A TITAN grid rendszer és kommunikációja

Péter Tömösközi, Roland Király, Zoltán Hernyák

2009. április 28.

Kivonat

A jelenleg elterjedt, vagy kevésbé ismert, de hatékony grid rendszerek és az elosztott programozáshoz használatos programozási nyelvek sokaságának tükrében hasznosnak ítéltük, hogy egy saját, az említett és a továbbiakban felsorolásra kerülő rendszerek legjobb tulajdonságait megvalósító, vagy éppen kihasználni képes rendszert hozzunk létre, mely minden tekintetben megfelel a grid rendszerekkel szemben támasztott elvárásoknak.

Az általunk kifejlesztett és jelenleg is fejlesztés alatt álló rendszer legfontosabb elemei egy elosztott futtató környezet és a hozzá tartozó köztes réteg, mely platformfüggetlen módon viselkedik és egy elosztott funkcionális programozási nyelv és annak fordítóprogramja, mely nyelv könnyen elsajátítható, nagy kifejező erővel rendelkezik, és támogatja a párhuzamos programozást, valamint segítségével a párhuzamos programozáshoz nem értők is tudnak elosztott programokat fejleszteni.

A rendszerünket a TITAN kódnévvel illettük, utalva ezzel a nyelv kifejező erejére és az elosztott rendszerek hatékonyságára. Ebben az írásban kísérletet teszünk arra, hogy a TITAN rendszerről és a hozzá tartozó nyelvről átfogó leírást adjunk.

1. Számítási teljesítmény és GRID

A számítógépek egyik kiemelt fejlesztési iránya a számítási teljesítmény fokozása és ezzel egyidejűleg hatalmas számítási teljesítményű eszközök készítése. A mai napig elsősorban nagy erőforrásokkal rendelkező kutatóintézetekben, kormányzati megrendelésekre folyik ilyen rendszerek tervezése és telepítése.

Az internet terjedésével, a protokollok kiépülésével, és szabvánnyá válásával a távoli számítógépek megcímzése, munkára fogása egyre gyakoribb megoldás. Projektek jelentek meg, melyek a gépek *idle*, üresjárati idejében jelen lévő szabad számítási kapacitások kihasználása segítségével gyakorlatilag ingyen oldják meg a nagy számítási igényű feladataikat. Az emberek egyre szívesebben csatlakoznak és ajánlják fel a gépeik szabad kapacitását tudományos célok megvalósítása érdekében. Ennek természetesen határt szab a bizalmatlanság, a vírusoktól való félelem, és egyéb tényezők.

Egyszerűbb az eset, amikor a felhasználandó gépek mindegyike egy adott vállalat birtokában van, és dedikált cél egy projekt futtatása. Ezen gépek akár kis távolságra is elhelyezekedhetnek egymástól, nagy sebességű (jellemzően 100Mb, gigabit) ethernet kábelen csatlakoznak egymáshoz. A projektek futtatását ütemezni kell. Az aránylag olcsó, kis teljesítményű gépek összekapcsolásával hatalmas számítási kapacitás érhető el. Amennyiben valamelyik gép meghibásodik, szervíz idejére a teljes rendszert általában nem kell leállítani, a teljesítmény kis csökkenése mellett is megőrízhető az üzemképesség.

Amennyiben a szabad számítási kapacitások már rendelkezésünkre állnak, megfelelő programozási nyelvet kell választani a feladat implementálására. Sok programozási nyelv támogatja az elosztott programok fejlesztését, de nagy a hiány olyan operációs rendszerbeli komponensekben, amelyek az elosztott futtatást és az elosztott filerendszert támogatnák.

A Linux operációs rendszereken szokásos megoldás az ftp, scp lehetőségek használatával a kód átmásolása a távoli gépre, és ssh segítségével a kód indítása. Ugyanakkor elég kényelmetlen a Linux-ok beállítása hogy ezt egy script is automatikusan el tudja végezni, és ez a megoldás eléggé Linux specifikus, nem tudja kezelni a gépek között esetleg elhelyezkedő Windows alapú gépeket.

Linux operációs rendszer alatt működőképes megoldást nyújt az MPI [1] library használata. Ez egy kommunikációs függvénygyőjtemény, mely üzenetküldési (*message passing*) primitívekkel támogatja az alkalmazások közötti üzenetküldést. Az alkalmazások távoli gépre juttatására semmilyen támogatást nem ad, de azok megtalálását már nagy automatizmussal végzi.

A funkcionális programozási nyelvek is egyre nagyobb támogatást adnak az elosztott programozási technikák fejlesztésének. A funkcionális programozási nyelvek két fontos előnnyel bírnak. Az egyik az, hogy matematikai számítások leírására kézenfekvőbb eszközök mint az imperatív programozási nyelvek. Ezért a számítási feladatok kódolása egyszerőbb ezeken a nyelveken. A másik előny, hogy a kiértékelési rendszerük szinte kínálja a párhuzamos feldolgozás lehetőségét. Sajnos a párhuzamos kiértékelés nehezen vihető át elosztott rendszerekbe, mivel a kiértékelési gráf mélységi másolása a gépek között nem triviális feladat. Nem elhanyagolható a funkcionális nyelvi elemek kifejezőereje és az ilyen típusú programok tömörsége sem, mely nagyban segíti a programok értelmezhetőségét.

Párhuzamos kiértékelési rendszerrel bír a *Concurrent Haskell* [3, 4]. Ez mindössze négy nyelvi primitívvel bővítette a Haskell nyelvet, melyek közül a *forkIO* a legjelentősebb: új szál indítása, mely paraméterként egy Haskell kifejezést kap.

A JoCaml nyelv az Objective Caml [7] nyelv kiegészítése a *join calculus* segítségével. A JoCaml nyelv kifejezetten párhuzamos és elosztott programozás fejlesztési támogatással rendelkezik. Ez egy nem tisztán funkcionális nyelv, mely imperatív és oop elemeket is tartalmaz [6].

Az ERLANG [8], amelyet az Ericson és az Ellemtel Computer Science Laboratories fejlesztett ki. Az Erlang egy funkcionális programozási nyelv, amely konkurrens, valós idejő, elosztott és nagy hibatőrő képességő rendszerek készítését teszi lehetővé. Az Ericsson az Erlang Open Telecom Platform kiterjesztését telekommunikációs rendszerek fejlesztésére használja. A nyelv beépített eszközökkel rendelkezik az elosztottság és az üzenetküldés területén, közös memória terület nélkül, üzenetküldésekkel valósítja meg az elosztottságot. Támogatja más nyelven írt programkomponensek integrálását, viszont a típusossága igen gyenge.

A *Hume* [2] erősen típusos, funkcionális alapokkal bíró programozási nyelv első verziója 2000 novemberében jelent meg. A *Hume* elsődleges prioritása a futási idő és erőforrás-felhasználás limitáltságának megőrzése. A *Hume* öt szintet különböztet meg, mely öt absztrakciós szinthez tartozik. A *Hume* egy aszinkron kommunikációs modellt használ, melynek alapeleme a *doboz*. A dobozoknak egyedi azonosítóik, be-, és kimenő adataik pedig típusosak.

A felsorolt eszközök és programozási nyelvek sokaságának tükrében hasznosnak ítéltük, hogy egy saját, a fent felsorolt rendszerek legjobb tulajdonságait megvalósító, vagy éppen kihasználni képes rendszert hozzunk létre, mely minden tekintetben megfelel az elvárásoknak. A kifejlesztett rendszer legfontosabb elemei egy elosztott futtató környezet és a hozzá tartozó köztes réteg, mely platformfüggetlen módon viselkedik és egy elosztott funkcionális programozási nyelv és annak fordítóprogramja, mely könnyen elsajátítható, nagy kifejező erővel rendelkezik, és támogatja párhuzamos programozást, valamint segítségével a párhuzamos programozáshoz nem értők is tudnak elosztott programokat fejleszteni.

A rendszerünket a TITAN kódnévvel illettük, utalva ezzel a nyelv kifejező erejére és az elosztott rendszerek hatékonyságára.

2. A TITAN

A *TITAN* egy elosztott számítást támogató környezet. Több komponensből áll:

- *TITAN-F* elosztott futtató rendszer
- *TITAN-A* alacsony absztrakciós szintő, imperatív jellemzőkkel bíró programozási nyelv
- *TITAN-M* magas szintő, matematiakai absztrakciót támogató, funkcionális programozási nyelv

A *TITAN-F* elsődleges célja, hogy a titan programok futtatását támogassa. Ezt a köztes rétegre épített komponensekkel oldjuk meg, melynek van C++, Erlang, C#, PHP, Java interface-e, ezért ezen felsorolt programozási nyelvek mindegyikében lehetőség van a komponensek fejlesztésére és összekapcsolására.

A futtató rendszer egyes komponenseit minden egyes, a számítási *grid*-ben részt vevő számítógépen el kell indítani. Ezen komponenseken keresztül kapcsolódik be a szóban forgó számítógép a rendszerbe. Ezek a komponensek az alábbiak:

- *TService* a futtató rendszer szolgáltatása, melyen keresztül az alap parancsokat ki lehet adni az adott gépre. Ezek lehetnek további komponensek indítására vonatkozó parancsok, állapotlekérési utasítások, komponensek verziófrissítését végző utasítások. Ezen szolgáltatáson keresztül lehet a számítógépeket a *TITAN* szemszögéből távmenedzselni.
- *TStarter* a futtató rendszer azon szolgáltatása, amelyen keresztül a *TITAN* kódtárolójában szereplő, bináris kód letöltését és indítását kezdeményezni
- *TDebugger* a futtató rendszer nyomkövető szolgáltatása, melyen keresztül az adott gépen futó kódok állapotát, futási vermét, egyéb jellemzőjét le lehet kérdezni

A további komponensek csak a grid egyes, kitüntetett gépein kell csak elindítani:

- *TNameServ* a futtató rendszer névszolgáltatója. Hasonló szerepet tölt be mint az internetes kommunikációban a DNS szerverek. Egy, a futtató rendszer valamely komponensének, vagy egy futó kód-példánynak azonosítója alapján megadja annak tényleges helyét (esetleg TCP/IP címét). Ezen szolgáltatás képes több példányban is létezni, ugyanazon alhálózatban, mely esetben mindegyik példány ugyanazon információkat tárolja. A duplikált létezést indokolja, hogy egyes példányok leállása és meghibásodása esetén a további példányok automatikusan képesek legyenek átvenni a szolgáltató szerepet.
- *TCodeServ* a futtató rendszer kódtárolója. A programozó által megírt, és a rendszerben eleve létező kódok forráskódját és lefordított állapotát őrző szerver. Erre a szerverre csak digitális aláírással ellátott kód kerülhet fel - mely igazolja a forrás hitelességét. A projekt indulásakor – amennyiben valamely futó program dinamikus kódbetöltést kezdeményezne, akkor erről a szerverről lehet azt letölteni. A szerveren lévő kód lehet publikus (minden felhasználó beépítheti a programjába), illetve privát, mely esetben az adott kódhoz csak az őt feltöltő programozó férhet hozzá.
- *TAuthServ* a felhasználók autentikációját végző szerver, melyen keresztül a rendszerbe a felhasználók beléphetnek, és azonosíthatják magukat. Ezen szerver által kiállított hitelesítési adatok alapján működnek a rendszer további szolgáltatásai.
- *TScheduler* az ütemező, mely képes bizonyos feltételek (alacsony terheltségi szint, előre beállított időpont) alapján a felhasználók távollétében is (akár éjszaka is) kezdeményezni egy projekt indítását.

3. A TITAN-M programozási nyelv

A *TITAN-M* egy matematikai absztrakciós szinttel jellemezhető, funkcionális alapokon nyugvó programozási nyelv, melyen keresztül a programozók megalkothatják a rendszerbe illeszthető publikus, illetve saját használatra fenntartott privát függvényeiket.

A nyelv fejlesztéséhez a *Maple*, a *Clean* és a *ERLANG* szolgáltatták az alapokat. A *TITAN-M* nyelven megírt függvényeket a paranccsori fordítóval lehet *TITAN-A* nyelvi kódra fordítani. Ezen szolgáltatást az interaktív jellemzőkkel is bíró IDE felületbe fogjuk integrálni, mely jelentősen leegyszerűsíti a kezelést. Az interaktív működés miatt az egyes függvények megírása után egyszerű közegben azokat azonnal tesztelni is lehet.

A *TITAN-A* generált forráskódot egy fordítóprogram továbbfordítja C# nyelvre, mely alapján a C# fordító a bináris kódot elkészíti. A bináris kódba a felhasználó azonosítója beépítésre kerül, csakúgy, mint olyan meta-információk, mint a kód publikussága. A *TITAN-M*, *TITAN-A*, C# nyelvő forráskódok, és a bináris kód feltöltésre kerül a *TCodeServ* szerverre, amely annak hitelességét és összetartozását ellenőrzi.

A kódok feltöltése után egyszerű parancsokkal lehet a rendszert a kód indítására utasítani.

4. Stratégia-paraméterezés

A *TITAN-M* nyelvű programok stratégiák segítségével támogatják a fooráskód egygépes- szekvenciális, egygépes-párhuzamos, és grid alapú-elosztott működését. A startégia paraméter egyfajta aktív metatagként funkcionál, mely befolyásolja a fordítás során generált *TITAN-A* nyelvő kódot, de futás közben is befolyásolja az adott kifejezés kiértékelését.

Kétfajta stratégia-kezelést kívánunk megvalósítani:

- *kód-stratégia*
- *adat-stratégia*

A kód alapú stratégia segítségével egyes operátorok kiértékelését vagy a függvényvégrehajtást lehet párhuzamosítani és elosztottá tenni, illetve kikényszeríteni az egyszálú kiértékelést. A startégia paraméterek előre beépített értékek közül választhatóak. Jelenlegi terveinkben három ilyen lehetőség szerepel:

- *s* startégia jelenti a szekvenciális működést, vagyis az adott gépen egy szálon történő kiértékelést. Ez a stratégia hasznos lehet olyan I/O kommunikációt végző folyamatokban, amelyek kötött sorrendet igényelnek, és nem végezhetők külön szálon sem.
- *p* stratégia jelöli a párhuzamos működést, mely esetben az adott gépen szálindítások történnek, és a kiértékelés a szálakon párhuzamosan folytatódik. Ezen működést a rendszer visszautasíthatja, és áttérhet szekvenciális működésre, ha úgy itéli meg, hogy a teljesítményt ezen működés hátrányosan érintené, vagy a külön szálon indított függvények meta-információi szerint nem alkalmasak ilyen futtatásra.
- *d* stratégia jelöli az elosztott működést. Ez esetben a művelet nem szálindításokat generál, hanem a futó kód egy része áttöltődik a grid egy másik gépére, és ott fut tovább. A másolás lustán történik, tehát a kód által hivatkozott memóriaértékek csak akkor másolódnak át szintén a túloldali gépre, ha arra az odamásolt kód hivatkozni készül.

TITAN-M

```
1  dpf Sinus‘s ( Double x ) -> Double
```

1. példa. A szekvenciális kiértékeléső Sinus fv

TITAN-M

```
1  dpf Sinus‘PS ( Double x ) -> Double
```

2. példa. Startégia-paraméterezett Sinus fv

TITAN-M

```
1  dpf Nfib’PS (Int x)->Int
2    | n<2          = 1
3    | otherwise    = Nfib‘PS(n-1) ś Nfib‘PS(n-2)
```

3. példa. Startégia-paraméterezett Fibonacci fv

A stratégia paraméter a rendszerben automatikus mőködéső, de a programozó a kód írása során lekérheti az aktuális startégia paramétert, megvizsgálhatja, illetve továbbadhatja azt.

Az 1 példában szereplő *dpf* szavak a *define public function* rövidítése, melyek egy TITAN-M nyelvi függvény prototípusának megadásákor használatosak. A függvény neve *Sinus*, mely ezen definíció szerint a *s*, vagyis szekvenciális kiértékeléső változat lesz. A függvény törzsében ekkor csakis olyan utasítások szerepelhetnek, melyek nem kezdeményeznek a kódban sem párhuzamos sem elosztott végrehajtást. Ezen függvény-változatra hivatkozó kód biztos lehet hogy a függvény eredményének kiértékeléseben szerepet játszó utasítások mindegyik az adott gépen fog végrehajtódni.

Ezen változat valójában egy mintaillesztéshez hasonló értelmezés miatt akkor kerül felhasználásra, ha a Sinus műveletet kérő kódban szintén a *s* stratégiával kérték a kiértékelést. A rendszerben (library kód) szerepelhet a *Sinus‘p* és *Sinusd* változatok is, melyek párhuzamos és elosztott kiértékeléső sinus műveleteket takarnak. Ha a hívás helyén konkrétan *p* vagy *d* kiértékelési startégiát írták elő, és ilyen stratégiájú Sinus függvény is szerepel, akkor azok választódnak ki. Ha *p* vagy *d* kiértékelést kértek, de csak *s* szekvenciális startégiájú változat áll rendelkezésre, akkor automatikusan ezen változat hívódik meg. A programozó kérhet fordításkor elemzést, amelybe a hivási és megvalósítási ütközések *warning* formában kilistázásra kerülnek.

A 2. példában szereplő paraméterezés esetében a Sinus függvény nincs statikusan a *s* szekvenciális kiértékelési startégiához kötve, a *PS* azonosítójú paraméter az *aktuális* stratégia-paramétert jelöli. Ha például a hívás helyén *p* párhuzamos kiértékelést kértek, akkor a *PS* értéke ezen *p* stratégia-érték lesz.

A függvény törzsében a *PS* paraméterre hasonlóan lehet vizsgálatokat végezni, mint az adatparaméterek esetén. A függvény törzsét ennek megfelelően az aktuálisan kért kiértékelési startégia alapján lehet elágaztatni.

5. Dinamikus stratégiák

Bonyolulabb esetekben a fenti stratégia-paraméterek nem biztosítanak elég nagy rugalmasságot. Vizsgáljuk meg a közismert *Fibonacci* függvény törzsét:

A 3. példában szereplő Fibonacci függvény szintén dinamikusan paraméterezett. A hívás helyén

TITAN-M

```
1  dpf sum`s([Int]`p{ 5} lst)->Int
```

4. példa. Stratégia-paraméterezett lista

specifikált, a kódban *PS*-el jelölt aktuális startégia értékét a rekurzív hívás során továbbadja saját példányainak, de a két részeredményt összesítő + műveletet *s* szekvenciális startégiával adja össze, vagyis a két részeredmény elosztott aktuális startégia esetén (PS=d) más-más gépeken futhat le, de az össszeadást ezen a gépen kell elvégezni.

Hatékony működés miatt fontos lenne két módosítás:

- utasítani a fordítót, hogy az összeadás műveletet azon a gépen végrehajtani, amelyiken valamelyik operandusz képződik, vagyis vagy azon, amelyiken az *NFib(n-1)*, vagy azon, amelyiken az *Nfib(n-2)* kiértélése történik. Ez esetben ugyanis csak az egyik operandusz-értéknek kellene utaznia a hálózaton keresztül.

- az elosztott kiértékelés a fibonacci működése miatt hatékony, hiszen a két részeredmény külön-külön is kiszámolható. Ugyanakkor minden rekurzív hívás két ágra szakítja a működést, nagyobb kiindulási N érték esetén a rendszerben túl sok kiértékelési szál keletkezne, melyek összességében hatékonyság csökkentéséhez vezetnek. Az elvárt viselkedés szerint a fibonacci függvények például a rekurzió 5 szintjéig viselkedjenek elosztott stratégiaként, de az ötödik szint után automatikusan alakuljanak szekvenciális *s* stratégiává. Ekkor a további rekurzív hívások már az adott gépen belüli kiértékeléssé alakulnak át.

6. Adat-stratégiák

Az adatok kiértékelési stratégiája nagy mennyiségő adat esetén lehet jelentős. Amennyiben egy listánk van, melyekben szereplő értékekkel műveleteket kívánunk végezni – kérhetjük, hogy a nagy méretű listát a rendszer bontsa fel rész-listákra, végezze el a műveleteket párhuzamos vagy elosztott módon, majd az eredményeket összesítse.

Az adat-stratégia egyelőre csak listák esetén van értelmezve.

A 4. példában szereplő *lst* függvény paraméter típusa szerint *[Int]*, melyet *p* párhuzamos stratégiával paramétereztünk fel. Ezt olyan függvények esetén tehetjük meg, amelyek elemenként feldolgozhatók. Ez jelen esetben azt fogja eredményezni, hogy a rendszer a listát minimum 5 részre fogja felbontani, a rész-listákra egyesével alkalmazza a *sum* függvényt, melyek mindegyik egy-egy *Int* típusú értéket állít elő. Az eredményeket a + operátorral összesíti a rendszer automatikusan, így állítván elő a feldolgozás eredményét.

Az adat-stratégia jelentősége egyrészt akkor kerül előtérbe, ha az alap feldolgozó függvény (jelen esetben a *sum*) nem képes maga megoldani a párhuzamos feldolgozást. A jelen példában szereplő *sum* függvény *s* szekvenciális stratégiájú, mely esetben a lista elemeinek összegét egyetlen gépen, egyetlen szálon készíteni el. Az adat-stratégia kikényszeríti a párhuzamos kiértékelést.

7. Célok és lehetőségek

A technikai leírás összefoglaljuk mindazt, amit célul tűztünk ki. A tapasztalatok azt mutatják, hogy az általunk létrehozott modell és annak implementációja hasznos, könnyedén használható és univerzális. Részben sikerült megvalósítanunk a kitűzött célokat, és vannak olyanok, amiket éppen a napokban

implementálunk. Mivel ez a projekt még gyerekcipőben jár és folyamatosan fejlődik, néha új irányokat vesz és remélhetőleg még nagyon sok új lehetőség rejlik benne. Fontosnak tartjuk azt, hogy a forráskódok, az ötletek és minden egyéb elem egyszerűen hozzáférhető és ingyenes, és talán ezen tényező miatt számos kutató csatlakozik ideig-óráig a kutatáshoz és néhányan jelenleg is részt vesznek a munkában.

Hivatkozások

[1] William Gropp, Ewing Lusk, Anthony Skjellum: *Using MPI - Portable Parallel Programming with Message-Passing Interface* MIT Press, 1999

[2] Kevin Hammond, Greg Michaelson, Robert Pointon: *The Hume Report, version 1.1*, `http://www-fp.cs.st-andrews.ac.uk/hume/report/`

[3] Jones, S. P., Gordon, A., Finne, S.: Concurrent Haskell, Conference Record of POPL '96: The 23rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, Glasgow, 1996, 11 pp.

[4] Finne, S. and Jones, S., P. J.: Concurrent Haskell, In Principles Of Programming Languages, St. Petersburg Beach, Florida, 1996, pp. 295-308

[5] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1-40 and 41-77, 1992.

[6] Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: *The JoCaml language beta release, Documentation and user's manual*, INRIA, 2001.

[7] Leroy X. et al. The Objective Caml Language (version 3.10). Software and documentation, available at `http://caml.inria.fr`, 2007.

[8] J. Barklund and R. Virding. Erlang Reference Manual, 1999. Available from `http://www.erlang.org/download/erl_spec47.ps.gz`. 2007.06.01

[9] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G. (Eds.): Research Directions in Parallel Functional Programming, pp. 289-303, Springer-Verlag, 1999.

[10] Rabhi, F.A., Gorlatch, S. (Eds.): *Patterns and Skeletons for Parallel and Distributed Computing*, Springer Verlag, 2002.

[11] Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, and Nils Weskamp: *High-Level Process Control in Eden*, In: Kosch, H., Böszörményi L., Hellwagner H. (eds.): Parallel Processing, 9th International Euro-Par Conference, Euro-Par 2003, Proceedings, Klagenfurt, Austria, August 27-29, 2003, Springer Verlag, LNCS Vol. 2790, pp. 723-741.

[12] Ricardo Pena , Fernando Rubio , Clara Segura: *Deriving Non-Hierarchical Process Topologies*, Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01), p.51-62, August 22-24, 2001

[13] Best, E., Hopkins, R. P.: B$(PN)^2$ - a Basic Petri Net Programming Notation, In: Bode, A., Reeve, M., Wolf, G. (Eds.): *Parallel Architectures and Languages Europe, 5th International PARLE Conference, PARLE'93*, Proceedings, Munich, Germany, June 14-17, 1993, Springer Verlag, LNCS Vol. 694, pp. 379-390.