Általános architektúra vizsgálata

1. DemoGRID beszámoló

2001-12-15

Kivonat

A DemoGRID projekt első szakaszában egy-egy önmagában álló, mintegy 25 gépes cluster szoftverarchitektúrájának általános vizsgálatát végeztük el. Ennek keretében megvizsgáltuk azt, hogy milyen típusú szoftvereszközökkel hogyan lehet alkalmazások készítését támogatni. Megvizsgáltuk azt is, hogy milyen módszerekkel lehet a GRID erőforrásainak használatát, illetve a rajta futó alkalmazásokat monitorozni.

1. Bevezetés

Az általános architektúra vizsgálat keretében megvizsgáltuk az alábbi szofvereszközöket abból a szempontból, hogy milyen eszközökkel és hogyan támogatják GRID alkalmazások készítését.

- PVM [Fro01]
- MPI [Fro01, Her01]
- CORBA. Kifejlesztettünk egy speciális keretrendszert CORBA alkalmazásokhoz (ORB(M) [Ulb01]) és egy Clean-CORBA interfészt ([Var01]).
- Funkcionális programozási nyelvi eszközei Haskellben és Cleanben [Tej01, ZsHo01, Var01]. Haskell modulok Clean használatához készítettünk egy Haskell - Clean leképezést megvalósító előfordítót [Heg01].

Az alkalmazások futtatása során adatokat kell gyűjteni azok futásidejéről és erőforrásfelhasználásáról. Erre készült a P-GRADE program [P-GRADE].

2. Eszközök GRID alkalmazások készítéséhez

2.1. PVM

Az elosztott architektúrájú párhuzamos rendszerekben, aminek a GRID is egy megvalósulása, a PVM jelenti az egyik legelterjedtebb kommunikációs eszközkészletet. Használójának, a programozónak (aki általában valamilyen tudományos számítást végez az elosztott rendszer segítségével) nem nyújt olyan magasszintű szolgáltatásokat, mint mondjuk a CORBA technológia, de mindenféleképpen magasabbszintű eszköz, mint például a TCP/IP protokoll. Elterjedtsége miatt mindenféleképpen érdemes megvizsgálnunk alkalmazhatóságát GRID környezetben, és hatékonyságát, valamint kényelmességét össze kell hasonlítanunk más rendszerekével. Az ELTE Informatika Tanszékcsoport Programozási Nyelvek laboratóriumában épített cluster-en, ami 25 Linux-os PC-ből áll, üzembe helyeztünk egy PVM rendszert. Ebben a rendszerben néhány fontos párhuzamos algoritmus (asszociatív műveletek, például mátrixszorzás elvégzése; adatcsatornára visszavezethető számítások, például hatványsorok közelítése) különböző C és C++ nyelvű implementációit teszteltük [PVMa]. Megvizsgáltuk, hogy milyen számítás- és kommunikációigényű feladatoknál lehet a gyakorlatban is kimutatni a párhuzamos algoritmusok elméleti hatékonysági jellemzőit (például az asszociatív műveletek elvégzésénél n adatra O(n) processzoron O(log n) végrehajtási idő, vagy az adatcsatornára visszavezethető számításoknál „elég sok” adatra a megközelítőleg lineáris speedup).

2.2. MPI

Az ELTE, ITCS, Programozási Nyelvek laboratoriumának clusterén installáltunk MPI függvénykönyvtárat, amelyet C nyelvű alkalmazásokkal teszteltünk [Her01].

A tesztelés során a legtöbb alapvető MPI rutin működését ellenőríztük. Az üzenetküldési technikák közül a plain és csomagolt, ill. a pont-pont és a broadcast jellegű üzenetküldést teszteltük. Megméréstük a rövid (néhány byte), és a hosszú üzenetküldés (több megabyte), ill. a blokkoló és nem blokkoló változat költségeit. Ezeken kívül vizsgáltuk a standard kimenet, bemenet használhatóságát, valamint a file kezelés lehetőségeit MPI alkalmazásokból. Az alábbi problémákat fedeztük fel: - kiderült, hogy az stdout minden processz számára elérhető, de az stdin csak a 0-s rangú processz számára. - A szeparált file i/o nem működött, talán konfigurálási okoknál fogva. - A nagyobb méretű (>30k) üzenetek küldését a rendszer nem tudta megbízhatóan kezelni (alkalmanként lefagyott).

Az MPI kisebb gondoktól eltekintve működőképesnek bizonyult.

2.3. CORBA

A GRID rendszerek egyik legfontosabb kérdése, hogy alkalmazhatók-e a meglévő objektum orientált elosztott keretrendszerek, mint például az egyes CORBA implementációk. Vizsgálataink eredményeként bebizonyosodott, hogy a CORBA alapú rendszerek csak részben képesek megfelelni a GRID által támasztot speciális követelményeknek. A hiányosságok egyik oka, a hívási szemantikák számának kötöttsége, illetve a rendszer nehéz adaptálhatósága a speciális feladatokhoz (pl. nagy mennyiségű adat mozgatása) és a számítási környezethez. A projekt keretében kifejlesztett ORB(M) rendszer, illetve mögötte álló (PSE - Pluggable Semantic Elements) modell ezen hiányosságokat küszöböli ki. A rendszer lehetövé teszi a felhasználó számára saját, a környezetet jobban kihasználó hívási szemntikák implementálását. Méréseink szerint a speciális szemantikák (GRID, multimédia, mobil és csoport kommunikáció) teljesítménye esetenként kétszerese is lehet az egyik leggyorsabb CORBA implementációnak (omniORB3).

2.4. Funkcionális programozási nyelvek eszközei

2.4.1. Clean

A Concurrent Clean (lusta kiertekelesű) funkcionális programozási nyelv a következő alapvető párhuzamos programozási nyelvi elemekkel rendelkezik: annotációk, üzenetküldő és fogadó csatornák, párhuzamos folyamatok, a TCP protokollon alapuló kommunikáció elemei, valamint párhuzamos kiértékelési stratégiák. Az ELTE, ITCS, Programozási Nyelvek laboratoriumának clusterén a Concurrent Clean, Linux operációs rendszerre írt 1.3.3.-as fordítóját installáltuk, mely a fent említett elemek közül a TCP protokollon alapuló kommunikáció elemeit tartalmazza. A további nyelvi elemeket a jelenlegi fordító nem támogatja, így azokat implementálni kell. Ezen nyelvi elemek segítségével felépíthetőek a GRID számításokhoz szükséges, paraméterezhető, általános párhuzamos számítási sémák, a szkeletonokat [ZsHo01].

Ezen túlmenően késztettünk egy Clean-CORBA interfészt is. Ily módon a Clean nyelven készített programok adatokat tudnak cserélni a CORBA szabványnak megfelelően bármely más nyelvű, CORBA interfészt használó alkalmazással is [Var01]

Haskellhez sok hasznos könyvtárat írtak, így Clean nyelvű alkalmazásokhoz ezek használhatóvá tétele jelentősen bővíti a Cleanben elkészíthető alkalmazások körét is. Ezen feladat megoldására készül egy Haskell-Clean előfordító, amely lehetővé teszi Haskell modulok Cleanből történő használatát [Heg01] A front-end a Clean 2.0 forditohoz készült, és képes Haskell kódot elemezni és az elemzett kódból a Clean 2.0 fordító számára használható szintaxisfát generálni. Ehhez szükséges a két nyelv összehasonlítása és a különféle Haskell struktúrákkal ekvivalens Clean struktúrák megtalálása. A front-end részei: lexikografikus elemző, ami tokenizálja a Haskell kódot és alapvető konverziókat végez a tokeneken (pl. az oktális vagy hexadecimális kóddal adott karaktereket átalakítja a Clean szintaxisának megfelelő formára); szintaktikus elemző, ami elemzi a tokensorozatot és létrehozza a Clean fordító számára használható szinaxisfát; preprocesszor, ami az előző két elemzővel föl nem oldható problémák egy részét szünteti meg.

2.4.2. Haskell

A Haskell funkcionális nyelvhez több, egymástól független kiegészítést is készítettek, melyek az elosztott működést segítik. Megvizsgáltuk, hogy ezek miben kulönböznek egymástól, illetve a konkrét környezetünkben melyiket célszerű alkalmazni. Jelenleg a vizsgált könyvtárak közül a Distributed Haskell bizonyult a számunkra leghasználhatóbbnak. Ezt a könyvtárat, illetve a használatához szükséges Glasgow Haskell Compiler 4.08-as változatát fel is telepítettük a rendszerünkre, és ki is próbaltuk egy egyszerű kliens - szerver alkalmazással.

3. Futó alkalmazások monitorozása GRID környezetben

Két módon fordulhat elő, hogy egy alkalmazásnak nem elegendőek a helyi erőforrások. Az egyik az, ha a futtatónak nincs elég kapacitása, de máshol van. A másik, ha sehol sincs akkora önálló kapacitás, ami elég lenne az alkalmazás lefuttatásához. Az első eset kevésbé érdekes, mert át lehet vinni az alkalmazást az erősebb erőforrásra; a második esetben azonban az egyetlen megoldás, ha az alkalmazás egyszerre több erőforrást is használ, azaz párhuzamosan működik.

A párhuzamos programozásnak nagy elméleti és gyakorlati tapasztalati háttere van. A legnagyobb gond, hogy számos probléma, ami szekvenciális esetben triviális, párhuzamosság megengedése esetén bonyolulttá válik. Ez azt eredményezi, hogy párhuzamos alkalmazások fejlesztése sokkal nagyobb szakértelmet igényel, mint a szekveniálisoké.

A SZTAKI által szuperszámítógépekre és klaszterekre kifejlesztett P-GRADE szoftver [P-GRADE] célja, hogy ezt a bonyolultságot megpróbálja csökkenteni. A P-GRADE egy egyszerűen használható grafikus fejlesztőkörnyezet, aminek segítségével könnyen és egyszerűen készíthetők párhuzamos programok, a fejlesztőknek nem kell az alacsony szintű részletekkel foglalkozniuk. A rendszer része egy integrált parallel debugger is, ami nagymértékben megkönnyíti a program tesztelését és a hibakeresést.

Hivatkozások

[P-GRADE] A Graphical Programming Environment for Multicomputers, 9 pages (dg-rep-1-arch-grade).

[Ulb01] Ulbert A.: The Architecture and Design of the ORB(M) Extensible Distributed Object Framework, October 27, 2001, 12 pages (dg-rep-1-arch-orbm).

[Var01] Varga Z.: Developing parallel applications using Clean and the Clean-CORBA interface, 2001, 2 pages (dg-rep-1-arch-cleancorba).

[ZsHo01] Zsók, V. - Horváth, Z.: Skeleton Algorithms in Concurrent Clean, draft paper, 2001, 6 pages (dg-rep-1-arch-skel).

[Fro01] Frohner Á.: Párhuzamos Programozási Eszközök Összehasonlítása, 2001 (dg-rep-1-arch-tools).

[Her01] Hernyák Z.: MPI tesztelése a nyelvi laboratóriumban, 2001.11.18., 4 oldal (dg-rep-1-arch-mpitest).

[PVMa] Balázs G.: Párhuzamos programok implementációja PVM/cluster környezetben - mérési eredmények, 10 oldal (dg-rep-1-arch-pvmtest).

[Tej01] Tejfel M.: A Haskell nyelv vizsgálata GRID környezetben készülő alkalmazások fejlesztéséhez, 2001, 1 oldal (dg-rep-1-arch-haskell).

[Heg01] Hegedűs H.: Transforming Haskell Structures to Clean, 2001, 6 pages (dg-rep-1-arch-h2clean).