Eötvös Loránd Tudományegyetem
Informatikai Kar
Komputeralgebra Tanszék

Cache optimalizált lineáris szita párhuzamos megvalósítása

Dr Vatai Emil
adjunktus

Huszti Péter
Programtervező Informatikus MSc

Budapest, 2017

Tartalomjegyzék

1. Bevezetés

1.1. A dolgozat célja, motiváció

Az alkalmazott matematikában nagyon fontos szerepet játszanak az prím számok, elég csak például a nyílt kulcsos titkosítási módszerekre gondolnunk. Ebből kifolyólag az évek során nagyon sok módszert fejlesztettek ki prímszázmok keresésére, például a Fermat-teszt, vagy a Miller-Rabin teszt. Ezen módszerekkel elég gyorsan el lehet dönteni egy darab számról, hogy prím-e vagy sem, és így ezért az ilyen algoritmusokkal nagyon nagy prímeket is meg lehet találni viszonylag gyorsan.

Előfordulhat azonban probléma, hogy egy adott intervallumban szeretnénk megtalálni az ott lévő összes prím számot. Az ilyen feladatok megoldására a leghatékonyabb módszerek a szitáló módszerek. Ezek úgy működnek, hogy kiválasztunk bizonyos számokat, és azokkal "végigszitálunk" a vizsgált intervallumon, és az érintetlenül hagyott számok lesznek a nekünk megfelelő, jelen esetben prím számok. Ezek közül is a legegyszerűbb egészen az ókorig nyúlik vissza, az ún. Erathosztenészi-szita. Ez az algoritmus ahhoz képest, hogy milyen rég óta ismert, meglepően hatékonyan működik. De természetesen vannak hátrányai, például hiába tűnik nagyon gyorsnak komplexitás szempontjából, elég nagy intervallumokra nagyon meg fog növekedni a memória olvasások száma, ami mint köztudott nagyságrendekkel lassabb, mint a processzorok utasítás végrehajtó képessége már a memóriában lévő adatokon. Nem meglepő módon sok féle képpen fel lehet javítani az algoritmus teljesítményét mai modern eszközökkel.

A dolgozat célja az, hogy a fent említett "naív" szitáló algoritmus teljesítményét növeljük, miközben a program skálázható is maradjon, és így a valós gyakorlati életben is alkalmazni lehessen. A dolgozat során két féle módon próbáltam meg javítani a teljesítményen.

Az első és talán legkézenfekvőbb módszer a program párhuzamosítása. Ezt viszonylag egyszerűen meg lehet tenni, mivel az algoritmust könnyedén fel lehet darabolni kisebb, egymással ekvivalens részfeladatokra, amiket szét lehet osztani a processzor szálai közt. Ráadásul a konkurens programokra jellenző veszélyek nem állnak fent, így sok nehézségtől meg tudjuk kímélni magunkat, ami a teljesítményt is javítja.

A másik megközelítés, hogy a memóriaműveletek számát próbáljuk meg minimalizálni. Erre egy hatékony megvalósítása az ún. COLS - cache optimalizált lineáris szita - algoritmus, aminek az megvalósítása is része a dolgozatnak.

Végül a fentiekből magától értetődik egy újabb gyorsítási lehetőség, hogy a COLS algoritmust is meg lehet valósítani párhuzamos szálakon, ami mint majd később látjuk megint csak nagy teljesítménynövekedéssel járhat.

A dolgozat során implementáltam a fent említett négy algoritmust, és azt vizsgáltam, hogy milyen esetekben (a probléma mérete, hardver specifikációja) mennyire tudják, ha egyáltalán lehetséges, felgyorsítani a fenit módszerek a probléma megoldását. Az algoritmusokat C++ nyelvben valósítottam meg, csak és kizárólag a standard C++11 szabvány által kínált lehetőségeket felhasználva.

Az elészített programmal könnyen és gyorsan lehet egy megadott intervallumon megkeresni a prím számokat, továbbá megfelelően paraméterezhető a rendelkezésre álló processzor(ok) tulajdonságainak ismeretében. A program jól skálázható, így nagy problémák megoldására is lehetőséget nyújt.

2. Matematikai háttér

2.1. Prím számok, faktorizáció

2.1. Definíció. Egy p természetes számot prímnek nevezünk, ha $\forall a, b$-re amire $p|a \cdot b \rightarrow (p|a \vee p|b)$.

Természetes számok körében ez a definíció ekvivalens azzal, hogy egy prím számnak kettő, és csak kettő osztója van, 1 és önmaga.

A prím számok kitüntetett szerepet játszanak a matematikában. Többek közt felhasználják őket hasítótáblákhoz, pszeudovéletlen számok generálásához vagy nyílt kulcsú titkosításokhoz. Utóbbiak széles körben elterjedtek, valószínűleg sokan ismerik például az RSA kódolást, az SSH-t vagy a HTTPS-t. Ezek mind fontos részét képezik a modern kornak. A nyílt kulcsú kódolások olyan matematikai problémákon alapulnak, amelyeket megoldani nehéz, vagyis a mai eszközeinkkel valós időben nem lehetséges, viszont ellenőrizni egy lehetséges megoldást gyors és egyszerű. A leggyakrabban használt ilyen probléma a prím faktorizáció.

Számelmélet alaptétele: minden pozitív szám felírható egyértelműen prímszámok szorzatára.

Viszont, ennek a felbontásnak a megkeresése NP-nehéz probléma, vagyis nem tudunk jelenleg sokkal jobb módszert annál, mint hogy kipróbáljuk az összes lehetséges prím számot, hogy oszthója-e a felbontani kívánt számnak.

Tehát jól látszik, hogy a prím számok megtalálása kiemelten fontos feladat. Rengeteg módszer létezik arra, hogy prímeket keressünk. A dolgozat az ún. szitáló módszerekkel foglalkozik, konkrétabban ezek felhasználása prímszámok keresésére. Ezeknek a módszereknek megvan az az előnye, hogy egy adott intervallumban megtalálják az összes ott előforduló prímet, viszont ha konkrétan csak egy darab számról akarjuk eldönteni, hogy prím-e, akkor ezeknél a módszereknél léteznek sokkal hatékonyabbak is.

2.2. Eratoszthenészi-szita

Az Eratoszthenészi-szita, mint a nevéből is látszik már egy nagyon régen ismert algoritmus. Ennek ellenére, bármilyen meglepő is, ha gyorsan meg akarjuk keresni egy intervallumban az összes prím számot, akkor ehhez az algoritmushoz kell visszanyúlnunk. Ez egy egyszerű kizárásos algoritmus. A számelmélet alaptétele szerint az intervallumunkban minden szám, amelyik nem prím, osztható nálánál kisebb prím szám(ok)kal. Tehát, ha lenne egy listánk a kisebb prímekről, akkor azoknak meg tudnánk találni az intervallumunkban lévő többszöröseit, és amely szám egyik kis prímnek sem többszöröse, az prím szám. Ez az alap ötlet.

Az algoritumus:

1. Készítünk egylistát a kisebb prímekről, amelyekkel majd ki fogjuk szitálni a vizsgált intervallumot. De mit is jelent az, hogy kis prímek? Jelöljük az intervallumunkat, ahol keressük a prímeket $[a, b]$-vel. A első ötlet természetesen, hogy vizsgálunk minden 1-nél nagyobb de a-nál kisebb prímszámot. De kicsit jobban belegondolva erre egyáltalán nincs szükség.

2.

2.3. COLS

3. Felhasználói dokumentáció

3.1. A program használata

3.2. A program kimenete

3.3. Rendszerkövetelmények

3.4. CD tartalma, telepítés

4. Fejlesztői dokumentáció

4.1. A program felépítése

4.2. Adatszerkezetek

4.2.1. Szitatábla

4.2.2. Szegmensek

4.2.3. Körök és edények

4.3. Párhuzamos megvalósítás

4.4. Skálázhatóság

4.5. Tesztelés

4.6. Továbbfejlesztési lehetőségek

5. Az algoritmusok összehasonlítása

5.1. Használt hardverek

5.2. Az eredmények

5.3. Konklúzió

6. Összegzés