Perl-programok teljesítménynövelése

Négy alaplépés, amellyel elkészült programjainkat felturbózhatjuk.

Fejlesztőtársam és jómagam egy olyan adatgyűjtő programon dolgoztunk, amelyik jobbára Perl nyelven íródott. Az alkalmazás mérési adatfájlokat gyűjtött össze könyvtárakból, értelmezte őket, elvégzett néhány statisztikai számítást, majd az eredményeket adatbázisba írta. Ahhoz azonban, hogy a termelési időszak alatt várható jelentős terhelést is elviselje, programunk teljesítményét meg kellett növelnünk.

E cikkből megismerhetjük a teljesítménynövelés négy alaplépését: az azonosítást, a teljesítménymérést, az újraírást (refactoring) és az ellenőrzést. Teljesítménynövelés céljából ezeket a lépéseket alkalmazhatjuk egy meglévő programra. Megállapítjuk, hogy melyik függvény okozhat teljesítménycsökkenést, majd alapszintű teljesítménymérést végzünk rajta, különféle teljesítménynövelő módosításokat hajtunk végre, végül a változtatások eredményét összehasonlítjuk alapszintű eredményeinkkel.

Teljesítménygondok beazonosítása

Amikor egy program teljesítményét megpróbáljuk növelni, az első lépés annak megtalálása, hogy mely része nem teljesít megfelelően. Erre a célra két eltérő módszert szoktam alkalmazni: a kódátnézést és a megfigyelést (profiling).

A teljesítménynövelési célú kódátnézés célja felderíteni a kódban található gyanús részeket. A kódátnézés előnye, hogy a vizsgálódó a programon belül figyelheti meg az adatok áramlását. Az adatok áramlásának megértése nagymértékben segítheti az eltávolítható vezérlési ciklusok felderítését, egyúttal megkönnyíti a később programmegfigyelés alá veendő kódrészletek beazonosítását. Nem javaslom, hogy a teljesítményvizsgálati kódátnézést egyéb célú, például szabványossági kódátnézéssel keverjük.

A programmegfigyelés folyamata során a program futását vizsgáljuk, hogy megállapítsuk, hol tölt el nagyobb időt, illetve hogy az egyes műveleteket milyen gyakran hajtja végre. Erre a célra a `Benchmark::Timer` Perl csomagot használtam; a csomagban található függvények segítségével megjelölhető a kérdéses kódrészlet eleje és vége. Az összes ilyen megjelölt kódrészletet egy-egy címke azonosítja. Amikor a program futtatása során egy megjelölt részletbe lépünk, a kérdéses szakasz által felhasznált időmennyiség rögzítésre kerül.

A programunkba illesztett megfigyelőrészek módszere eléggé erőszakos; ugyanis megváltoztatja a kód viselkedését. Más szóval, könnyen előfordulhat, hogy a megfigyelő kód elfed vagy meghamisít valamilyen teljesítménygondot (Performance Problems). A teljesítménynövelés korai szakaszában ez nem feltétlenül okoz nehézséget, hiszen a teljesítmény problémája nagyságrendileg nagyobb, mint a megfigyelő kód hatása a teljesítményre. A teljesítménygondok kiküszöbölése után azonban egyre valószínűbb, hogy a további teljesítménykérdéseket már nehezebb lesz megkülönböztetni. Mint sok más dolog, a teljesítmény növelése is iteratív folyamat.

Esetünkben néhány rész megfigyelése azt mutatta, hogy jelentős idő fordítódik a gépekről begyűjtött adatok statisztikáinak elkészítésére. Átnéztem a statisztikák készítéséhez kapcsolódó kódokat, és kiderült, hogy igen gyakran használjuk a normálszórást előállító `std_dev` függvényt.

A `std_dev` két okból is szemet szúrt: először is azért, mert a normálszórás számításához a teljes mérési készlet átlagát és négyzetátlagát is ki kell számítani. Az `std_dev` függvényhez használt számítások két ciklust használnak, holott egy is elegendő lenne. Másodszor észrevettem, hogy a vermen keresztül – egyszerű hivatkozás helyett – a teljes adatkészletet átadjuk az `std_dev` függvénynek. Úgy gondoltam, hogy ennek a két dolognak a hatása a teljesítményre már megéri a vizsgálódást.

1. lista A std_dev alapmegoldása

```
sub mean {
  my $result;
  foreach (@_) { $result += $_ }
  return $result / @_;
}


sub std_dev {
  my $mean = mean(@_);
  my @elem_squared;
  foreach (@_) {
    push (@elem_squared, ($_ **2));
  }
  return sqrt( mean(@elem_squared) -
              ➥($mean ** 2));
}
```

Teljesítménymérés

Miután beazonosítottuk a fejlesztésre szoruló függvényt, áttértem a következő lépésre, a teljesítménymérésre. A teljesítménymérés során összehasonlítási alapként megállapítjuk a kezdeti teljesítményt. Kizárólag a teljesítménypróba segítségével tudhatjuk meg, hogy változtatásaink javítottak-e bármit is a teljesítményen. Az itt bemutatott valamennyi teljesítménypróba időalapú. Szerencsére létezik egy külön időalapú teljesítménymérésre kifejlesztett Perl-csomag, a Benchmark.

A `std_dev` függvényt (*1. lista*) a programból átmásoltam a próbaparancsfájlba. Azzal, hogy a kérdéses részt a próbarendszerbe másoltam, az adatgyűjtő program hatásai nélkül mérhetem a teljesítményt. Mivel mérvadó teljesítményadatokat szeretnék kapni, az adatgyűjtő program körülményeihez hasonló terhelést kell előállítanom.

Az adatgyűjtő program által feldolgozott adatokat megvizsgálva úgy találtam, hogy 0 és 999,999 közötti számok keveréke megfelelő lesz. Ahhoz, hogy helyes teljesítmény eredményt kapjunk, az `std_dev` függvényt többször meg kell ismételnünk. Minél több alkalommal hajtódik végre a függvény, annál megbízhatóbb és egységesebb lesz a teljesítménypróba. A Perl Benchmark csomagban beállíthatjuk, hogy hányszor szeretnénk megismételni a próbát. Példaképpen tesztünket 10 000 alkalommal futtassuk. Másik lehetőségként egy adott időintervallumot is megadhatunk, ilyenkor a próbaprogram a megadott időn belül annyiszor futtatja a függvényt, ahányszor csak bírja.

A próbában bemutatott valamennyi teljesítménypróba 10 másodperces időintervallumot használ. 1 000 000 adatelem normálszórásának legalább 10 másodpercig tartó kiszámítása a következő eredményt adta:

```
12 wallclock secs (10.57 usr + 0.02 sys =
➥10.59 CPU) @ 0.28/s (n = 3)
```

A fenti sorokból megtudhatjuk, hogy a teljesítménymérés 12 másodpercig tartott. A teljesítménymérő eszköz másodpercenként 0,28 alkalommal tudta a függvényt futtatni, azaz – megfordítva – egy ciklus 3,5 másodpercig tartott. A teljesítménymérő eszköz a megadott 10 processzor-másodperc alatt mindössze három alkalommal tudta futtatni a függvényt (n = 3). A továbbiakban cikkünkben a teljesítményeredményeket másodperc/iteráció mértékegységben közöljük (s/iter). Minél alacsonyabb ez a szám, annál nagyobb a teljesítmény. Példaképpen egy végtelen gyors függvény 0 s/iter értéket adna, egy nagyon rossz függvény mondjuk 60 s/iter értéket. Az `std_dev` alapteljesítményének megállapítását követően már meg lehet állapítani a függvény újraírásának a hatását.

Az `std_dev` számítással kapcsolatos nehézségek feltárásához ugyan három ciklus is elegendő, mélyebb teljesítményelemzést azonban csak további minták alapján érdemes végezni.

Újraírás és ellenőrzés

Az első listában közölt teljesítményteszt végrehajtása után két lépésben újraírtam az `std_dev` algoritmust. Az első, `std_dev_ref` nevű változatban az értékátadás módszerét változtattam meg, „érték szerinti átadás”-ról „cím szerinti átadás”-ra az `std_dev` függvényben és az `std_dev` által meghívott átlagfüggvényben. Az eredményül kapott függvények a *2. listában* találhatók. Ez a változtatás elméletileg mindkét függvény esetében növeli a teljesítményt, hiszen a programnak az `std_dev`, majd az azt követő átlagfüggvény meghívása előtt nem szükséges a teljes adatállományt a verembe másolnia.

2. lista Érték szerinti átadás helyettesítése cím szerinti átadással

```
sub mean_ref {
  my $result;
  my $ar = shift;
  foreach (@$ar) { $result += $_ }
  return $result / scalar(@$ar);
}

sub std_dev_ref {
  my $ar = shift;
  my $mean = mean_ref($ar);
  my @elem_squared;
  foreach (@$ar) {
    push (@elem_squared, ($_ **2));
  }
  return sqrt( mean_ref(\@elem_squared) -
               ($mean ** 2));
}
```

3. lista Az átlagfüggvény eltávolítása után

```
sub std_dev_ref_sum {
  my $ar = shift;
  my $elements = scalar @$ar;
  my $sum = 0;
  my $sumsq = 0;

  foreach (@$ar) {
    $sum += $_;
    $sumsq += ($_ **2);
  }
  return sqrt( $sumsq/$elements -
               (($sum/$elements) ** 2));
}
```

A második módosításban, amelynek a `std_dev_ref_sum` nevet adtam, az átlagfüggvényt egy az egyben eltávolítottam. Az átlag és a négyzetátlag számítását a teljes adatkészletre vonatkoztatott egyetlen ciklusba helyeztem át. Ez a *3. listában* olvasható módosítás legalább két, az összes adaton végigfutó ciklust távolít el. Az *1. táblázat* a teljesítménymérési időeredményeket foglalja össze. Ahogy reméltük, az 1. táblázatban minden egyes lépésnél teljesítménynövekedést tapasztalhatunk. Az `std_dev` és `std_dev_ref` függvények közt 20 százalék növekedés

1. táblázat *Az alap és két módosítás*

	másodperc/iteráció
std_dev	3.53
std_dev_ref	2.93
std_dev_ref_sum	1.37

4. lista Az std_dev_pm függvény

```
sub std_dev_pm {
  my $stat = new
     ➥Statistics::Descriptive::Sparse();
  $stat->add_data(@_);
  return $stat->standard_deviation();
}
```

látszik, a `std_dev` és `std_dev_ref_sum` függvények közt pedig 158 százalék növekedést sikerült elérni. Ez bizonyítani látszik azt a feltevésemet, hogy a cím szerinti átadás Perlben gyorsabb, mint az érték szerinti (Nagy adatkészletekre igaz, néhány bájtos átadásnál nem feltétlenül – a ford.). Hasonlóképpen az adatokat kezelő két ciklus eltávolítása ismét növelte az `std_dev_ref_sum` függvény teljesítményét. A két módosítást követően a függvény 1,37 másodperc alatt képes kiszámítani 1 000 000 elem normálszórását. Igaz ugyan, hogy ez már lényegesen jobb, mint az eredeti, de azért azt hiszem, még mindig lehet javítani rajta.

Nem készítette el már valaki?

Rengeteg nyílt forrású Perl-csomag érhető el. Szerencsés esetben ráakadhatok egy olyan normálszórás-függvényre, amely gyorsabb, mint az eddigi legjobb próbálkozásom. A CPAN-on valóban rá is akadtam a `Statistics::Descriptive` nevű csomagra, amit letöltöttem. Létrehoztam egy `Statistics::Descriptive` csomagot használó `std_dev_pm` nevű függvényt; a függvényhez tartozó programot a *4. listában* találjuk. A függvény azonban csak 6,80 s/iter teljesítményt ért el, ami 48 százalékkal rosszabb, mint az alapul szolgáló `std_dev` függvényé. Ha belegondolunk, hogy a `Statistics::Descriptive` csomag objektum csatoló-felületet használ, ez igazából nem is meglepő eredmény. Minden számításban többletmunkát jelent ugyanis a `Statistics::Descriptive::Sparse` objektumok létrehozása és törlése. Ez nem azt jelenti, hogy a `Statistics::Descriptive` rossz csomag lenne. Figyelemre méltó mennyiségű Perlben írt statisztikai függvényt tartalmaz és nagyon könnyen használható olyan számításokra, amelyeknél a sebesség nem döntő tényező. Csakhogy a mi esetünkben éppen a sebesség a fontosabb.

Egy nyelven kívüli élmény

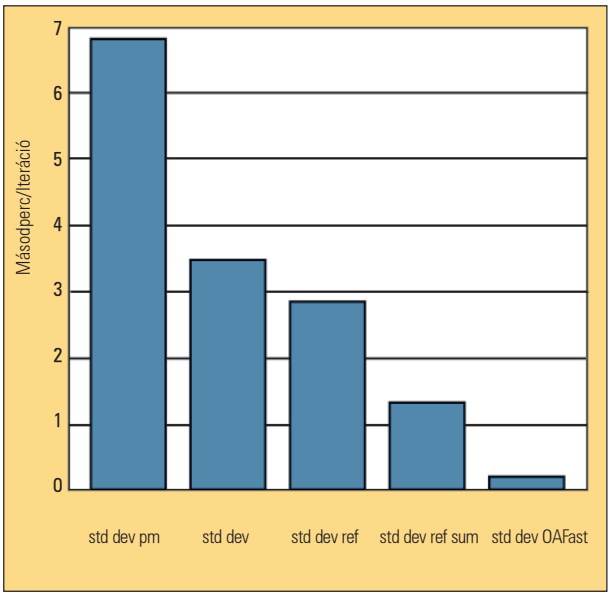
Minden nyelvnek van jó és rossz oldala. A Perl például kiváló általános célú nyelv, de tömeges számfeldolgo-

5. lista Az XS alapú megoldás

```
double
std_dev(sv)
    INPUT:
        SV *            sv
    CODE:
        double sum = 0;
        double sumsq = 0;
        double mean = 0;

        /* Dereference a scalar to retrieve
           an array value */
        AV* data = (AV*)SvRV(sv);

        /* Determine the length of the array */
        I32 arrayLen = av_len(data);

        if(arrayLen > 0)
        {
          for(I32 i = 0; i <= arrayLen; i++)
            {
              /* Fetch the scalar located at i
                 from the array.*/
              SV** pvalue = av_fetch(data,i,0);

              /* Dereference the scalar into
                 a numeric value. */
              double value = SvNV(*pvalue);

              /* collect the sum and the
                 sum of squares. */
              sum += value;
              sumsq += value * value;

            }
          mean = (sum/(arrayLen+1));
          RETVAL = sqrt((sumsq/(arrayLen+1)) -
                        (mean * mean));
        }
        else
        {
          RETVAL = 0;
        }

    OUTPUT:
        RETVAL
```

2. táblázat *Az alap és a leggyorsabb Perl-megoldás összehasonlítása a C-ével*

	másodperc/iteráció
std_dev	3.53
std_dev_ref_sum	1.37
std_dev_OAFast	0.277



Valamennyi megoldás összehasonlítása

zásra nem a legjobb. Ezt szem előtt tartva úgy döntöttem, hogy a normálszórás-függvényt C nyelven újraírom, hogy lássam, növeli-e a teljesítményt.

Az adatgyűjtő alkalmazás esetében nem lett volna kifizetődő a teljes programnak C nyelven történő újraírása. A legtöbb alkalmazás elkészítésére számos különleges függvénye a Perlt teszi a legalkalmasabb nyelvvé. Az alkalmazás újraírásának legjobb módszere, ha csak a teljesítmény növeléséhez szükséges függvényeket írjuk újra. Ezt a C nyelven megírt normálszórás-függvény Perl-modulba burkolásával érjük el. A C-függvény beburkolása révén a program nagyobbik része Perlben futhat, de a szükséges helyeken C- és C++-kódot is alkalmazhatunk.

A már létező C- vagy C++-csatolófelület fölé az XS használatával írhatunk Perl-burkolót (wrapper). Az XS eszköz a Perl-csomagban megtalálható, dokumentációját a ***perlxs*** Perl-dokumentumban találjuk. Szükségünk lesz néhány, a perlguts dokumentumban található információra is.

Az XS segítségével létrehoztam egy `OAFastStats` nevű Perl-csomagot, amely a C alapú normálszórás függvényt tartalmazza. Ezt az *5. listában* bemutatott függvényt azután Perlből már közvetlenül meg lehet hívni. Az összehasonlíthatóság kedvéért ezt a normálszórás függvényt `std_dev_OAFast` névre kereszteltem.

Az alap normálszórás- és a XS-el burkolt C alapú-függvény teljesítménye közti eltérést a *2. táblázat* mutatja be, miszerint jelentős sebességnövekedést tapasztalhatunk. A C-függvény (`std_dev_ref_OAFast`) 1175 százalékkal fürgébb, mint az alapul használt függvényünk (`std_dev`), és 395 százalékkal gyorsabb, mint a legjobb Perl-megoldás (`std_dev_ref_sum`).

Összegzés

A feladat során beazonosítottam azt a függvényt, amelyik valószínűleg nem teljesít olyan jól, mint kellene. A Perl nyelvű számítások újraírásával képes voltam bizonyos sebességnövekedést elérni. Kipróbáltam egy nyílt forrású csomagot, de kiderült, hogy 48 százalékkal rosszabb teljesítményt nyújt, mint az eredeti függvényem. Végül C nyelven elkészítettem a normálszórás függvényt, és az XS rétegen keresztül beépítettem a Perlbe. A C-változat az eredeti Perl-változathoz képest 1175 százalékos sebességnövekedést mutatott. Az egyes fejlesztéseket az *ábránkon* foglaltam össze.

A legtöbb esetben úgy tűnik, hogy a Perl teljesítménye képes versenyezni a C-évelt – a mostani azonban nyilvánvalóan nem az az eset. A Perl kiváló általános célú nyelv, amelynek az egyik előnye éppen az, hogy képes átlépni a nyelv határait, és alacsonyabb szintű nyelvek kódjait beépíteni. Ne féljünk a nyelvek keverésétől, ha tényleg szükség van a teljesítmény növelésére; ám tudnunk kell, hogy ez karbantartási többletmunkát jelent. A további nyelvek bevezetésének az a hátulütője, hogy az alkalmazást később karbantartani kívánókra többletterhet ró: ismerniük kell a C nyelvet és érteniük kell az XS függvényt. Esetünkben azonban a megnövekedett teljesítmény lényegesen nagyobb súllyal esett latba, mint az XS támogatásával járó nehézségek.

Linux Journal 2004. február 116. szám

Bruce W. Lowther (blowther@micron.com) Idaho államban a boisei Micron Technology, Inc. programmérnöke. Kilenc éve dolgozik a Micronnál, ahol az utóbbi öt évben olyan eszközökön dolgozik, amelyek félvezető felszereléseket segítenek beépíteni a Micron gyártásfolyamatába.