használható változók definiálásakor.

A globális mutatók (pl. int *global a) különálló memóriákban futó folyamatok számára is közösen láthatóvá tesznek memóriarészeket. A különbséget az egyes folyamatok számára alapvetően az elérés ideje jelenti majd. Az a folyamat, amely memóriájában tartalmazza a változót sokkal gyorsabban tudja majd elérni azt, mint egy távoli társa. A hivatkozásban a két folyamat között nem lesz semmi jelölésbeli különbség, csak az elérés módszerében.

```
int *global pi;
int * *global     ppi;
```

```
C *global pC;
```

Ezek a mutatók bármilyen típusra mutathatnak, kivéve a függvényeket.

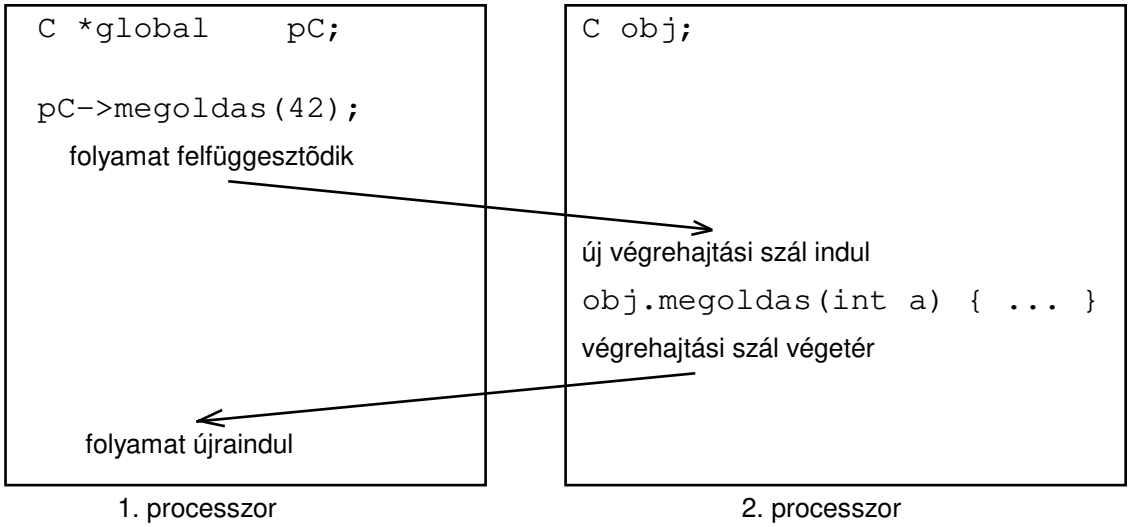
Globális változók elérése a rendes mutatókkal megegyező:

```
int x = *pi + 1;
int x += **pi;
```

A használat során a programozónak nem kell figyelnie arra, hogy a változó valójában melyik processzor memóriájában foglal helyet.

Ha az adott mutató egy objektum, akkor annak tagfüggvényei is elérhetőek a mutatón keresztül:

```
pC->megoldas(42);
```



12. Ábra: Távoli eljáráshívás CC++-ban

Ennél a lépésnél az éppen futó folyamat futása felfüggesztődik, és a távoli objektum tagfüggvénye távoli eljáráshívással végrehajtódik. A végrehajtás idejére a globális mutatóval jelzett objektum processzorán egy új szál jön létre a tagfüggvény lefuttatására. Globális mutatóval jelölt objektumok tagfüggvényei közül ezért egyszerre több is futhat. A távoli objektum konzisztenciája érdekében az ilyen távoli eljáráshívásban résztvevő tagfüggvényeket *atomic* vagy *sync* kulcsszavakkal leírható konstrukciókkal kell szinkronizálni.

A távoli eljáráshívás során a paramétereket a kommunikációhoz definiált *CCVoid* adatfolyammal adja át a rendszer.

Globális pointereket csak a == és != operátorokkal lehet összehasonlítani, a < és > operátorokkal nem.

Megengedett a globális pointerek explicit kényszerítése lokális pointerré, de ha az adott mutató nem helyi memóriát jelöl, akkor ez futásidejű hibához vezet.

Virtuális processzorok

Virtuális processzorokkal adattagok és kódok foghatóak össze, amelyek együtt egy processzoron végrehajtható programrészt definiálnak, melyek közös memóriaterületet látnak. A virtuális processzoron futó folyamatok feltétlenül közös memóriát kell hogy használjanak, még ha a futtató rendszer több processzoron tudja is ezeket a folyamatokat futtatni.

Több virtuális processzor is leképezhető egyetlen fizikai processzorra. Ezt a programozó határozhatja meg a *proc_t* típuson keresztül.

Virtuális processzorokat normál C++ osztályok segítségével lehet alkotni, melyeket a *global* típusmódosítóval kell ellátni, pl.:

```
// Udvozlo.h
#include <iostream.h>
global class Udvozlo { // ez virtualis processzor lesz
      public:
            Udv() {};
            void mondj_hellot(int id);
};

// Hello.cc++
#include <stdlib.h>
#include "Udvozlo.h"
int main(int argc, char *argv[])
{
      int P = atoi(argv[1]);
      parfor(int p=0; p<P; p++)
      {
            Udvozlo     *global     U;
            proc_t elhelyezes = proc_t("Udvozlo", argv[2+p]);
            U = new (elhelyezes) Udvozlo();
            U->mondj_hellot(p);
            delete U;
      }
      return 0;
}

// Udvozlo.cc++
#include "Udvozlo.h"
void Udvozlo::mondj_hellot(int id)
{
      cout << "Hello a " << id << ". processzorrol! " << endl;
}
```

Ebben a példában az *Udvozlo* osztály egy virtuális processzort definiál. Az *Udvozlo.cc++* forrást külön kell lefordítani, hogy a virtuális processzorhoz tartozó programot elkészíthesse a fordító. Ennek a programnak az indításáról a rendszer gondoskodik, amikor létrejön egy új példánya a *new* utasítás hatására.

Ha egy osztály a *global* módosítóval van ellátva, akkor abból egy virtuális processzor lesz. Az összes adattagját és publikus eljárását elérhetik globális mutatón keresztül más processzorokról az ott futó folyamatok.

A *global* módosítószó nem befolyásolja az öröklődést, azaz a *private* és *protected* láthatóságú osztály részeket a C++-hoz hasonlóan lehet elérni a leszármazott osztályokban.

Virtuális processzorok létrehozásakor mindig meg kell adni az elhelyezésüket is. Ez a CC++ nyelvben a *proc_t* típuson keresztül történik. Ebben az implementáció függő típusban általában a virtuális processzort futtató fizikai processzor, vagy gép nevét lehet megadni, valamint a virtuális processzor futtatható kódját tartalmazó file elérési útját.

Az így létrehozott elhelyezési információval már használható a *new* operátor virtuális processzorok dinamikus létrehozására.

Ezen speciális objektumok elérése – mint az a fenti példákból kiderült – a hagyományos objektumok elérésével egyezik meg, csak költségesebb.

Egy dinamikusan lefoglalt virtuális processzor, vagy processzor objektumra mutató globális mutató felszabadításával meg lehet szüntetni az adott virtuális processzort:

```
delete U;
```

Minden CC++ programban lehet használni a *::this* mutatót, amely mindig az adott processzoron futó processzor objektumra mutat.

Adatfolyam a kommunikációra

Fizikai processzorok közötti kommunikációkban – távoli adattagok elérésekor, illetve távoli eljárások hívásakor – a CC++ rendszernek át kell küldeni a lokális adatokat.

Erre e feladatra a rendszer használhatná a memória másolást is, ez azonban a C programok mutatókkal teletűzdelt világában ritkán vezet jó eredményre. A probléma megoldására a CC++ nyelvben definiáltak egy új adatfolyamot, a **CCVoid** típust.

Minden kommunikációban résztvevő típusnál definiálni kell a következő operátorokat:

```
CCVoid& operator<<(CCVoid&, const TYPE& obj_in);
CCVoid& operator>>(CCVoid&, TYPE& obj_out);
```

Ezek az összes beépített típusra definiálva vannak, így a műveletek összeállításakor felhasználhatóak ezek a primitívek, pl.:

```
class Point {
      float x;
      float y;
      friend CCVoid& operator<<(CCVoid&, const Point& p_in);
      friend CCVoid& operator>>(CCVoid&, Point& p_out);
};
CCVoid& operator<<(CCVoid& v, const Point& p_in)
{
      v << p_out.x << p_out.y;
      return v;
}
CCVoid& operator>>(CCVoid& v, Point& p_out);
{
      v >> p_in.x >> p_in.y;
      return v;
}
```

Az egész módszer a *istream* és *ostream* folyamok használatával egyezik meg.

Ha a programozó definiálja ezeket az operátorokat, akkor azt a CC++ fordító automatikusan felhasználja, ha szüksége van rá. Ha nincsenek definiálva ezek, akkor a fordító megpróbál generálni ilyeneket, hogy a processzorok közötti kommunikációt megvalósíthassa. A fordító által generált kódok persze nem valószínű, hogy a programozó elvárásának megfelelően kezelik a mutatókat és struktúrákat.

2.4.3. Összefoglalás

A CC++ programozási nyelv a C++ nyelv taszk párhuzamosságot támogató kiterjesztéseként készült el, ezért főleg bonyolult algoritmusokat igénylő párhuzamos **feladatok** megoldására alkalmas (pl. funkcionálisan szétoszthető feladatok, vagy adatok szerint szétoszthető feladatok, amelyek kiegyensúlyozatlanok).

A CC++ nyelv kiterjesztései jól támogatják rugalmas, helyzethez alkalmazkodó programok írását. A C++ nyelv kivételkezelésével kombinálva nagyon jó, és hibatűrő alkalmazások írását teszi ez lehetővé.

Kliens/szerver megoldások implementálására is nagyon jól felhasználható a nyelv, viszont nem definiál olyan eszközöket, amely futásidőben lehetővé tenni futó alkalmazások összekapcsolását, ezért a *PVM*-hez hasonló gondokkal kell megküzdenie az ilyen programok íróinak.

A CC++ nyelven írt programok hatékonysága és biztonsága az alatta lévő kommunikációs rétegtől függ. Jelenleg ez a nyelv alkotói által biztosított *Nexus* rendszer. Ez nem teszi lehetővé osztott operációs rendszer írását, de ennek lecserélésével a nyelv erre is alkalmas lehet.

A CC++ nyelv sok szinten támogatja a folyamatok **párhuzamosságát**. A *par* és *parfor* kulcsszavak az utasítások , kifejezések szintjén, a *spawn* utasítás a végrehajtási szálak szintjén és a processzor objektumok a párhuzamosan futó önálló memóriájú programok szintjén.

Folyamat modellje dinamikus, azaz a program bármely pontján lehet új végrehajtási szálajkat, illetve párhuzamos programokat indítani és megszüntetni.

Virtuális processzorok definiálásával a programozó az erőforrások elosztására is megkötéseket tehet.

A nyelv elfedi a programozó elöl az architektúra **kommunikációs** módszereit és egy egységes felületet biztosít. Ebben osztott memóriát – szálak közös memóriaterülete, globális mutatók –, és távoli eljáráshívásokat is lehet használni folyamatok közötti kommunikációra. Mindkét kommunikációs mód aszimetrikus, azaz a küldő, illetve kérő írja elő a címzettet. A fogadó oldal nem tehet megkötéseket arra, hogy kitől fogadja el a kommunikációs kérelmeket. Ezek a módszerek emellett szinkron, blokkoló kommunikációval járnak. A kommunikáció ilyen formája azért lehet mégis gyors, mert minden egyes kérelem kiszolgálásához egy külön végrehajtási szál indulhat el.

A *CC++* nyelv két **szinkronizációs** eszközt ad a programozónak. Az *atomic* módosítóval definiált eljárásokra a monitorokhoz hasonló atomi végrehajtás lesz jellemző. Ez az atomicitás nem az egész osztott objektumra, hanem csak az adott tagfüggvényre vonatkozik!

Bonyolultabb útkifejezések megvalósítására, illetve más szinkronizációs problémák megoldására csak a *sync* típusmódosító használható, amely működésében egy egyszer használatos szemaforhoz hasonlítható. Ez az elem eléggé sajátos programtervezési módszereket igényel, bár a nyelv alkotói szerint minden probléma megoldható ezzel is.

A CC++ nyelv tehát egy nagyon kellemes eszköz folyamat párhuzamosságot tartalmazó problémák megoldására, bár hatékony felhasználása – az objektumorientáltság és a párhuzamosság kombinálásából származó gondok miatt is – új programtervezési módszereket igényel.

2.5. HPF

High Performance Fortran az adatpárhuzamosságot támogató kiterjesztése a Fortran 77 nyelvnek.

2.5.1. Bevezető az eszközről

Az adatpárhuzamosságot támogató nyelvek a vektor-műveleteket támogató nagyszámítógépek fordítóiból fejlődtek ki. Ezekben a nyelvekben a program szekvenciális kódjában a hardware által támogatott vektor műveletek is szerepelhettek, ami nagy mértékben gyorsította az egyszerű szekvenciális kód futását. Az adat párhuzamosságot támogató nyelvek tehát szekvenciális program írására alkalmas nyelvek, amelyek sok beépített elemet tartalmaznak nagy mennyiségű adat párhuzamos kezelésére. A nagy méretű tömbök, mátrixok elosztását a processzorok memóriájában, és a velük végzett műveletek folyamatának szervezését automatikusan elvégzik.

Leggyakrabban SIMD számítógépeken találhatóak meg, de egyre több MIMD architektúrájú gépre is készítenek ilyen fordítókat.

Áttekintés a nyelvi változatokról

Fortran 77 óta a Fortran nyelvnek számtalan kiterjesztése született. Az ilyen kiterjesztési törekvések összefogására 1990-ben definiálták a nyelv egy új változatát, melyet Fortran 90-nek neveztek el. Ez a nyelv a 77-es változatot mutatókkal, felhasználó által definiált adattípusokkal, modulokkal, rekurzív alprogramok írásának lehetőségével, dinamikus tárkezeléssel, tömb műveletekkel, sok új beépített operátorral és még sok más aprósággal terjesztette ki. Ez a nyelv az új lehetőségek alapján bármelyik modern programozási nyelvvel felveszi a versenyt.

A Fortran 90 magában sok olyan lehetőséget tartalmaz amelyek lehetővé teszik adat párhuzamos programok írását (pl. tömb műveletek), de vannak hiányosságai, amik miatt nem biztos, hogy optimális kódot generál a fordító minden számítógépen.

Ezen hiányosságok kiküszöbölésére a Fortran 90 nyelvet többen is kiterjesztették fordítási direktívákkal és új nyelvi elemekkel. Ilyenek a CRAFT, a CM-Fortran (Connection Machine Fortran) és ezen törekvések egybefogására egy fórum által szabványosított (HPFF – High Performance Fortran Forum) *HPF* nyelv.

Az adatpárhuzamosságot más programozási nyelvekben is támogatják. A HPF-hez hasonlóan több hagyományos nyelvet is kiterjesztettek, és így született a C*, a *LISP és a MODULA-2* nyelv. Törekvések vannak a C++ kiterjesztésére is: pC++ és HPC++.

Ezek a kiterjesztések általában ugyanezeket az elveket követik – sőt sok változatuk egyenesen az HPF-ből veszi a terminológiáját (pC++, HPC++ és MODULA-2*) –, ezért a HPF nyelv megismerése ezen nyelvek megtanulásában is nagy segítség lehet.

Egy bevezető példa

Az adat párhuzamosságot támogató nyelvek működésének megértéséhez legjobb egy egyszerű példát megtekinteni, amely jól szemlélteti a különbséget a hagyományos nyelvekhez képest:

F77 **F90**

```
REAL a(64), b(64)                    REAL a(64), b(64)
DO i = 1, 64
      a(i) = 2.0                     a = 2.0
      b(i) = b(i) * a(i)                   b = b * a
END DO
```

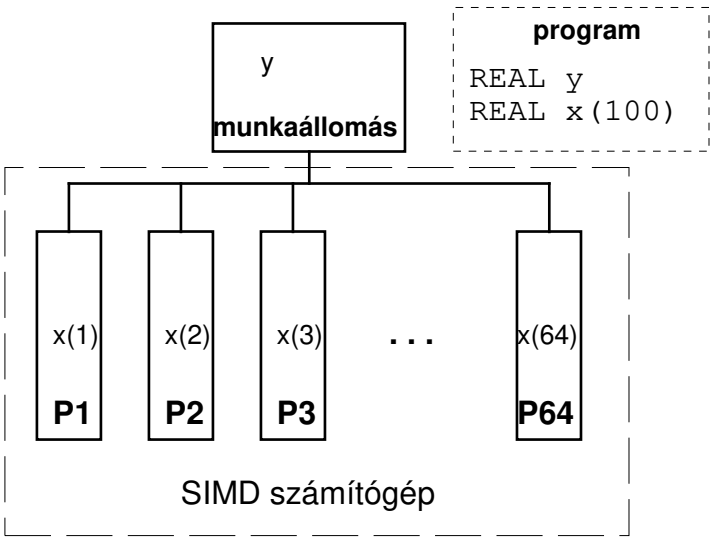
F90 programban a tömböt szét lehet osztani a processzorok között – ha van elég, akkor akár minden tömbelemet más-más processzor kezelhet –, és a rajtuk végzett műveleteket párhuzamosan lehet elvégezni. Ennek az implicit párhuzamosság kifejezésére az F77 program nem alkalmas, mert a fordító programnak ki kéne találni, hogy mit a ciklus értelme.

Az F90-ben a tömbökkel ugyanúgy lehet műveleteket végezni, mint egyszerű változókkal, és ekkor az adott művelet minden tömbelemre alkalmazva lesz.

Programok szerkezete

Egy adatpárhuzamosságot támogató nyelven írt program általában szekvenciálisan végrehajtható és párhuzamosítható részekre bontható. Ezen részeket általában más-más gépek hajtják végre.

A szekvenciális részeket egy nagy teljesítményű munkaállomás, a párhuzamosított részeket egyszerűbb – de nagyon sok (CM 200-nál akár 65536 darab) – kis processzor hajtja végre. A sok kis processzor koordinálását is a nagy munkaállomás végzi el.



13. Ábra: Adatpárhuzamos program elosztása

2.5.2. Felhasználói felület

A felhasználói felület leírásában a HPF és a Fortran 90 párhuzamossággal kapcsolatos elemei találhatóak meg. Az itt található elemeket a *HPF* fordítóprogramok mind támogatják. Ha egy elem Fortran 90-ben is megvan, az külön ki lesz emelve.

A *HPF* sok megoldását fordítási direktívákkal el lehet rejteni, így a *HPF* program egy helyes Fortran 90 program is lehet. Ez a megoldás megjegyzés sorokba írt utasításokkal érhető el:

```
!HPF$ PROCESSORS pr(3)
            INTEGER a(8)
            ...
```

Ezeket a speciális (!HPF$) megjegyzéseket a HPF fordító nem hagyja ki.

Tömb műveletek

A Fortran 90 nyelv alapvető nyelvi újdonsága – amely a párhuzamosság bevezetését lehetővé tette – a tömbökkel végzett műveletek egyszerűbb írásmódja.

Az F77-ben megszokott skalár műveleteket a Fortran 90-ben már tömbökre is lehet alkalmazni. Az így leírt műveleteket a fordító a kifejezésben szereplő tömbök megfelelő elemeire fogja alkalmazni. A műveletekre ezért az a megkötés van, hogy a benne szereplő tömbök dimenzióinak és méreteinek meg kell egyezniük.

A következő példában a *B* tömb megfelelő elemeinek és *c* változónak az összege kerül bele *A* tömb megfelelő elemeibe. A beépített függvények – mint a gyökvonás – is használhatóak.

```
integer A(10,10), B(10,10), c
A = B + c
A = A + 1.0
A = SQRT(A)
```

A tömbműveleteknél a kifejezésekbe persze nem csak egész tömbök, hanem azok részei is behelyettesíthetőek. Résztömböt a megfelelő koordináta helyére írt számhármassal lehet megadni: *alsó-határ:felső-határ:lépésköz.* A tömb lépésköznyi távolságra lévő elemei az alsó határtól a felső határig lépkedve lesznek a résztömb elemei. Ilyen specifikációt hiányosan és több koordináta helyén is meg lehet adni.

```
integer A(4,8), B(4,8)
A(2, :)                         ! A masodik sora
A(2, 2:7)                       ! A masodik soranak kozepe
A(2, 1:8:2)              ! A masodik sorabol minden
                                !   masodik elem

! B tomb eltolasa, es hozzaadasa A-hoz
A(:, 1:7) = A(:, 1:7) + B(:, 2:8)
```

Tömb operátorok

A Fortran 90 nyelv egy argumentumú beépített műveleteknél – mint a gyökvonásnál – megengedi azok alkalmazását tömbökre. Ebben az esetben az operátor a tömb elemekre külön-külön lesz alkalmazva. Ilyen pl.: SQRT, SIN, TAN, ABS ...

Vannak persze olyan műveletek is, amelyek egész tömböket is érintenek. Ezek közül a következőek vannak definiálva:

- **SUM(A)** A elemeinek az összege
- **PRODUCT(A)** A elemeinek szorzata
- **MAXVAL(A)** A legkisebb eleme
- **MINVAL(A)** A legnagyobb eleme
- **MAXLOC(A)** A legkisebb elemének az indexei
- **MINLOC(A)** A legnagyobb elemének az indexei
- **MATMUL(A,B)** A és B mátrixok szorzata

- **DOTPRODUCT(A,B)** A és B vektorok szorzata
- **TRANSPOSE(A)** A transzponáltja
- CSIHIFT(A, DIM, SHIFT) A mátrix eltoltja

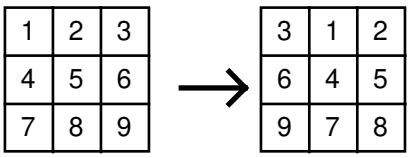
Az összeg, szorzat, minimum és maximum műveletek az egész tömbre értendők, tekintet nélkül annak alakjára, dimenzióinak számára.

A maximum és minimum helyek keresésekor az eredményt az argumentum tömb dimenziószámának megfelelő nagyságú tömbbe fogja beletenni az operátor:

```
integer     A(10,10), B(20, 10), P(2), s
...
P = MAXLOC(A)
s = SUM(A)
B(11:20, :) = MATMUL(A, B(1:10, :)
```

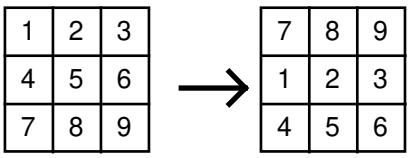
Egyes műveleteknél szükség lehet a szomszédos mátrix elemekre. Ilyen esetekben használható a *CSHIFT* művelet, mellyel a mátrixot valamelyik dimenzió irányában körkörösen el lehet forgatni:

```
result = CSHIFT(X, 2, -1)     ! masodik dimenzioban egy
                                    ! elemmel visszaforgat
```



14. Ábra: CSHIFT(X, 2, -1)

```
result = CSHIFT(X, -1)        ! elso dimenzional eleg
                                    ! egy parameter
```



15. Ábra: CSHIFT(X, -1)

A következő átlagolási feladat az eltolást használva sokkal egyszerűbben leírható:

```
real X(0:99), B(0:99)
do i = 0,99
  B(i) = ( X(mod(i+99,100) + X(mod(i+1,100)) )/2
enddo
```

F90-ben:

```
real X(100), B(100), L(100), R(100)
L = CSHIFT(X,+1)
R = CSHIFT(X,-1)
B = ( L + R )/2
```

,vagy még egyszerűbben:

```
real X(100), B(100)
B = ( CSHIFT(X,+1) + CSHIFT(X,-1) )/2
```

A *HPF*, illetve más párhuzamos Fortran kiterjesztések ezeken kívül még operátorok gazdag választékát nyújtják, például asszociatív műveletek részeredményeit is kiszámoló eszközöket.

Adatok elosztása

Az előző részben definiált nyelvi eszközök lehetővé teszik, hogy a fordítóprogram felismerje a párhuzamosítható részeket, de még semmit nem mondanak az adatok eloszlásáról.

Igen költséges módszerekkel a program adatfolyamának eltérképezésével esetleg egy Fortran 90 fordítóprogram is képes lehet optimálisan elosztani nagy tömböket sok processzoron, viszont a *HPF* erre jobb megoldást kínál. A programozó fordítási direktívákkal tájékoztathatja a fordítót, hogy szerinte hogyan kéne az adatokat elosztani, hogy minél kevesebb legyen a kommunikáció a számítások végzése során.

16. Ábra: Tömbök elosztása HPF-ben

Az adatok elosztására szolgáló elemek csak direktívák, azaz a fordító figyelmen kívül is hagyhatja azokat, ha úgy ítéli meg, hogy csökkentik a hatásfokot. Ezt nyugodtan megteheti, hiszen ezek csak a tömbök elosztását a folyamatok szervezését befolyásolhatják, de a program algoritmusát, működését nem. Elhagyásuk, vagy megtartásuk esetén ugyanazt a végeredményt kell produkálnia a kész programnak.

Az adatok elosztását három szinten lehet leírni:

- **Virtuális processzor:** A programozó virtuális processzor mátrixokat definiálhat az adott feladat igényeinek megfelelően. Ezen virtuális processzorok fizikai processzorokhoz rendelése implementációfüggő.

```
!HPF$ PROCESSORS procik(10,3)
!HPF$ PROCESSORS masleosztas(30)
```

Mindkét esetben 30 virtuális processzor használható.

- **Elosztás:** Tömböket bizonyos irányelvek alapján el lehet osztani a processzorok között. Minden dimenzióra elő lehet írni, hogy hogyan illeszkedjenek az elemek a fenti módon definiált virtuális processzorokhoz.
  - ***** Nincs elosztás, azaz a tömb elemei ebben a dimenzióban egy virtuális processzorra kerüljenek rá.
  - **BLOCK** A tömb elemeit ebben a dimenzióban blokkokra bontva kell elosztani, azaz folytonos darabokra kell szétszedni, és az így képződött résztömböket kell egy-egy virtuális processzorhoz hozzárendelni.
  - **CYCLIC** A tömb elemeit ciklikusan kell elosztani ebben a dimenzióban. Ez azt jelenti, hogy a tömb elemei egyenként hozzárendelődnek a szóba jöhető virtuális

processzorokhoz. Ha a processzorok elfogynak, akkor újra az elsőhöz rendelődik hozzá a tömb következő eleme.

```
!HPF$            PROCESSORS p(4,8)

                 real D(1024), E(1024)

! D elosztasa sorokbol allo blokkokkal, a sorok folytonosak
!HPF$            DISTRIBUTE D(BLOCK, *)

! E elosztasa p virtualis processzormatrixon, az elosztas
! oszlopokbol allo blokkokra bontassal, es ezen oszlopok
! cirkularis szetosztasav tortenik meg
!HPF$            DISTRIBUTE E(CYCLIC, BLOCK) ONTO p
```

- **Igazítás:** Ha egy műveletben több tömb is résztvesz, akkor ezeket egymáshoz is lehet igazítani. Az igazítás eredményeképpen az egy kifejezésben szereplő tömbök által elfoglalt processzorok között sokkal kevesebb lesz a kommunikáció, mint a processzorokon belül, így sokkal hatékonyabb lesz a program.

Igazítás legegyszerűbb formája, ha egy tömböt egy másikhoz van igazítva:

```
      real A(10), B(10)
!HPF$ ALIGN B WITH A
```

Ezzel a direktívával bonyolultabb kapcsolatokat is meg lehet határozni:

```
ALIGN A(I) WITH B(I+2)  ! A(1)-et B(3)-hoz igazitja
ALIGN A(I) WITH B(2*I)  ! A(1)-et B(2)-hoz, A(2)-t B(4)-hez...
ALIGN A(I/2) WITH B(I)  ! A(1:2)-t B(1)-hez
ALIGN D(I,J) WITH E(J,I)      ! D-t E transzponaltjahoz
ALIGN D(:,*) WITH A(:)  ! D sorait A elemeihez
```

Adatok olvasása általában sokkal kevesebb időt igényel, mint az adatok írása – egyes fura architektúráktól eltekintve, pl. Cray T3D – ezért érdemes lehet egy-egy gyakran használt vektort minden processzoron lemásolni, ha a műveletek csak olvasnak belőle:

```
!HPF$            ALIGN A(*) WITH E(*, J)
```

Az igazításhoz nem feltétlenül kell igazi tömböket felhasználni, lehet mintákat is készíteni.

Az *HPF* által definiált adatelosztási módszert hatékonysága, jól definiáltsága és viszonylag egyszerű fogalmai miatt egyre több adatpárhuzamosságot támogató nyelv is átvette, pl. pC++ és HPC++.

Explicit párhuzamosság

Az eddig ismertetett eszközök csak matematikai formulákkal a tömbök egészére megfogalmazható műveletekkel megadható algoritmusok párhuzamosítására voltak alkalmasak. Néhány esetben szükség van arra, hogy tömb elemekkel egyenként foglalkozzon az algoritmus. Ekkor az egyes elemekre esetleg más-más műveleteket kell elvégezni, amely más megfogalmazást igényel.

WHERE

Tömbök minden egyes elemére végrehajtott műveleteknél használható ez a konstrukció:

```
      real A(1024), B(1024)
```

```
...
WHERE (A .NE. 0)
  A = 1 / A
  B = A * 10
ELSE WHERE
  B = 1
END WHERE
```

Minden A-beli elemre kiértékelődik a feltétel, és ennek eredményétől függően végrehajtódik a két utasításblokk közül az egyik.

FORALL

Párhuzamos ciklus formája a nyelvben a *FORALL* kulcsszóval írható le. A kulcsszó után index értelmezési tartományát lehet meghatározni érték-hármasokkal, illetve egy opcionális feltétellel:

```
! tombok inicializalasa
FORALL(I = 1:1024)  A(I) = I
FORALL(I=1:32, J=1:32:2) X(I,J) = I+J

! felso haromszog matrix nullazasa
FORALL(I=1:32, J=1:32, I<J) X(I,J) = 0.0
```

A műveletekben nem történhet hivatkozás más lépésekben kiszámolt értékekre, hiszen ezek végrehajtási sorrendje definiálatlan. Ezeket az utasításokat a fordító megvizsgálhatja, de helyességük eldöntése néhány esetben csak futásidőben történhet meg:

```
FORALL(I=1:32) A(permut(I)) = B(I)
```

INDEPENDENT

Hagyományos ciklusok is párhuzamosíthatóak az *INDEPENDENT* direktívával:

```
!HPF$ INDEPENDENT
            DO I= 1, n
                  A(I) = I
            ENDDO
```

Az így megírt ciklusból a fordító párhuzamosan is végrehajtható ciklusmagokat is generálhat. Ebben az esetben minden ciklusmagban meg lesz a ciklusváltozó egy másolata. Ezekben az utasításokban átmeneti – de globálisan deklarált – változókra is lehet hivatkozni, amely veszélyes lehet. Az ilyen gondok elkerülésére egyes változókból minden ciklusmag számára készülhet egy-egy lokális másolat:

```
!HPF$ INDEPENDENT, NEW(temp)
            DO I=1, n
                  temp = A(I+1) + A(I-1)
                  A(I) = temp/2
            ENDDO
```

PURE

A párhuzamosan végrehajtható ciklusokban nagyon csúnya dolgokat lehetne csinálni, ha bármilyen függvényt meg lehetne belőlük hívni. Ilyen helyzetek miatt ezekben az esetekben csak *PURE* direktívával definiált függvények hívhatóak meg.

A *PURE* direktíva egyike a Fortran 90-ben bevezetett függvény attribútumait leíró direktíváknak. Azt határozza meg, hogy a függvényben nem lehet globális változóra hivatkozni, mutatót használni, és a paraméterek értékét nem lehet megváltoztatni. Az így definiált függvény helyességét a fordító ellenőrzi.

A *PURE* tulajdonsággal rendelkező függvényeknek tehát semmiféle mellékhatása nem lehet, így nyugodtan lehet őket használni párhuzamos ciklusokban is.

Egyebek

Alprogramok írásánál a Fortran 90 sok új lehetőséget biztosít, amelyek szekvenciális programok írásánál is hasznosak lehetnek.

F90-ben alprogram elején is lehet lokális tömböket deklarálni, melyek méretét akár a paraméterlista is befolyásolhatja:

```
subroutine valami(a, b, c, m, n)
real dimension(n,n), intent(in) :: A      ! csak olvashato
real dimension(m,m) :: b,c
intent(out) :: b                          ! kimenetre
intent(inout) :: c
real dimension(n,m) :: temp               ! lokalis tomb
...
```

Ezek az információk az alprogram törzsétől külön is definiálhatóak, amik a hívások típusellenőrzését teszik lehetővé.

Lokális tömbök definiálásakor hasznosak lehetnek az alábbi beépített függvények (pl.: A(2, -1:3, 10) :

- **SHAPE(A)** A tömb alakját írja le
- **LBOUND(A)** A tömb alsó indexeit adja vissza (pl.: [1, -1, 1])
- **UBOUND(A)** A tömb felső indexeit adja vissza (pl.: [2, 3, 10])
- **SIZE(A)** A tömb méreteit adja meg (pl.: [2, 5, 10])

Alprogramok hívásakor a főprogramban lévő tömb elosztás nem biztos, hogy megfelel az alprogram algoritmusának. Ilyen esetekben az alprogram paramétereiben szereplő tömböket újra szét kell osztani a **REDISTRIBUTE** és a **REALIGN** direktívák használatával.

Ezek a megoldások persze nagyon költségesek lehetnek, ezért ha az algoritmus egy lokális tömbjében egyszerűen egy paraméterét szeretné utánozni, akkor használhatja a **INHERIT** direktívát az elosztás leírására.

A programok írásánál persze az is jól jöhet, ha a fordításkor már ismert a célarchitektúra felépítése. Ezekről a helyi lehetőségekről a **PROCESSORS_SHAPE** és a **NUMBER_OF_PROCESSORS** függvények segítségével lehet érdeklődni.

2.5.3. Példák

Mandelbrot halmaz számítása

HPF nyelvű megoldásában a feladatnak lényegében csak a probléma algoritmikus megfogalmazásával kell a programozónak törődni, a párhuzamosítás feladatát a fordító automatikusan elvégzi.

```
! Mandelbrot halmaz szamitasa (-2,-2)-(2,2)
      PROGRAM mandel
      IMPLICIT NONE

      INTEGER, PARAMETER :: N=512, RESOLUTION=255
```

```
      INTEGER :: i, j, start, stop

      INTEGER, DIMENSION(N,N) :: colour
      REAL, DIMENSION(N,N)      :: zr, zi, cr, ci, zrs, zis
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: zr, zi, cr, ci, zrs, zis
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: colour

! A tombok inicializalasa a FORALL utasitas hasznalataval
!
      FORALL (i=1:n, j=1:n) zr(i,j) = REAL(i-1)/REAL(n-1)
      FORALL (i=1:n, j=1:n) zi(i,j) = REAL(j-1)/REAL(n-1)

! Segedtombok beallitasa
!
      cr = zr
      ci = zi

      zrs = zr*zr
      zis = zi*zi

      colour = 0

! A fo szamitasi muvelet
!
      DO i = 0, RESOLUTION

        WHERE ((zrs + zis) .le. 4.0)

          zrs = zr*zr
          zis = zi*zi

          zi = 2.0*zr*zi + ci
          zr = zrs - zis + cr

          colour = i

        END WHERE

      END DO

! A kep kiirasa PGM formatumban
!
     OPEN(UNIT=10, FILE='mandel.pgm')
     WRITE(10, FMT='(''P2'',/,i3,2x,i3,/,i3)') N, N, RESOLUTION
     WRITE(10,*) colour
     CLOSE(UNIT=10)

     END
```

2.5.4. Összefoglalás

A *High Performance Fortran* nyelvet a feladatok egy szűkebb körének megoldására lehet használni, mint az eddig ismertetett eszközöket. Főleg matematikai számítások, kiegyensúlyozott adatpárhuzamos feladatok megoldására alkalmas, viszont ezek leírására nagyon jó eszközöket biztosít.

Az HPF nyelven megírt program szerkezete egy hagyományos program szerkezetének felel meg, ezért a hagyományos nyelvekben használt programtervezési, és helyességbizonyítási módszerek használhatóak benne.

A nyelv nem alkalmas – vagy sok nehézséget okoz – kiegyensúlyozatlan problémák kezelésére, MIMD stílusú párhuzamos programok írásárasa, tehát osztott operációs rendszer, vagy kliens/szerver modellre épülő adatbázis-kezelő nem írható benne.

A nyelv kifejezetten az **adat-árhuzamosság** támogatására készült, a szekvenciális kódból a fordítóprogram készít párhuzamos szerkezeteket. Az adatokra csak azok egészét érintő műveletek fogalmazhatóak meg, teljesen független utasítássorozatok nem.

Ezen megkötés alól a párhuzamosított ciklusokból függvények hívása kibúvót jelenthetne, de ez a megkötések miatt nem jelent lényegi különbséget (a *PURE* attribútummal rendelkező függvényekre vonatkozó megkötések miatt ugyanazt a feladatot lehet elvégezni velük, mintha a függvények kódját behelyettesítenénk a ciklusmagba).

A nyelv nem definiálja, hogy a program pontosan milyen programot eredményezzen egy adott párhuzamos architektúrán. Ezt a részt minden *HPF* fordító implementáció függő módon írja le.

Ebben az esetben nem lehet beszélni **folyamat modellekről** sem, hiszen a program logikailag egyetlen folyamat a programozó számára, és az implementációtól függően ez egyes pontokon több processzoron is végrehajtódó részeket tartalmazhat.

A párhuzamos programozást támogató programozási eszközök további vizsgálati szempontjai között szereplő **kommunikációs** és **szinkronizációs** lehetőségeket is elfedi a nyelv, ezért ezekről elég keveset lehet mondani:

Kommunikáció szempontjából egy osztott memóriával rendelkező gépet modellez. Az összes párhuzamosítható szerkezet vége egy implicit szinkronizációs pontot jelent a programban létrejött párhuzamosan végrehajtott folyamatszálak számára.

A HPF és Fortran 90 nyelvek **jellemzői**:

- **helyesség:** a HPF és Fortran 90 nyelven írt programok szerkezete szekvenciális programoknak felel meg, ezért az ott bevezetett programhelyesség vizsgáló módszerek alkalmazhatóak rájuk.
- **megbízhatóság:** a nyelvben kivételes helyzetek kezelésére nincsenek beépített eszközök, mindig az adott implementációtól függ ezek megoldása.
- **hatékonyság:** a szűk feladatosztályra ezek a nyelvek nagyon erős absztrakciós lehetőségeket biztosítanak, ezzel sok lehetőséget adnak jól optimalizált kódok generálására. A *HPF* nyelvben bevezetett fordítási direktívák az automatikus optimalizálási lehetőségeket kibővítik, a programozó számára lehetőséget adnak, hogy további ismeretekkel elősegítse – az adott architektúrán – még hatékonyabb kód készítését. Egyszerűbb esetekben a fordító maga is képes lehet jó kód generálására, de igazán hatékony program írása érdekében a programozónak meg kell értenie az adott párhuzamos architektúrát, és meg kell ismernie a fordító kódgenerálási alapelveit is.
- **hordozhatóság:** a nyelv magas szintű absztrakciós lehetőségeket biztosít programok írására, amelyből bárhol lehet jó programot készíteni.
- **biztonság:** nem ilyen problémák kezelésére készült a nyelv
- tesztelés, optimalizálás támogatása: implementációtól függ a támogatás
- **felület a fejlesztésre:** implementációfüggő a támogatás, de a hagyományos programfejlesztési eszközök/környezetek továbbra is használhatóak hozzá.

Tehát a HPF nyelv matematikai problémák kezelésére alkalmas, de ezen a téren kezelhetőségben messze megelőzi az összes többi eszközt.

3. Függelék

3.1. Irodalomjegyzék

- PVM User's Guide And Reference Manual; Oak Ridge National Laboratory, September, 1994; ORNL/TM-12187
- The World's Fastest Computers; 44-64 pages, BYTE, January, 1996
- MPI: A Message Passing Interface Standard; Message Passing Interface Forum, June 12, 1995
- Introduction to Parallel Computing; Edinburgh Parallel Computer Centre; The University of Edinburgh
- Writing Message-Passing Parallel Programs with MPI; Edinburgh Parallel Computing Centre, The University of Edinburgh
- Az Ada programozási nyelv; Kozics Sándor, Budapest, 1992; ELTE TTK
- Magasszintű procedurális nyelvek; Kozics Sándor, Az ALGOL 60, a FORTRAN, a COBOL és a PL/I programozási nyelvek, Budapest, 1992; ELTE TTK
- A Communication Backend for Parallel Language Compilers; James M. Stichnoth and Thomas Gross; Languages and Compilers for Parallel Computing, p224-238; Lecture Notes in Computer Science 1033
- Virtual Topologies; J. Philbin, S. Jagannathan, R, Mirani; Languages and Compilers for Parallel Computing, p450-464; Lecture Notes in Computer Science 1033
- Principles of Concurrent Programming; M. Ben-Ari
- User Guide for a Portable Parallel C++ Programming System, pC++; Dennis Gannon, Shelby X. Yang, Peter Beckman; Indiana University, Iniana, U.S.A, 1994
- CC++ Tutorial; Caltech University, California, U.S.A, 1994
- MPI Document; http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html
- High Performance Fortran Report 1.1
- Introduction to Programming the Connection Machine, Harvey Richardson, Thinking Machines Corporation