Excel@FIT 2015
http://excel.fit.vutbr.cz

Paralelní faktorizace celých čísel metodou SIQS

Dominik Breitenbacher*

Abstrakt
Práce se zabývá faktorizací celých čísel, tedy rozkladem složeného čísla na jeho faktory. Faktorizace je nejznámější a nejpoužívanější metodou k lámání RSA. V rámci této práce byla vybrána faktorizační metoda zvaná SIQS. SIQS je považována za nejrychlejší metodu k faktorizaci čísel, které mají do 100 dekadických číslic, a tato metoda byla v rámci práce i naimplementována. Implementace byla řádně zdokumentována, čímž se práce snaží vyplnit mezeru mezi teoretickým popisem SIQS a existujícími implementacemi.
I když se jedná o nejrychlejší metodu (do 100 číslic), není možné ji efektivně počítat v polynomiálním čase, a tak se hledají různé možnosti, jak tuto metodu co nejvíce urychlit. Jako první se nabízí paralelizace. K tomuto účelu bylo využito OpenMP jakožto výkonného nástroje pro práci s vlákny, jeho použití je přitom velice jednoduché. Další možností je pak optimalizace kódu.
Cílem této práce je ukázat, jak jednoduše lze v mnoha případech využít paralelizace kódu a dále také, jak díky podrobné analýze kódu lze dosáhnout poměrně velkého urychlení. Metodika iteračního provádění optimalizací se ukázala jako velmi účinná a je možné ji použít obecně, nejenom v této úloze. Touto metodikou byla implementace SIQS vylepšena tak, že faktorizace byla urychlena až 99-krát, v některých částech kódu dokonce ještě více.

Klíčová slova: Faktorizace — SIQS — OpenMP — MPIR — Profilace

Přiložené materiály: Downloadable Code

*xbreit00@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

Prvním krokem šifry RSA při generování klíčů je vynásobení dvou velkých prvočísel (dnes obvykle o délce 384 bitů a více), z nichž vznikne složené číslo n [1]. Vygenerování čísla n je založeno problému faktorizace. Faktorizace je proces, při kterém se snažíme zpětně nalézt všechny dělitele složeného čísla. Proces faktorizace je však výpočetně tak náročný, že by se případnému útočníkovi zpětné hledání faktorů daného čísla n nevyplatilo. Díky RSA tak vznikla soutěž, kde se jednotliví programátoři a matematici dodnes předhání, kdo rozloží větší číslo[1]. Při tom vzniklo mnoho nových algoritmů a jedním z nich je i SIQS, který byl popsán panem Continim v tomto článku [2].

SIQS je nejrychlejší metodou pro faktorizování čísel do 100 dekadických číslic a obecně je metoda druhá nejrychlejší z doposud objevených. Jak bylo zmíněno, problémem ale je její poměrně velká náročnost na pochopení, což může odradit některé případné zájemce o tuto metodu. SIQS lze rozdělit do mnoha částí a implementace každé takové části může být provedena různými způsoby. Jak ale bylo zmíněno, faktorizace touto metodou jako celku bude vždy časově velice náročná, a tedy je nutné jednotlivé části implementovat co nejefektivněji.

SIQS bylo vybráno a implementováno v rámci této práce hned z několika důvodů. Prvním z nich byla snaha implementovat rychlou faktorizační metodu v jazyce C++ s využitím datových typů, které jsou poskytovány standardními knihovnami jazyka, a metodu pak dále paralelizovat pomocí OpenMP. Vytvořená implementace pak bude srovnána s nejrychlejší dostupnou implementací metody SIQS a bude diskutováno, jaký přínos měl způsob implementace vytvořené v rámci této práce.

[1] http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm

Druhým důvodem volby metody SIQS byl fakt, že daná metoda byla sice již mnohokrát implementována, nikde však není popsán přesný postup, jak metodu implementovat. Princip a postup faktorizace metody SIQS je asi nejlépe popsán v článku [2]. Postup metody SIQS je zde však popsán pouze na teoretické úrovni a v některých případech i na poměrně vysoké abstraktní úrovni. Samotná realizace implementace na základě tohoto článku tak může být pro případného zájemce až nemožná, pokud není dostatečně znalý matematických základů a postupů, pomocí kterých je zde metoda vysvětlena. Cílem této práce tak je metodu přiblížit a vysvětlit všem zájemcům o faktorizaci. Vytvořený dokument se sice specializuje na metodu SIQS, snaží se ale přitom pokrýt co možná nejvíce z problematiky faktorizace a je tak vhodný ke vzdělávacím účelům.

Implementace probíhala ve dvou fázích. V první fázi byl naprogramován kompletní a funkční algoritmus, a to nejjednodušším způsobem, zatím bez přihlédnutí k požadavkům na rychlost provádění. Na číslech s menší délkou (a nižší časovou náročností) bylo ověřeno, že algoritmus byl správně pochopen a korektně naimplementován. Takto provedený algoritmus posloužil jako referenční implementace s ověřenou funkčností. Bylo ovšem zřejmé, že směrem k delším číslům jsou nutné jisté úpravy, zejména s ohledem na rychlost provádění.

Optimalizace na rychlost byla druhou fází řešení celého zadání. Metodika iteračního provádění optimalizací se ukázala jako velmi účinná a je použitelná obecně, nejenom v této úloze.

Algoritmus SIQS sestává z několika kroků, které na sebe navazují. Jednotlivé kroky mají ovšem různou třídu složitosti od lineární až po kubickou. Směrem k vyšším číslům tak byla v jednotlivých iteracích optimalizace odhalována další a další úzká hrdla provádění, která se u kratších čísel zatím jevila jako bezvýznamná.

Optimalizace spočívala v důsledné paralelizaci všude tam, kde to bylo možné z podstaty algoritmu možné, v dalších iteracích pak bylo nutné zaměnit některé datové typy za jiné, vhodnější (mapy na vektory, bool na uint64 apod.). Také bylo nutné minimalizovat počet paměťových alokací a uvolňování.

Algoritmus metody SIQS, který byl implementován v rámci této práce, však není jedinou implementací. Asi nejznámější implementací SIQS je **msieve**. Msieve je již několik let vyvíjen skupinkou programátorů a matematiků, implementace SIQS je tak velmi optimalizována. Msieve dále umožňuje provádět faktorizaci na clusterech s využitím OpenMPI, což implementace této práce také neumožňuje. Naopak ale implementace této práce využívá paralelizace za použití OpenMP, a tedy aplikace je schopna běžet paralelně i na jednom stroji. Nespornou výhodou implementace této práce je ale čitelnost zdrojových kódů a dokumentace. V dokumentaci je metoda SIQS velice detailně popsána a jsou zde popsány všechny problémy, se kterými se případný zájemce může setkat a jak tyto problémy vyřešit. Pokud by čtenáři dokumentace činil problém pochopit princip SIQS, je zde popsán konkrétní příklad, kde je metoda SIQS popsána v jednotlivých krocích i se všemi výstupy. Pokud se čtenář rozhodne si algoritmus SIQS naimplementovat sám, může si tak díky uvedenému příkladu jednoduše zkontrolovat, zda jeho vlastní implementace SIQS generuje v jednotlivých částech správný výstup a tedy je implementace s nejvyšší pravděpodobnosti v pořádku. Něco takového Msieve nenabízí, zdrojové kódy jsou sice dobře popsány, ale samotná implementace, jak sám i autor ve zdrojových kódech přiznává, je značně zatemněná a tedy extrémně složitá k pochopení.

2. SIQS – Self-Initializing quadratic sieve

Kvadratické síto je jedna z nejpoužívanějších metod pro faktorizaci velkých čísel, kterou popsal pan Pomerance v [3], [4]. Metoda má své kořeny ve Fermatově faktorizaci, která tvrdí, že každé liché číslo jsme schopni zapsat jako rozklad dvou čtverců, $n = u^2 - v^2$, neboli $n = (u+v)(u-v)$. U Fermatovy metody může být hledání dvou čtverců, které vedou k rozkladu zadaného čísla, zdlouhavé, pokud se faktory nenalézají blízko sebe, protože metoda na začátku položí $u = \lceil \sqrt{n} \rceil$ a pokud $u^2 - n \neq v^2$, hodnota u je zvýšena o 1 a výpočet se opakuje.

S vylepšeným postupem přišel pan Kraitchik [4], který si uvědomil, že místo toho, abychom hledali, kdy bude platit $n = u^2 - v^2$, stačí najít u a v takové, že $u^2 - v^2$ bude násobkem rozkládaného čísla n, což lze také zapsat jako $u^2 \equiv v^2 \pmod{n}$. Bystrý čtenář si může uvědomit, že tento postup může nalézt i triviální dělitele, kteří jsou pro nás nezajímaví. Rozklad bude netriviální, pokud bude platit $u \equiv \pm v \pmod{n}$. Jednotlivé faktory pak získáme jako největší společný dělitel $GCD(u-v,n)$, případně $GCD(u+v,n)$.

Kraitchik tedy navrhl hledat taková čísla u_i, pro která platí:

$$u^2 = u_1^2 * \ldots * u_k^2 \equiv \\ \equiv \left(u_1^2 \bmod n\right) * \ldots * \left(u_k^2 \bmod n\right) = v^2 \qquad (1)$$

Každé číslo m jsme schopni rozložit na prvočísla, tedy $m = \prod p_i^{e_i}$, kde p_i je prvočíslem, a počet těchto prvočísel, dále značen B, si na základě nějakého rozhodnutí omezíme. Pak jsme z exponentů těchto prvočísel schopni vyjádřit vektor $e(m) = (e_1, e_2, \ldots)$. Jelikož

hledáme čtverec, dané hodnoty exponentů pro nás obsahují nepotřebné informace a bude nám postačovat, pokud budeme mít exponenty vyjádřeny v rámci modulo 2. Najdeme-li aspoň dva vektory takové, že jejich součtem bude nulový vektor, pak hodnota všech exponentů je sudá, a tak jsme získali číslo, které je čtvercem. Cílem je tedy hledat taková m, s jejichž pomocí jsme schopni získat nulový vektor. Kolik takových m je nutné nalézt, blíže popisují páni Brillhart a Morrison v článku [5]. Čísla m, která jsme schopni rozložit na prvních B prvočísel, říkáme relace anebo B-hladká čísla.

SIQS je vylepšením kvadratického síta (QS – Quadratic Sieve). Metoda na rozdíl od kvadratického síta, které používá polynom:

$$Q(x) = x^2 - n \tag{2}$$

pracuje s polynomem:

$$Q_{a,b}(x) = ax^2 + 2bx + c \tag{3}$$

Jak získat jednotlivé koeficienty polynomu je možné nalézt v článku [2].

Kvadratické síto používá ke hledání relací pouze jeden polynom. Naopak SIQS díky zavedeným koeficientům může polynomy měnit. SIQS tak pracuje s proměnnou x pouze ve stanoveném intervalu a po vyčerpání všech hodnot z intervalu je polynom vyměněn za jiný. Tím se výsledná hodnota polynomu udrží na menších hodnotách a nalezení relace je tak pravděpodobnější.

3. Návrh implementace SIQS

Projekt bude tvořen v jazyce C++ na architektuře x86. Jazyk C++ byl vybrán, protože má ve standardních knihovnách implementovány datové typy, které budou v rámci projektu značně využívány. Tento jazyk byl zvolen také z důvodu, že pro něj existuje mnoho profilovacích nástrojů, a je tak možné sledovat, ve kterém místě kódu se spotřebovává mnoho času, a případně tuto část optimalizovat. Dalším důvodem pro nasazení jazyka C++ je podpora OpenMP, které bude použito pro paralelizaci.

U faktorizace je však nutné řešit jeden důležitý problém. Tím problémem je fakt, že tradiční programovací jazyky ani architektury neposkytují datový typ, jenž by byl schopen pojmout čísla, která se mají faktorizovat. Z tohoto důvodu bude použita knihovna MPIR, která s takto velkými čísly pracovat dokáže a definuje nad nimi řadu operací.

Generování polynomu bude probíhat tak, jak je popsáno v [2]. Každé vlákno si vypočte koeficient a, na jehož základě si vygeneruje polynom, s nímž bude pracovat. Toto a musí být v rámci běhu jedinečné, pokud by si někdy později jakékoli vlákno vypočetlo stejné a, došlo by k nalezení úplně stejných B-hladkých čísel, což je nežádoucí. Dále si vlákno spočte první koeficient b a z něj odvodí koeficient c. V případě nutnosti vygenerovat nový polynom si vlákno vypočte následující b dle Grayova kódu a opět odvodí koeficient c.

Když je dokončena fáze generování polynomu, přechází se k fázi prosívání, tedy hledání relací. Každé vlákno dostane hodnotu M, která bude určovat rozsah $\langle -M; M \rangle$, přes který se bude iterovat a hledat B-hladké hodnoty. Každé vlákno dále dostane shodnou faktorizační bázi, na základě které se bude určovat, zda je zkoumané číslo B-hladké. V rámci této práce se ve fázi prosévání nebudou hledat pouze relace, ale také tzv. částečné relace. Vylepšení používající i částečné relace nazýváme Large Prime Variation. Jeho bližší popis lze nalézt například zde [2].

Výstupem této fáze je matice vektorů exponentů. Na základě velikosti faktorizační báze je určeno, kolik relací potřebujeme k sestavení matice. Každé vlákno, které nalezne relaci, ji uloží do matice a zvýší čítač nalezených relací o 1. Teoreticky by mělo stačit nalezenout o jednu relaci více, než je počet použitých prvočísel.

Pro nalezení lineární závislosti mezi vektory exponentů bude použita Gaussova eliminační metoda. Implementována bude taková varianta, kterou bude možno jednoduše paralelizovat. Spočtením lineární závislosti získáme vztah $u^2 \equiv v^2 \pmod{N}$, pomocí $GCD(u - v, N)$ stačí tedy ověřit, že jsme získali netriviálního dělitele.

4. Implementace SIQS

4.1 Získání faktorizační báze

Nejdříve je na základě zadaného čísla zvolena faktorizační báze (počet použitých prvočísel) tak, že se z předpřipravené tabulky načte optimální počet prvočísel. Tato velikost je následně zredukovaná tak, že faktorizační báze bude obsahovat pouze prvočísla, pro které je Legendreův symbol roven 1. Každé číslo, které se dělením některých prvočísel z faktorizační báze zredukuje na 1, bude tvořit relaci.

4.2 Generování koeficientu a polynomu

Před samotným vygenerováním koeficientu a je nutné nejdříve spočíst jeho ideální hodnotu, tu získáme ze vztahu $a \approx \frac{\sqrt{2n}}{M}$. Jelikož koeficient a se sestavuje z prvočísel z faktorizační báze, vybereme taková, jejichž součet logaritmů bude nejblíže logaritmu ideální

hodnoty a. Čím bližší hodnotu se nám podaří nalézt, tím větší je pravděpodobnost nalezení relací. Naopak rozdíl by neměl být větší než 0.01. V takovém případě se nám sice bude dařit relace nalézt, ty ale budou velmi pravděpodobně duplicitní s jinou již nějakou nalezenou relací.

Generování koeficientu a v této práci je řešeno postupem, který představili páni Carrier a Wagsraff [6] využívající binární vyhledávací strom pro uchovávání trojic a algoritmus NEXKSB pro lexikografický výběr k-tic prvočísel [7].

4.3 Generování polynomu

Když je spočten koeficient a, jsme schopni spočítat všechny hodnoty koeficientu b. Čím více dělitelů koeficient a má, tím více jsme schopni vygenerovat koeficientů b. Výpočet koeficientu b je výpočetně mnohem méně náročný než výpočet a, a proto se snažíme mít dělitelů koeficientu a co nejvíce. Zároveň ale dělitelé koeficientu a nesmí být malí, jinak bude mnoho nalezených relací duplicitních. Počet dělitelů tak musí být zvolen vhodně. V případě nutnosti vygenerovat nový polynom je následující hodnota koeficientu b spočtena pomocí Grayova kódu. Pokud již byly použity všechny hodnoty koeficientu b, byl polynom vyčerpán a je nutné vygenerovat nový koeficient a. Jako poslední zbývá zmínit koeficient c. Ten je při každém vygenerování koeficientu b získán dle vzorce $n = b^2 - ac$.

4.4 Prosévání

Prosévání je časově nejnáročnější částí algoritmu, kdy se snažíme nalézt potřebný počet relací k pozdějšímu hledání nulového vektoru. Prosévání se skládá z hledání kořenů prvočísel pro jednotlivé polynomy, hledání kandidátů na B-hladké číslo a samotné ověření, zda je kandidát opravdu B-hladkým číslem.

4.4.1 Hledání kandidátů B-hladkých čísel

Na začátku prosévání novým polynomem je nejprve nutné si spočítat kořeny jednotlivých prvočísel. Jednotlivé kořeny pro jednotlivá prvočísla nad daným polynomem počítáme dle postupu uvedeného v tomto článku [2]. Kořeny nám slouží k rychlému určení, že vypočtená hodnota polynomu pro dané x bude dělitelná daným prvočíslem.

Když jsou kořeny spočteny, provádí se výpočet hodnot z polynomu pro zvolený interval. Každé x z daného intervalu slouží zároveň jako index do pole. Pro každé x přičteme do pole na příslušný index hodnotu $\log(p)$ každého prvočísla p, pro které je jeho kořen či násobek kořene roven aktuálnímu x. V rámci této práce je ukládán na daný index i příslušný dělitel. Následně projdeme celé pole a pokud je na nějakém indexu uložená hodnota vyšší než $\log(2x\sqrt{N})$, pak hodnotu spočtenou pro dané x považujeme za kandidáta na získání relace.

4.4.2 Ověření kandidátů

Ověření, že kandidát je opravdu B-hladké číslo, provádíme tak, že jej podělíme všemi děliteli, které máme uložené na daném indexu kandidáta. Je nutné brát do úvahy, že číslo by mohlo být složeno z mocnin daných prvočísel. Pokud je výsledkem po dělení všemi prvočísly 1, nalezli jsme relaci. Nastala-li tato situace, vytvoříme vektor exponentů a relaci uložíme. Není-li výsledkem 1, pak kontrolujeme, zda je výsledek prvočíslem. Pokud ano a toto prvočíslo je v námi stanovené hranici, pak se jedná o částečnou relaci. Z takového čísla opět vytvoříme vektor exponentů a uložíme si danou relaci i s daným velkým prvočíslem pro další zpracování v rámci Large Prime Variation.

Pokud po projití všech kandidátů jsme nenasbírali ještě dostatek relací, pak je nutné vygenerovat nový polynom a proces opakovat. Díky využití Large Prime Variation nám stačí nasbírat o něco málo více než polovinu relací. Zbytek potřebných relací bude vytvořen z relací částečných. Může však nastat situace, kdy částečných relací nebude dostatek. V takovém případě je algoritmus nucen opět spustit proces prosévání.

4.4.3 Zavedený paralelizmus

Jak generování polynomu, tak prosévání běží paralelně. Toho bylo dosaženo použitím OpenMP, které využívá všechna dostupná vlákna na stroji. Vlákna běží po celý čas prosévání poměrně nezávisle na sobě. Vlákna se mohou ovlivňovat pouze v kritických sekcích, kdy ostatní vlákna musí čekat na to, než vlákno, které obsadilo kritickou sekci, dokončí svůj výpočet. Kritické sekce se skládají z velmi triviálních úloh. Jedna kritická sekce zajišťuje, že si vlákno vygeneruje novou sadu prvočísel, ze kterých bude složen koeficient a, pomocí algoritmu NEXKSB. Druhá pak zajišťuje, že se budou relace nebo částečné relace v daný okamžik ukládat pouze jedním vláknem, aby nedošlo k nekonzistenci.

4.5 Vyhledání lineární závislosti vektorů a dokončení faktorizace

Když je prosévání ukončeno, dochází k sestavení dalších relací z nasbíraných částečných relací. Poté je nutné relace projít a odstranit ty, které by vedly k selhání při hledání lineární závislosti vektorů anebo by pravděpodobnost správného výsledku snížily. Relace, které by mohly způsobit selhání, jsou buď relace duplicitní anebo tzv. singletony. Singletony jsou relace

obsahující prvočíslo, které není obsaženo v žádné jiné relaci. Ty přímo nevedou k selhání při hledání lineární závislosti, ale jelikož neexistuje relace, která by obsahovala dané prvočíslo, tak relace nikdy nemůže být součástí vektorů, které tvoří nulový vektor.

Odstraněním všech těchto nežádoucích relací však může dojít k situaci, kdy bude celkový počet nasbíraných relací menší než počet, který je potřebný. V takovém případě je nutné opět spustit prosévání a dosbírat potřebný počet relací.

Z nasbíraných relací je následně vytvořena matice. V této matici se hledají lineárně závislé vektory. K tomuto účelu byla použita rychlá Gaussova eliminační metoda, která je popsána v článku [8]. Výsledkem je několik možností, jak získat nulový vektor, a tedy dělitele zadaného čísla. Jelikož každá možnost má 50% na úspěch, tedy že výsledkem nebude triviální dělitel, stačí si ze všech možností uložit prvních deset a ty vyzkoušet. Výpočet dělitele je proveden tak, jak bylo popsáno v kapitole 2.

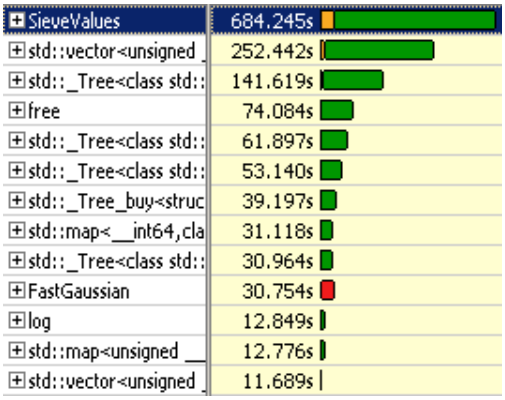
5. Měření a optimalizace

Po implementaci bylo provedeno měření a zkoumalo se, jak je implementace rychlá. Měření probíhalo na 10 číslech o délce 40 dekadických číslic a na 10 číslech o délce 50 dekadických číslic. Metoda byla měřena jak v sériovém režimu, tak v paralelním režimu. Pro každé číslo o 40 i 50 dekadických číslicích bylo SIQS puštěno v obou režimech 5-krát. Demonstračně pak byla měřena i faktorizace čísla o 60 a 70 dekadických číslicích. Tato čísla však byla faktorizována pouze jednou a to v paralelním režimu. Uvedené hodnoty v tabulkách pak vyjadřují průměrné zpracování jednoho čísla o dané velikosti. Měření byla prováděna na notebooku vybaveném procesorem Intel i7 4700MQ se zapnutým Hyper-Threadingem a Turbo Boost.[2] Instalovaný operační systém byl Windows 8.1. První výsledky měření jsou zaznamenány v tabulce 1.

Úloha	Sériově	Paralelně	Urychlení paralelizmem
40 dec	72.68s	22.71s	3.19
50 dec	984.07s	307.96s	3.19
60 dec	-	3217.55s	-

Tabulka 1. Naměřená doba faktorizace při naivní implementaci

Měření podle očekávání ukázalo, že faktorizace pomocí prvotní implementace trvá příliš dlouho. Byla tak provedena profilace, aby se ukázala místa v implementaci, která jsou časově nejnáročnější. Jako profilační nástroj byl použit Intel VTune Amplifier XE 2013. Výsledek profilace je vyobrazen na obrázku 1. Profilace byla provedena na čísle o 60 dekadických číslicích.



Obrázek 1. Profilace SIQS

Z obrázků lze vypozorovat, že nejvíce času se spotřebovává při prosévání, konkrétně u metody *find()*, která je součástí implementace mapy ze standardní knihovny jazyka C++. V sekci 4.4 je popsáno místo, kam se ukládají jednotliví dělitelé pro jednotlivé hodnoty proměnné x jako pole. Ve skutečnosti se jedná o mapu, protože se předpokládalo, že budou existovat hodnoty x, které nebudou dělitelné ani jedním z prvočísel faktorizační báze a že počet takových hodnot nebude zanedbatelný. Metoda *find()* zde tak ošetřuje případ, že pro danou hodnotu x ještě neexistuje záznam a došlo by tak k pokusu o přístup k neexistující položce. Jedním z problémů je fakt, že toto hledání je provedeno pro každý násobek kořene každého prvočísla, to znamená, že tato metoda je vyvolávána opravdu často. Druhým problémem je, že vyhledávání v mapě má logaritmickou složitost a s rostoucí mapou je každé takové hledání pomalejší. Dále se ukázalo, že počet hodnot proměnné x, pro kterou nebudou existovat žádní dělitelé, je zanedbatelný.

Navrženým řešením je tedy přepsání mapy na vektor. Jelikož počet položek, které bude vektor maximálně obsahovat je znám po celou dobu prosévání, bude před prosíváním vektor inicializován tak, aby byl schopen pojmout všechny položky a nevznikla tak potřeba vektor zvětšovat. Dále všechna pole budou alokována hned, jak bude známa jejich velikost a dealokována až v době, kdy nebudou potřeba. Práce s pamětí tak bude efektivnější. Hodnoty logaritmů pro prvočísla budou uložena do pole. Nevýhodou sice bude větší spotřeba místa, protože i když předem víme, z kolika prvočísel se bude získávat faktorizační báze, nevíme, kolik prvočísel skutečně bude ve faktorizační bázi. Na druhou stranu ale získáme konstantní přístup k přísluš-

[2] Bližší informace o procesoru lze nalézt zde: http://cpuboss.com/cpu/Intel-Core-i7-4700MQ

nému logaritmu. Výpočet prahu, jenž určuje, která spočtená hodnota polynomu je kandidátem, bude upraven tak, že jeho konstantní část bude spočtena předem a bude se dopočítávat pouze část proměnná.

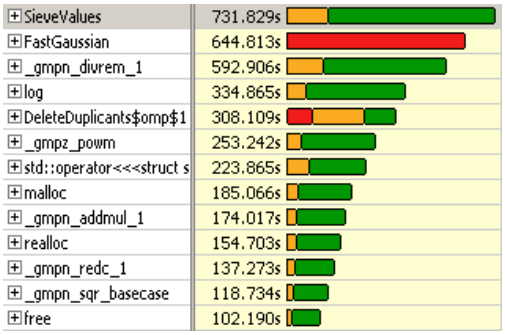
Navržené řešení bylo realizováno a následně bylo opět provedeno měření, aby se zjistilo, jaký vliv na rychlost tato změna měla. Při tomto měření bylo demonstračně faktorizováno i číslo o 70 dekadických číslicích. Výsledky měření jsou zaneseny do tabulky 2.

Úloha	Sériově	Paralelně	Urychlení optimalizací
40 dec	10.00s	4.25s	5.33
50 dec	157.86s	59.24s	5.19
60 dec	-	597.24s	5.38
70 dec	-	5059.17s	-

Tabulka 2. Naměřená doba faktorizace po 1. optimalizaci

Provedené změny přinesly zhruba 5-násobné zrychlení. Bylo ale zjištěno, že ukládání dělitelů všech hodnot x je nutné přepracovat či odstranit. Výhodou této techniky je, že při ověření kandidátů se bude dělit pouze prvočísly, které hodnotu polynomu pro dané x opravdu dělí. Tím se bude kandidát dělit mnohem méně prvočísly, než kdybychom tuto informaci neměli a museli tak dělit všemi prvočísly z faktorizační báze. Problémem je, že sběr těchto dělitelů při prosévání se provádí pro každou hodnotu x. Ukládání dělitelů je tak časově velice náročné. Dokonce se ukázalo, že je ukládání mnohem náročnější, než kdyby se tato informace neukládala a každý kandidát by se dělil všemi prvočísly z faktorizační báze. Ukládání této informace tedy bylo odstraněno. Dále byl doimplementován Knuth-Schroeppeluv algoritmus [9]. Výstupem tohoto algoritmu je konstanta, kterou je nejvhodnější zadané číslo N vynásobit. U mnoha čísel totiž nastává situace, kdy faktorizační báze nebude obsahovat malá prvočísla jako 2, 3, 5, 7, 11 atd. Čím více ale faktorizační báze takovýchto malých prvočísel obsahuje, tím pravděpodobnější je nalezení B-hladkého čísla. Vynásobením tedy zadaného čísla tímto násobitelem nám může přinést značné urychlení. Také bylo paralelizováno odstranění duplicitních relací pomocí OpenMP direktivy *#pragma omp parallel for*, protože se prochází všechny dvojice a kontroluje se, zda nejsou zkoumané relace shodné. Paralelizací se tak kontrola značně urychlí. Po těchto úpravách byla opět provedena profilace. Výsledek je ukázán na obrázku 2. Profilace byla provedena na čísle o 70 dekadických číslicích, aby se projevilo co možná nejvíce slabých míst.

Z obrázku 2 lze vypozorovat, že doba strávená ve



Obrázek 2. Profilace SIQS po provedených úpravách

funkci *SieveValue()* je velice podobná té z obrázku 1. Je nutné si ale uvědomit, že v 1. případě se prováděla profilace nad číslem s 60 dekadickými číslicemi, na obrázku 2 je však profilace s číslem o 70 dekadických číslicích, a tedy obrázek poukazuje na značné urychlení faktorizace. Pro demonstraci, číslo o 60 dekadických číslicích, které původně bylo faktorizováno za 597s, jak je ukázáno v tabulce 2, bylo po provedených úpravách faktorizováno za 119s, což přináší pro faktorizaci takto velkého čísla další zhruba 5-násobné urychlení. Je patrné, že Gaussova eliminační metoda pro hledání lineární závislosti začíná být velký problém. Pokud by Gaussova eliminační metoda nebyla upravena, nebylo by reálně možné faktorizovat číslo o 80 dekadických číslicích.

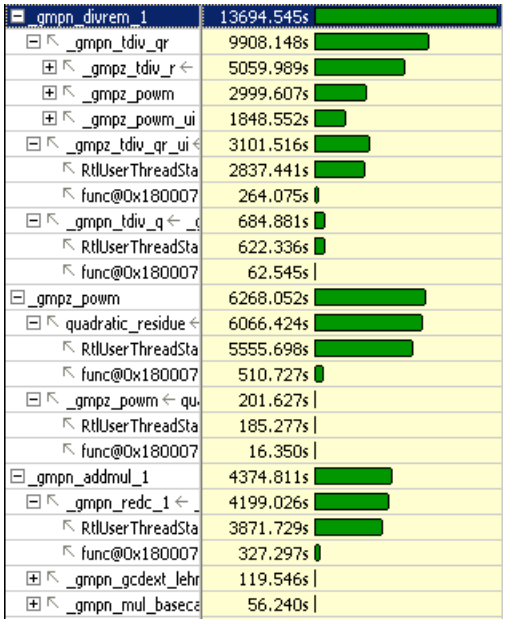
Byla provedena tedy paralelizace této metody. Ukázalo se ale, že i přes paralelizaci této metody byla doba zpracování matice příliš dlouhá. Důvodem je způsob uchovávání relací. Relace, tedy vektory exponentů, jsou ukládány do pole datového typu *bool*. Toto řešení je na jednu stranu velice pohodlné, protože každé prvočíslo je obsaženo v jednom boolu a dobře se tak s ním pracuje. Na druhou stranu ale pokud se musí provést větší kvantum operací nad relacemi, což se v případě Gaussovy eliminační metody provádí, je zpracování relací velice neefektivní. Navrženým řešením tedy je, aby se ukládání relaci provádělo do pole datového typu *integer*, kde bude uloženo tolik prvočísel, kolik bitů daný integer obsahuje. Když se pak bude provádět například operace *xor*, bude se provádět nad několika prvočísly zároveň, což povede k velké úspoře času. Dále paralelizovat odstranění singletonů, aby částí, které běží v SIQS sériově, bylo minimum. Výsledek úpravy je zanesen do tabulky 3.

Provedené úpravy přinesly opět značné urychlení. Například u čísla s 60 dekadickými číslicemi došlo oproti původní referenční implementaci k 99-násobnému urychlení. Když se SIQS zdálo dostatečně rychlé, byla provedena faktorizace čísla s 80 dekadickými číslicemi. Z důvodu velké časové náročnosti ale tak velké číslo již nebylo faktorizováno na notebooku, kde

Úloha	Sériově	Paralelně	Urychlení optimalizací
40 dec	2.12s	1.32s	3.22
50 dec	18.53s	10.72s	5.53
60 dec	102.19s	32.23s	18.53
70 dec	-	583.67s	8.67

Tabulka 3. Naměřená doba faktorizace po 2. optimalizaci

byl SIQS vyvíjen, ale na stroji s Intel Xeon E5-1650 v2[3] a 32GB pamětí. Faktorizace tohoto čísla byla úspěšná a trvala 1h 54min a 25s. Při této faktorizaci byla zároveň prováděna profilace. Výsledek je zobrazen na obrázku 3.



Obrázek 3. Profilace po změně způsobu ukládání relací

Faktorizace čísla s 80 dekadickými číslicemi proběhla nad očekávání rychle, a tak byla rovnou provedena faktorizace čísla o 90 dekadických číslicích. Faktorizace zvoleného čísla byla úspěšně dokončena za 5 hodin a 58 minut. Informace získané faktorizací tohoto čísla byly blíže analyzovány. Paměťová náročnost byla na daném stroji stále na nízké úrovni, a tak by velmi pravděpodobně bylo možné faktorizovat i číslo se 100 dekadickými číslicemi. Byl tedy učiněn pokus o pokoření této „magické" hodnoty. Faktorizace takto velkého čísla je však značně náročnější než faktorizace čísla s 90 dekadickými číslicemi a to jak z pohledu časového, tak z pohledu náročnosti na paměť. Doba faktorizace tak byla odhadována na několik dní. Faktorizace čísla o 100 dekadických číslicích byla nakonec úspěšně dokončena za 2 dny 16 hodin a 13 minut. Jelikož číslo o 100 dekadických číslicích je 332 bitů velké, ukázala tato práce, že šifrování pomocí RSA-332 by bylo velkým bezpečnostním rizikem, protože klíč této velikosti by byl nalezen do 3 dnů.

Z výsledků profilace na obrázku 3 lze však dále vypozorovat, že nyní největší spotřeba času se nachází v aritmetických operacích MPIR knihovny. Další úpravy by tak musely vést k redukci počtu těchto operací. Z tabulky 3 je však možné ještě vypozorovat, že i přes použití všech osmi logických jader na procesoru Intel i7 4700MQ, nebylo dosaženo příslušně velkého urychlení. Tento fakt má několik důvodů. Jedním z nich je, že faktorizace čísel se 40 nebo 50 dekadickými číslicemi je již velmi rychlá a měřené časy tak ovlivňuje příprava SIQS, která vždy běží sériově. Příprava SIQS pro faktorizaci čísel od 50 dekadických čísel výše je vždy stejně časově náročná, a tak čím vyšší číslo se snažíme faktorizovat, tím více je čas strávený přípravou SIQS zanedbatelnější. Tabulka 4 tak ukazuje časy naměřené při provádění samotné faktorizace.

Úloha	Sériově	Paralelně	Urychlení paralelizmem
40 dec	1.05s	0.26s	4.04
50 dec	11.60s	2.99s	3.88
60 dec	100.25s	29.65s	3.38

Tabulka 4. Naměřená doba pouze při provádění faktorizace samotné

Úpravy kódu v rámci optimalizace mnohokrát znamenaly vytvoření nějakého pole s vypočtenými hodnotami. Tyto hodnoty byly obvykle použity při faktorizaci ve větším počtu a předvypočtením se tak ušetřilo mnoho znovu opakujících se výpočtů. Tyto úpravy obvykle přinesly výrazné urychlení faktorizace. S těmito úpravami se ale vyskytl problém. Čím více těchto polí s předvypočtenými hodnotami je, tím více vzniká požadavek na paměť. Pole nejsou tak veliká, aby představovala problém pro hlavní paměť, ale jsou dostatečně velká na to, aby nastal problém s jejich uložením a setrvání v rychlé vyrovnávací paměti. Problém se stupňuje s většími čísly, jež chceme faktorizovat, protože s každým větším číslem narůstá i faktorizační báze, podle čehož se úměrně zvětšují i zmíněná pole. Jelikož faktorizace běží paralelizace, je tento problém o to větší. Nastává tak situace, kdy další úpravy podobného typu sice vedou k urychlení čísel kolem 60 dekadických čísel a méně, naopak ale vedou ke zpomalení větších čísel. Další úprava by tak vedla k vytvoření více „cache-friendly" kódu. To by ovšem znamenalo provedení hluboké analýzy kódu a následně poměrně rozsáhlé změny. Proto další úpravy zatím provedeny

[3] http://cpuboss.com/cpu/Intel-Xeon-E5-1650-v2

nebyly.

Kromě výše navrhnuté úpravy se nabízí hned několik dalších vylepšení. Jedním z nich je například paralelizace pomocí OpenMPI. Faktorizace by tak mohla probíhat na clusterech. Urychlení by tak bylo veliké. Bylo by také možné nahradit OpenMP vlastní implementací, dosáhnout tak větší kontroly nad paralelními částmi kódu a v některých případech mít možnost použít více vláken, protože OpenMP je v tomto ohledu omezené. Jinou možností je vytvořit si vlastní implementaci vektorů a map, a tedy přizpůsobit fungování těchto datových typů přímo pro potřeby SIQS. Jak je vidět, nápadů na vylepšení je mnoho. Tyto nápady budou postupem času do projektu zapracovávány a bude tak vytvořen velmi výkonný nástroj k faktorizaci čísel.

Pro srovnání SIQS implementace této práce s nejrychlejší dostupnou implementací SIQS, Msieve, byla vytvořena tabulka 5. Implementace SIQS této práce je v tabulce nazvána prostě SIQS.

Úloha	SIQS	MSieve
40 dec	1.32s	0.15s
50 dec	10.72s	0.56s
60 dec	32.23s	3.55s
70 dec	583.67s	39.03s

Tabulka 5. Srovnání SIQS této práce a Msieve

Jak je z tabulky 5 vidět, implementace vytvořená v rámci této práce nedosahuje rychlosti Msieve. Je nutné si ale uvědomit, že čas investovaný do vývoje implementace této práce, jenž je diplomovou prací, je v porovnání s časem investovaným do vývoje z pohledu člověkoroků výrazně rozdílný a to se také odráží v rychlosti jednotlivých metod. Cílem této práce také nebylo překonat rychlost Msieve, ale v rámci možností vytvořit efektivní SIQS a zároveň tuto implementaci řádně zdokumentovat, aby mohla dále sloužit k výukovým účelům. Navíc vývoj implementace ještě nebyl ukončen, a jak bylo zmíněno výše, je zde ještě mnoho nápadů, jak metodu vylepšit a urychlit.

6. Závěr

Výsledky potvrdily metodickou správnost rozdělení implementace na vytvoření referenční implementace v první fázi a na rychlostní optimalizaci ve fázi druhé. Vždy tak bylo možno rychlostní úpravy porovnat s referenční verzí a ověřit si, jestli nedošlo k porušení funkčnosti algoritmu. Měření potvrdila, že díky důsledné optimalizaci bylo dosaženo několika násobného urychlení oproti referenční implementaci. Profilace kódu byla provedena v několika iteracích a v každé iteraci byly upravena místa v kódu, která spotřebovávala nejvíce času. Podařilo se také faktorizovat číslo o 100 dekadických číslicích.

Bylo ukázáno, že i přes velmi komplikovanou teorii stojící za SIQS, je možné implementaci této metody provést nenáročně. V mnoha případech bylo komplikované najít v článcích týkajících se SIQS vysvětlení, jak daný problém prakticky vyřešit anebo co je důvodem, že některá z částí SIQS nepracuje podle předpokladů. Všechny tyto problémy byly nakonec vyřešeny a popsány v dokumentaci vytvořené v rámci této práce. Dokumentace zároveň obsahuje i konkrétní případ faktorizace čísla o 20 dekadických číslicích metodou SIQS. Kdokoli si tak může tento příklad projít a buď pochopit, jak SIQS pracuje, pokud byla teorie pro něj příliš náročná, anebo si může čtenář ověřit, že jeho vlastní implementace pracuje podle předpokladů.

Literatura

[1] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[2] Scott Patrick Contini. Factoring integers with the self-initializing quadratic sieve. 1997.

[3] Carl Pomerance. The quadratic sieve factoring algorithm. In *Advances in cryptology*, pages 169–182. Springer, 1985.

[4] Carl Pomerance. A tale of two sieves. *Biscuits of Number Theory*, 85, 2008.

[5] Michael A Morrison and John Brillhart. A method of factoring and the factorization of f_7. *Mathematics of Computation*, 29(129):183–205, 1975.

[6] Brian Carrier and Samuel S Wagstaff Jr. Implementing the hypercube quadratic sieve with two large primes. In *International Conference on Number Theory for Secure Communications*, pages 51–64, 2003.

[7] Albert Nijenhuis and Herbert S Wilf. *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014.

[8] Çetin K Koç and Sarath N Arachchige. A fast algorithm for gaussian elimination over gf (2) and its implementation on the gapp. *Journal of Parallel and Distributed Computing*, 13(1):118–122, 1991.

[9] Robert D Silverman. The multiple polynomial quadratic sieve. *Mathematics of Computation*, 48(177):329–339, 1987.