A párhuzamos programozás jelene

Legyen az ember akár tudós, grafikus, zenész vagy filmrendező, könnyen lehet, hogy munkája során egyszer hasznosnak találja majd a mai nagy teljesítményű Beowulf klaszterek képességeit.

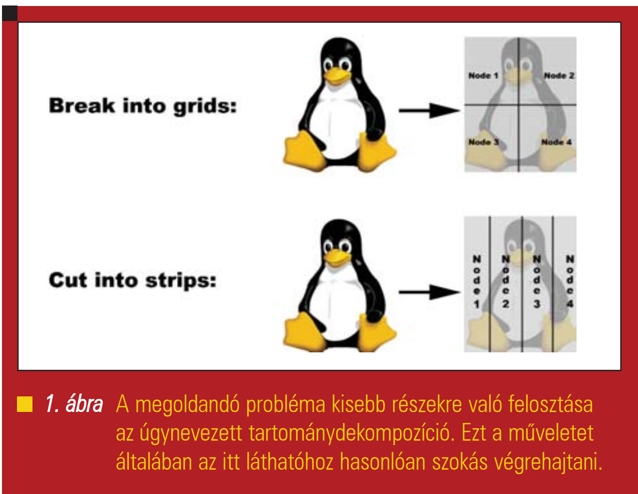
Egy tipikus párhuzamos rendszer

Először is szükségünk lesz néhány azonos számítógépre, amelyeken ***Linux*** fut, és amelyek nagy sebességű ***Ethernet*** hálózattal vannak összekötve. A legjobb a ***Gigabit Ethernet***, hiszen a hálózat sebessége az egyik olyan dolog, ami erősen visszafoghatja egy klaszter teljesítményét. Szintén szükségünk lesz valamilyen elosztott fájlrendszerre, és persze a klasztertechnológiával kapcsolatos könyvtárakra, szoftverekre. A legtöbb klaszterben egyszerűen ***NFS***-t használnak elosztott fájlrendszerként, bár tény, hogy létezik néhány ennél egzotikusabb megoldás is. Ilyen például az ***IBM GPFS*** (***General Parallel Filesystem***) rendszere. Ami a klaszteren történő számítások szoftveres támogatását illeti, itt is van néhány választási lehetőségünk. Napjainkban a de facto szabvány az ***MPI*** (***Message Passing Interface***), de a ***PVM*** (***Parallel Virtual Machine***) könyvtár is kiválóan működik. Az elmúlt

Amikor ***Donald Becker*** az 1990-es évek elején a ***NASA***-nál dolgozva felvetette a ***Beowulf klaszter*** megépítésének lehetőségét, mindörökre megváltoztatta a nagy teljesítményű számítógépek fejlődéstörténetét.
Ez az ötlet ugyanis nem kevesebbet jelentett, mint hogy a korabeli szuperszámítógépek teljesítményét azok árának töredékéért is el lehetett érni. Korábban ha egy szervezet egy ilyen szuperszámítógépet szeretett volna beszerezni, akkor egy több millió dolláros számlára számíthatott.
Egy ugyanekkora teljesítménnyel bíró ***Beowulf klasztert*** viszont már néhány százezerből is meg lehetett építeni, ami még mindig nem kevés, de az előbbi összeghez viszonyítva szinte baráti ár. Ha vetünk egy pillantást a ***TOP500***-as listára (a világ ötszáz leggyorsabb szuperszámítógépe), láthatjuk, hogy ez az egyszerű ötlet mekkora hatással volt a számítástechnika fejlődésére. A ***Beowulf klaszterek*** két legfontosabb közös tulajdonsága hogy közönséges, vagyis boltban bárki által megvásárolható alkatrészekből állnak, valamint hogy ***Linux*** fut rajtuk. Elterjedésüknek amúgy volt egy – eredetileg talán nem is sejtett – hatása is: szerte a világon megmozgatták a legjobb programozók fantáziáját.
Ennek ellenére számos ember a mai napig úgy gondolja, hogy a ***Beowulf klaszterek*** a mindennapi munkára alkalmatlanok, vagy legalábbis nem könnyű megtalálni a helyüket ezen a téren. Ebben persze van némi igazság is. Én például garantáltan nem adnék pénzt a ***Quake 4*** egy olyan változatáért, ami képes egy ilyen klaszteren futni. Az igazi felhasználások spektrumának egyik végén jelenleg az olyan filmes vállalkozások állnak, mint például a ***Pixar***, amelyek a legújabb filmekben alkalmazott digitális trükköket ilyen klasztereken számoltatják ki, a másikon pedig azok a tudósok, akik a magreakcióktól kezdve az emberi genomig megszámlálhatatlanul sok dolgot kutatnak a segítségükkel. Bátran állíthatjuk tehát, hogy a fő felhasználások valóban elég távol esnek a mindennapi élettől. Ugyanakkor jó hír, hogy némi programozási tudással a klasztereket nem csak a tudósok és a hollywoodi stúdiók tudják kihasználni, hanem akár mi, közönséges földi halandók is.
Persze mielőtt párhuzamosítani kezdenénk egy alkalmazást, előbb nem árt elgondolkodni azon, mekkora is lesz a nyereség. Párhuzamos programot az ember általában azért ír, mert a feldolgozandó adatmennyiség nem fér el egyetlen ***PC*** memóriájában, vagy mert az elvégzendő számítás túlságosan hosszú ideig tartana, ha egyetlen processzor hajtaná végre. Mármost ha egy program párhuzamos változata egy másodperccel rövidebb idő alatt fut le, azért szinte biztosan nem éri meg az algoritmus átírásával tölteni az időnket. Ugyanakkor – amint azt a cikkben bemutatott példával demonstrálni fogom – jócskán akadhatnak olyan helyzetek is, amikor a párhuzamosítás egészen kis munka befektetésével elvégezhető, az eredmény pedig kifejezetten látványos.
Van az algoritmusoknak egy egész csoportja, amelyek bár olyan, első látásra erősen különböző területekről származnak mint a képfeldolgozás vagy a hangjelek átalakítása, ugyanazokkal a módszerekkel ugyanolyan könnyen felbonthatók részfeladatokra. A cikkben ezt a felbontási módszert fogom bemutatni: egy ***Tux***-ot ábrázoló képre fogunk alkalmazni egy egyszerű konvolúciós szűrőt.

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva



1. ábra A megoldandó probléma kisebb részekre való felosztása az úgynevezett tartománydekompozíció. Ezt a műveletet általában az itt láthatóhoz hasonlóan szokás végrehajtani.

időszakban meglehetősen sokak figyelme fordult a ***MOSIX*** és ***openMOSIX*** megoldások felé, de általánosságban elmondható, hogy ezeket elsősorban nem kifejezetten klaszterekre íródott programok futtatására használják, hanem arra, hogy áthidalják a szakadékot a szekvenciális és párhuzamos világ között. Szintén közös jellemzőjük, hogy működésük során a többszálú programok szálait osztják szét fizikailag elkülönült csomópontok között. Ebben a cikkben a továbbiakban feltételezem, hogy az olvasó rendelkezik egy telepített és működő ***MPI*** rendszerrel, bár a párhuzamosítás logikája a ***PVM*** használatakor is teljesen hasonló. Akinek esetleg teljesen új az ***MPI***, és még soha nem telepített ilyen rendszert, az olvassa el a ***Linux Journal*** webhelyén ***Stan Blank*** és ***Roman Zaritski*** cikkét a témáról. Ők ketten kiválóan leírták, hogyan kell egy ***MPI*** rendszert üzembe helyezni.

A program inicializálása

Valamennyi ***MPI*** program elején kötelezően végre kell hajtanunk néhány olyan rutint, amelyek beállítják a csomópontok közti kommunikációt, és megállapítják minden egyes csomópont rangját (***rank***). A rang egy egész szám, amely a kérdéses gépet egyedileg azonosítja a klaszterben. A számozás nullától indul, és a gépek száma mínusz egyig folytatódik. A nullás rangú csomópont általában a klaszter központja, vagyis ez irányítja az összes többi csomópont munkáját is. Aztán ha a csomópontok mindegyike elvégezte a rá kiszabott feladatot, akkor – szintén kötelezően – végre kell hajtanunk egy záró függvényhívást is, mielőtt a program kilépne. Egy ***MPI*** program váza tehát a következőképpen fest:

```
#include <mpi.h>
#include <stdlib.h>
int main (void) {
           int myRank, clusterSize;
           int imgHeight, lowerBoundY,
            ➥upperBoundY,
              boxSize;
          // Initialize MPI
         MPI_Init((void *) 0, (void *) 0);
         // Get which node number we are.
         MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
         // Get how many total nodes there are.
         MPI_Comm_size(MPI_COMM_WORLD,
           ➥&clusterSize);
        // boxSize - the amount of the image
        //each node
        //                  will process
        boxSize = imgHeight / clusterSize;
         // lowerBoundY - where each node starts
         //processing.
         lowerBoundY = myRank*boxSize;
        // upperBoundY - where each node stops
        //processing.
        upperBoundY = lowerBoundY + boxSize;
        // Body of program goes here
        // Clean-up and exit:
        MPI_Finalize(MPI_COMM_WORLD);
        return 0;
}
```

Ez a kód valamennyi csomóponton önállóan fut, vagyis a `lowerBoundY` és az `upperBoundY` értéke minden gépen más és más lesz. Ezt a következő szakaszban természetesen ki is fogjuk használni.

A feldolgozandó kép felbontása

Ha egy digitálisra képre egy konvolúciós szűrőt alkalmazunk, a művelet percekig, vagy akár órákig is eltarthat a szűrő bonyolultságától, a kép nagyságától és a számítógép sebességétől függően. Ezen egy klaszter birtokában nyilván úgy segíthetünk, ha a képet kisebb darabokra bontjuk, és ezeket szétosztjuk több számítógép között. Az *1. ábrán* ennek a legegyszerűbb és ezért leggyakrabban alkalmazott módját láthatjuk: a képet csíkokra bontjuk. Ha tehát van egy nagy méretű digitális képünk, akkor *C/C++* nyelv használatát feltételezve a probléma particionálása a következőképpen oldható meg:

```
FILE *imageFile = fopen("image_in.ppm", "rb");
// Safety check.
if (imageFile != NULL) {
   // Read in the header.
   fread(imageHeader, sizeof(char),
                    HEADER_LENGTH, imageFile);
   // fseek puts us at the point in the image
   // that this node will process.
   fseek(imageFile, lowerBoundY*WIDTH*3,
             SEEK_SET);
   // Here is where we read in the colors:
   // i is the current row in the image.
   // j is the current column in the image.
   // k is the color, 0 for red, 1 for blue,
   //    and 2 for green.
   for (i=0; i<boxSize+1; i++) {
      for (j=0; j<WIDTH; j++) {
         for(k=0; k<3; k++) {
           fread(&byte, 1, 1, imageFile);
            pixelIndex = i*WIDTH+j+k;
            origImage[pixelIndex] = byte;
         }
      }
   }
}
fclose(imageFile);
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

2. ábra Az élszűrés (edge detect) művelet ezzel a konvolúciós mátrixszal írható le. A piros négyszög az éppen feldolgozás alatt álló pixelnek fele meg, a többi szám pedig a szomszédos pixelek értékének felel meg.

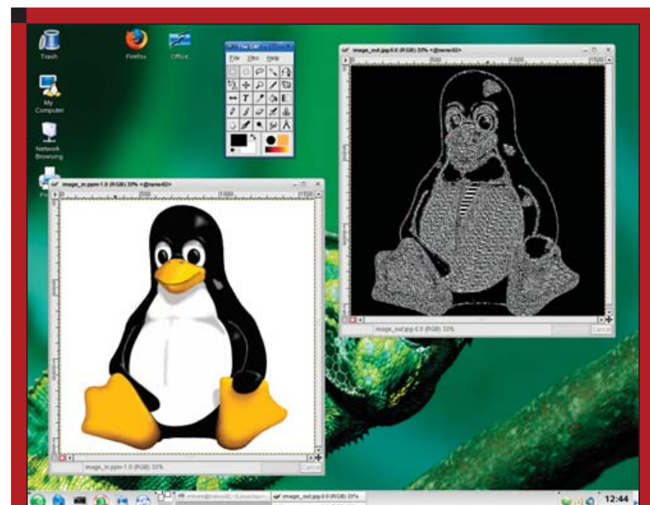
A szűrő alkalmazása

Most, hogy már mindegyik csomópont megkapta a képnek azt a részét, amit neki kell feldolgoznia, nekiláthatunk a tényleges munkának, vagyis a szűrő alkalmazásának. Hogy hogyan is kell végigszámoltatni egy konvolúciós szűrőt, az kiválóan le van írva a *GIMP* dokumentációjában. Ami azt illeti számos olyan képfeldolgozási eljárás van, amelyek tulajdonképpen egy-egy ügyesen kitalált konvolúciós mátrixnak feleltethetők meg. Ilyen például az élesítés (***sharpen***), az elmosás (***blur***), a Gauss-féle elmosás (***Gaussian blur***), az élszűrés (***edge detect***) vagy az élkiemelés (***edge enhance***). A konvolúciós szűrő úgy működik, hogy minden egyes pixel értékét a saját és a szomszédai értékének függvényében változtatja meg. Ebben a cikkben az élszűrés (***edge detect***) szűrőt fogjuk alkalmazni. Az ehhez tartozó mátrixot a 2. *ábra* mutatja.

A szűrő alkalmazása jelen esetben azt jelenti, hogy minden egyes pixel értékét megszorozzuk -4-gyel, majd az így kapott értékhez hozzáadjuk a fölötte, alatta, valamint a tőle jobbra és balra levő pixelek értékét. Ez lesz az adott pixel új értéke. Mivel a mátrix sarkaiban csupa nulla van, ezért a hatékonyság növelése érdekében a számítások során nem is vesszük figyelembe valamennyi mátrixelemet, ami szigorú matematikai értelemben csalás ugyan, de esetünkben az eredményen mit sem változtat. Az alábbi kódrészlet a szűrő alkalmazását, míg a 3. *ábra* a végeredményül kapott képet mutatja:

```
for (i=0; i<boxSize; i++) {
   for (j=0; j<WIDTH; j++) {
      if (i>0 && i<(HEIGHT-1) &&
          j>0 && j<(WIDTH-1)){
        // Now we apply the filter matrix
        // First to the current pixel.
        pixelIndex = i*WIDTH + j;
        r = origImage[pixelIndex];
        g = origImage[pixelIndex+1];
        b = origImage[pixelIndex+2];
        filter_r = -4*r;
        filter_g = -4*g;
        filter_b = -4*b;
        // Next to the left neighbor.
        pixelIndex = i*WIDTH + j - 1;
        r = origImage[pixelIndex];
        g = origImage[pixelIndex+1];
        b = origImage[pixelIndex+2];
        filter_r += 1*r;
        filter_g += 1*g;
        filter_b += 1*b;
        // Next to the right neighbor.
        pixelIndex = i*WIDTH + j + 1;
        r = origImage[pixelIndex];
        g = origImage[pixelIndex+1];
        b = origImage[pixelIndex+2];
        filter_r += 1*r;
        filter_g += 1*g;
        filter_b += 1*b;
        // The neighbor above.
        pixelIndex = (i-1)*WIDTH + j;
        r = origImage[pixelIndex];
        g = origImage[pixelIndex+1];
        b = origImage[pixelIndex+2];
        filter_r += 1*r;
        filter_g += 1*g;
        filter_b += 1*b;
        // The neighbor below.
        pixelIndex = (i+1)*WIDTH + j;
        r = origImage[pixelIndex];
        g = origImage[pixelIndex+1];
        b = origImage[pixelIndex+2];
        filter_r += 1*r;
        filter_g += 1*g;
        filter_b += 1*b;
      }
      // Record the new pixel.
      pixelIndex = i*WIDTH + j;
      filterImage[pixelIndex]   = filter_r;
      filterImage[pixelIndex+1] = filter_g;
      filterImage[pixelIndex+2] = filter_b;
   }
}
```

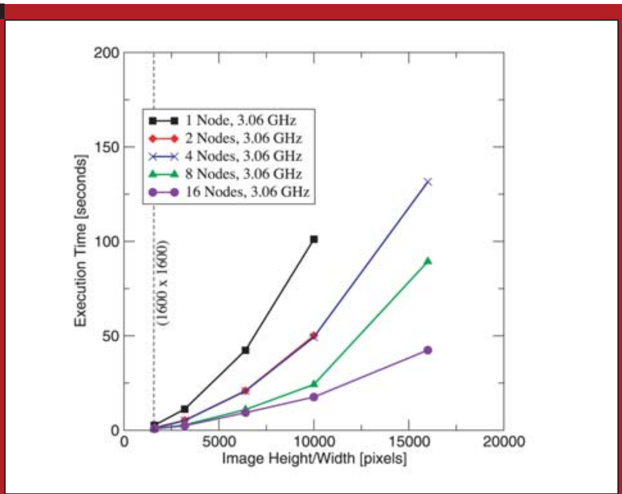
A `readImage()` rutinnak természetesen kell legyen egy `writeImage()` megfelelője is, amely lemezre írja a kép részleteit.



3. ábra Balra az eredeti kép látható, míg a jobb oldali ábra az élszűrés eredményét mutatja

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva



4. ábra A működési idő függése a kép méretétől és a klaszter csomópontjainak számától. A kép mérete 1.600x1.600 pixeltől 16.000x16.000 pixelig változott. Így a legnagyobb kép feldolgozásához legalább négy csomópontból álló klaszterre volt szükség.

A kód lefordítása és futtatása

Az ***MPI*** mindkét, ***Linux*** alatt elérhető megvalósítása (***LAM*** és ***MPICH***) tartalmaz olyan szkripteket, amelyek segítségével a felhasználó könnyebben fordíthatja le az általa írt alkalmazást úgy, hogy ahhoz a megfelelő ***MPI*** könyvtárak is hozzálinkelődjenek. Ezekkel tulajdonképpen a megfelelő kapcsolókat adhatjuk át a ***GCC***-nek pont ugyanúgy, ahogy azt általában is tesszük. Az egyes nyelvekhez a következő szkriptek használhatók:

`mpicc` : ***C*** nyelvű programok
`mpi++` : ***C++*** programok
`mpif77` : ***FORTRAN 77*** programok

Az elkészült kódot az `mpirun` parancs segítségével futtathatjuk. Ha tehát egy programot ***parallel.c***-nek hívnak, akkor a fordítását a

```
mpicc -O3 -o parallel parallel.c
```

paranccsal, míg a futtatását a

```
mpirun n0 ./parallel
```

paranccsal végezhetjük. Utóbbiban az `n0` azt jelenti, hogy a programot kizárólag a nullás rangú csomóponton kell futtatni. Ha több csomópont között akarjuk elosztani a munkát, akkor egy nyolc processzoros rendszer esetében az `n0...n7` értékeket használhatjuk. Ha pedig azt akarjuk jelezni a rendszernek, hogy az összes aktuálisan rendelkezésre álló számítási kapacitást ki akarjuk használni, akkor az

```
mpirun C
```

parancsot kell használnunk.

Mekkora többletteljesítményt nyertünk?

Most tehát ott tartunk, hogy néhány egészen egyszerű ***MPI*** hívással párhuzamosítottunk egy olyan programot, ami egy konvolúciós szűrőt tud alkalmazni egy digitális képre. A lényegesebb kérdés mostantól nyilván az, hogy miért is érte ez meg nekünk? Vajon mennyivel nagyobb teljesítményt nyertünk? A nyereségnek persze többféle mértékegysége lehet attól függően, hogy kinek mi a fontosabb. Nyerhetünk teljesítményt úgy, hogy gyorsabban fut le a programunk, de az is előfordulhat, hogy valakinek nem ez az igazán lényeges szempont, hanem hogy mennyi számítást tud egyszerre elvégeztetni a rendszerével. Ha például van egy 16.000x16.000 pixel méretű digitális képünk, akkor ennek a betöltéséhez egy 768.000.000 elemű tömbre lesz szükségünk. Ez pedig egyszerűen túl nagy. A ***GCC***-től csak egy nyájas hibaüzenetet fogunk kapni, hogy nem tud ekkora tömböt létrehozni. (***A probléma máshogyan persze megoldható – a szerk.***) Ha viszont az imént bemutatott módon fölszabdaljuk a képet több kisebb csíkra, az egyes darabok már kezelhetőnek bizonyulnak.
A cikkben bemutatott kódot egy 16 csomópontból álló ***Beolwulf*** klaszteren teszteltem. Minden csomópontnak ***1 GB*** memóriája és ***3.06 GHz***-es ***Pentium 4*** processzora volt, és valamennyin ***Fedora Core 1*** futott. Az összeköttetést ***Gigabit Ethernet*** biztosította, a csomópontok közti adatcsere pedig egy ***NFS*** partíció segítségével volt megoldva. A kép beolvasásához, feldolgozásához és lemezre való visszaírásához szükséges idő nagyságát a *4. ábrán* láthatjuk.
Amint azt a *4. ábrán* jól látható, a feldolgozás párhuzamosítása már a kisebb méretű képek esetében is jelentős gyorsulást eredményezett, az igazi előny azonban a kifejezetten nagy képeknél látható. Ami azt illeti, a 10.000x10.000 pixelnél nagyobb képek esetében maga a feldolgozás szekvenciális üzemmódban el sem végezhető a korábban említett memóriakorlát miatt, így ezekben az esetekben egy legalább négy csomópontból álló klaszteren kellett a futtatást végezni. Az ábráról azt is leolvashatjuk, hol volt értelme a párhuzamosításnak, és hol nem. Ami azt illeti, 1.600x1.600 pixeltől 3.200x3.200 pixelig nincs különösebben nagy eltérés a futás sebességében. Ezek a képek ráadásul annyira kicsik, hogy a memória sem jelent korlátot, vagyis a párhuzamosítás még ezzel az érvvel sem támasztható alá.
Hogy a kvalitatív értékelésen túl néhány számot is mondjak, egyetlen ***3,06 GHz***-es processzorral szerelt gépen egy 6.400x6.400 pixeles kép beolvasása, feldolgozása és lemezre való visszaírása körülbelül 50 másodpercig tart.
Egy ugyanilyen csomópontokból álló 16 processzoros klaszter ezzel szemben 10 másodperc alatt végez ugyanezzel a művelettel. Mi több, egy 16 processzoros klaszter még a 16.000x16.000 pixeles képpel is hamarabb végez, mint az egyprocesszoros gép a 6,25-ször kisebb képpel.

Lehetséges gyakorlati alkalmazások

Ebben a cikkben a nagy teljesítményű ***Beowulf*** klaszterek felhasználásának csupán egyetlen lehetőségét tudom bemutatni, ugyanakkor az alkalmazott alapelvek és módszerek mindenütt azonosak. A feldolgozandó adatok szintjén párhuzamosítható alkalmazások mindegyike pontosan úgy működik, mint a most bemutatott képfeldolgozó eljárás: minden csomópont beolvassa az adatok egy részét, feldolgozza azokat, majd vagy visszaküldi az eredményt a központi csomópontnak, vagy maga írja ki azt lemezre. A következőkben felsorolok négy olyan területet, ahol a párhuzamosításra meglátásom szerint nagy jövő vár.

© Kiskapu Kft. Minden jog fenntartva

Néhány alapvető és hasznos MPI szubrutin

Az *MPI* könyvtár több mint 200 függvényből áll, és mindegyik hasznos bizonyos helyzetekben. Azért van ilyen sok eleme ennek az interfésznek, mert az *MPI* számos különböző architektúrán képes működni, és így meglehetősen sokféle igénynek kell megfelelnie egyszerre. A következőkben felsorolom közülük azt a néhányat, amelyek a cikkben bemutatotthoz hasonló párhuzamos algoritmusok megvalósítása során hasznosak lehetnek.

`MPI_Init((void*) 0, (void*) 0)` – Inicializálja az *MPI* rendszert.
`MPI_Comm_size(MPI_COMM_WORLD, &clstSize)` – A `cltSize` (egész) változóban visszaadja a klaszter méretét.
`MPI_Comm_rank(MPI_COMM_WORLD, &myRank)` – A `myRank` (egész) változóban visszaadja a kérdéses csomópont rangját.
`MPI_Barrier(MPI_COMM_WORLD)` – Megállítja a végrehajtást mindaddig, amíg a klaszter valamennyi csomópontja el nem jut a kódnak erre a pontjára.
`MPI_Wtime()` – Visszaadja egy önkényes, múltbeli időpillanat óta eltelt időt. A szubrutinok és egyéb folyamatok időzítésére használható.
`MPI_Finalize(MPI_COMM_WORLD)` – Leállít valamennyi *MPI* folyamatot. Ezt a rutint valamennyi programnak meg kell hívnia leállás előtt.

1. ***Képszűrők:*** Amint azt a fenti példa is jól demonstrálta, a párhuzamos feldolgozás kiválóan használható a képkezelő eljárások terén, hiszen nem csak felgyorsítja a folyamatokat, hanem lehetőséget ad kifejezetten nagy képek kezelésére is. Éppen ezért nyilván komoly előrelépést jelentene, ha az olyan képfeldolgozó alkalmazásokhoz, mint amilyen például a ***Gimp*** elkészülne egy párhuzamos szűrőket tartalmazó csomag.

2. ***Hangfeldolgozás:*** Szűrési módszereket nem csak a képek, hanem a hangfájlok feldolgozásában is használnak, és ezek is jelentős feldolgozási időt igényelnek. Éppen ezért az olyan nyílt forrású alkalmazások, mint az ***Audacity*** szintén sokat profitálhatnak a párhuzamosítási módszerek alkalmazásából.

3. ***Adatbázis műveletek:*** A kifejezetten nagy mennyiségű adat feldolgozásával járó műveletek párhuzamosíthatók, és ez által felgyorsíthatók úgy, hogy az egyes csomópontok olyan részlekérdezéseket futtatnak, amelyek a szükséges adatoknak csak egy részét adják vissza. Ez után valamennyi csomópont el is végezheti az általa megszerzett adatokon a szükséges műveleteket.

4. ***Biztonsági rendszerek:*** A klaszterek segítségével a rendszergazdák sokkal gyorsabban győződhetnek meg arról, mennyire biztonságos jelszavakat használnak a felhasználók. A ***/etc/shadow*** tartalmának nyers erővel történő visszafejtése köztudottan elég időigényes, ha azonban a feladatot célszerűen szétosztjuk egy klaszter csomópontjai között, sokkal gyorsabban is lefut a teszt. Ezzel aztán nem csak időt nyerünk, hanem a lelkünk nyugalma is visszatérhet, hiszen pontos képet kapunk rendszerünk biztonságáról.

Záró megjegyzések

Őszintén remélem, hogy ezzel a cikkel sikerült rámutatnom, miért gondolom úgy, hogy a párhuzamos programozásnak a mindennapi életben is hely van. Fel is soroltam néhány olyan területet, ahol a párhuzamosítás előnyeit feltehetőleg már a közeljövőben ki fogják használni a fejlesztők és a felhasználók egyaránt. Egyúttal arról is meg vagyok győződve, hogy ilyen potenciális alkalmazási terület még számos akad.

Létezik néhány olyan alapelv, amelyek betartása általában biztosítja azt, hogy a párhuzamos kód hatékonyabban működhessen, mint szekvenciális elődе. Az első ilyen ökölszabály a hálózati adatforgalom minimális szinten tartása. A csomópontok közti adatcsere általában sokkal több időt vesz igénybe, mint a magukon a gépeken elvégezhető műveletek. A fenti bemutatott példában a csomópontok között egyáltalán nem volt kommunikáció, hiszen az általuk végrehajtott műveletek logikailag teljesen függetlenek voltak egymástól. Ugyanakkor ez inkább a kivétel, nem a szabály. A legtöbb esetben a csomópontoknak a számítások során is adatokat kell cserélniük. A második szabály a bemenő adatok kezelésére vonatkozik: ha egy csomópontnak adatokat kell beolvasni a lemezről, csak annyit adatot olvastassunk be vele, amennyire ténylegesen szüksége van. Ezzel nyilván hatékonyabban használhatjuk a memóriát és a művelet sebessége is nő. Végezetül legyünk óvatosak olyan esetekben, amikor a csomópontoknak szinkronban kell maradniuk a számítások során, ez ugyanis a klaszterekben nem történik meg automatikusan. Egyes gépek kicsit gyorsabban működnek, míg mások lassabban, vagyis a szinkronizációval kapcsolatban nem élhetünk semmiféle előfeltevéssel. A keretes részben összefoglaltam néhány olyan ***MPI*** rutint, amelyekkel könnyen megoldhatjuk a csomópontok szinkronizációját.
Azt gondolom, hogy a jövőben a klaszterek egyre nagyobb szerepet játszanak majd mindennapi életünkben is. Egyúttal remélem, hogy ezzel a cikkel sikerült meggyőznöm az olvasót arról, hogy a párhuzamos alkalmazások fejlesztése nem különösebben ördöngös feladat, másrészt pedig az így kapott teljesítménytöbblet messze megér a befektetett munkát, és lehetőséget teremt egészen különleges területeken való alkalmazásra is.

Végezetül szeretném köszönetemet kifejezni ***Dr. Mohamed Larajdinak***, aki lehetővé tette, hogy programjaimat a ***Memphisi Egyetem Fizika Tanszékének Beowulf*** klaszterén teszteljem.

Linux Journal 2006. szeptember, 149. szám

Kapcsolódó anyagok: ➲***www.linuxjournal.com/article/9135***

Michael-Jon Ainsley Hore