P–GRADE programok JGrid rendszerben futtatható Java kódjának generálása

Lövei László

2003. június 30.

1. Bevezető

Ez a dokumentum a P–GRADE[1] grafikus fejlesztőrendszerrel készített elosztott programok JGrid[2] rendszerben futtatható Java nyelvű kódjának az automatikus elkészítéséről szól.

A JGrid egy Jini[3] alapú Grid rendszer, amelyben a folyamatok Java objektumokként jelennek meg. Egy P–GRADE program párhuzamosan futó processzekből áll, minden processzt egy objektum reprezentál a gridben. Ezek az objektumok a JGrid rendszer felé egy metódust nyújtanak interfészként, amely a processz főprogramját tartalmazza. A JGrid számítási szolgáltatás ezt a főprogramot futtatja le egy önálló thread-ben.

A P–GRADE program leírja a processzek kódjait, valamint a processz példányok kommunikációs kapcsolatait. Minden processz kódot egy külön Java osztály ír le, ezekből jönnek létre a processz objektumok a program felépítésének megfelelően. A kommunikációs template-ek kivételével általában minden kódhoz csak egy példány tartozik.

A processzeket csoportokba lehet szervezni, és ezeket önálló kommunikációs végpontként lehet kezelni. A csoportokhoz nem tartozik programkód, ezért ezeket olyan osztályok írják le, amelyek csak a kommunikációs struktúra inicializálásában vesznek részt, a példányaik nem kerülnek ki a gridbe.

A processzekhez generálandó kódot két rétegre érdemes bontani. A kód egy része a programtól függetlenül leírható, ezért ezt érdemes egy belső könyvtár formájában elkészíteni. A kód programtól függő részeit ennek a könyvtárnak a szolgáltatásaira lehet építeni.

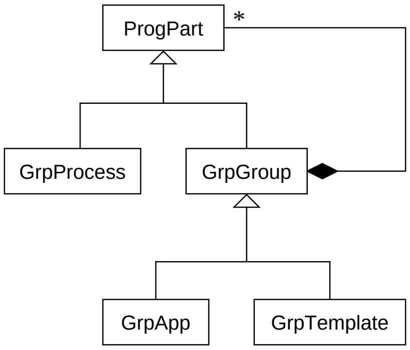
A processzek kommunikációs struktúrájának felépítését egy önálló vezérlő folyamat végzi a program indulása során. Ezt a menedzser folyamatot is egy külön osztály írja le, ennek az egyetlen példánya vezéreli a program indítását, azaz a processzeket megvalósító objektumok létrehozását és elindítását, valamint a kölcsönös kommunikációhoz szükséges inicializációk végrehajtását. Ezt a kódot is két rétegre bontva érdemes elkészíteni a processzekhez hasonlóan.

A támogató könyvtár legfontosabb feladata a kiszolgáló rendszer interfészeinek elrejtése egy olyan felület mögé, amelyhez könnyű kódot generálni. Ez lehetővé teszi például a kommunikációs infrastruktúra kicserélését a későbbiekben. Jelenleg az MPI szabvány[4] Java nyelvű interfészének[5] egy részét megvalósító kommunikációs felület áll rendelkezésre. További feladata a GRAPNEL könyvtár P–GRADE programok felé nyújtott szolgáltatásainak megőrzése, ennek nyomait a `grp` kezdetű függvénynevek jelzik.

2. A generált kódot támogató könyvtár

Mivel a processzek kódját egy adott interfészt megvalósító osztálynak kell tartalmaznia, érdemes a támogató könyvtárat osztályokként elkészíteni, amelyek biztosítják ezt az interfészt. A kódgenerálás során ezekből az osztályokból leszármaztatva kell kell elkészíteni a processzek kódját.

A támogató osztályok diagrammja az 1. ábrán látható. A GrpGroup osztály leszármazottjai írják le a processz csoportokat; maga az osztály a csoportban található processzeket képes elindítani a gridben. A GrpApp osztály ennek egy speciális esete; ez nem egy valódi csoportot ír le, hanem az alkalmazás csoporthierarchiájának legkülső szintjén levő elemeket tartalmazza. Ez az osztály tartalmazza a vezérlő kódot. A GrpTemplate osztály leszármazottjai a template-ek speciális struktúrájának felépítését segítik. A GrpProcess osztály leszármazottjai írják le a processzek kódját.



1. ábra. A támogató könyvtár interfészének osztálydiagrammja

2.1. A kommunikációs struktúra támogatása

A processzek és csoportok közös őseként szolgáló ProgPart osztály egy közös funkciót valósít meg: a portok kezelését. A generált kód számára elég a processzek és csoportok portjaihoz egy sorszámot rendelni, valamint a csoportok inicializálása során megadni a sorszámok segítségével a portok közti kapcsolatokat, a támogató osztályok a további részleteket elfedik. A ProgPart osztály nem nyújt közvetlenül interfészt a generált kód felé.

A csoport portok kezelése speciális olyan szempontból, hogy a csoporton kívülről egy portként jelennek meg, a csoporton belül viszont minden processznek el kell tudnia érni úgy, mintha nem csoport port lenne. Ennek az a megoldása, hogy a csoport portokhoz a processzek szintjén is tartozik egy-egy port, amelyet a csoport inicializálása során kell összekötni a valódi csoport porttal.

2.2. A processz csoportok támogatása

A csoportokat támogató GrpGroup osztály a csoport processzeinek elindításához és a kommunikációs struktúrájuk inicializálásához nyújt segítséget. A felhasználás során minden processzhez rendelni kell egy sorszámot (N processzből álló csoportban 0 és N-1 között), az osztály műveletei pedig ezen sorszámokhoz rendelnek processz objektumokat, amelyek a program indulása során kikerülnek a gridbe, valamint inicializálják a kommunikációs struktúrájukat.

A csoport struktúrájának megadásához az interfészben leírt metódusokat lehet használni, a kódot a támogató osztály init nevű metódusát felülírva kell elkészíteni.

A GrpGroup osztály interfésze.

GrpGroup(GrpGroup parent, int elNum, int pgID, int portNum)

> Az osztály konstruktorának a parent paraméterében meg kell adni az őt tartalmazó csoport objektumot, az elNum paraméterében azt, hogy hány eleme lesz a csoportnak, a pgID paraméterben a belső P–GRADE azonosítót, a portNum paraméterben pedig a csoport portjainak a számát.

void createGroup(int ID, GrpGroup group)

> Az ID sorszámhoz hozzárendeli a group csoport objektumot.

void createProcess(int ID, GrpProcess process);

> Az ID sorszámhoz hozzárendeli a process processz objektumot.

void initPort(int port);

> A port azonosítójú csoport portot inicializálja. A csoport minden elemében ugyanazt a sorszámot kell hozzárendelni a csoport porthoz, mint amit a csoportban. Ez lényegében azt jelenti, hogy a csoportokba tartozó processzekben a portok felsorolását a csoport portokkal kell kezdeni, és a processz portokkal folytatni (lásd 3.4. pont).

void connectPorts(int fromID, int fromPort, int toID, int toPort);

> A fromID sorszámú elem fromPort sorszámú portját összeköti a toID sorszámú elem toPort sorszámú portjával. A létrejött kapcsolat mindig kétirányú.

abstract void init();

> Ennek a metódusnak kell tartalmaznia az inicializáló kódot, azaz a processzek létrehozását és a portok összekötését az előző műveletek segítségével.

2.3. A vezérlő folyamat támogatása

A GrpApp osztály tartalmazza a program inicializálásához és futtatásához szükséges műveleteket. Egyrészt ez egy speciális csoport, amelyben a 2.2 pontban ismertetett interfész segítségével lehet leírni az alkalmazás legkülső szintjén álló elemeket. Ezen kívül tartalmaz még egy metódust, amellyel a programot el lehet indítani.

A GrpApp osztály interfésze. Az osztály a GrpGroup műveletein kívül a következőket teszi elérhetővé.

GrpApp(int elNum);

> A konstruktornak az elNum paraméterben meg kell adni az alkalmazás külső szintjén levő elemeinek számát.

void start();

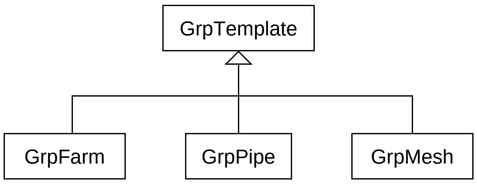
> Elindítja a gridben a teljes programot.

2.4. A template-ek támogatása

A kommunikációs template-ek speciális csoportok, melyeknek előre adott a belső kommunikációs struktúrájuk. Ezeknek a struktúráknak a kialakítását is meg lehet valósítani általánosan, ezt teszik meg a GrpTemplate osztály 2. ábrán látható leszármazottjai.

Az egyes template-ek felépítése más és más, ezért a GrpTemplate osztály nem nyújt önálló interfészt a generált kód felé. Minden speciális template esetén igaz azonban, hogy van egy defineProcess metódusa, amellyel meg lehet adni a template egyes elemeit. A template-be tartozó processzek tetszőlegesek lehetnek, egy megkötéssel: a template által definiált processz portok sorszámait kötött nevű mezőkben kell tárolni (a 3.4. pontban leírt nevekkel azonos formában, a P–GRADE által generált port-azonosítóknak megfelelően).

A template csoport inicializálása során meg kell adni a template processzek osztályait, amelyből a template osztály automatikusan példányosít. Ha vannak, meg kell még adni a csoport-portok összerendeléseit a GrpGroup osztályban megtalálható initPort metódus segítségével. Más inicializálásra nincs szükség.



2. ábra. A template-eket támogató osztályok

2.4.1. A Farm template

A Farm template-nek nincs belső kommunikációs struktúrája, egy processz kódját tartalmazza, amiből példányokat hoz létre. Egy paramétere van, a létrehozandó példányok száma.

A GrpFarm osztály interfésze.

GrpFarm(String name, int pgID, int portNum, int n);

> A konstruktornak a name paraméterben a template példány nevét, az n paraméterben pedig a méretét kell megadni. A pgID és portNum paraméterek megegyeznek a GrpGroup konstruktorának paramétereivel.

defineProcess(Class code);

A code paraméterben megadott osztály (kötelezően a GrpProcess leszármazottja) hozzárendelése a template (egyetlen) processz kódjához.

abstract void initFarm();

Ebben a metódusban kell leírni az inicializáló kódot.

2.4.2. A Pipe template

A Pipe egy egydimenziós template, három processz kódot igényelhet: a pipe első és utolsó elemének, valamint a kettő közötti elemeknek a kódját. Paraméterei a processzek száma, valamint a kommunikációs struktúra ciklikus vagy aciklikus tulajdonsága. Ciklikus struktúra esetén a két szélső processz kódját nem kell megadni.

A processzeket leíró kódokban két port-sorszámot kell definiálni, PORT_0 ill. PORT_1 néven. Aciklikus pipe esetén az első és utolsó elem 0. illetve 1. portja elhagyható.

A GrpPipe osztály interfésze.

GrpPipe(String name, int pgID, int portNum, int n, int type);

A konstruktornak a name paraméterben a template példány nevét, az n paraméterben pedig a méretét kell megadni. A type paraméter értéke a következő konstansok valamelyike lehet:

- TYPE_CYCLIC,
- TYPE_ACYCLIC.

A pgID és portNum paraméterek megegyeznek a GrpGroup konstruktorának paramétereivel.

defineProcess(int pos, Class code);

A code paraméterben megadott osztály (kötelezően a GrpProcess leszármazottja) hozzárendelése a template pos pozíciójában levő processz kódjához. A pos értéke POS_*N* lehet, ahol *N* értéke 0, 1 vagy 2 lehet (a 0 az első, a 2 az utolsó, az 1 a középső processzek kódját jelöli).

abstract void initPipe();

Ebben a metódusban kell leírni az inicializáló kódot.

2.4.3. A Mesh template

A Mesh egy kétdimenziós template, négyzetrács-jellegű kommunikációs topológiával. Ez kilenc különböző processz kódot igényelhet, valamint a méret és a kommunikációs paramétereket mindkét dimenzióhoz. Ciklikus kommunikáció esetén a szélső processzek kódjára itt sincs szükség.

A processzekben négy portra lehet szükség (a processz pozíciójától függően), ezek sorszámait a PORT_*N* nevekkel kell elérhetővé tenni. A portok szerepei: a 0. balra, a 1. jobbra, a 2. felfelé, a 3. lefelé kapcsolódik. Aciklikus esetben a szélső portok elhagyhatóak.

A GrpMesh osztály interfésze.

GrpMesh(String name, int pgID, int portNum, int x, int y, int type);

> A konstruktornak a name paraméterben a template példány nevét, az x és y paraméterekben pedig a vízszintes ill. függőleges méretét kell megadni. A type paraméter értékét a következő konstansokból lehet összeállítani bináris vagy művelettel:
>
> - TYPE_HORIZ_CYCLIC,
> - TYPE_HORIZ_ACYCLIC,
> - TYPE_VERT_CYCLIC,
> - TYPE_VERT_ACYCLIC.

defineProcess(int pos, Class code);

> A code paraméterben megadott osztály (kötelezően a GrpProcess leszármazottja) hozzárendelése a template pos pozíciójában levő processz kódjához. A pos értéke POS_X_Y lehet, ahol X és Y értéke 0 és 2 közötti lehet (a 0 és a 2 a két szélső, az 1 a középső processzek kódját jelöli).

abstract void initMesh();

> Ebben a metódusban kell leírni az inicializáló kódot.

2.5. A processzek támogatása

A generált kódnak a kommunikáció és a monitorozás terén van szüksége támogatásra, a többi processz-szintű művelet könnyen megfeleltethető primitív Java nyelvi eszközöknek. Emellett még a szövegesen megadott programkódok támogatásához szükség van néhány információs függvényre. A processz főprogramját a csoportok felépítéséhez hasonlóan egy absztrakt metódus implementálásával lehet megadni.

A GrpProcess osztály általános metódusai.

GrpProcess(int pgID, int portNum, int[] portInf);

> A konstruktoroknak a pgID paraméterben a processz belső P–GRADE azonosítóját kell megadni, a portNum paraméterben pedig a portjainak a számát. A portokat nem kell külön inicializálni, minden port képes input és output kommunikációra is, a kapcsolatai pedig a csoport inicializáláskor derülnek ki. A portInf paraméter egy információs táblázat, a tartalma a 3.4. pontban van leírva.

abstract void main();

> Ebben a metódusban kell leírni a processz főprogramját. Ez a metódus fog lefutni a számítási szolgáltatáson a gridben.

2.5.1. A kommunikáció támogatása

A kommunikációhoz szükséges a portok egyedi azonosítása. Ezt egy 0 és M-1 közötti sorszámmal kell megtenni, ahol M a processz portjainak száma, a csoportportokat is beleértve. A kommunikáció során nincs megkülönböztetés a csoport és a normál portok között.

A kommunikációt egy csomagpufferen keresztül támogatja az osztály. Küldés esetén a pufferben kell összeállítani a protokollnak megfelelő csomagot, fogadás esetén pedig a pufferből lehet azt kiolvasni.

A kommunikációval kapcsolatos monitorozási tevékenységet teljesen elvégzi az osztály.

A kommunikációt támogató metódusok. A *type* lehetséges értékei: char, int, long, float, double, String.

void initSend(int portID, int proto, boolean sync);

> A portID sorszámú porton a proto protokollt használva a sync paraméter true ill. false értékétől függően szinkron vagy aszinkron adatküldést készít elő. Törli a puffert.

void pk*type*(*type* value);

> Elhelyezi a puffer következő pozíciójára a value értéket.

void send(int index);

> Elküldi az initSend paramétereként megadott porthoz kapcsolódó, index absztrakt indexű processznek a puffer tartalmát. Törli a puffert. Meghívja a megfelelő monitorozó metódust.

void send();

> Elküldi az initSend paramétereként megadott porthoz kapcsolódó összes processznek a puffer tartalmát (multicast üzenetküldés). Törli a puffert. Meghívja a megfelelő monitorozó metódust.

void initRecv(int portID, int proto);

> A portID sorszámú porton proto protokollú adatok fogadását készíti elő.

void recv(int index);

> Az initRecv paramétereként megadott porthoz kapcsolódó index absztrakt indexű processztől fogad egy adatot (szinkron input kommunikáció). A kapott adatot beteszi a pufferba. Meghívja a megfelelő monitorozó metódust.

void altrecv(int[] portIDs, int[] protos);

> Az initRecv metódustól függetlenül hívható metódus. A portIDs tömbben felsorolt portok protos tömbbeli protokolljainak megfelelő üzenetek közül fogad pontosan egyet (szinkron alternatív input kommunikáció). A kapott adatot elhelyezi a pufferben. Meghívja a megfelelő monitorozó metódust.

void getAltPort();

> Annak a portnak a sorszámát adja vissza, amellyel a legutóbbi alternatív input kommunikáció történt.

void getAltProto();

> Annak a protokollnak a számát adja vissza, amelyet a legutóbbi alternatív input kommunikáció használt.

type upk*type*();

> A puffer következő adatát olvassa ki.

2.5.2. A monitorozás támogatása

A monitorozáshoz a GRM instrumentációs API-hoz[6] való hozzáférést kell biztosítani a generált kód számára. A processzeket támogató osztály ezért minden GRM függvényhez tartalmaz egy neki megfelelő metódust:

grmStart(String appName, String procName, int id);

> Ez a metódus inicializálja a monitorozó rendszert, és jelzi a processz indulását. A paraméterként megadott információkat felhasználja a későbbiek során.

grmExit();

> Jelzi a processz futásának végét, és lezárja a monitorozó rendszert.

grmBeginBlock(int type, int id);

> Jelzi egy blokk végrehajtásának a kezdetét.

grmEndBlock(int type, int id);

> Jelzi egy blokk végrehajtásának a végét.

grmSend(int targetID, int commID, int bytes);

> Jelzi egy üzenetküldő utasítás végrehajtását.

grmReceive(int sourceID, int commID, int bytes);

> Jelzi egy üzenetfogadó utasítás végrehajtását.

grmMcast(int[] targIDs, int commID, int bytes);

> Jelzi egy multicast üzenetküldő utasítás végrehajtását.

grmDefine(int eid, String desc);

> Egy saját eseménytípust definiál.

grmEvent(int eid, String event);

> Egy saját típusú eseményt generál.

A kommunikációval kapcsolatos instrumentációs metódusok hívására a generált kódnak nincs szüksége, mert ezt a támogató osztály elvégzi.

2.5.3. A szövegesen megadott kódban használható metódusok

A szöveges kódokban szükség van a template paraméterek elérésére, amelyeket nem célszerű „beledrótozni” a programba. Emellett az alternatív input kommunikáció után le kell tudni kérdezni, hogy melyik partnertől érkezett az üzenet. Ezeket az információkat az alábbi függvények segítségével lehet elérni.

int[] grp_dimensions();

> Egy template-en belüli processz hívhatja, visszaadja egy tömbben a template dimenzióinak méretét.

static int[] grp_get_dimensions(String name);

> Bárhonnan hívható, a name nevű template dimenzióit adja vissza.

int[] grp_myindex();

> Egy template-en belüli processznek adja vissza a saját indexeit a template-en belül. Minden dimenzióhoz tartozik egy index.

int grp_partners(int port);

> A port azonosítójú porthoz kapcsolódó processzek számát adja vissza.

int grp_which_port();

> A legutóbbi alternatív input kommunikációs partner portazonosítóját adja vissza.

int grp_which_partner();

> A legutóbbi alternatív input kommunikációs partner absztrakt indexét adja vissza.

int grp_which_proto();

> A legutóbbi alternatív input kommunikációs partner protokoll azonosítóját adja vissza.

2.5.4. Felhasználói csoportkommunikáció

A csoportos küldés és (nem alternatív) fogadás során lehetőség van arra, hogy a felhasználó definiálja azt, hogy mely partnerekkel pontosan milyen kommunikáció zajlik. Ezt egy szövegesen megadható kóddal írhatja le, amelyben az információs függvények mellett a grp_in és grp_out fogadó és küldő metódusokat használhatóak, valamint egy speciális információs függvény, a grp_partners, amely az éppen aktuális csoportkommunikációban résztvevő partnerek számát adja meg.

Az eredeti GRAPNEL könyvtárban a grp_in és grp_out függvények változó paraméterlistát használtak, mert az általuk fogadott ill. generált üzenetet a protokolltól függő típusú és számú változók tárolják. Java nyelven ezt csak úgy lehet megoldani, hogy minden protokollhoz készül két megfelelő fejlécű kommunikációs metódus. Ezeket a fordítás során lehet generálni, ennek pontos menete, valamint a függvények leírása a 3.5.5. pontban található.

3. A kódgenerálás

A kódgenerálás során létrejön a P–GRADE program Java nyelvű forráskódja, valamint a grafikus elemek és a forráskód kapcsolatait leíró referencia lista. A forráskód a támogató osztályokból leszármaztatott, a vezérlő folyamat részeit (a csoportok és a főprogram osztályai) és a processzeket implementáló osztályokat írja le.

3.1. A vezérlő folyamat generálása

A vezérlő folyamat osztályát a GrpApp osztályból leszármaztatva kell létrehozni, a neve megegyezik az alkalmazás nevével. Két szerepet tölt be: egyrészt a program külső szintű elemeinek a csoportjaként működik, másrészt az egész program indítását végzi. A program indításához létre kell hozni egy példányt az osztályból, és annak meghívni a start metódusát.

3.2. A csoportok kódjának generálása

A csoport osztályát a GrpGroup osztályból kell leszármaztatni, a neve a csoport kódjának neve (ez általában *GroupName*_code alakú). A generált kód négy részből áll: a processzek és portok sorszámainak definíciójából, a processz- és csoport-hozzárendelésekből, a csoport portok inicializálásából és a csatornák felépítéséből.

Minden processzhez rendelni kell egy sorszámot. A sorszámok későbbi hivatkozásának egyszerűsítéséhez minden sorszámot egy néven keresztül lehet elérni. A vezérlő folyamat osztályának első része ezeket sorolja fel osztályszintű deklarációkként, processzek esetén

final static int PROC_*ProcessName* = *N*;

alakban, ahol *ProcessName* a processz példány neve, *N* pedig egy nemnegatív egész szám, a processz sorszáma. Csoportok (és template-ek) esetén a név alakja

final static int GROUP_*GroupName* = *N*;

ahol *GroupName* a csoport példány neve. A sorszámok kiosztásának módjára nincs megkötés.

Ha a csoportnak vannak csoport portjai, akkor azokhoz sorszámot kell rendelni. Ha a csoport egy másik csoportnak az eleme, akkor a külső csoporthoz is kell rendelni belső portot. Ezeket a sorszám-kiosztásokat ugyanúgy kell megtenni, mint ahogy a 3.4. pontban a processzeknél van leírva.

A három inicializáló szakaszt az osztály init metódusában kell elhelyezni. Az első szakasz a sorszámok megfeleltetése az objektumoknak, amely egyszerűen createProcess és createGroup metódushívásokat tartalmaz, minden processz- és csoportpéldányra. A hívás alakja:

createProcess(PROC_*ProcessName*, new *ProcessCode*());

ahol *ProcessName* a processz példány neve, *ProcessCode* pedig a processz osztályának a neve. Csoport esetén:

createGroup(GROUP_*GroupName*, new *GroupCode*());

ahol *GroupName* a csoport példány neve, *GroupCode* pedig a csoport osztály neve.

A második inicializáló szakasz csak akkor jelenik meg, ha vannak csoport portok. Ekkor minden csoport porthoz meg kell hívni az inicializáló metódust. A hívás alakja:

initPort(PORT_*Id*);

ahol *Id* a csoport port azonosítója. Ennél a hívásnál fontos, hogy a csoport portok sorszámai megegyeznek a processzeken belül nekik megfelelő portok sorszámaival.

A harmadik inicializáló szakasz a processzek portjainak összekötését végzi el. Ehhez szükség van a processzek portjainak sorszámára, ezt a processzek kódját tartalmazó osztályban deklarált konstansok segítségével lehet megkapni. A kapcsolatot létrehozó hívás alakja:

connectPorts(*FromName*, *FromCode*.PORT_*FromPort*,
ToName, *ToCode*.PORT_*ToPort*);

ahol *FromName* és *ToName* a processz illetve csoport példányok sorszámának azonosítói, *FromCode* és *ToCode* a processz vagy csoport kódot leíró osztályok nevei, *FromPort* és *ToPort* pedig a portok szimbolikus nevei (ld. 3.4).

3.3. Kommunikációs template-ek kezelése

A template-ek támogató osztáyai speciális csoport osztáyok, amelyek önállóan inicializálják a belső kommunikációs struktúrát. Itt a csoportinicializációs lépésekből csak a csoport portok megfeleltetését kell elvégezni, valamint a processz osztályokat kell megadni. A template-ekben található processzek kódját ugyanúgy kell generálni, mint a nem-template processzekét.

3.3.1. Farm generálása

A processzek kódjának megadása itt elég egyszerű, a következő hívással tehető meg:

defineProcess(Class.forName("*ProcCode*"));

ahol a *ProcCode* a processz kód osztályának neve.

3.3.2. Pipe generálása

A processzek kódjának megadása itt a

defineProcess(POS_*N*, Class.forName("*ProcCode*$_N$"));

hívással történik, ahol *N* értéke aciklikus pipe esetén 0 és 2 között fut, ciklikus pipe esetén pedig csak 1. A *ProcCode*$_N$ az N. pozícióhoz rendelt processz kód osztályának neve.

3.3.3. Mesh generálása

A processzek kódját itt a

defineProcess(POS_*X*_*Y*, Class.forName("*ProcCode*$_{X,Y}$"));

hívással lehet megadni. *X* és *Y* értéke teljesen aciklikus esetben 0 és 2 között fut, vízszintes ciklikus esetben *Y* csak 1 lehet, függőleges ciklikus esetben *X* értéke csak 1 lehet. A *ProcCode*$_{X,Y}$ az (X, Y) pozícióhoz rendelt processz kód osztályának neve.

3.4. A processzek kódjának generálása

A processzek osztályait a GrpProcess osztályból kell leszármaztatni, az osztály neve a processz kód neve lesz (ez általában *ProcName*_code alakú). Egy ilyen osztály két részből áll. Az első rész a processz portjaihoz rendel nevet és a névhez sorszámot, a második rész pedig a főprogramot írja le a main metódusban.

A portok kezelésénél külön figyelmet igényel, hogy a portoknak van egy egész szám azonosítójuk. Ezek az azonosítók azonban nem mindig folytonosan vannak kiosztva, és önmagukban nem is egyediek, csak akkor, ha a csoport portokhoz hozzávesszük a csoport azonosítóját is. Implementációs szempontból ezért célszerűbb egy új sorszámozás bevezetése.

Portok adatai. A sorszámokhoz rendelt név a csoport és a port azonosítójából áll elő. Nem csoport portok esetében ennek a formája PORT_*PortId*, csoport port esetén pedig PORT_*GrpId*_*PortId*, ahol a *PortId* a port azonosítója, a *GrpId* pedig a csoport belső azonosítója (a belső azonosító nem látszik a P–GRADE felületen keresztül, de az implementációhoz megfelel).

A csoport portok használatánál ügyelni kell arra, hogy a sorszám ugyanaz legyen, mint a csoport kódjában definiált. Ugyanez a probléma merül fel akkor is, ha egy csoportnak van egy csoport eleme: a belső csoportnak is ugyanúgy kell portokat kiosztani, mint egy processznek. A sorszámok megegyezését úgy lehet biztosítani, hogy a sorszámoknál először felsoroljuk a legkülső szintű csoport portjait (ugyanabban a sorrendben, ahogy a csoportban, pl. azonosító szerint növekvő sorrendben), ezután a következő csoportét, majd a legbelső csoport után soroljuk fel a processz (vagy csoport) saját portjait.

A 2.5.3 pontban leírt információs függvényeknek az eredeti portazonosítót kell visszaadniuk, ezért kell egy olyan táblázat, amely az egyes portsorszámokhoz megadja az eredeti értékeket. Ezt egy egész számokat tartalmazó vektorral lehet leírni, amelynek annyi eleme van, ahány port tartozik a processzhez, és minden elem a megfelelő sorszámú port P–GRADE azonosítóját tartalmazza.

Speciális szöveges kódrészek. A processz kódjába minden esetben el kell helyezni azokat a szöveges kódokat, amelyeket a felhasználó a P–GRADE-ben fejléc, globális illetve lokális kódként adott meg. Ezeket úgy lehet itt értelmezni, hogy a fejléc kód tartalmát a processz osztályán kívülre helyezzük (ebben célszerűen import utasítások lesznek), a globális kód tartalmát az osztályon belülre, de a metódusokon kívülre (ide például metódusokat lehet elhelyezni), a lokális kód tartalmát pedig a main metóduson belülre (lokális változók deklarációjához).

3.5. A kommunikációs műveletek generálása

A kommunikációs műveletek élesen elkülönítik egymástól az ugyanazon portokon definiált különböző protokollokat, egymástól független csatornaként kezelve őket. A továbbiakban ezért amikor portról, port azonosításáról van szó, mindig a (port, protokoll) pár megadására kell gondolni.

A kommunikációs műveletekhez lehet megadni őrfeltételeket is. A generálás során az őrfeltételként megadott kódot egy if() utasítás fejlécében kell elhelyezni, a törzsébe pedig a feltétellel őrzött porthoz tartozó teljes kommunikációs kód kerül.

3.5.1. Protokollok értelmezése

A P–GRADE rendszerben a protokollok az üzenetcsomagok tartalmát írják le. Ez egy felsorolás, amely megadja, hogy a csomag milyen típusú adatból mennyit tartalmaz. Egy ilyen protokoll formálisan leírható $(type, size)$ párok sorozataként. A kódgenerálás során az üzenet értelmezése küldés esetén a puffer feltöltését, vagy fogadás esetén a puffer kiolvasását jelenti.

Ha adott a $(type_1, size_1), \ldots, (type_n, size_n)$ sorozattal leírt protokoll, valamint egy $\mathsf{v}_1, \ldots, \mathsf{v}_n$ változósorozat, akkor a következő protokoll-értelmező kódot lehet hozzá generálni. A protokoll minden eleméhez tartozik egy értelmező lépés; ezeket a lépéseket szekvenciálisan kell végrehajtani a protokollnak megfelelő sorrendben. A sorozat i. tagjának megfelelő lépés a következő:

- Üzenet fogadás esetén ha $size_i = 1$, akkor a v_i változó típusa kötelezően $type_i$, és a lépés:

  $$\mathsf{v}_i = \mathsf{upk}type_i\mathsf{();}$$

- Üzenet fogadás esetén ha $size_i > 1$, akkor a v_i változó típusa kötelezően $type_i$ elemek tömbje, és a lépés:

  $$\begin{gathered}\mathsf{v}_i = \mathsf{new}\ type_i[size_i]; \\ \mathsf{for\ (int\ j{=}0;\ j{<}}size_i\mathsf{;\ {+}{+}j)} \\ \mathsf{v}_i\mathsf{[j]} = \mathsf{upk}type_i\mathsf{();}\end{gathered}$$

- Üzenet küldés esetén ha $size_i = 1$, akkor a v_i változó típusa kötelezően $type_i$, és a lépés:

  $$\mathsf{pk}type_i(\mathsf{v}_i);$$

- Üzenet küldés esetén ha $size_i > 1$, akkor a v_i változó típusa kötelezően $type_i$ elemek tömbje, a tömb mérete legalább $size_i$, és a lépés:

  $$\begin{gathered}\mathsf{for\ (int\ j{=}0;\ j{<}}size_i\mathsf{;\ {+}{+}j)} \\ \mathsf{pk}type_i(\mathsf{v}_i\mathsf{[j]});\end{gathered}$$

3.5.2. Üzenet küldése (CAO elem)

Minden CAO művelet egy vagy több porton keresztül küld üzenetet. Az egyes portokra vonatkozó műveletek függetlenek egymástól, szekvenciálisan végrehajthatóak, a sorrendjük nem kötött. A továbbiakban az egy portra történő üzenetküldésről lesz szó.

Minden portra vonatkozó küldő művelet az initSend(portID, proto, sync) hívással kezdődik, ahol a portID és a proto a portot azonosító sorszám illetve protokollszám, a sync pedig a művelet paraméterei között megadott „szinkron küldés" tulajdonságot jelölő logikai érték. Ezt követi az üzenet(ek) összeállítása és elküldése, amelynek formája függ a kommunikáció típusától.

Pont-pont küldés: a protokollnak megfelelő pk* hívásokkal fel kell tölteni a küldő puffert, majd a send(0) hívással [1] elküldeni az üzenetet.

Multicast típusú küldés: a protokollnak megfelelő pk* hívásokkal fel kell tölteni a küldő puffert, majd a send() hívással elküldeni minden partnerének az üzenetet.

Scatter típusú küldés: egy iterációban (0 és grp_partners()-1 közötti indexekre) a protokollnak megfelelő pk* hívásokkal fel kell tölteni az puffert a megfelelő adatokkal, és a send(i) hívással elküldeni a soron következő partnernek.

Felhasználói típusú küldés: a felhasználó által megadott kódot kell végrehajtani.

Az elküldendő adatokat az első két esetben a felhasználó által megadott változókból kell venni. Scatter típusú küldés esetén a megadott változókat tömbként kell értelmezni, amelyek legalább annyi elemet tartalmaznak, mint ahány partnere van a portnak, és minden partnernek a neki megfelelő indexű tömbelemet kell elküldeni.

3.5.3. Üzenet fogadása (CAI elem)

A CAO elemekhez hasonlóan a CAI elemek is több porttal kommunikálnak egymástól függetlenül, ezek sorrendje tetszőleges, így itt is csak az egy porttal való kommunikációról lesz szó.

A fogadó művelet első hívása az initRecv(portID, proto) hívás, ahol a portID és a proto a port sorszáma és a protokoll azonosítója. A további műveletek itt is a kommunikáció típusától függenek.

Pont-pont fogadás: a recv(0) hívással megtörténik az üzenet fogadása, ezután a protokollnak megfelelő upk* hívásokkal el kell tárolni a kapott adatokat a felhasználó által megadott változókba.

Gather típusú fogadás: egy iterációban (0 és grp_partners()-1 közötti indexekre) a recv(i) hívással megtörténik egy üzenet fogadása, ezután a protokollnak megfelelő upk* hívásokkal el kell tárolni az adatokat. A scatter típusú küldéshez hasonlóan itt is tömb típusú változókat kell megadni, és az egyes partnerektől kapott adatok a neki megfelelő indexű tömbelembe kerülnek.

Felhasználói típusú fogadás: a felhasználó által megadott kódot kell végrehajtani.

[1] A 0 index az egyetlen partnert jelöli. Azért kell megadni a paramétert, mert a paraméter nélküli send multicast küldést jelez a monitorozó rendszernek.

3.5.4. Alternatív üzenet fogadás (CAIALT elem)

A CAIALT típusú műveleteknél a megadott portokon nem egymástól függetlenül történik a kommunikáció. Egy speciális támogató művelettel lehet végrehajtani, a hívása: altrecv(portIDs, protos), ahol mind a portIDs, mind a protos egy egészeket tartalmazó tömb, ugyanakkora mérettel, és a kommunikációban résztvevő portokat írják le. A művelet hatására megtörténik az üzenet fogadása.

A portIDs és protos tömbök tartalmát nem lehet fordítási időben előállítani, mert minden porthoz külön őrfeltétel adható meg. Ezért egy olyan kódot kell generálni, amely két Vector típusú változót használ, minden porthoz kiértékeli az őrfeltételként megadott kódot, és csak a feltétel teljesülése esetén tárolja el a port/protokoll párt a két vektorban. Az altrecv paramétere az ilyen módon dinamikusan felépített két vektor lesz.

A kapott adat eltárolásához meg kell állapítani, hogy melyik portról érkezett az üzenet. Ezt

if (getAltPort() == PORT_*Id* && getAltProto() == *proto*)

alakú őrfeltételekkel lehet megtenni, az őrfeltétel után pedig a megadott változókba az adott protokollnak megfelelő upk* hívásokkal el lehet tárolni a kapott üzenetet.

3.5.5. A felhasználói csoportkommunikáció műveletei

A felhasználói típusú csoportkommunikáció során a felhasználó megad egy kódrészletet, amelyben a grp_in és a grp_out függvények segítségével tetszőleges kommunikációs stratégiát le tud írni. A kommunikációs partner csoport elemeit el tudja érni az absztrakt indexeik segítségével, és mindegyikhez meg tudja adni, hogy milyen adatokkal történjen vele a kommunikáció.

Egy ilyen kommunikációs függvényhívás a csoport egy tagjával kommunikál, egy üzenet küldését ill. fogadását végzi el. Ehhez paraméterként meg kell adni a partner absztrakt indexét, valamint az adathoz rendelt változókat. Ez utóbbiak megadása függ a használt protokolltól, ezért minden, a processzben használható protokollhoz generálni kell a két, megfelelő paraméterekkel rendelkező kommunikációs függvényt.

Ha egy processzben megjelenik a $(type_1, size_1), \ldots, (type_n, size_n)$ protokoll, akkor a következő függvényeket kell generálni a processz kódjába.

void grp_out(int index, type_1[] t1, ...,type_n[] tn);

> A függvény a paraméterként kapott adatokból összeállított üzenetet elküldi az index absztrakt indexű partnernek.

void grp_in(int index, type_1[] t1, ...,type_n[] tn);

> Az index absztrakt indexű partnertől fogad egy üzenetet, és a kapott adatokat eltárolja a paraméterként megadott tömbökben.

Mindkét függvény törzsét a 3.5.1. pont alapján kell generálni. Egy különbség van, mégpedig hogy ha a protokoll egy elemének mérete 1, akkor nem primitív típusú változót, hanem a tömb változó 0 indexű elemét kell használni.[2]

[2] Azért szükséges mindenhol tömb típust használni, mert csak így lehet megoldani az output paraméterek kezelését.

Mindkét függvény feltételezi, hogy a paraméterként megadott tömbök mérete megfelel a protokollban megadott size_i méreteknek. Ennek biztosítása a felhasználó felelőssége.

Mivel ugyanaz a protokoll többször is előfordulhat egy processzben, és több különböző protokollhoz is tartozhat ugyanolyan szignatúra, ez fordítási hibákat okozhatna, ezért több ugyanolyan szignatúrájú függvényt igénylő protokollhoz is csak egy függvénypárt szabad generálni.

3.6. Egyéb processz szintű elemek kódjának generálása

A P–GRADE-ben a kommunikáción kívül a következő processz szintű elemek találhatóak:

Szöveges blokk (SEQ): A kódgenerálás annyiból áll, hogy a megadott kódot be kell illeszteni a programszövegbe.

Feltételes elágazás (COND): Az elágazás forrásszövege egy if utasítás fejlécét tartalmazza C szintaxissal, amely a megegyezik a Java nyelvű szintaxissal. A kódgenerálás során így ezt változatlanul át lehet venni, az elágazás két ágának a kódját pedig egy else kulcsszóval elválasztva be lehet tenni a fejléc után a kódba.

Ciklus konstrukció (LOOP): A forrásszöveg egy C szintaxisú ciklusfejlécet tartalmaz, amelyet itt is változatlanul át lehet venni. A ciklus magjának kódját be lehet tenni a Java ciklus blokkjába.

Grafikus blokk (GRPH): A blokk egy ugyanolyan felépítésű program, mint a főprogram, így azzal megegyezően lehet belőle kódot generálni, majd azt betenni a programszövegbe.

3.7. Monitorozó utasítások generálása

A processz main metódusának elejére el kell helyezni egy grmStart(app, proc, id), a végére pedig egy grmExit() hívást. A grmStart hívás app paramétere az alkalmazás neve, a proc paraméter a processz neve, az id a processz belső azonosítója. A kommunikációs műveletek estében a monitorozást a támogató osztály végzi.

A többi processz szintű művelet esetében (lásd 3.6) a művelet kódja elé és mögé egy grmBeginBlock(type, id) illetve egy grmEndBlock() hívást kell generálni. A type paraméter a művelet belső P–GRADE kódja, az id paraméter a művelet processzen belüli azonosítószáma (szintén egy belső P–GRADE azonosító).

3.8. A programkód referencia listájának generálása

A grafikusan megadott és a generált kód közötti kapcsolatot a referencia lista írja le. Ebben a listában az egyes programelemek és a generált kód részeinek a megfeleltetése van. Ezt fel lehet használni a fordítási hibák forrásának megkeresésére vagy a debugoláshoz.

A lista szövegfile-okba kerül a kódgenerálás során, minden Java forrásfile-hoz készül egy referencia file. A file neve megegyezik a forráskódéval, csak a kiterjesztése .java helyett .crf. Minden sora egy programszakasz leírását tartalmazza a következő formában:

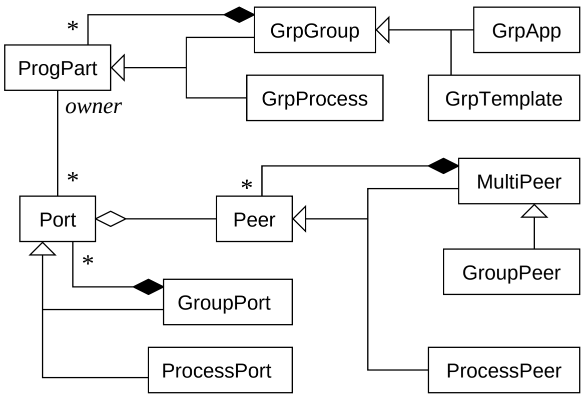
Típus Első Utolsó Név Azonosító

A Típus mező értéke a D vagy az U karakter lehet, a D egy grafikus blokkot, a U egy szövegesen megadott kódblokkot jelöl. Az Első a blokkhoz tartozó forrásszöveg első sorának száma, az Utolsó az utolsó sor száma. A Név a processz kód neve (ez általában megegyezik a file nevével), az Azonosító pedig a programrésznek megfelelő blokk P–GRADE azonosítója.

4. A támogató könyvtár implementációja

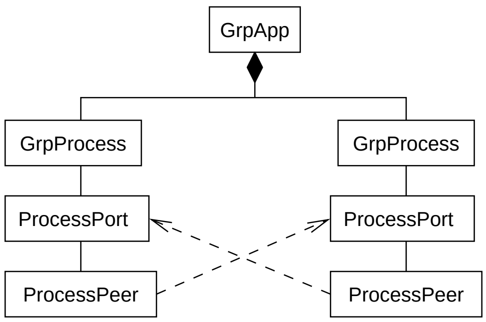
A támogató könyvtár feladatai két fő csoportra oszthatóak: az alkalmazás elindításával illetve a kommunikációval kapcsolatos műveletek implementálására. Az indítást a GrpGroup és a GrpApp osztályok kódja végzi, ez az indító node-on fut. Az indítás során a processzek kódjai eljutnak a futtató node-ra. A kommunikációhoz szükség van arra, hogy az egyes processzek ismerjék a partnereik elérhetőségét. Ezt az információt tárolják a Port és Peer osztályok, ezeket használja a GrpProcess osztály kódja az üzenetek küldéséhez és fogadásához.

A megvalósításhoz használt osztályok a 3. ábrán láthatóak. Az osztályok egy belső interfészt nyújtanak egymás felé, amely a generált kód számára nem érhető el.



3. ábra. A támogató könyvtár implementációjának osztálydiagrammja

Egy processzt leíró GrpProcess objektumhoz pontosan annyi ProcessPort objektum tartozik, ahány portja van a processznek (a csoport portokat is beleértve). Minden port objektumhoz tartozik egy Peer objektum, amely a porthoz kapcsolódó többi port elérési információit tartalmazza. A 4. ábrán a legyeszerűbb, két processzből és köztük egy csatornából álló program objektumdiagrammját lehet látni. A szaggatott vonalak az üzenetküldési lehetőségeket mutatják.



4. ábra. A legegyszerűbb kommunikációjú program objektumdiagrammja

4.1. A ProgPart osztály

Ez az osztály a processzek és csoportok egy közös funkcióját írja le: a portok kezelését. Ehhez egy Port típusú elemekből álló tömböt tartalmaz, amelyet GroupPort vagy ProcessPort típusú objektumokkal inicializál, ennek az elemeit lehet elérni a következő belső interfészen keresztül.

ProgPart(int n, int type);

> A konstruktornak az n paraméterben meg kell adni a portok számát. A type paraméter értéke PART_GROUP vagy PART_PROCESS lehet, ettől függően GroupPort vagy ProcessPort típusú portokkal fog rendelkezni az inicializált objektum.

Port getPort(int i);

> Visszaadja az i sorszámú portot.

4.2. A GrpGroup osztály

A csoportok elemeinek nyilvántartására az osztály egy ProgPart típusú elemekből álló tömböt használ, melynek méretét a konstruktor kapja meg. A tömb elemeit a createProcess és createGroup hívások inicializálják az init metódus futása során.

Ez az osztály végzi a portok összekötését is, a ProgPart osztály getPort és a Port osztály toPeer és addPeer metódusai segítségével. Az összekötést mindig mindkét irányba el kell végezni, egy port típusa (in, out vagy inout) ezen a szinten már nem jelenik meg. A csoport portok inicializálásának a kódja is itt van, ezt a GroupPort osztály addMember metódusával (lásd a 4.6. pontot) lehet megtenni.

Az indítás előtt a processzeknek el kell juttatni néhány információt az alkalmazás egészéről. Az információk összegyűjtését és szétterítését is a csoporthierarchián keresztül érdemes végezni. Speciális információ a processzek kommunikációs azonosítója, amelyet a processz regisztrálása során az alkalmazás objektum oszt ki nekik; a regisztrálás is a csoporthierarchián keresztül történik.

Végül a processzek gridben való elindításának kódját is ez az osztály tartalmazza.

Az osztály belső interfésze a következő:

void registerProcess(GrpProcess proc);

> Elvégzi a proc processz regisztrálását. Ebben az osztályban ez a metódus továbbadja a regisztrációs kérelmet az őt tartalmazó csoportnak, a hierarchia alján álló alkalmazás objektum végzi el a regisztrálást.

void runParts();

> Elindítja a csoportban található processzeket a gridben, valamint meghívja a benne levő csoportok runParts metódusát.

Map getInfo();

> Összeállítja a csoport elemeiről a szükséges globális információkat, és egy HashMap-ben elhelyezve visszaadja azokat. A tartalomról lásd a 4.5.2. pontot.

setInfo(Map info);

Szétteríti az info paraméterben kapott globális információkat a csoport elemei közt.

4.3. A GrpApp osztály

Ez az osztály felelős az alkalmazás valódi indításáért. Ennek a kódja a start metódusban van, amely elvégzi az indításhoz szükséges inicializációkat, majd meghívja a runParts metódust.

Ez az osztály végzi el a kommunikációs azonosítók kiosztását. A program minden processzéről érkezik egy registerProcess hívás a csoporthierarchián keresztül. Az alkalmazás megállapítja a következő szabad MPI rank értéket, és a setRank hívással átadja azt a processz objektumnak.

Az inicializáció része a globális információk kigyűjtése és szétterítése az alkalmazásban. Ezt a

setInfo(getInfo());

hívással lehet megtenni az alkalmazás szintjén: a getInfo összegyűjti az alkalmazás egészéből az információkat, a setInfo pedig elküldi minden processznek.

4.4. A GrpTemplate osztály és leszármazottjai

A GrpTemplate osztály elsősorban a template-ek paramétereinek adminisztrációját végzi, amire azért van szükség, hogy a kódban használható információs függvényekhez eljussanak a megfelelő adatok. Emellett ad néhány kényelmi funkciót a leszármazottjai felé, a processzek példányosításához és portjaik eléréséhez szükséges Java Reflection műveleteket rejti el. A következő belső interfészt nyújtja:

GrpTemplate(String name, int elNum, int pgID, int portNum);

A konstruktornak meg kell adni a name paraméterben a template nevét, a többi paraméter megegyezik a GrpGroup osztály konstruktorának paramétereivel.

void createProcess(int ID, Class code, int[] index, int[] dim);

Ezzel a metódussal egy új processz példányt lehet létrehozni a templateben. A ID paraméter a processz azonosítója, a code a processz kódját tartalmazó osztály (ebből hoz létre egy új példányt a metódus). Az index és a dim tömb azokat az információkat tartalmazza, amelyeket a processz grp_myindex() és grp_dimensions() hívásainak kell visszaadniuk.

int getPort(int ID, String name);

A kommunikációs struktúra felépítését segítő függvény, az ID azonosítójú processz name néven deklarált portsorszámát adja vissza.

A leszármazottak implementációja. Mindhárom speciális template osztály funkciója a processzek létrehozása és a belső kommunikáció felépítése. A defineProcess metódus végzi a processzek létrehozását a konstruktorban megadott méretek alapján. Az init metódus végzi el a kommunikációs struktúra felépítését a tíspu-specifikus inicializáló metódus (pl. initFarm()) hívása után.

4.5. A GrpProcess osztály

Ennek az osztálynak a leszármazottjai lesznek a gridben futó tényleges processzek, ezért ez az osztály a Task interfész implementációjaként készül el. A taszk főprogramja hívja meg a leszármazott osztályba generált main metódust.

A processz elindítása előtt minden olyan információt el kell juttatni a processzhez, amelyre szüksége lehet a futás során. Itt elsősorban a többi processz eléréséhez szükséges adatokról van szó, amely a csoport inicializálás során a Peer objektumokon keresztül jut el a processzhez, valamint a P–GRADE program strukturális információiról. Ez utóbbiakat a következő belső interfészen keresztül adja meg a GrpGroup osztály:

void setIndex(int[] ind);

> Egy template-en belüli processznek a processz indexét vagy indexeit megadó metódus. A grp_myindex() függvény a későbbiek során az ind paraméter tartalmát adja vissza.

void setDimensions(int[] dim);

> Egy template-en belüli processznek a template dimenzióinak méretét megadó metódus. A grp_dimensions() függvény a későbbiek során a dim paraméter tartalmát adja vissza.

void setInfo(Map info);

> Egy processznek az alkalmazás strukturális információit megadó metódus. Az info paraméter jelentése a 4.5.2. pontban van leírva.

void setRank(int rank);

> A processz regisztrálása során az alkalmazás objektum ezen a metóduson keresztül juttatja el a processzhez a kommunikációs azonosítóját.

int getRank();

> A ProcessPort osztály ezen a metóduson keresztül kérdezi le a processz kommunikációs azonosítóját a portra mutató ProcessPeer objektum létrehozása során.

4.5.1. Kommunikáció

A kommunikációs műveletek két feladatot látnak el: lehetővé teszik az üzenet tartalmának pufferelését, valamint az üzenet küldését és fogadását. A küldést és fogadást a Peer osztály metódusai valósítják meg, amelyek byte típusú tömböket tudnak küldeni ill. fogadni. Erre az osztályra marad a tömb feltöltése illetve tartalmának értelmezése.

Mind a feltöltést, mind a kiolvasást könnyen meg lehet oldani a java.io csomag szolgáltatásaival. A ByteArrayOutputStream és DataOutputStream osztályok kombinálásával tetszőleges primitív típusú adatok sorozatából lehet készíteni byte tömböt, a ByteArrayInputStream és DataInputStream osztályokkal pedig ugyanezt vissza lehet olvasni.

A kommunikáció monitorozása. Minden kommunikációs metódus elvégzi a szükséges monitorozó hívásokat. A hívásokhoz szükséges a partnerek belső P–GRADE azonosítóinak ismerete, ezt a Peer osztály getProcID() metódusával lehet lekérdezni.

Nem multicast üzenetküldés: a send(int) metódus minden hívásakor meghívja a grmSend(targID, commID, bytes) metódust. A targID paraméter a célpont processz belső azonosítója, a commID a protokoll azonosítója, a bytes pedig az üzenet mérete.

Multicast üzenetküldés: a send() metódus (multicast verzió) minden hívásakor meghívja a grmMcast(targIDs, commID, bytes) metódust. A targIDs egy tömb, ami a célpont processzek belső azonosítóit tartalmazza, a másik két paraméter ugyanaz, mint az előző pontban.

Üzenet fogadás: a recv() és a altrecv(int[],int[]) metódusok minden hívásukkor meghívják a grmReceive(srcID, commID, bytes) metódust. A srcID a kapott adat forrásának belső azonosítója, a többi paraméter ugyanaz, mint a többi esetben.

4.5.2. A szöveges kódból elérhető metódusok

A szöveges kódból kommunikációs és információs műveleteket lehet elérni. A kommunikációs műveletek közvetlenül megfeleltethetőek az osztály kommunikációs interfészének. A legutolsó kommunikációval kapcsolatos információs függvények (grp_which_* alakú metódusok) a kommunikáció műveletek által beállított példányváltozókat kérdezik le. A grp_myindex() és a grp_dimensions() hívások a korábbi inicializáció során kapott értékeket adják vissza. A grp_partners() függvény a Port osztály interfészén keresztül kapja meg az információt.

A grp_get_dimensions() függvény az eddigiekkel ellentétben globális jellegű információt ad meg, amihez a processz setInfo metódusával beállított információk szükségesek. Ezt egy Map típusú objektummal lehet leírni, amely az alkalmazás template-jeinek leírását tartalmazza. Mivel csak a dimenziók méreteire lehet szükség, más nincs is benne: minden kulcs objektum String típusú, és egy template nevét tartalmazza; a hozzárendelt érték pedig int tömb típusú, és a dimenziók méreteit írja le.

4.5.3. Monitorozó metódusok

A monitorozást támogató metódusok egyszerűen továbbadják a paramétereiket a monitorozást ténylegesen végző könyvtár metódusainak.

4.6. A Port osztály

A Port osztály leszármazottjai reprezentálják a P–GRADE program portjait. A funkciójuk a kapcsolataik felépítése és nyilvántartása. A felépítés a program inicializálása során történik, ennek során kell egy olyan nyilvántartó struktúrát felépíteni, amely átvihető a grid bármely pontjára úgy, hogy nem veszít el semmilyen információt. Ezt segíti a Peer osztály, amely egy másik processz (vagy csoport) portjának elérhetőségét tárolja.

Két port összekapcsolása során a célpontként megadott port elkészít egy saját magára hivatkozó Peer objektumot, ezt tárolja el a forrásként megadott port. A következő belső interfészt nyújtja az osztály:

abstract Peer toPeer();

> Ez a metódus készíti el egy adott Port példányra hivatkozó Peer objektumot. A megvalósítása attól függ, hogy processz vagy csoport portról van szó.

Peer getPeer();

> Lekérdezi a porthoz kapcsolódó partner(eke)t leíró Peer objektumot.

void addPeer(Peer peer);

> Felvesz egy új partnert az eddig tároltak közé. Ha eddig nem volt partner, akkor egyszerűen eltárolja a peer objektumot. Ha eddig egy nem MultiPeer partnere volt, akkor készít belőle egy MultiPeer-t. Ezután mindenképp MultiPeer partnere van, ebbe felveszi a peer-t. Lásd még a 4.7. pontot.

4.6.1. A **ProcessPort** osztály

Ez az osztály processz portokat reprezentál, ezért csak GrpProcess objektumhoz tartozhat. A következő belső interfészt nyújtja:

ProcessPort(GrpProcess owner);

> A konstruktornak az owner paraméterben meg kell adni azt, hogy melyik processznek a portja jön létre.

Peer toPeer();

> Létrehoz egy ProcessPeer objektumot, amely saját magára hivatkozik.

4.6.2. A **GroupPort** osztály

Ez az osztály egy csoport portot reprezentál olyan módon, hogy nyilvántartja a csoport tagjaiban ehhez a csoport porthoz rendelt belső portokat. Csak GrpGroup objektumhoz tartozhat. A következő belső interfészt nyújtja:

GroupPort(GrpGroup owner);

> A konstruktornak az owner paraméterben meg kell adni azt, hogy melyik csoportnak a portja jön létre.

void addMember(Port port);

> Hozzáadja a belső portok listájához a port portot.

Peer toPeer();

> Létrehoz egy GroupPeer objektumot, amely a hozzá rendelt belső portok Peer-jeit tartalmazza.

void addPeer(Peer peer);

> A csoport porthoz hozzádott új partnert a csoport tagjanak át kell adni, mintha minden tag külön-külön megkapta volna.

4.7. A Peer osztály

4.7.1. Az osztály feladata és interfésze

Az osztály leszármazottjainak feladata egy port partnerének vagy partnereinek a nyilvántartása, valamint a velük való kommunikáció lehetővé tétele. A ProcessPeer objektumok kapcsolódnak közvetlenül a processzekhez, ezek végzik a tényleges adatküldést/fogadást. A MultiPeer objektumok több, egymástól független Peer objektumot fognak össze. A GroupPeer objektumok csoport portokkal való kommunikációt valósítanak meg, amely a multicast üzenetek kivételével megegyezik a független partnerekkel való kommunikációval.

A ProcessPeer osztály tárolja a „túloldali” processz kommunikációs szintű azonosítóját, valamint a port sorszámát. A küldéskor/fogadáskor megadott protokoll azonosítóval együtt ez egyértelműen azonosítja egy P–GRADE csatorna végpontját, és lehetővé teszi vele a kommunikációt.

A MultiPeer osztály két tömböt tárol. Az egyikben Peer objektumok vannak, amelyek lehetnek ProcessPeer vagy GroupPeer típusúak is, a port csatornavégpontjainak típusai szerint. A csoportok ismeretét a multicast küldés során lehet hasznosítani. A másik tömbben csak ProcessPeer-ek vannak. Itt is megvan minden végpont, de eltűnik a csoport-hierarchia. Ezt a unicast küldéskor lehet használni, amikor indexszel kell hivatkozni a partnerekre.

A Peer osztály belső interfésze a következő:

int getProcID(int index);

> Visszaadja az index absztrakt indexű partner processz P–GRADE azonosítóját (a monitorozáshoz szükséges).

int[] getProcID();

> Visszaadja az összes partner processz P–GRADE azonosítóját (a multicast monitorozáshoz szükséges).

ProcessPeer[] flatPeer();

> Visszaadja az objektumhoz tartozó processz végpontokat a csoport-hierarchia nélkül.

void setLocal(int rank, int portID);

> Ezen a metóduson keresztül kapja meg az objektum a lokális portra vonatkozó információkat, amelyek kellenek az üzenetek fejléceinek a kitöltéséhez.

void send(int proto, boolean sync, byte[] data);

> Multicast adatküldést megvalósító metódus (proto protokoll azonosítóval). A sync paraméter azt jelzi, hogy meg kell-e várni az üzenet megérkezését.

void send(int index, int proto, boolean sync, byte[] data);

> Az index absztrakt indexű partner felé proto protokoll azonosítójú unicast adatküldést megvalósító metódus. A sync paraméter azt jelzi, hogy meg kell-e várni az üzenet megérkezését.

byte[] recv(int index, int proto);

> Az index absztrakt indexű partner felől proto protokoll azonosítójú üzenet fogadását megvalósító metódus.

static AltResult altrecv(Peer[] peers, int[] protos);

> Alternatív fogadást megvalósító osztálymetódus. A peers és a protos tömbök méretei megegyeznek, az azonos indexű elemeik egy lehetséges forrást azonosítanak egy Peer objektum és a protokoll segítségével.

Az altrecv metódus visszatérési értékének típusa AltResult, amely statikus belső osztály, a következő publikus változókkal:

int port;

> A kommunikációban résztvevő port sorszámát tartalmazza.

int partner;

> A résztvevő partner absztrakt indexét tartalamazza.

int proto;

> A kommunikáció protokoll-azonosítóját tartalmazza.

byte[] data;

> A kapott adatot tartalmazza.

A Peer osztály műveletei absztrakt műveletek, az egyes leszármazottak implementálják őket a megfelelő módon. A leszármazottak interfészeiben csak a konstruktorok térnek el:

ProcessPeer(int rank, int portID, int pgID);

> A processz végpontnak a rank paraméterben a végpont processz MPI azonosítóját, a portID-ben a port sorszámát, a pgID-ben pedig a processz P–GRADE azonosítóját kell megadni.

MultiPeer(Peer[] peers);

> A csoportos végpont létrehozásához csak a csoport elemeit kell megadni.

GroupPeer(*groupID*, Peer[] peers);

> A valódi processz csoport végponthoz meg kell adni a csoport kommunikációs azonosítóját is. Amíg a rendszer nem támogatja a csoportokat, addig a *GroupID* paraméter elmarad.

4.7.2. A kommunikáció implementációja

A kommunikációs műveletek megvalósítása a ProcessPeer osztályban található. A MultiPeer és GroupPeer osztályok műveletei a flatPeer metódus segítségével megkapják a megfelelő ProcessPeer példányokat, és azoknak a műveleteit hívják meg.

Az alternatív input (CAIALT) műveleteket az MPI nem blokkoló üzenetfogadó hívásaival lehet megvalósítani, ezért a program futása során minden porthoz függőben vannak ilyen üzenetfogadások. A porthoz tartozó protokollokat külön csatornaként kell kezelni, ezért minden protokollhoz rendelünk egy MPI tag-et és egy külön üzenetfogadást. Minden üzenetfogadáshoz tárolni kell egy MPI_Request objektumot és egy adatpuffert, ezeket dinamikus vektorokban tárolja az osztály, amelyeket a protokoll azonosítójával indexel.

Két processz között több csatorna is lehet, ezeket a portok azonosítói különböztetik meg. Az üzenetek MPI tag-jeibe ezért a portok generált sorszámait is bele kell venni. A tag-ek 32 bites, előjeles egész számok, ezért célszerű a felső 16 biten tárolni a port sorszámát, az alsó 16 biten pedig a protokoll azonosítóját.

A P–GRADE üzenetek két MPI üzenetként kerülnek továbbításra. Az első üzenet egy fejléc, amely két egész számot tárol: az első az üzenet forrása, a második pedig a mérete. A forrást a hozzá rendelt MPI tag írja le. Ha ez a szám negálva van, az szinkron üzenetküldést jelez, ekkor nyugtázó üzenet küldése szükséges, amelyben a negált tag-et kell használni. A második MPI üzenet tartalmazza az adatokat, annyi byte típusú elemet, amennyit a fejléc első mezője megad. Az esetleges nyugtázó üzenet üres.

A processz indulásakor egyetlen üzenetfogadás sem kezdődik meg. Minden csatorna első üzenetfogadását a rá vonatkozó első `recv` vagy `altrecv` hívás indítja. Alternatív input esetén több fogadás is elindul és ezek közül csak egy fejeződik be, a többi aktív marad. Ezeket az aktív fogadó műveleteket a következő `recv` típusú művelet használja fel.

A `Peer` osztály kommunikációs műveleteit tehát a következőképpen lehet megvalósítani:[3]

`void send(int index, int proto, int sync, byte[] data);`

> Összeállítja a fejlécet a forrás oldali port számból, a paraméterként megadott protokollból (amely megegyezik mind a forrás, mind a cél oldalon), és abból, hogy szinkron küldésről van-e szó (szintén paraméter). Elküldi a fejlécet és az adatokat az `MPI_Send` hívással. Szinkron küldés esetén az `MPI_Recv` hívással megvárja a nyugtázó üzenetet.

`byte[] recv(int index, int proto);`

> Megvizsgálja, hogy van-e aktív üzenetfogadás a paraméterként megadott protokollhoz; ha nincs, létrehozza az `MPI_Irecv` hívással. Megvárja a fejlécet az `MPI_Wait` hívással, majd fogadja az adatokat az `MPI_Recv` hívással. Ha a fejléc szinkron küldést jelez, visszaküld egy nyugtát.

`static AltResult altrecv(Peer[] peers, int[] protos);`

> Megvizsgálja, hogy a paraméterben felsorolt partnerekhez és protokollokhoz van-e aktív üzenetfogadás, amelyikhez nincs, ahhoz az `MPI_Irecv` hívással létrehozza. Az `MPI_Waitany` hívással megvárja az első sikeres fogadást, majd ugyanúgy működik tovább, mint a `recv` metódus.

[3] Az `MPI_Recv` és `MPI_Wait` hívások emulálhatóak az `MPI_Irecv` és `MPI_Waitany` hívásokkal.

Hivatkozások

[1] P–GRADE User's Manual
http://www.lpds.sztaki.hu/pgrade/p_grade/manual.html

[2] JGrid Design Document – Project Report

[3] Jini Network Technology
http://www.jini.org

[4] MPI: A Message-Passing Interface Standard
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[5] mpiJava 1.2: API Specification
http://grids.ucs.indiana.edu/ptliupages/projects/HPJava/mpiJava.html

[6] GRM/PROVE Users' Guide
http://www.lpds.sztaki.hu/~pnorbert/grm/manual