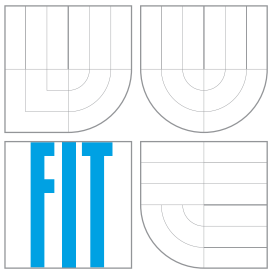
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PARALELIZACE FAKTORIZACE CELÝCH ČÍSEL Z POHLEDU LÁMÁNÍ RSA

PARALLELIZATION OF INTEGER FACTORIZATION

FROM THE VIEW OF RSA BREAKING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DOMINIK BREITENBACHER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. IVAN HOMOLIAK

BRNO 2015

Abstrakt

Práce se zabývá faktorizací celých čísel. Faktorizace je nejznámější a nejpoužívanější metodou kryptoanalýzy RSA. V rámci této práce byla vybrána a implementována faktorizační metoda zvaná SIQS. I když se jedná o nejrychlejší metodu (do 100 dekadických 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. Další možností je optimalizace kódu. Cílem této práce je také ukázat, jak jednoduše lze v mnoha případech využít paralelizace kódu a dále, jak díky podrobné analýze kódu lze dosáhnout poměrně velkého urychlení. Použitá metodika iteračního provádění optimalizací se ukázala jako velmi účinná. Touto metodikou byla implementace SIQS vylepšena tak, že faktorizace byla urychlena až 100-krát, v některých částech kódu dokonce ještě více.

Abstract

This paper follows up the factorization of integers. Factorization is the most popular and used method for RSA cryptoanalysis. The SIQS was chosen as a factorization method that will be used in this paper. Although SIQS is the fastest method (up to 100 digits), it can't be effectively computed at polynomial time, so it's needed to look up for options, how to speed up the method as much as possible. One of the possible ways is paralelization. In this case OpenMP was used. Other possible way is optimalization. The goal of this paper is also to show, how easily is possible to use paralelizion and thanks to detailed analyzation the source codes one can reach relatively large speed up. Used method of iterative optimalization showed itself as a very effective tool. Using this method the implementation of SIQS achieved almost 100 multiplied speed up and at some parts of the code even more.

Klíčová slova

Faktorizace, kvadratické síto, MPQS, SIQS, Pollardova Rho metoda, Fermatova faktorizace, Eliptické křivky, NFS, test prvočíselnosti, paralelizace, OpenMP, kryptoanalýza RSA

Keywords

Factorization, Quadratic sieve, MPQS, SIQS, Pollard Rho method, Fermat factorization, Elliptic curves, NFS, primality test, parallelization, OpenMP, RSA cryptoanalysis

Citace

Dominik Breitenbacher: Paralelizace faktorizace celých čísel
z pohledu lámání RSA, diplomová práce, Brno, FIT VUT v Brně, 2015

Paralelizace faktorizace celých čísel z pohledu lámání RSA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Ivana Homoliaka

.....
Bc. Dominik Breitenbacher
26. května 2015

Poděkování

Poděkovat bych chtěl svému vedoucímu diplomové práce Ing. Ivanu Homoliakovi za poskytnutí cenných rad, jak diplomovou práci vypracovávat, kde hledat potřebné materiály a jak s nimi efektivně pracovat, a také za ochotu pomoci při jakémkoli problému, který se mi naskytl.

Také bych rád poděkoval svému otci, Zdeňku Breitenbacherovi, za poskytnutí profilačních nástrojů a strojů, kde bylo možné práci efektivně analyzovat.

Neméně bych chtěl také poděkovat členům fóra mersenneforum, kteří díky své ochotě a svým nabylím zkušenostem v této problematice, mi byli schopni vysvětlit a objasnit problém, se kterým jsem se potkal.

Obsah

1 Úvod **4**
- 1.1 Cíle práce . . . 5
- 1.2 Struktura práce . . . 5

2 Pozadí teorie faktorizace **7**
- 2.1 Vlastní a nevlastní dělitelé . . . 7
- 2.2 Hladké číslo . . . 7
- 2.3 Eukleidovo lemma . . . 7
- 2.4 Narozeninový paradox . . . 8
- 2.5 Eulerova funkce . . . 8
- 2.6 Eulerovo kritérium . . . 9
- 2.7 Legendreův symbol . . . 9
  - 2.7.1 Jacobiho symbol . . . 9
- 2.8 Malá Fermatova věta . . . 10
- 2.9 Čínská věta o zbytcích . . . 10
- 2.10 Kongruence zbytkových tříd . . . 11
- 2.11 Algebra – Grupy, okruhy, tělesa, pole . . . 12

3 Pomocné algoritmy a kódy **13**
- 3.1 Grayův kód . . . 13
- 3.2 Euklideův algoritmus - GCD (Greatest common divisor) . . . 14
- 3.3 Eratosthenovo síto . . . 14
- 3.4 Tonelliho-Shanksův algoritmus . . . 15

4 RSA **16**
- 4.1 Generování klíče . . . 16
- 4.2 Šifrování a dešifrování . . . 17
  - 4.2.1 Výpočet příkladu . . . 17
- 4.3 Podepisování pomocí RSA . . . 18
- 4.4 Zajištění důvěrnosti, integrity, autentizace a nepopiratelnosti . . . 19

5 Test Prvočíselnosti **20**
- 5.1 Naivní testy prvočíselnosti . . . 20
- 5.2 Fermatův test prvočíselnosti . . . 20
- 5.3 Millerův-Rabinův test prvočíselnosti . . . 21
- 5.4 Solovayův-Strassenův test prvočíselnosti . . . 22

6 Faktorizační algoritmy **24**
6.1 Zkusmé dělení . . . 24
6.2 Pollardova ρ metoda . . . 25
6.3 Pollardova $p-1$ metoda . . . 26
6.4 Metoda eliptických křivek . . . 27
6.5 Fermatova faktorizace . . . 31
6.6 Kvadratické síto (QS) . . . 33
6.6.1 Prosívání . . . 34
6.6.2 Metody pro nalezení lineární závislosti . . . 36
6.6.3 Large Prime Variation . . . 37
6.7 Multipolynomiální kvadratické síto (MPQS) . . . 38
6.8 Self-Initialization quadratic sieve (SIQS) . . . 39
6.9 Obecné číselně teoretické síto (GNFS) . . . 40

7 Zvolená metoda faktorizace a návrh implementace **42**
7.1 Rozbor metod . . . 42
7.2 Zvolená faktorizační metoda a návrh paralelizace . . . 42
7.3 Možnosti paralelizace . . . 43
7.3.1 Klasická vlákna . . . 43
7.3.2 nVidia CUDA . . . 44
7.3.3 OpenMP . . . 44
7.3.4 OpenMPI . . . 44
7.4 Návrh řešení a postup implementace . . . 45

8 Praktický příklad faktorizace metodou SIQS **48**
8.1 Zadání . . . 48
8.2 Příprava SIQS . . . 48
8.3 Generování polynomu . . . 49
8.3.1 Generování koeficientu a . . . 49
8.3.2 Získání koeficientů b a c . . . 50
8.4 Fáze prosévání . . . 51
8.4.1 Hledání kandidátů B-hladkých čísel . . . 51
8.4.2 Ověření kandidátů . . . 52
8.5 Předzpracování matice . . . 53
8.6 Fáze lineární algebry . . . 53
8.7 Spočtení dělitele zadaného čísla . . . 54
8.8 Dodatek k příkladu . . . 54

9 Implementace SIQS **55**
9.1 Zadání čísla k faktorizaci . . . 55
9.2 Příprava SIQS . . . 55
9.3 Generování polynomu . . . 56
9.3.1 Generování koeficientu a . . . 56
9.3.2 Generování koeficientu b a c . . . 57
9.3.3 Zavedený paralelizmus . . . 57
9.4 Fáze prosévání . . . 58
9.4.1 Výpočet kořenů . . . 58
9.4.2 Hledání kandidátů B-hladkých čísel . . . 58

9.4.3 Ověření kandidátů . . . 58
9.5 Zpracování relací . . . 59
9.6 Fáze lineární algebry . . . 60
9.7 Získání dělitele zadaného čísla . . . 61

10 Měření a optimalizace **62**
10.1 Referenční verze . . . 62
10.2 Optimalizace referenční verze . . . 63
10.2.1 Optimalizace pro rychlejší faktorizaci 60 dekadických číslic . . . 63
10.2.2 Optimalizace pro rychlejší faktorizaci 70 dekadických číslic . . . 66
10.3 Paměťové náročnosti . . . 73
10.4 Porovnání procesorů . . . 73
10.5 Porovnání s Msieve . . . 74
10.6 Porovnání Pollard ρ metody a SIQS . . . 74
10.7 Návrhy na další vylepšení . . . 75

11 Závěr **76**

A Obsah DVD **80**

Kapitola 1

Úvod

V dnešní době internetu lze většinu věcí vyřídit na počítači z pohodlí domova. Příkladem může být provedení nějaké bankovní transakce, kdy již nemusíme jít na pobočku, ale můžeme se přihlásit ke svému účtu přes webový prohlížeč a danou transakci provést sami. Jiným příkladem by mohlo být objednávání zboží přes e-shop, kde si najdeme potřebné zboží, koupíme a v případě potřeby si jej můžeme nechat přivézt až domů.

To vše by ale nebylo možné, kdyby neexistoval nějaký mechanizmus, který nám zaručuje, že všechny přenášené informace, které jsou většinou velice citlivé z pohledu našeho soukromí, jsou nějakým způsobem chráněny před zneužitím. V opačném případě by totiž útočník mohl například sledovat naši komunikaci s bankou a vysledovat naše přihlašovací údaje. Tím by získal přístup k účtu a náš účet velmi pravděpodobně zneužil. Byl tedy zaveden mechanizmus, který nazýváme šifrování. Tento mechanizmus zajišťuje, že vždy, když jsou odesílány nějaké informace, jsou tyto informace určitým způsobem transformovány, aby byly pro případného útočníka, který se snaží naši komunikaci sledovat, nesrozumitelné a pro něj tedy nepoužitelné. Obvykle se informace šifrují na základě nějakého klíče, kdy klíč mají obě komunikující strany. Druhé straně nevadí, že dostane naše informace v nesrozumitelné podobě, protože má příslušný klíč, kterým si tyto informace zase transformuje zpátky. Šifrování rozdělujeme na dva typy, takzvané symetrické, kdy obě strany používají k šifrované komunikaci stejný klíč, anebo takzvané asymetrické, kdy jedna strana má privátní klíč, který se nezveřejňuje, a druhá strana má klíč veřejný.

Veřejný klíč obecně slouží k zašifrování zprávy, kterou chce jeden z komunikujících poslat. Zašifrované informace dokáže zpátky rozšifrovat pouze vlastník privátního klíče. Délka jak privátního, tak veřejného klíče je pro danou dvojici vždy stejná a dnes je obvyklá velikost klíče 1024 či 2048 bitů.

Typickým a velice užívaným představitelem asymetrické šifry je RSA (viz kap. 4). Klíč je u této šifry vygenerován na základě vynásobení dvou prvočísel a bezpečnost je tak založena na předpokladu, že nalézt prvočísla, z kterých klíč vznikl, je výpočetně nemožné. Obecně totiž vynásobit dvě čísla je triviální záležitostí, najít však zpětně ze kterých čísel je výsledné číslo složeno, je problém velice náročný, dokonce tak náročný, že jej obecně nelze řešit v konečném čase. Proces hledání čísel, ze kterých bylo dané číslo složeno, nazýváme faktorizace.

Pro faktorizaci celých čísel bylo objeveno mnoho různých metod, které se snaží najít faktory složeného čísla. Nepovedlo se však najít metodu, která by byla schopna řešit tento problém dostatečně rychle, což je také důsledkem neřešitelnosti problému faktorizace

v rozumném čase. Přesto bylo nalezeno několik velmi sofistikovaných metod, které hledání faktorů velice urychlilo. Díky těmto metodám musí být dnes délky klíčů asymetrických šifer dlouhé minimálně 1024 bitů, protože takto dlouhé klíče i s použitím sofistikovaných metod nejsme schopni na dostupných výpočetních prostředcích faktorizovat dostatečně rychle. Největší faktorizovaný RSA klíč má velikost 768 bitů. Tohoto úspěchu se povedlo dosáhnout v roce 2009, a i když získání faktorů trvalo přibližně dva roky, není doporučeno klíče této délky používat [8].

1.1 Cíle práce

Cílem této práce je prozkoumat různé faktorizační metody, zhodnotit jejich možnosti paralelizace a nejvhodnější metodu implementovat. K implementaci zvolené metody budou použity moderní nástroje a metody. Na základě implementace a výsledků měření bude diskutováno, jaký přínos mělo nasazení těchto moderních metod či nástrojů a případně, zda existují i jiné či další možnosti, jak implementovanou metodu vylepšit, aby faktorizace touto metodou byla co nejefektivnější.

1.2 Struktura práce

V kapitole 2 jsou tak popsány nejdůležitější pojmy, které se používají v problematice faktorizace, a čtenář si je tak má možnost v případě neznalosti doplnit bez toho, aby musel tyto pojmy hledat v jiné literatuře.

Kapitola 3 popisuje algoritmy, které jsou nedílnou součástí některých faktorizačních metod. Například Euklideův algoritmus (viz kap. 3.2) pro hledání největšího společného dělitele používají téměř všechny faktorizační metody k finálnímu nalezení faktoru složeného čísla.

RSA je popsáno v kapitole 4. Čtenář zde najde popis šifrovací metody RSA, jak se postupuje při generování veřejného a soukromého klíče a jak se provádí šifrování a dešifrování zprávy. Kromě toho je zde také předveden konkrétní příklad šifrování a dešifrování, aby čtenář v případě potřeby funkčnost RSA lépe pochopil. Kapitola také popisuje, jak lze pomocí RSA vytvářet digitální podpisy a co udělat, aby byla u přenášené zprávy zajištěna důvěrnost, integrita, autentizace a nepopiratelnost.

Kapitola 5 prezentuje některé testy prvočíselnosti. Test prvočíselnosti je důležité provést vždy před samotnou faktorizací, protože pokud bychom se pokusili provést faktorizaci prvočísla, žádného rozumného výsledku bychom nedosáhli a navíc bychom na tento výsledek museli čekat po celý čas procesu faktorizace.

Kapitola 6 pak popisuje jak základní, tak ty nejznámější a nejpoužívanější metody faktorizace. Kapitola je rozdělena na dvě části. V první jsou popisovány faktorizační metody s exponenciální časovou složitostí, které dnes používáme spíše pro faktorizaci čísel do 30 dekadických číslic. V druhé části jsou pak popsány faktorizační metody se subexponenciální časovou složitostí. Metody popsané v této části se používají pro faktorizaci čísel, které mají nad 30 dekadických číslic, konkrétně pak metoda GNFS (viz kap. 6.9) se dnes používá pro faktorizaci těch největších čísel.

Výběr vhodné metody k implementování a zdůvodnění výběru je popsáno v kapitole 7. V této kapitole se také nachází rozbor jednotlivých částí algoritmu a je zde popsán podrobný

návrh, jak jednotlivé části algoritmu budou implementovány. Kapitola také obsahuje popis, jak bude použit paralelizmus, aby došlo k maximálnímu urychlení metody.

Jelikož vybraná metoda SIQS (viz kap. 6.8) je poměrně složitá a tak i náročná na pochopení, byl vytvořen příklad, pomocí kterého se postupně prochází jednotlivými částmi tohoto algoritmu. Příklad se tak snaží co nejdetailněji metodu popsat, aby čtenář mohl metodu pochopit, i kdyby se mu nepodařilo metodu plně pochopit z jejího teoretického popisu. Demonstrační příklad je možné nalézt v kapitole 8.

Samotná implementace metody SIQS je popsána v kapitole 9. V této kapitole je možné se tedy dozvědět, jak metodu lze konkrétně implementovat. Jsou zde také popsány problémy, se kterými je možné se při implementaci setkat, a je zde popsáno, jak tyto problémy řešit či jak se jim vyvarovat.

Po dokončení implementace byla provedena měření rychlosti metody. Na základě výsledků byla provedená analýza a navrženy změny k dosažení zvýšení rychlosti. Následně bylo opět provedeno měření a profilace. Takto bylo provedeno několik iterací. Při této metodice bylo dosaženo až 100-násobného zrychlení implementace. Průběh měření, profilací a optimalizací je popsán v kapitole 10.

Kapitola 2

Pozadí teorie faktorizace

Kapitola vysvětluje pojmy, které jsou použity v této práci. Pokud bude v textu zmíněn algoritmus, který není přímo faktorizační metodou, pak bude tento algoritmus popsán v této kapitole. To platí i u vět, formulí, definic apod. Čtenář tak bude mít vždy možnost, v případě neznalosti pojmu, nahlédnout do této kapitoly, kde bude daný pojem blíže popsán.

2.1 Vlastní a nevlastní dělitelé

- Nevlastní dělitelé pro číslo p jsou-1, 1, p a $-p$
- Existují-li pro číslo p další dělitelé, pak se jedná o dělitele vlastní

V rámci problému faktorizace jsou pro nás nevlastní dělitelé nezajímaví a naopak, pokud faktorizace vede k zisku nevlastního dělitele, znamená to pro nás nastavit faktorizační metodě nové parametry a proces faktorizace opakovat. Jinou možností je, že jsme se pokusili faktorizovat prvočíslo. Takové situaci bychom měli předcházet spuštěním testu prvočíselnosti (viz kap. 5) před samotným spuštěním procesu faktorizace.

2.2 Hladké číslo

Celé číslo je označeno jako B-hladké, pokud toto číslo nemá většího prvočíselného dělitele, než zadané celé číslo B.

Například číslo 30 má prvočíselný rozklad $2 * 3 * 5$, a je tedy možné toto číslo označit jako 5-hladké, protože neexistuje prvočíslo $p > 5$, pro které by platilo $p|30$.

Dále se také rozlišuje pojem B-mocné. Celé číslo m je B-mocné, pokud platí:

$$m \quad = \quad \prod p_i^{\alpha_i},\ kde\ \alpha_i\ \geq 2\ a\ p_i^{\alpha_i} \leq B \tag{2.1}$$

Jedná se tedy o taková celá čísla, jež lze rozložit na mocniny prvočísel menších než B.

2.3 Eukleidovo lemma

Eukleidovo lemma tvrdí, že pokud je zkoumané celé číslo n vytvořeno součinem dvou celých čísel a a b, pak máme-li prvočíslo p, pro které platí $p|n$, pak také platí, že $p|a$ nebo $p|b$.

Tato vlastnost odlišuje způsob faktorizace některých faktorizačních metod. Pokud totiž n je složeným číslem, kde čísla a a b jsou jeho faktory, pak i tato čísla mohou být dále složená. Některé faktorizační metody tak rozkládají zadané číslo tak dlouho, dokud jej nerozloží na součin prvočísel. Faktorizační metody, které se pokouší rozložit čísla o velkém počtu dekadických číslic, řekněme čísla o více jak 60 dekadických číslicích, však ukončí svůj výpočet již při nalezení prvního faktoru. Takové metody se obvykle pokouší najít faktor nějakého RSA klíče (viz kap. 4), kdy klíč je získán vynásobením dvou prvočísel, a tak nalezení pouze jednoho faktoru postačuje.

2.4 Narozeninový paradox

Narozeninový paradox se zabývá pravděpodobností, s jakou je možno najít ve skupině lidí o počtu n dva takové, že budou mít narozeniny ve stejný den. Pokud máme skupinu o 23 lidech a vybereme jednoho, pro kterého chceme zjistit, zda má narozeniny ve stejný den jako někdo jiný ze skupiny, pak pravděpodobnost takové události je $^{22}/_{365}$. Pokud ale zkoušíme najít dvojici lidí se stejným dnem narození u všech lidí ze skupiny, pak je těchto možností $23 * \frac{22}{2} = 253$. Výpočet pravděpodobnosti pro libovolný počet n lidí ve skupině je jednodušší, když jej spočteme jako komplement k pravděpodobnosti, že všichni ve skupině mají den narození rozdílný. Poté pravděpodobnost vypočteme takto:

$$\overline{p}(n) = 1 * (1 - \frac{1}{365}) * (1 - \frac{2}{365}) * \cdots * (1 - \frac{n-1}{365}) = \frac{365!}{365^n * (365 - n)!} \quad (2.2)$$

$$p(n) = 1 - \overline{p}(n) \quad (2.3)$$

Narozeninový paradox lze využít pro jakýkoli problém, ve kterém hledáme výskyt stejné události pro daný počet prvků. Proto lze například zaznamenat použití narozeninového paradoxu pro odhad, kdy se začne přibližně periodicky opakovat sekvence iterací Pollardovy ρ metody (viz kap 6.2). Výpočet je pak nutno upravit takto:

$$\overline{p}(n) = 1 * (1 - \frac{1}{x}) * (1 - \frac{2}{x}) * \cdots * (1 - \frac{n-1}{x}) = \frac{x!}{x^n * (x - n)!} \quad (2.4)$$

$$p(n) = 1 - \overline{p}(n), \quad (2.5)$$

kde x je počet prvků ve skupině.

Přesný výpočet pravděpodobnosti je poměrně zdlouhavý, a proto existují aproximace výpočtu, které jsou značně rychlejší, a získaná hodnota pravděpodobnosti nám postačuje i s případnou odchylkou. V této práci se bude využívat varianty aproximace pomocí odmocniny. Pokud budeme chtít zjistit, kolik prvků z kolekce je třeba k nalezení shodné dvojice s pravděpodobností $p(n) = 0.5$, bude výpočet proveden takto:

$$\#(n) = \sqrt{n} \quad (2.6)$$

2.5 Eulerova funkce

Jedná se o funkci $\varphi : \mathbb{N} \mapsto \mathbb{N}$, která pro zadané číslo počítá, kolik čísel menších než číslo zadané je nesoudělných s tímto číslem, tedy platí $\varphi(n) = \#(k)$, kde $1 \leq k < n$ a $GCD(k, n) = 1$. Z uvedených vlastností vyplývá:

$$\begin{array}{lcl} \varphi(1) & = & 1 \\ \varphi(p) & = & p-1, \textit{ kde } p \textit{ je prvočíslo} \\ \varphi(p^m) & = & (p-1)*p^{m-1}, \textit{ kde } p \textit{ je prvočíslo a } m \textit{ je kladné celé číslo} \end{array} \tag{2.7}$$

Dále platí, že pokud je n složené číslo, tedy $n = a * b$, pak:

$$\varphi(n) = \varphi(a) * \varphi(b) \tag{2.8}$$

Eulerova funkce se například využívá při generování veřejného a privátního klíče asymetrické šifry RSA (viz kap. 4).

2.6 Eulerovo kritérium

Eulerovo kritérium je formule, která nám říká, zda existuje nějaké celé číslo, jež je kvadratickým zbytkem modulo p, kde p je prvočíslo.

Mějme liché prvočíslo p a nějaké celé číslo a, jež je nesoudělné s p, tzn. $GCD(a,p) = 1$ (GCD viz kap. 3.2). Pak je Eulerovo kritérium definováno takto:

$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod p & \textit{Existuje-li celé číslo } x \textit{ takové, že } a \equiv x^2 \pmod p \\ -1 \pmod p & \textit{Pokud takové celé číslo neexistuje} \end{cases} \tag{2.9}$$

Definici je také možno zapsat pomocí Legendreova symbolu (viz kap. 2.7):

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p \tag{2.10}$$

2.7 Legendreův symbol

Legendreův symbol $\left(\frac{a}{p}\right)$ je funkce, která je pro liché prvočíslo p definována takto:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \textit{pokud } a \equiv 0 \pmod p, \\ 1, & \textit{pokud } a \textit{ je kvadratický zbytek} \pmod p, \\ -1, & \textit{pokud } a \textit{ je kvadratický nezbytek} \pmod p \end{cases} \tag{2.11}$$

2.7.1 Jacobiho symbol

Jacobiho symbol $\left(\frac{a}{n}\right)$ je zobecněním Legendreova symbolu a pro lichá čísla $n \geq 3$ je definován na základě prvočíselného rozkladu čísla n takto:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} * \left(\frac{a}{p_2}\right)^{\alpha_2} * \cdots * \left(\frac{a}{p_k}\right)^{\alpha_k}, \tag{2.12}$$

kde $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$. Jacobiho symbol je také definován pro $n = 1$, kde platí $\left(\frac{a}{1}\right) = 1$.

2.8 Malá Fermatova věta

Tato věta tvrdí, že pro každé prvočíslo p a pro každé celé číslo a takové, že $GCD(a, p) = 1$, platí:

$$\begin{aligned} a^{p-1} &\equiv 1\,(mod\,p),\ nebo\ také \\ a^{p} &\equiv a\,(mod\,p) \end{aligned} \tag{2.13}$$

Tvrzení Malé Fermatovy věty je využito u mnoha testů prvočíselnosti jako například Fermatův test prvočíselnosti (viz kap. 5.2) či Millerův-Rabinův test prvočíselnosti (viz kap. 5.3). Základ v Malé Fermatově větě však také nachází faktorizační metoda Pollard $p - 1$ (viz kap. 6.3).

2.9 Čínská věta o zbytcích

Toto tvrzení popsal v problému číslo 26 čínský matematik Sun Tsu Suan-Ching ve 3. až 5. století našeho letopočtu [6]. Řešeným problémem bylo spočítat počet vojáků, pokud známe několik variant seřazení vojáku do řad o určitém počtu a počet vojáků v poslední řadě při zvolené variantě. Blíže matematicky zapsáno, mějme po dvojicích navzájem nesoudělná přirozená čísla $n_1, n_2, \ldots, n_k$, kdy $n_i \geq 2$ pro $i = 1, \ldots, k$. Pak máme soustavu rovnic:

$$\begin{aligned} x &\equiv a_1\,(mod\,n_1) \\ x &\equiv a_2\,(mod\,n_2) \\ &\vdots \\ x &\equiv a_k\,(mod\,n_k), \end{aligned} \tag{2.14}$$

kde x je hledaný počet vojáků, $n_1, n_2, \ldots, n_k$ jsou počty vojáků v řadě a $a_1, a_2, \ldots, a_k$ vyjadřují počty vojáků v poslední řadě. Ze soustavy rovnic lze vyčíst, že hledané x bude ležet v intervalu daném produktem všech $n_1, n_2, \ldots, n_k$, které označíme N, tzn. $N = \prod n_i$ pro $i = 1, \ldots, k$ a tedy $x \in \langle 1; N \rangle$.

Příklad, který výpočet lépe vysvětlí. Vojáky seřadíme do řad o 5 vojácích a v poslední řadě jsou 4. Následně je seřadíme do řad o 7 vojácích, zbude 1. Kolik je celkem vojáků? Vytvořme soustavu rovnic:

$$\begin{aligned} x &= 5 * k + 4 \\ x &= 7 * l + 1 \end{aligned} \tag{2.15}$$

Víme, že x bude v rozmezí 1 až 35, protože $n_1 = 5$ a $n_2 = 7$ a tedy $N = n_1 * n_2 = 35$. Nyní soustavu přepíšeme dle Čínské větě o zbytcích:

$$\begin{aligned} x &\equiv 4\,(mod\,n_1) \\ x &\equiv 1\,(mod\,n_2) \end{aligned} \tag{2.16}$$

Provedeme substituci:

$$5 * k + 4 \equiv 1\,(mod\,7) \tag{2.17}$$

Upravíme:

$$5 * k \equiv -3\ (mod\,7) \tag{2.18}$$

Zbavíme se záporných hodnot, a tedy:

$$5 * k \equiv 4 \,(mod\, 7) \tag{2.19}$$

Potřebujeme se zbavit násobku k, rovnici tedy vynásobíme inverzním číslem čísla 5 pro modulo 7, tj. 3, protože $5 * 3 \equiv 1 \,(mod\, 7)$. Tím získáme:

$$k \equiv 5 \,(mod\, 7) \tag{2.20}$$

Při dosazení k do původní rovnice dostaneme:

$$x = 5 * 5 + 4 = 29 \tag{2.21}$$

Počet vojáků je tedy 29.

2.10 Kongruence zbytkových tříd

Kongruence zbytkových tříd, značená operátorem $\equiv$, je relace daná vztahem:

$$a \quad \equiv \quad b \,(mod\, n) \Leftrightarrow n|\,(b - a)\,, \tag{2.22}$$

kde $n \geq 2$ a vyjadřuje tak rozklad množiny $\mathbb{Z}$ na zbytkové třídy po dělení číslem n. Takový prostor pak označujeme $\mathbb{Z}_n$. Vlastnosti kongruence zbytkových tříd jsou následující:

$$\begin{aligned} a \equiv b \wedge c \equiv d \quad &\Longrightarrow \quad a + c \equiv b + d \\ a \equiv b \wedge c \equiv d \quad &\Longrightarrow \quad a - c \equiv b - d \\ a \equiv b \wedge c \equiv d \quad &\Longrightarrow \quad a * c \equiv b * d \end{aligned} \tag{2.23}$$

Dále by ještě bylo možno vyjádřit operaci dělení. Operaci dělení x/y lze však transformovat na násobení $x * y^{-1}$ a inverzní prvek y^{-1} spočteme podle vztahu:

$$y * y^{-1} \quad \equiv \quad 1 \,(mod\, n) \tag{2.24}$$

Každá zbytková třída v $\mathbb{Z}_n$ má svého zástupce. Ten zastupuje všechna celá čísla z původní množiny $\mathbb{Z}$, která spadají do stejné zbytkové třídy. Pokud máme například $n = 4$, pak pracujeme v prostoru $\mathbb{Z}_4$ a zbytkové třídy tedy budou následující:

$$\begin{aligned} C_0 &= \{0,\, 4,\, 8, \ldots\} \\ C_1 &= \{1,\, 5,\, 9, \ldots\} \\ C_2 &= \{2,\, 6,\, 10, \ldots\} \\ C_3 &= \{3,\, 7,\, 11, \ldots\}\,, \end{aligned} \tag{2.25}$$

kde zástupce třídy C_0 je 0, zástupcem třídy C_1 je 1, třídu C_2 zastupuje 2 a poslední třídu C_3 zastupuje 3. Jelikož ale pracujeme s celými čísly, pak musíme uvažovat i záporná čísla. U záporných čísel je těžší určit, do které zbytkové třídy patří. Je však možné si vypomoci vztahem:

$$c * n + x \quad = \quad y, \tag{2.26}$$

kde $x \in \mathbb{Z}$ je záporné číslo, u kterého chceme zjistit, do jaké zbytkové třídy v rámci prostoru $\mathbb{Z}_n$ náleží. Proměnná c vyjadřuje násobek čísla n takový, aby platilo $c*n > x$ a y je zástupce, který jednoznačně určuje zbytkovou třídu, do které číslo x spadá. Ve skutečnosti tak množiny zbytkových tříd vypadají následovně:

$$\begin{aligned} C_0 &= \{\ldots, -8, -4, 0, 4, 8, \ldots\} \\ C_1 &= \{\ldots, -7, -3, 1, 5, 9, \ldots\} \\ C_2 &= \{\ldots, -6, -2, 2, 6, 10, \ldots\} \\ C_3 &= \{\ldots, -5, -1, 3, 7, 11, \ldots\} \end{aligned} \tag{2.27}$$

2.11 Algebra – Grupy, okruhy, tělesa, pole

Definice 1. Algebra $(A, \cdot)$ se nazývá *grupoid*, pokud operace $\cdot$ je definována jako zobrazení $\cdot : A \times A \to A$. Tím, že je operace $\cdot$ binární, říkáme, že Algebra $(A, \cdot)$ je typu (2).

Definice 2. Grupoid $(H, \cdot)$ se nazývá *pologrupa* právě tehdy, když $\cdot$ je asociativní.

Definice 3. Pologrupa $(H, \cdot)$ se nazývá *monoid* právě tehdy, když existuje neutrální prvek e, to znamená, že existuje prvek takový, že platí $x \cdot e = x$, kde $x \in H$.

Definice 4. Monoid $(G, \cdot)$ se nazývá *grupa* právě tehdy, když pro každý prvek $x \in G$ platí, že je invertibilní, to znamená, že existuje $x^{-1} \in G$ takové, že $x \cdot x^{-1} = x^{-1} \cdot x = e$.

Definice 5. Grupa $(G, \cdot)$ se nazývá *Abelovská grupa* právě tehdy, když $\cdot$ je komutativní.

Definice 6. Algebra $(R, +, \cdot)$ typu (2,2) se nazývá *okruh* právě tehdy, když algebra $(R, +)$ je komutativní grupa, algebra $(R, \cdot)$ je pologrupa a zároveň mezi operacemi $+$ a $\cdot$ platí distributivita.

Definice 7. Algebra $(R, +, 0, -, \cdot, 1)$ typu (2,0,1,2,0) se nazývá *okruh s jednotkovým prvkem* právě tehdy, když $(R, +, 0, -, \cdot)$ je okruh a pro operaci $\cdot$ je 1 neutrálním prvkem.

Definice 8. Okruh s jednotkovým prvkem $(R, +, 0, -, \cdot, 1)$ se nazývá *těleso* právě tehdy, když $0 \neq 1$ a $(R \setminus \{0\}, \cdot)$ je grupa.

Definice 9. Okruh s jednotkovým prvkem $(R, +, 0, -, \cdot, 1)$ se nazývá *pole* právě tehdy, když $\cdot$ je komutativní.

Kapitola 3

Pomocné algoritmy a kódy

3.1 Grayův kód

Grayův kód, někdy označován také jako zrcadlový binární kód, vyjadřuje binární soustavu, ve které se dvě sousední čísla od sebe liší pouze v jednom bitu. Grayův kód se nejčastěji používá pro opravu chyb v digitální komunikaci. V rámci problému faktorizace čísel najdeme použití tohoto kódu například u metody SIQS (viz kap. 6.8), kde je používán pro vybírání následující kombinace hodnot na základě aktuální kombinace. Ukázka, jak může Grayův kód vypadat:

Dekadicky	Binárně	Grayův kód
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Obrázek 3.1: Srovnání klasické binární soustavy s Grayovým kódem

3.2 Euklideův algoritmus - GCD (Greatest common divisor)

Tento algoritmus určuje největšího společného dělitele dvou přirozených čísel a je základem mnoha faktorizačních metod.

Algoritmus 1: Eukleidův algoritmus

Input: x, y
Output: a

```
1: while y ≠ 0 do
2:     t = b;
3:     b = a % b;
4:     a = t;
5: end while
6: return a
```

3.3 Eratosthenovo síto

Eratosthenovo síto je algoritmus, který prosívá přirozená čísla z intervalu $\langle 2; n \rangle$, kde n je celé číslo vyjadřující hranici. Principiálně se postupuje tak, že se vybere první prvek x z množiny zadané intervalem a vyřadí se všechna čísla y_i, pro která platí $x|y_i$. Poté je vybrán další prvek a opět jsou vyřazeny všechny násobky tohoto prvku. Postupujeme tak dlouho, dokud nejsou vyřazeny všechny násobky všech prvočísel, a v množině tak zůstanou pouze prvočísla. Tato situace nastává, když je vybráno první číslo, které je větší $\sqrt{n}$. Je jisté, že všechna čísla za touto hodnotou jsou již prvočísla a všechny násobky předešlých prvočísel již byla vyřazena.

Algoritmus pro Eratosthenovo síto by mohl vypadat následovně:

Algoritmus 2: Eratosthenovo síto

Input: $n, bool \quad X[]$
Output: $bool \quad X[]$

```
1: for 2 ≤ i ≤ √n do
2:     if X[i] = True then
3:         for j ≤ n, s krokem i do
4:             X[i] = False;
5:         end for
6:     end if
7: end for
8: return X[]
```

3.4 Tonelliho-Shanksův algoritmus

Tento algoritmus řeší kongruenci:

$$x^2 \equiv n \ (mod\, p) \tag{3.1}$$

kde n je kvadratický zbytek čísla x^2 modulo p, jež je lichým prvočíslem. Ověřit, zda číslo n je kvadratický zbytek, je možno provést například pomocí Legendreova symbolu (viz kap. 2.7). Tonelliho-Shanksův algoritmus je využit například při faktorizaci metodou MPQS (viz kap. 6.7) či SIQS (viz kap. 6.8), kde je použit pro výpočet koeficientů polynomu [1].

Algoritmus vychází z Eulerova kritéria (viz kap. 2.6). Číslo $p-1$ je možno zapsat ve tvaru $2^s d$, kde d je liché číslo. Pokud pro zadané prvočíslo p platí $p \equiv 3 \ (mod\, 4)$, pak je možné výpočet Tonelliho-Shanksova algoritmu úplně vynechat a spočíst x přímo pomocí vztahu $x \equiv \pm n^{\frac{p+1}{4}}$. Není-li podmínka splněna, pak pseudokód algoritmu vypadá následovně:

Algoritmus 3: Tonelliho-Shanksův algoritmus

Input: n, p, s, d, z, které splňuje podmínku $\left(\frac{z}{p}\right) = -1$
Output: x

$c \equiv z^d$ (mod p) ;
$x \equiv n^{\frac{d+1}{2}}$ (mod p) ;
$t \equiv n^d$ (mod p) ;
$M = s$ **while** $t \not\equiv 1$ *(mod p)* **do**
    Najdi nejmenší $i, 0 < i < M$ takové, že bude splněna podmínka $t^{2^i} \equiv 1$ (mod p) ;
    $b \equiv c^{2^{M-i-1}}$ (mod p) ;
    $x \equiv xb$ (mod p) ;
    $t \equiv tb^2$ (mod p) ;
    $c \equiv b^2$ (mod p) ;
    $M = i$
end while
return x

V pseudokódu je vrácen výsledek jen jednoho řešení, tato úloha má však řešení dvě. Druhé řešení získáme výpočtem $x_2 = p - x$.

Kapitola 4

RSA

Pánové Ronald L. Rivest, Leonard Adleman a Adi Shamir přišli roku 1977 s implementací asymetrické šifry, kterou pojmenovali dle svých počátečních písmen příjmení, RSA [20]. RSA je dnes jednou z nejpoužívanějších šifrovacích metod, kterou lze použít i pro podepisování. Zajímavé na metodě je, že není založena na složitých matematických základech, ale naopak na prostém násobení. Základem metody je totiž násobení dvou velkých prvočísel. Platí, že násobení je triviální operace, kdežto faktorizace složeného čísla, a tedy zpětné nalezení daných prvočísel je problém výpočetně velice náročný. Faktorizace je považovaná za problém obecně neřešitelný v konečném čase, a tak tvůrci RSA předpokládají, že při zvolení klíče rozumné délky odradí útočníka fakt, že faktorizace klíče by trvala příliš mnoho času vzhledem k tomu, jakou cenu má informace zašifrovaná tímto klíčem. Dnes je obvyklá délka klíčů 1024 nebo 2048 bitů, protože na dostupných výpočetních prostředcích se stávajícími faktorizačními metodami je faktorizace takto velkých klíčů v rozumném čase nemožná.

4.1 Generování klíče

Nejdříve je nutné si vygenerovat privátní a veřejný klíč. Privátní klíč obecně slouží k dešifrování zpráv zaslaných druhou stranou, která zprávu zašifrovala veřejným klíčem. Vygenerování klíčů probíhá tak, že si zvolíme dvě velká prvočísla p a q a spočteme:

$$n = p * q \tag{4.1}$$

Následně spočteme Eulerovu funkci (viz kap. 2.5):

$$\varphi(n) = (p-1) * (q-1) \tag{4.2}$$

Poté si zvolíme číslo e takové, pro které platí:

$$GCD(e, \varphi(n)) = 1 \wedge e < \varphi(n) \tag{4.3}$$

Nyní máme vše potřebné pro veřejný klíč:

$$k_{pub} = (n, e) \tag{4.4}$$

Z e spočteme d, jež je multiplikativní inverzí e modulo $\varphi(n)$ (viz kap. 2.10), tedy:

$$e * d \quad = \quad 1 \ (mod\, \varphi(n)) \tag{4.5}$$

neboli:

$$d \quad = \quad e^{-1} \ (mod\, \varphi(n))$$

Privátní klíč je pak tvořen dvojicí

$$k_{priv} \quad = \quad (n, d) \tag{4.6}$$

4.2 Šifrování a dešifrování

Šifrování následně probíhá tak, že například text, který chceme zašifrovat a přenést, musíme nejdříve vyjádřit celočíselně. Pokud by zpráva měla být delší neboli větší než n, pak musíme tuto zprávu rozdělit na části tak, aby hodnota každé části byla menší než n. Jednotlivé části pak šifrujeme dle vzorce:

$$C \quad = \quad M^e \ (mod\, n)\,, \tag{4.7}$$

kde M je část či celá zpráva a C je výsledek operace, tedy zašifrovaná zpráva. Dešifrování je inverzní operací a řídí se tak vzorcem:

$$M \quad = \quad C^d \ (mod\, n) \tag{4.8}$$

Šifrováním pomocí RSA získáváme důvěrnost, tedy zajišťujeme, že případný útočník není schopen přenášenou zprávu číst, a jediný, kdo si může zprávu přečíst, je vlastník privátního klíče.

4.2.1 Výpočet příkladu

Mějme tak například zprávu „ITS ALL “, kterou chceme zašifrovat. Nejdříve musíme vygenerovat klíče. Jako prvočísla tedy zvolíme $p = 47$ a $q = 59$. Podle uvedeného postupu spočteme veřejný modulus n:

$$n \quad = \quad 47 * 59 = 2773 \tag{4.9}$$

Eulerova funkce $\varphi(n)$ pak vychází:

$$\varphi(2773) \quad = \quad (47 - 1) * (59 - 1) = 2668 \tag{4.10}$$

Veřejný exponent volíme tak, aby byly splněny podmínky na něj kladené:

$$e = 17 \tag{4.11}$$

Z veřejného exponentu e spočteme soukromý exponent:

$$d = 17^{-1} \ (mod\, 2668) = 157 \tag{4.12}$$

Získané klíče jsou tedy:

$$\begin{aligned} k_{pub} &= (2773,\ 17) \\ k_{priv} &= (2773,\ 157) \end{aligned} \tag{4.13}$$

Nyní můžeme přejít k samotnému šifrování zprávy. Jak bylo zmíněno, šifrovanou zprávu je nutné vyjádřit celočíselně, a tedy převedeme text do celočíselné podoby tak, že *mezera* bude reprezentována jako 00, A bude 01, ..., Z bude 26. Následně text rozdělíme tak, aby získaná hodnota jednotlivých bloku byla menší než n.

$$\text{„}ITS\ ALL\text{“} = 0920\ 1900\ 0112\ 1200 \tag{4.14}$$

Zašifrování tedy bude vypadat následovně:

$$\begin{aligned} C_1 &= 0920^{17}\ mod\, 2773 = 0948 \\ C_2 &= 1900^{17}\ mod\, 2773 = 2342 \\ C_3 &= 0112^{17}\ mod\, 2773 = 1084 \\ C_4 &= 1200^{17}\ mod\, 2773 = 1444 \end{aligned} \tag{4.15}$$

Příjemce pak dešifruje přijatý šifrovaný text takto:

$$\begin{aligned} M_1 &= 0948^{157}\ mod\, 2773 = 0920 \\ M_2 &= 2342^{157}\ mod\, 2773 = 1900 \\ M_3 &= 1084^{157}\ mod\, 2773 = 0112 \\ M_4 &= 1444^{157}\ mod\, 2773 = 1200 \end{aligned} \tag{4.16}$$

Složením dešifrovaných bloků získáme původní text.

4.3 Podepisování pomocí RSA

Jak bylo zmíněno výše, šifra RSA neumožňuje jen zprávy šifrovat, ale i podepisovat. Vytvořený digitální podpis můžeme chápat jako náhradu vlastnoručního podpisu. Proces podepisování je opačný procesu šifrování. Zprávu tedy podepisujeme takto:

$$S = M^d \ (mod\, n)$$

a podpis následně ověřujeme:

$$M = S^e \ (mod\, n)$$

Podepsanou zprávu, vzhledem k tomu, že je podepsaná pomocí privátního klíče, mohou číst všichni, kteří vlastní veřejný klíč, tedy i útočník. To je ovšem v pořádku, protože cílem digitálního podpisu je zajištění integrity, autentizace a nepopiratelnosti. Jinak řečeno, digitálním podpisem si zajistíme, že zprávu není možné při přenosu pozměnit, autorem zprávy je vlastník soukromého klíče a ten vytvoření zprávy nemůže popřít, protože je jediným vlastníkem soukromého klíče.

4.4 Zajištění důvěrnosti, integrity, autentizace a nepopiratelnosti

Je možné si povšimnout, že pokud zprávu šifrujeme, zajistíme si pouze důvěrnost. Pokud ovšem jenom podepisujeme, pak si zajistíme integritu, autentizaci a nepopiratelnost, ale nezajistíme si důvěrnost. Co když ale potřebujeme u zprávy zajistit všechny vlastnosti? Jediným řešením tak je zprávu jak zašifrovat, tak podepsat. V jakém to uděláme pořadí, je ve skutečnosti nepodstatné, protože budou fungovat oba přístupy, obecně ale postupujeme tak, jak kdybychom psali dopis. Na papír napíše zprávu, kterou chceme poslat, to budou naše data. Aby bylo jasné, že jsme zprávu psali my, na papír se podepíšeme. Podepsanou zprávu následně vložíme do obálky a zalepíme, tedy podepsanou zprávu zašifrujeme.

Pro to, aby bylo možné zprávy posílat mezi dvěma stranami podepsané i zašifrované, je nutné, aby obě strany měly vygenerovány vlastní pár RSA klíčů a veřejné klíče si vyměnily. Následně strana, která chce zprávu odeslat, podepíše zprávu vlastním soukromým klíčem a zašifruje podepsanou zprávu veřejným klíčem druhé strany. Podepsanou a zašifrovanou zprávu pak odešle. Druhá strana, která zprávu přijala, nejdříve zprávu dešifruje pomocí vlastního soukromého klíče a následně ověří podpis veřejným klíčem druhé strany.

Kapitola 5

Test Prvočíselnosti

Test prvočíselností nám vyhodnocuje, zda zkoumané číslo je prvočíslem nebo číslem složeným. Testy prvočíselnosti se tak provádí před samotnou faktorizací, pokud by totiž zadané číslo bylo prvočíslo, nemá smysl jej faktorizovat.

5.1 Naivní testy prvočíselnosti

Asi úplně nejnaivnějším testem prvočíselnosti je zkusmé dělení (viz kap. 6.1). Zkoumané číslo zkoušíme postupně dělit všemi čísly v rozsahu $\langle 1; \lceil \sqrt{n} \rceil \rangle$, a pokud je nalezen nějaký dělitel, pak zkoumané číslo je číslem složeným. V opačném případě se jedná o prvočíslo. Metoda sice přesně určí, zda zkoumané číslo je či není prvočíslem, avšak časová složitost této metody je tak velká, že se v praxi pro testování prvočísel spíše nepoužívá.

Další naivní metodou je využití Eratosthenova síta (viz kap. 3.3). Výhodou této metody je stejně jako v případě zkusmého dělení, že jednoznačně určí, zda zkoumané číslo je prvočíslem. Eratosthenovo síto má také stejnou nevýhodu jako zkusmé dělení, což je příliš velká časová náročnost.

5.2 Fermatův test prvočíselnosti

Tato metoda je založena na tvrzení Malé Fermatovy věty (viz kap. 2.8). Jelikož ale nevíme, zda zkoumané číslo n je či není prvočíslem, je nutné platnost Malé Fermatovy věty ověřit. Vybereme si tedy nějaké $a \in \mathbb{Z}^+$, a pokud získáme $a^{n-1} \not\equiv 1 \ (mod\, n)$, pak víme, že číslo n je číslem složeným. Potvrdit ale, že číslo n je prvočíslem, je náročnější [6].

Pokud bychom chtěli potvrdit, že číslo n je zcela jistě prvočíslem, pak bychom museli ověřit platnost Malé Fermatovy věty pro všechna a v intervalu $\langle 1; n-1 \rangle$. Takový výpočet by byl ovšem velmi časově náročný, a proto se většinou spokojíme, když ověříme pouze nějaké kvantum čísel z daného rozsahu. Pokud pro všechna čísla z kvanta bude platit Malá Fermatova věta, pak zkoumané číslo n prohlásíme za prvočíslo, i když to je jen s nějakou pravděpodobností. Samozřejmě čím více čísel otestujeme, tím větší je pravděpodobnost, že naše tvrzení je pravdivé. Čísla pro ověření vybíráme ze zmíněného rozsahu náhodně.

Někdy může nastat situace, kdy i pro složené číslo bude Malá Fermatova věta platit. Číslo a, které způsobí takovou situaci, nazýváme Fermatovým lhářem. Příkladem může být ověření, zda číslo $n = 15$ je prvočíslem a pro ověření zvolíme $a = 4$. V takovém případě nám totiž vyjde $4^{14} \equiv 1 \ (mod\, 15)$. Proto obecně vždy postupujeme tak, že vygenerujeme

náhodně číslo a z daného intervalu, a pokud Malá Fermatova věta platí, pak generujeme další a. Bude-li pro nějaké a porušena platnost Malé Fermatovy věty, s výpočtem skončíme a číslo n prohlásíme za složené. V opačném případě vygenerujeme zvolený počet a, a jelikož ani jednou nenastala situace, kdy by Malá Fermatova věta neplatila, prohlásíme číslo n za prvočíslo.

Pseudokód pro Fermatův test prvočíselnosti tedy vypadá následovně:

Algoritmus 4: Fermatův test prvočíselnosti

Input: n
Output: n není/pravděpodobně je prvočíslo

1: **for** $1 \leq k \leq$ *Maximální počet opakování* **do**
2: Náhodně zvol a z rozsahu $\langle 1; n-1 \rangle$;
3: **if** $a^{n-1} \not\equiv 1 \ (mod\ n)$ **then**
4: **return** n ***není*** *prvočíslo*
5: **end if**
6: **end for**
7: **return** n ***pravděpodobně je*** *prvočíslo*

5.3 Millerův-Rabinův test prvočíselnosti

Metoda vychází, obdobně jako Fermatův test prvočíselnosti (viz kap. 5.2), z Malé Fermatovy věty (kap. 2.8), ale přistupuje k ní jinak [19]. Malou Fermatovu větu můžeme zapsat jako:

$$a^{p-1} - 1 \equiv 0 \ (mod\, p) \tag{5.1}$$

Víme, že prvočíslo p je liché, a proto $p-1$ je sudé číslo, které můžeme rozepsat na $2^s * d$, kde s a d jsou kladná celá čísla, přičemž d je liché. Tím dostaneme:

$$a^{2^s * d} - 1 \equiv 0 \ (mod\, p) \tag{5.2}$$

To lze dále rozepsat na:

$$\begin{aligned} & a^{2^s * d} - 1 = \\ & = \left(a^{2^{s-1} * d} - 1\right) * \left(a^{2^{s-1} * d} + 1\right) = \\ & = \left(a^d - 1\right) * \left(a^d + 1\right) * \left(a^{2d} + 1\right) * \cdots * \left(a^{2^{s-1} * d} + 1\right) \end{aligned} \tag{5.3}$$

Takže:

$$\begin{aligned} & a^{2^s * d} - 1 = \\ & = \left(a^{2^{s-1} * d} - 1\right) * \left(a^{2^{s-1} * d} + 1\right) = \\ & = \left(a^d - 1\right) * \left(a^d + 1\right) * \left(a^{2d} + 1\right) * \cdots * \left(a^{2^{s-1} * d} + 1\right) \equiv 0 \ (mod\, p)\,, \end{aligned} \tag{5.4}$$

což znamená, že prvočíslo p dělí jeden z činitelů, a tedy pro každé a bude platit:

$$a^d \equiv 1 \ (mod\, p) \tag{5.5}$$

nebo

$$a^{2^r * d} \equiv -1 \ (mod\, p) \ pro\, nějaké\, 0 \leq r \leq s-1 \tag{5.6}$$

Millerův-Rabinův test prvočíselnosti se tak snaží najít a takové, že:

$$a^d \not\equiv 1 \ (mod\, n) \tag{5.7}$$

a

$$a^{2^r * d} \not\equiv -1 \ (mod\, n) \ pro\, každé\, 0 \leq r \leq s-1, \tag{5.8}$$

kde n je zkoumané číslo. Pokud se nám podaří takové a najít, pak zkoumané číslo je složené. Pseudokód pro Millerův-Rabinův test prvočíselnosti by vypadal následovně (poznamenejme, že $n-1 \equiv -1 \ (mod\, n)$):

Algoritmus 5: Millerův-Rabinův test prvočíselnosti

Input: n
Output: n není/pravděpodobně je prvočíslo

```
1:  for 1 ≤ k ≤ Maximální počet opakování do
2:      Náhodně zvol a z rozsahu ⟨2; n − 2⟩ ;
3:      x = a^d (mod n) ;
4:      if x = 1 nebo x = n − 1 then
5:          continue ;
6:      end if
7:      for 1 ≤ r ≤ s − 1 do
8:          x = x^2 (mod n) ;
9:          if x = 1 then
10:             return n není prvočíslo
11:         end if
12:         if x = n − 1 then
13:             break ;
14:         end if
15:     end for
16:     return n není prvočíslo
17: end for
18: return n pravděpodobně je prvočíslo
```

5.4 Solovayův-Strassenův test prvočíselnosti

Solovayův-Strassenův test prvočíselnosti, blíže popsán v [22], je principiálně velmi podobný Fermatovu testu prvočíselnosti, který je popsán v kapitole 5.2. Tato metoda nepoužívá k ověření čísla Malou Fermatovu větu (viz kap. 2.8), ale využívá Eulerova kritéria (viz kap. 2.6). Obdobně jako u Fermatovy metody, při testu prvočíselnosti nevíme, jestli zkoumané číslo je prvočíslem anebo se jedná o číslo složené, a tak nemůžeme použít Legendreův symbol, který je určen pro práci s prvočísly. Proto místo Legendreova symbolu používáme Jacobiho symbol (viz kap. 2.7.1). Solovayova-Strassenova metoda tedy pro ověření, zda je zkoumané číslo n prvočíslem, používá vztah:

$$a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \ (mod\, n) \tag{5.9}$$

Postup při ověřování, zda je číslo n prvočíslem, je shodný s Fermatovou metodou, pseudokód tedy vypadá následovně:

Algoritmus 6: Solovayův-Strassenův test prvočíselnosti

Input: n
Output: n není/pravděpodobně je prvočíslo

1: **for** $1 \leq k \leq$ *Maximální počet opakování* **do**
2: Náhodně zvol a z rozsahu $\langle 2; n-1 \rangle$;
3: **if** $\left(\frac{a}{n}\right) = 0$ *nebo* $a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right)$ *(mod n)* **then**
4: **return** n ***není*** *prvočíslo*
5: **end if**
6: **end for**
7: **return** n ***pravděpodobně je*** *prvočíslo*

Kapitola 6

Faktorizační algoritmy

Kapitola popisuje známe faktorizační metody. Jsou zde popsány i ty úplně nejjednodušší metody. Je tak například z důvodu, aby čtenář, který se v daném tématu neorientuje, pochopil jednotlivé metody od základu, a také proto, že složitosti sofistikovanějších metod se často porovnávají právě s jednoduchými metodami, aby vynikl smysl jejich použití.

Faktorizační metody se nalézají ve dvou třídách složitosti. Jedná se o exponenciální složitost a subexponenciální složitost. Exponenciální složitost mají jednodušší metody jako například *Metoda zkusmého dělení* (viz 6.1) nebo *Pollardova ρ metoda* (viz 6.2). I když tyto metody mají exponenciální složitost, stále najdou své uplatnění, hlavně při hledání menších čísel, řekněme do 30 dekadických číslic. Mezi metody se subexponenciální složitostí řadíme například *Metodu kvadratického síta* (viz 6.6) či *Obecné číselně teoretické číslo* (viz 6.9). Metody se subexponenciální složitostí se dnes používají nejčastěji pro faktorizaci velkých čísel, kdy například *Multi-polynomiální kvadratické síto* (viz 6.7), jež je rozšířením metody kvadratického síta, používáme k faktorizaci čísel mající do 100 dekadických číslic a Obecné číselně teoretické síto používáme pro faktorizaci čísel, která mají více jak 100 dekadických číslic.

Metody se subexponenciální složitostí sice vynikají jejich rychlostí v porovnání s metodami s exponenciální složitostí, ovšem jejich náročnost na pochopení je mnohem vyšší.

6.1 Zkusmé dělení

Jedná se o nejjednodušší metodu, kterou je možné použít pro faktorizaci celých čísel. Metoda pouze inkrementuje dělitele, a pokud dělitel dělí zadané číslo N beze zbytku, pak jsme nalezli faktor zadaného čísla.

Metodu je možné vylepšit omezením inkrementace dělitele pouze do hodnoty $\left\lceil \sqrt{N} \right\rceil$. Pokud ani při použití poslední možné hodnoty nebyl nalezen faktor, znamená to, že faktory jsou pouze nevlastní dělitelé (viz kap. 2.1). Metodu je ještě možné dále vylepšit. Místo inkrementace dělitele až po hodnotu $\left\lceil \sqrt{N} \right\rceil$ je výhodnější si vytvořit tabulku všech prvočísel menších než $\left\lceil \sqrt{N} \right\rceil$ a pak zkoušet zadané číslo N dělit pouze těmito prvočísly, protože jeden z faktorů musí zcela jistě být prvočíslem.

Pseudokód pro základní variantu zkusmého dělení vypadá následovně:

Algoritmus 7: Zkusmé dělení

Input: n
Output: V případě úspěchu x, jinak n

```
1: for 0 ≤ x ≤ ⌈√n⌉ do
2:     if x|n then
3:         return x
4:     end if
5: end for
6: return n
```

6.2 Pollardova ρ metoda

Tuto metodu představil pan Pollard v [16]. Metoda je velice efektivní, pokud faktorizujeme číslo s malými faktory, a je také velice jednoduchá na implementaci, jak bude ukázáno v pseudokódu.

Mějme náhodnou funkci $f : \mathbb{Z}_n \to \mathbb{Z}_n$ a náhodné číslo $x \in \mathbb{Z}_n$. Následně pozorujme sekvenci:

$$x,\ f(x),\ f^2(x),\ \ldots \tag{6.1}$$

Jelikož je množina $\mathbb{Z}_n$ konečná, musí zcela jistě nastat situace, kdy $f^j(x) = f^k(x)$, a tedy se obrazy začnou periodicky opakovat. Chování sekvence můžeme vyobrazit symbolem ρ, kdy ocas značí preperiodu a okruh pak danou periodu. Délka ocasu a okruhu bude dle narozeninového paradoxu (viz kap. 2.4) přibližně $O(\sqrt{n})$.

Pollardova ρ metoda využívá tohoto principu k faktorizaci zadaného čísla n. Při hledání dělitele p čísla n pracujeme v prostoru $\mathbb{Z}_n$ a snažíme se najít iterace $f^j(x)$ a $f^k(x)$ takové, pro které bude platit $f^j(x) \equiv f^k(x)\ (mod\, p)$. Pokud se nám podaří takový vztah najít, pak získáme p spočtením $GCD(f^j(x) - f^k(x),\ n)$.

Hledat ale $f^j(x)$ a $f^k(x)$ takové, aby platilo $f^j(x) \equiv f^k(x)\ (mod\, p)$, by bylo velice náročné a složitost takového výpočtu by překročila složitost zkusmého dělení. Proto se obvykle omezíme na hledání $f^i(x) \equiv f^{2i}(x)\ (mod\, p)$, pro $i \geq 1$. Poznamenejme, že jako náhodná funkce se obvykle volí $f(x) = x^2 + 1$ a výchozí hodnota čísla x obvykle bývá 2 [7].

Algoritmus pro faktorizaci pomocí Pollardovy ρ metody by mohl vypadat následovně:

Algoritmus 8: Pollardova ρ metoda

Input: n
Output: r

1: Náhodně vygeneruj $a \in \langle 0; n-1 \rangle$;
2: Definuj funkci $f(x) = x^2 + 1$;
3: $x_i = a$;
4: $x_{2i} = a$;
5: $r = 1$;
6: **while** $r = 1$ **do**
7: $\quad x_i = f(x_i)$;
8: $\quad x_{2i} = f(f(x_{2i}))$;
9: $\quad r = GCD(x_i - x_{2i}, n)$;
10: **end while**
11: **if** $r = n$ **then**
12: $\quad$ **goto** krok1 ;
13: **else**
14: $\quad$ **return** r ;
15: **end if**

6.3 Pollardova $p-1$ metoda

Metoda vychází z Malé Fermatovy věty (viz kap. 2.8), $a^{p-1} \equiv 1 (mod p)$. Pokud p je dělitelem zadaného čísla n, pak toto p lze získat jako výsledek $GCD\left(a^{p-1} - 1, n\right)$. Zkoušet ale, zda platí $GCD\left(a^{p-1} - 1, n\right) > 1$ pro každé p zvlášť, by bylo velice neefektivní. Je zde tedy nějaká možnost, jak vyzkoušet více prvočísel najednou? Pan Pollard našel řešení [15]. Malá Fermatova věta bude pravdivá i pro jakékoli celé číslo M, pro které platí $(p-1) \,|M$. Jelikož Malá Fermatova věta bude platit pro jakýkoli násobek $p-1$, je tedy například možné vytvořit M takové, aby bylo složeno jako násobek čísel $(p_1 - 1) * (p_2 - 1)$ pro nějaká dvě prvočísla p_1 a p_2. Následně pomocí $GCD\left(a^M - 1, n\right)$ je možné ověřit, zda p_1 nebo p_2 je dělitelem zkoumaného čísla n. Tímto principem můžeme vytvořit M takové, aby pokrývalo co možná nejvíce prvočísel.

Pokud chceme zajistit, aby M pokrývalo co možná nejvíce prvočísel, pak určíme hranici B a hledáme nejmenší společný násobek všech prvočísel $p_i \geq 3$ i jejich mocnin, takových, že $p_i^{\alpha_i} \leq B$, $kde\, \alpha_i \geq 1$. Číslo M tak zcela jistě bude B-hladké (viz kap. 2.2) a zároveň pro každé prvočíslo p_i bude platit $(p_i - 1) \,|M$.

Najít zmíněný společný násobek není nijak těžké. Mějme na počátku $M = 1$ a $k = 1$, tedy $M\,(k) = 1$. Pokud $k+1$ je prvočíslem p či jeho mocninou, pak $M\,(k+1) = p * M\,(k)$. V opačném případě $M\,(k+1) = M\,(k)$. Číslo k inkrementujeme tak dlouho, dokud $k \leq B$.

Algoritmus pro faktorizaci čísla n pomocí Pollardovy $p-1$ metody by tedy mohl vypadat následovně:

Algoritmus 9: Pollardova $p-1$ metoda

Input: n
Output: g

1: Najdi všechna prvočísla $p_i \leq B$, $i = 1 \ldots m$ a jejich mocniny $p_i^{\alpha_i} \leq B$ a $\alpha_i \geq 1$;
2: $c = a$; `// Obvykle se volí 2 nebo 3`
3: **for** $1 \leq i \leq m$ **do**
4: **for** $1 \leq j \leq \alpha_i$ **do**
5: $c = c^{p_i} \pmod{n}$;
6: **end for**
7: **end for**
8: $g = GCD(c-1, n)$ **return** g

Tato metoda může selhat ve 2 případech. Jedním z nich je situace, kdy nám vyjde $GCD\left(a^M - 1, n\right) = 1$. To znamená, že číslo n není dělitelné ani jedním z prvočísel, které jsme zkoušeli. Řešením může být zvětšení hranice B a vyzkoušení metody znova. Obvykle se ale tato metoda používá pouze k rychlému vyzkoušení, zda náhodou zadané číslo n nemá prvočíselného dělitele z množiny prvočísel, které jsou menší než zadaná hranice B. Pokud tato metoda selže, přechází se na nějakou jinou metodu. Pollardovu $p-1$ metodu je možné použít pro rozložení jakéhokoli čísla n ($B = \sqrt{n}$), nicméně asymptotická složitost by byla horší než u zkusmého dělení, a proto se tato metoda nepoužívá jako plnohodnotná faktorizační metoda.

Dalším případem selhání metody je situace, kdy $GCD\left(a^M - 1, n\right) = n$. Taková situace se řeší znovuspuštěním metody, ale v každém kroku zkoušíme, zda již platí $GCD\left(a^M - 1, n\right) > 1$. Situace, kdy $GCD\left(a^M - 1, n\right) = n$, nastává velice zřídka.

6.4 Metoda eliptických křivek

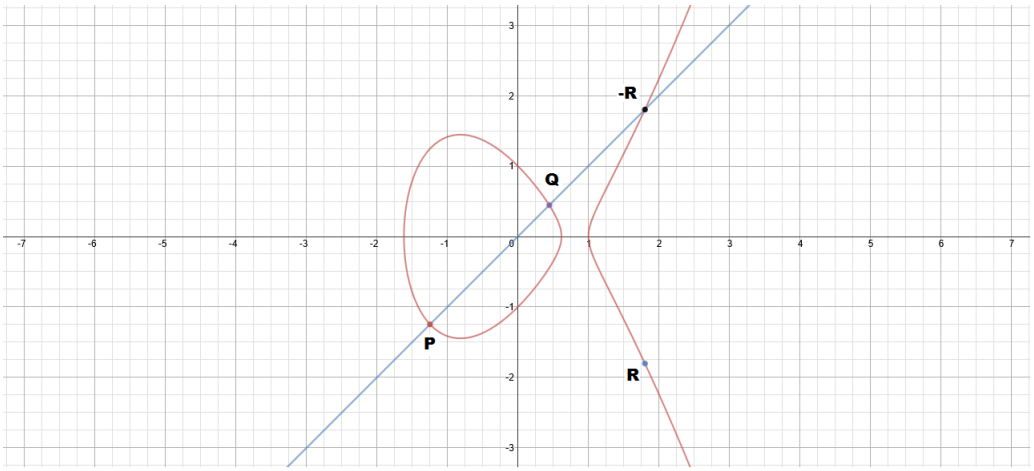
Jak název napovídá, metoda je založena na eliptických křivkách. S křivkami pracujeme jako s polynomy 3. řádu ve tvaru:

$$y^2 = x^3 + ax + b, \quad a, b \in R \tag{6.2}$$

Tyto polynomy jsou dále nereducibilní. Zápis křivky je někdy upraven do tvaru:

$$y^2 = x^3 + Cx^2 + Ax + B, \quad A, B, C \in R \tag{6.3}$$

Eliptická křivka by tedy mohla vypadat například takto:



Obrázek 6.1: Eliptická křivka: $y^2 = x^3 - 2x + 1$

Z obrázku je možné si všimnout, že nad eliptickými křivkami lze provádět operace sčítaní a odečítání. Každé sečtení či odečtení dvou bodů vede k získání jiného bodu eliptické křivky. Na obrázku je jako příklad uvedeno sečtení dvou bodů P a Q. Tyto body se proloží přímkou a v bodě, ve kterém přímka protne eliptickou křivku, se nachází takzvaný *opačný bod*, $-R$, výsledného bodu R, který je symetrický podle osy x ke svému opačnému bodu. V případě, že bychom se pokusili spočíst součet bodu R a jeho opačného bodu $-R$, zjistíme, že proložená přímka žádný další bod eliptické křivky neprotne, resp. protne, ale v nekonečnu. Pro takové případy byl definován tzv. „nulový bod“, O, který slouží jako neutrální prvek. Když máme definovány všechny body nad eliptickými křivkami, zbývá jen definovat samotné operace sčítání a odečítání:

$$\begin{aligned} -O &= O \\ -P_1 &= [x_1; -y_1] \\ O + P_1 &= P_1 \\ Pokud\ P_2 &= -P_1,\ pak\ P_1 + P_2 = O \\ Pokud\ P_2 &\neq -P_1,\ pak\ P_1 + P_2 = P_3 = [x_3; y_3]\,, \end{aligned} \tag{6.4}$$

kde $[x_3; y_3]$ spočteme za pomoci vztahu:

$$\begin{aligned} x_3 &= s^2 - x_1 - x_2 \\ y_3 &= s\,(x_1 - x_3)\,, \end{aligned} \tag{6.5}$$

kde s je směrnicí přímky, a pokud $P_1 = P_2$, tak tuto směrnici vypočteme takto:

$$s = \frac{3x_1^2 + a}{2y_1} \tag{6.6}$$

jinak:

$$s = \frac{y_2 - y_1}{x_2 - x_1}. \tag{6.7}$$

Doposud jsme pracovali v prostoru reálných čísel R, avšak při faktorizaci musíme pracovat v prostoru celých čísel. S eliptickými křivkami budeme tedy pracovat v Galoisově tělese $GF(p)$, tedy v tělese (viz kap. 2.11) s množinou A, jež má konečný počet prvků, tj. $A = \{0, \ldots, p-1\}$ a p je prvočíslem. V tomto prostoru pracujeme se všemi operacemi nad eliptickými křivkami stejně jako v prostoru reálných čísel, jen je každý výsledek operace modulován číslem p. Eliptickou křivku můžeme tedy chápat jako množinu souřadnic a zápis bude vypadat takto:

$$E(GF(p)) = \left\{[x, y] \in GF(p)^2 : y^2 = x^3 + ax + b \ (mod\, p)\right\} \cup \{O\} \tag{6.8}$$

či:

$$E(GF(p)) = \left\{[x, y] \in GF(p)^2 : y^2 = x^3 + Cx^2 + Ax + B \ (mod\, p)\right\} \cup \{O\} \tag{6.9}$$

pro upravený tvar eliptické křivky. Ještě než je eliptická křivka převedena do prostoru $GF(p)$, je nutné ověřit, že tato eliptická křivka splňuje podmínku $4a^3 + 27b^2 \not\equiv 0 \ (mod\, p)$, jinak by množina $E(GF(p))$ netvořila grupu. Nad grupou $E(GF(p))$ zavádíme pojem *řád křivky*, $\#E(GF(p))$, který vyjadřuje počet bodů na křivce plus 1 pro nulový bod. Dále používáme značení nP pro vyjádření mnohonásobného sečtení bodu:

$$nP = P + P + \ldots + P, \tag{6.10}$$

kde n vyjadřuje počet kopií P. Jelikož prostor $GF(p)$ je konečný, pak bude platit $(\#E(GF(p)))\, P = O$, což znamená, že pokud sečteme bod P přesně tolikrát, jako je počet řádu křivky, získáme nulový bod. Pokud bychom tedy provedli ještě jedno přičtení bodu P, získali bychom opět bod P. Délka periody pro bod P však ve skutečnosti může být kratší, a proto se ještě zavádí pojem *řád bodu*, r, pro nějž tedy platí $rP = O$. Vysvětlením všech důležitých pojmů týkajících se eliptických křivek je nyní možné přejít k popsání faktorizace nad eliptickými křivkami.

Faktorizace se provádí nad $\mathbb{Z}_n$, kde n je složené číslo, a tedy aritmetické operace nad eliptickými křivkami, které budeme chtít použít, budeme používat nad $E_{a,b}(\mathbb{Z}_n)$. Toto však není pravá eliptická křivka, protože jak bylo zmíněno, n je složené číslo. $E_{a,b}(\mathbb{Z}_n)$ nazýváme eliptickou pseudokřivkou mající tvar:

$$E_{a,b}(\mathbb{Z}_n) = \left\{[x, y] \in \mathbb{Z}_n^2 : y^2 = x^3 + ax + b\right\} \cup \{O\}, \tag{6.11}$$

pokud $GCD(n, 6) = 1$ a koeficienty a, b splňují podmínku $GCD\left(4a^3 + 27b^2, n\right) = 1$. Jak je vidět ze zápisu tvaru eliptické pseudokřivky, připouští se opět jeden nulový bod. Ve skutečnosti ale tím, že n je složené číslo, budou existovat další páry bodů, pro něž operace sčítání nebude definovaná. Tuto skutečnost bychom měli zjistit při pokusu o vypočtení směrnice přímky, jež je součástí provedení operace sečtení. Právě této vlastnosti eliptických

pseudokřivek je využito u faktorizace k nalezení faktoru složeného čísla n. Nejdříve bude předveden pseudokód Lenstrovy faktorizační metody a na tomto pseudokódu bude následně vysvětleno, jak se postupuje při hledání faktoru.

Algoritmus 10: Lenstrova metoda faktorizace za pomocí eliptických křivek

Input: n
Output: g

1: $B_1 = 50000$;
2: Náhodně zvol $x, y, a \in [0, n-1]$;
3: $b = (y^2 - x^3 - ax) \, mod \, n$;
4: $g = GCD(4a^3 + 27b^2, n)$;
5: **if** $g = n$ **then**
6: **goto** krok 2 ;
7: **end if**
8: **if** $g > 1$ **then**
9: **return** g ; `// Byl nalezen faktor čísla` n
10: **end if**
11: $E = E_{a,b}(\mathbb{Z}_n)$;
12: $P = (x, y)$;
`// Cyklus přes všechna prvočísla` $p_i \leq B_1$
13: **for** $1 \leq i \leq \pi(B_1)$ **do**
14: Najdi největší celé číslo α_i takové, že $p_i^{\alpha_i} \leq B_1$;
15: **for** $1 \leq j \leq \alpha_i$ **do**
16: $P = p_i P$, kdy zastavíme výpočet, pokud nějaké d^{-1}, jež je inverzí ke jmenovateli směrnice d, signalizuje netriviální rozklad, spočteme $g = GCD(n, d)$ a v takovém případě **return** g
17: **end for**
18: **end for**
19: Inkrementuj B_1 ; `// Faktor nebyl nalezen`
20: **goto** krok 2 ;

Nejdříve si zvolíme velikost faktorizační báze B_1. Tu využijeme v algoritmu později, ale aby algoritmus správně fungoval, je nutné faktorizační bázi inicializovat hned v prvním kroku. Následně náhodně vygenerujeme souřadnice bodu x a y a koeficient eliptické pseudokřivky a. Na základě souřadnic bodu a koeficientu a vygenerujeme druhý koeficient b. Nyní je potřeba ověřit, zda jsme neporušili podmínku pro eliptickou křivku. Pokud $g = n$, podmínka byla porušena a jsme nuceni generovat souřadnice bodu a koeficient a znovu. Mohli jsme mít ale takové štěstí, že při ověřování podmínky pro eliptickou křivku jsme nalezli faktor zkoumaného čísla n a algoritmus můžeme ukončit. Získanou eliptickou pseudokřivku a bod si uložíme. Nyní budeme procházet faktorizační bázi. Pro každé prvočíslo z faktorizační báze spočteme jeho mocninu takovou, aby platilo, že tato mocnina je menší než nastavená hranice faktorizační báze. Tento exponent nám bude určovat počet iterací cyklu, které musíme provést. V tomto cyklu počítáme $p_i P$ a doufáme, že nám nějaké d^{-1} zasignalizuje, že operace sčítání není pro dané body definována. Pokud se tak stane, pak stačí spočíst největšího společného dělitele pro d a n a získáme hledaný faktor. Projdeme-li všechna prvočísla i s jejich mocninami a faktor se nám nepodaří najít, jsme nuceni inkrementovat hranici faktorizační báze B_1 a hledat faktor znovu.

6.5 Fermatova faktorizace

Tato metoda položila základy většině moderním faktorizačním metodám, které se dnes používají. Pan Fermat si všiml, že každé liché číslo lze rozložit na rozdíl dvou čtverců $n = a^2 - b^2$. Uvedený vztah může mít více než jedno řešení. Zápis $n = a^2 - b^2$ lze samozřejmě rozložit na $n = (a+b)(a-b)$. Pokud tedy $a+b > 1$ nebo $a-b > 1$, pak se jedná o netriviální rozklad.

Při hledání netriviálního rozkladu postupujeme tak, že vybereme hodnotu a a zkoumáme jak bude vypadat $a^2 - n$. Výsledkem bude nějaké číslo x. Pokud toto x je čtvercem, řekněme b^2, pak jsme našli hledaný vztah $n = a^2 - b^2$. Zbývá už jen ověřit, zda se jedná o netriviální rozklad, tj. $a + b > 1$ nebo $a - b > 1$. Pokud podmínka neplatí, jedná se o triviální rozklad, který si nepřejeme. V takovém případě se musí začít hledat znova.

Při algoritmizaci Fermatovy metody se většinou nehledá a zcela náhodně, ale položí se $a = \lceil \sqrt{n} \rceil$ a pokud $a^2 - n \neq b^2$, pak $a = a + 1$ a zkoušíme znova, zda jsme nalezli b^2. To má hned několik výhod. Jelikož $a \geq \lceil \sqrt{n} \rceil$, pak nikdy nepracujeme se zápornými čísly. Další výhodou je, že pokud n je součinem dvou blízkých čísel, pak je tato metoda objeví velmi rychle.

Algoritmus pro Fermatovu faktorizaci by mohl vypadat následovně:

Algoritmus 11: Fermatova faktorizace

Input: n
Output: faktor zadaného čísla n

1: **for** $\lceil \sqrt{n} \rceil \leq a \leq n$ **do**
2: **if** $b = \sqrt{a^2 - n}$ ***and*** b *je celým číslem* **then**
3: **return** $(a + b)$; `// Můžeme vracet i(a- b)`
4: **end if**
5: **end for**
6: **return** n ; `// n je prvočíslo`

Pro lepší pochopení předvedu Fermatovu metodu na praktickém příkladu. Pan Pomerance, jež je například autorem faktorizační metody *kvadratické síto* (viz kap. 6.6), se na střední škole zúčastnil matematické soutěže, kdy jednou z úloh bylo najít dělitele čísla 8051. V té době znal *metodu zkusmého dělení* (kap. 6.1), která mu ovšem přišla pro tak velké číslo příliš zdlouhavá a věřil, že zadavatel úlohy po nich chce, aby použili nějakou obratnější metodu než prosté zkoušení dělitelů pomocí zkusmého dělení. Na tuto úlohu měl Pomerance 5 minut. Několik minut z tohoto času investoval do hledání nějaké obratné metody, což se mu nepovedlo, a tak se rozhodl, že ve zbývajících minutách zkusí úlohu vyřešit pomocí zkusmého dělení. Na řešení bohužel nepřišel, protože mu vypršel čas vyhrazený pro řešení této úlohy. Pokud by ovšem v té době znal Fermatovu metodu faktorizace, na získání řešení by zdaleka nepotřeboval 5 minut a postupoval by takto:

Zadané číslo je:

$$n = 8051 \tag{6.12}$$

Fermatova metoda rozkládá liché číslo na rozdíl dvou čtverců:

$$n = a^2 - b^2 \tag{6.13}$$

Což můžeme přepsat na:

$$b^2 = a^2 - n \tag{6.14}$$

Prvním krokem je získání hodnoty neznáme a, kterou stanovíme na $\lceil \sqrt{n} \rceil$ a tedy:

$$a = \lceil \sqrt{n} \rceil = 90 \tag{6.15}$$

Nyní jsme schopni spočítat hodnotu neznámé b:

$$\begin{aligned} b^2 &= 8100 - 8051 = 49 \\ b &= 7 \end{aligned} \tag{6.16}$$

Jelikož známe hodnoty proměnných a i b, spočteme faktory zadaného čísla n takto:

$$n = 90^2 - 7^2 = (90 - 7) * (90 + 7) = 83 * 97 \tag{6.17}$$

Je důležité si uvědomit, že číslo 8051 bylo vybráno úmyslně, abychom našli oba čtverce a^2 i b^2 hned v první iteraci. Pro jiné číslo bychom při odmocnění hodnoty získané z $a^2 - n$ mohli dostat číslo, které již není celým číslem, a tedy nevede k řešení. V takovém případě bychom zvýšili hodnotu proměnné a o 1 a zkoušeli získat celé číslo b znovu. To bychom prováděli tak dlouho, dokud by se nám to nepovedlo.

Taková situace nastane při pokusu o faktorizaci například čísla $n = 1649$. Zde by výpočet vypadal takto:

$$x = 41^2 - 1649 = 32 \tag{6.18}$$

Jelikož $x = 32$ není čtvercem, jsme nuceni inkrementovat hodnotu proměnné a a provést výpočet znovu:

$$\begin{aligned} x &= 42^2 - 1649 = 115 \\ x &= 43^2 - 1649 = 200 \\ &\vdots \\ x &= 57^2 - 1649 = 1600 = 40^2 \end{aligned} \tag{6.19}$$

Konečně jsme nalezli hodnotu, která je čtvercem a tedy:

$$n = 57^2 - 40^2 = (57 - 40) * (57 + 40) = 17 * 97 \tag{6.20}$$

Na uvedených příkladech lze vidět, že Fermatova metoda je velice efektivní, pokud je číslo n složeno z velmi blízkých čísel, tedy čísel, které jsou blízko $\sqrt{n}$. To bylo ukázáno na prvním příkladu. Naopak čím více se faktory od sebe vzdalují, tím více se vzdalují od $\sqrt{n}$ a získat tyto faktory pomocí Fermatovy metody je náročnější a zdlouhavější. To bylo ukázáno v druhém příkladu. Problém této metody je, že iteruje vždy od $\sqrt{n}$ směrem nahoru a není tak schopna nalézt malé faktory. Mnoho složených čísel má jeden z faktorů právě poměrně malý a v takových případech se může stát, že rozklad pomocí Fermatovy faktorizace bude trvat déle než rozklad pomocí zkusmého dělení.

Jenom poznamenám, že druhý uvedený příklad jsme schopni vyřešit mnohem rychleji než pomocí Fermatovy metody, přitom je postup založen na stejném principu. Více o tomto rychlejším postupu je napsáno v kapitole 6.6, která pojednává o faktorizaci pomocí kvadratického síta.

6.6 Kvadratické síto (QS)

Kvadratické síto je jedna z nejpoužívanějších metod pro faktorizaci velkých čísel, kterou popsal pan Pomerance v [17, 18]. Zároveň se jedná o nejrychlejší metodu, pokud chceme faktorizovat čísla, která mají do 100 dekadických číslic. Pokud chceme faktorizovat čísla větší, volíme většinou Obecné číselně teoretické síto (viz kap. 6.9).

Metoda má své kořeny ve Fermatově faktorizaci (viz kap. 6.5). Jak jsme si ale u Fermatovy metody ukázali, nalezení dvou čtverců, které vedou k rozkladu zadaného čísla, může být zdlouhavé, pokud se faktory nenalézají blízko sebe. Zároveň jsem poznamenal, že druhý příklad, který jsme u Fermatovy metody řešili „mnoha“ kroky, jsme schopni vyřešit během pár kroků a přitom stále budeme pracovat na podobném principu jako Fermatova metoda. S vylepšeným postupem přišel pan Kraitchik [18], 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 $u^2 \equiv v^2 \ (mod\, n)$. Bystrý čtenář si může uvědomit, že tento postup může nalézt i triviální dělitele, kteří jsou pro nás nezajímaví. Rozklad bude netriviální, pokud bude platit $u \equiv \pm v \ (mod\, n)$. Jednotlivé faktory pak získáme jako největší společný dělitel $GCD\,(u - v, n)$, případně $GCD\,(u + v, n)$.

Jak tedy ale hledat takové u a v, aby platilo $u^2 \equiv v^2 \ (mod\, n)$ a zároveň zajistit, aby metoda byla výhodnější než metoda Fermatova? Vezměme si opět číslo pro rozklad $n = 1649$ a udělejme pár kroků Fermatovy metody s tím, že budeme výsledek modulovat zadaným číslem n:

$$\begin{aligned} u_1^2 &= 41^2 = 1681 \equiv 32 \ (mod\, 1649) \\ u_2^2 &= 42^2 = 1764 \equiv 115 \ (mod\, 1649) \\ u_3^2 &= 43^2 = 1849 \equiv 200 \ (mod\, 1649) \end{aligned} \tag{6.21}$$

S Fermatovou metodou, jak jsme si ukázali, bychom iterovali až k hodnotě $u^2 = 57^2$. Avšak s Kraitchikovou myšlenkou můžeme přestat iterovat po třech krocích. Je tak proto,

že pokud si vezmeme u_1 a u_3 a jejich druhé mocniny mezi sebou vynásobíme, dostáváme $u_1^2 * u_3^2 = (u_1 * u_3)^2$, to si označíme jako u^2. Následně pokud vynásobíme mezi sebou hodnoty druhých mocnin u_1 a u_3 v rámci modulo 1649, dostaneme $32 * 200 = 6400 = 80^2$. Označme číslo 80 jako v a tedy $v^2 = 80^2$. Pokud vše spojíme, dostaneme:

$$(41 * 43)^2 \equiv 80^2 \ (mod\, 1649) \tag{6.22}$$

a tedy platí:

$$u^2 \equiv v^2 \ (mod\, n) \tag{6.23}$$

Pro úspěšné dokončení výpočtu nám zbývá ověřit, že jsme dostali netriviální rozklad, a nalézt největšího společného dělitele. Pokud porovnáme u a v, pak zjistíme, že $114 \not\equiv 80 \ (mod\, n)$, a tedy největší společný dělitel povede k získání netriviálního dělitele. Opravdu, $GCD\,(114 - 80, 1649) = 17$. Dále pak není těžké zjistit, že číslo $1649 = 17 * 97$.

Kraitchikovou myšlenkou je tedy hledat taková čísla u_i, že platí:

$$u^2 = u_1^2 * \ldots u_k^2 \equiv \left(u_1^2 \ mod\, n\right) * \ldots * \left(u_k^2 \ mod\, n\right) = v^2 \tag{6.24}$$

V uvedeném příkladu jsme dosáhli řešení, protože jsme věděli, že máme použít u_1 a u_3. Pokud ale budeme počítat jakékoli jiné číslo, nebudeme vědět, kolik iterací musíme provést, abychom mohli říci, že jich máme dost k nalezení kombinace, která povede k vytvoření čtverce. I v případě, že bychom takovou informaci měli, zkoušet všechny možné kombinace, než bychom nalezli tu správnou, by mohlo být časově velmi náročné. Naštěstí páni John Brillhart a Michael Morrison přišli na způsob, jak vybírat taková čísla, abychom jejich kombinací získali potřebný čtverec [12].

6.6.1 Prosívání

Každé číslo m jsme schopni rozložit na prvočísla, tedy $m = \prod p_i^{e_i}$, kde p_i je prvočíslem, a počet těchto prvočísel si na základě nějakého rozhodnutí omezíme. Pak jsme z exponentů těchto prvočísel schopni vyjádřit vektor $e\,(m) = (e_1, e_2, \ldots)$. Podívejme se opět na příklad, kdy se snažíme rozložit číslo $n = 1649$. Víme, že:

$$\begin{aligned} u_1^2 &= 41^2 = 1681 \equiv 32 \ (mod\, 1649) \\ u_2^2 &= 42^2 = 1764 \equiv 115 \ (mod\, 1649) \\ u_3^2 &= 43^2 = 1849 \equiv 200 \ (mod\, 1649) \end{aligned} \tag{6.25}$$

Označme $m_1 = 32$, $m_2 = 115$, $m_3 = 200$ a pokusme se je rozložit na prvočísla, kdy se omezíme, že se rozkládat bude maximálně na první 3 prvočísla, což jsou 2, 3 a 5:

$$\begin{aligned} m_1 &= 32 = 2^5 \\ m_3 &= 200 = 2^3 * 5^2 \end{aligned} \tag{6.26}$$

Jak můžeme vidět, při daném omezení se nám číslo m_2 nepodařilo rozložit, protože jedním z faktorů je prvočíslo 23, které nepatří mezi první 3 prvočísla. Povšimněme si, že

spojením čísel m_1 a m_3 dostaneme $m_{13} = (2^5) * (2^3 * 5^2) = 2^8 * 5^2 = \left(2^4 * 5\right)^2$, což je čtverec. Vyjádřeme si ale nyní vektory exponentů:

$$\begin{aligned} e\left(m_1\right) &= (5,0,0) \\ e\left(m_3\right) &= (3,0,2) \end{aligned} \tag{6.27}$$

Jelikož hledáme čtverec, dané hodnoty exponentů pro nás obsahují nepotřebné informace. Nám bude stačit, pokud budeme mít exponenty vyjádřeny v rámci modulo 2:

$$\begin{aligned} e\left(m_1\right) &\equiv (1,0,0)\ mod\, 2 \\ e\left(m_3\right) &\equiv (1,0,0)\ mod\, 2 \end{aligned} \tag{6.28}$$

Pokud sečteme vektory $e\left(m_1\right)$ a $e\left(m_3\right)$, dostaneme nulový vektor v rámci modulo 2, což opět znamená, že složené číslo je čtverec.

Tento příklad je velice jednoduchý a pouze jsme si na něm demonstrovali princip, jak se postupuje při hledání vhodných čísel pro získání čtverce. Ukažme si ještě jeden příklad, který bude složitější a na první pohled nebude jasné, která čísla zkombinovat, abychom získali v^2. Mějme tedy číslo $n = 2041$, které se snažíme rozložit. Opět budeme iterovat:

$$\begin{aligned} u_1^2 &= 46^2 = 2116 \equiv 75\ (mod\, 2041) \\ u_2^2 &= 47^2 = 2209 \equiv 168\ (mod\, 2041) \\ u_3^2 &= 48^2 = 2304 \equiv 263\ (mod\, 2041) \\ u_4^2 &= 49^2 = 2401 \equiv 360\ (mod\, 2041) \\ u_5^2 &= 50^2 = 2500 \equiv 459\ (mod\, 2041) \\ u_6^2 &= 51^2 = 2601 \equiv 560\ (mod\, 2041) \end{aligned} \tag{6.29}$$

Zatím se nám nepodařilo nalézt žádný čtverec, což by pro Fermatovu metodu znamenalo iterovat dále. Poznamenejme, že číslo $n = 2041 = 13 * 157$, což později dokážeme, a tedy Fermatova metoda by iterovala až k číslu 157! Nám postačí těchto 6 iterací. Omezme se, že budeme čísla rozkládat pouze na první 4 prvočísla, tj. na prvočísla 2, 3, 5 a 7. Proveďme tedy rozklad:

$$\begin{aligned} m_1 &= 75 = 3 * 5^2 \\ m_2 &= 168 = 2^3 * 3 * 7 \\ m_4 &= 360 = 2^3 * 3^2 * 5 \\ m_6 &= 560 = 2^4 * 5 * 7 \end{aligned} \tag{6.30}$$

Nepodařilo se nám tedy rozložit čísla m_3a m_5. Nyní vytvořme vektory exponentů:

$$\begin{aligned} e(m_1) &= (0,1,2,0) \\ e(m_2) &= (3,1,0,1) \\ e(m_4) &= (3,2,1,0) \\ e(m_6) &= (4,0,1,1) \end{aligned} \tag{6.31}$$

V prostoru modulo 2 vypadají vektory následně:

$$\begin{aligned} e(m_1) &= (0,1,0,0) \\ e(m_2) &= (1,1,0,1) \\ e(m_4) &= (1,0,1,0) \\ e(m_6) &= (0,0,1,1) \end{aligned} \tag{6.32}$$

Pokud všechny tyto vektory sečteme, dostaneme opět nulový vektor, což značí, že pokud vynásobíme čísla m_1, m_2, m_4 a m_6, získáme čtverec.

Tento příklad a příklad předešlý jsou vykonstruované pro prezentaci Kraitchikovy ideje. Vždy provedeme tolik iterací, kolik potřebujeme, a vždy si vybereme maximální počet prvočísel, který potřebujeme, abychom vyeliminovali nepotřebná čísla a pouze sečetli zbylé vektory k získání čtverce. V praxi samozřejmě musíme nějak vhodně odhadnout, kolik iterací budeme muset provést a na kolik prvočísel rozkládat, abychom byli schopni kombinací nějakých získaných čísel dostat čtverec. Proto páni Brillhart a Morrison navrhují zvolit si nějakou hranici B, kde B vyjadřuje počet prvních prvočísel. V takovém případě budeme získávat pouze taková čísla, jež jsou rozložitelná prvními B prvočísly. Z každého takového čísla si vytvoříme vektor exponentů, který bude B-dimenzionální. Je důležité poznamenat, že vektory budou vždy vyjádřeny ve vektorovém prostoru $\mathbb{F}_2^B$. Pokud najdeme takových vektorů $B+1$, pak zcela jistě budou tyto vektory lineárně závislé. Díky lineární závislosti budeme schopni nalézt nulový vektor a tedy kombinaci čísel, jejichž vynásobením získáme čtverec. Pro hledání lineární závislosti existuje mnoho algoritmů, některé z nich jsou popsány v kap. 6.6.2.

Jak ale zvolit hodnotu B? To je otázka, na kterou nelze jednoznačně odpovědět. Pokud zvolíme B malé, pak potřebujeme jen málo čísel k nalezení čtverce. Problémem je, že kvůli nízké hodnotě B bude těžké nalézt taková čísla, která lze rozložit pouze na B prvočísel. Naopak pokud zvolíme B velké, pak budeme čísla, která lze rozložit na B prvočísel, získávat rychleji, ale za to jich budeme potřebovat hodně. Musíme tedy najít vhodný kompromis a každá zvolená hodnota B může mít velký vliv na rychlost výpočtu.

Je důležité zmínit, že pokud nasbíráme dostatečný počet vektorů a najdeme mezi nimi lineární závislost, není zaručeno, že kombinace čísel tvořící čtverec povede k získání netriviálního rozkladu. Vedou-li nalezené vektory k triviálnímu rozkladu, je nutné proces prosívání opakovat.

6.6.2 Metody pro nalezení lineární závislosti

Nejjednodušší variantou pro hledání lineární závislosti je Gaussova eliminační metoda. Tato metoda ovšem není pro použití u faktorizace úplně nejvhodnější, protože metoda nedokáže

využít vlastností řídkých matic, které budou ve fázi prosévání vznikat. Časová složitost této metody je kubická, a tak doba zpracování matice s vyššími čísly bude značně narůstat. Tato metoda se využívala hlavně v době, kdy ostatní algoritmy pro řešení tohoto problému neexistovaly. Velkou výhodou této metody je její velmi jednoduchá implementace [9]. Jak ale bylo zmíněno, nevýhodou je obecně její časová náročnost, a tak zpracování matice touto metodou je únosné pro čísla do 110 dekadických číslic.

Proto se používají alternativy ke Gaussově eliminační metodě, které jsou mnohem efektivnější. Jednou z takových metod je Lanczos metoda [4, 14]. Lanczos metoda stejně jako Gaussova eliminační metoda slouží k řešení soustavy rovnic tvaru $Ax = y$. Páni Montgomery a Coppersmith nezávisle na sobě však upravili metodu tak, aby hledala pouze vektor reprezentující řádky, které jsou na sobě závislé [5, 11]. Cílem metody je tedy najít vektor x takový, aby platilo $Ax = 0$. Metoda pro svoji správnou funkci potřebuje symetrickou matici. Pokud matice není nesymetrická, což je případ matice vektorů exponentů, pak je nejdříve nutné vstupní matici upravit podle vztahu $A = B^T B$. Důležité je, že jak upravená metoda pana Montgomeryho, tak metoda pana Coppersmitha pracuje nad prostorem $\mathbb{F}_2$. To je prostor, se kterým pracujeme i v rámci matice vektorů exponentů a použití jedné z jmenovaných metod je tedy pro účely hledání lineární závislosti v matici vektorů exponentů velice efektivní. Potvrzením tohoto tvrzení je pak časová složitost, která u Montgomeryho metody nabývá hodnoty dn^2, což je oproti Gaussovy eliminační metodě značný rozdíl.

6.6.3 Large Prime Variation

Jak jsme si vysvětlili v kapitole 6.6.1, hlavní úlohou kvadratického síta je hledání relací $x_i^2 \equiv y_i \ (mod\, n)$, kde y_i je B-hladké (viz kap. 2.2). To, že je y_i B-hladké, nám umožňuje jej rozložit na prvočísla $p_i < B$, kdy exponenty ukládáme do vektoru v prostoru $\mathbb{F}_2^B$. Když máme dostatečný počet vektorů, snažíme se mezi nimi nalézt lineární závislost a vybrat tedy ta y_i, která svým produktem mezi sebou vytvoří čtverec Y^2. Jelikož ale hledáme pouze ta y_i, která jsou B-hladká, budeme mít mnoho relací, které tuto podmínku nesplnily, a tak je pro získání faktoru nepoužijeme. Pokud by ale existovala možnost, jak využít aspoň nějakou část nevyužitých relací, čas prosívání by se mohl znatelně urychlit, což by napomohlo k faktorizaci větších čísel, kde každé urychlení je znát.

Řešením je vylepšení kvadratického síta, nazývané Large Prime Variation. Pokud máme faktorizační bázi, tedy množinu všech prvočísel ohraničenou hodnotou B, pak při nalezení B-hladkého y_i jej rozložíme na prvočísla z faktorizační báze a zbude nám číslo 1. Jestliže ale y_i B-hladké nebylo, pak nás zkoumaná relace nezajímala, protože při rozkladu y_i zbylo nějaké číslo $P > 1$. Zaměřme se ale na zbylé P. Platí-li pro P, že $B < P \leq B^2$, pak víme, že P je zcela jistě prvočíslo. Podobného poznatku jsme si mohli všimnout u Eratosthenova síta, kdy jsme pro zadanou hranici n prosívali pouze do hodnoty $\lceil \sqrt{n} \rceil$, protože všechna prvočísla $p \leq \lceil \sqrt{n} \rceil$ nám odstranila z množiny všechny jejich násobky v intervalu $(\lceil \sqrt{n} \rceil ; n\rangle$, a tedy v tomto intervalu nám zůstanou pouze prvočísla. V takovém případě můžeme zkoumanou relaci zapsat jako $x^2 \equiv yP \ (mod\, n)$ a yP nazýváme číslo částečně B-hladké. Nalezneme-li alespoň dvě relace, které mají jako zbytek po rozkladu prvočíslo P, pak můžeme spojit tyto relace takto:

$$(x_1 x_2)^2 \quad \equiv \quad y_1 y_2 P^2 \ (mod\, n) \tag{6.33}$$

Při vytváření vektoru exponentů vytvoříme vektor pouze o B dimenzích. To si můžeme dovolit, protože máme P^2 a víme, že $2 \equiv 0 \ (mod\, 2)$, což pro nás znamená, že nemusíme

udržovat ve vektoru exponentů žádnou informaci o exponentu prvočísla P. Pokud najdeme více relací se stejným P, pak můžeme vytvářet plnohodnotné relace tímto způsobem:

$$(x_1 x_i)^2 \equiv y_1 y_i P^2 \ (mod\, n)\,,\ pro\, i = 2 \ldots k, \tag{6.34}$$

kde k je počet nalezených relací se stejným prvočíslem P.

Toto vylepšení kvadratického síta nám tedy pomáhá najít potřebné relace rychleji a přitom není třeba vyvinout téměř žádné úsilí navíc. Pokud budeme uchovávat relace s jedním prvočíslem mimo faktorizační bázi, tedy budeme postupovat podle toho, co bylo napsáno výše, nazýváme toto vylepšení kvadratického síta jako Single Large Prime Variation. Můžeme se ovšem rozhodnout, že kromě relací s jedním prvočíslem mimo faktorizační bázi budeme ukládat i relace se dvěma prvočísly mimo faktorizační bázi. Toto vylepšení nazýváme Double Large Prime Variation. Pro některé případy toto vylepšení přináší ještě další urychlení oproti Single Large Prime Variation, avšak přináší s sebou problémy, které je nutné řešit. Další varianty Large Prime Variation se obecně nepoužívají, protože problémy, které přinášejí, zastiňují případný zisk.

6.7 Multipolynomiální kvadratické síto (MPQS)

Pro prosívání v kvadratickém sítu, které bylo probráno v kapitole 6.6.1, používáme polynom $Q(x) = x^2 - n$, kdy x inicializujeme na $x = \lceil \sqrt{n} \rceil$ a postupně iterujeme x, přičemž se snažíme najít relace, kdy $Q(x)$ bude B-hladkým číslem. Tím, že x inicializujeme na $\lceil \sqrt{n} \rceil$, zajistíme, že budou hodnoty $Q(x)$ v počátečních iteracích malé, a tedy bude velmi pravděpodobné, že tyto hodnoty $Q(x)$ budou B-hladké. Problémem je, že s každou iterací x se budeme od $\lceil \sqrt{n} \rceil$ vzdalovat, hodnoty $Q(x)$ budou větší a tak pravděpodobnost, že $Q(x)$ je B-hladké, bude klesat. Navíc jelikož $Q(x) = x^2 - n$, porostou hodnoty $Q(x)$ poměrně rychle, a tedy pravděpodobnost na B-hladkost bude klesat také poměrně rychle. Proto se začalo uvažovat nad možnostmi, které by tento problém vyřešily.

Řešením je Multipolynomiální kvadratické síto [17]. Jak název napovídá, tato metoda bude používat více polynomů k hledání relací. Tím si totiž zaručíme, že vždycky když budeme vzdáleni od $\lceil \sqrt{n} \rceil$ tak, že pravděpodobnost výskytu B-hladké hodnoty bude velice nízká, budeme si moct vygenerovat polynom nový a opět pokračovat od hodnot, které budou blízké $\lceil \sqrt{n} \rceil$. To samozřejmě vyžaduje úpravu původního polynomu. Tvar polynomu, který se používá u metody MPQS má tvar:

$$Q_{a,b}(x) = ax^2 + 2bx + c \tag{6.35}$$

kde a,b,c jsou celá čísla a n je dále vyjádřeno jako $n = b^2 - ac$. Pak:

$$aQ_{a,b}(x) = a^2x^2 + 2abx + ac = (ax + b)^2 - n \tag{6.36}$$

a tedy získáváme:

$$(ax + b)^2 \equiv aQ_{a,b}(x) \ (mod\, n) \tag{6.37}$$

oproti:

$$x^2 \equiv Q(x) \ (mod\, n), \tag{6.38}$$

který jsme používali u klasického kvadratického síta.

Hodnoty koeficientů a,b,c volíme podle toho, na jak velkém intervalu chceme prosívat pomocí jednoho polynomu. Prosívání u MPQS probíhá vždy v intervalu $\langle -M; M \rangle$, kde M je nějaké celé číslo, a u každého polynomu tedy provedeme $2M+1$ iterací. Hodnotu koeficientu b volíme vždy v závislosti na hodnotě a tak, aby platilo $0 < |b| < \frac{1}{2}a$. Díky vytvoření této závislosti je zaručeno, že budeme moci pracovat s proměnnou x přesně v intervalu $\langle -M; M \rangle$. Při dosazení minimální nebo maximální hodnoty intervalu do polynomu získáme výraz pro největší hodnotu polynomu $Q(M) \approx {}^{(a^2M^2-n)}/_{a}$. Dosazením $x = 0$ získáme výraz pro nejmenší hodnotu, což bude $Q(0) \approx {}^{-n}/_{a}$. Hodnoty jsou přibližné, protože ve skutečnosti budeme pracovat s celými čísly, pokud by zde bylo výsledkem reálné číslo, bylo by nutné jej zaokrouhlit. Na základě získaných výrazů zvolíme $a \approx {}^{\sqrt{2n}}/_{M}$, aby po dosazení byly největší a nejmenší hodnoty přibližně podobné.

Pouze zvolení $a \approx {}^{\sqrt{2n}}/_{M}$ však nestačí. U QS jsme hledali takové relace, pro něž pravá strana této relace obsahovala hodnotu, která byla B-hladká. Toho stejného potřebujeme dosáhnout i u MPQS. Potřebujeme tak, aby hodnota $Q_{a,b}(x)$ byla opět B-hladká, a musíme tedy zajistit, aby a bylo čtvercem B-hladkého čísla. Tím, že a bude čtverec B-hladkého čísla zaručíme, že daná relace bude plnohodnotná a a nijak neovlivní vektor exponentů.

Výše bylo zmíněno, že omezení pro koeficient b je $0 < b < \frac{1}{2}a$, avšak nebylo řečeno, jak takové b získat. Pokud získáme b pomocí $b^2 \equiv n \ (mod\, a)$, pak zajistíme, že bude platit první zmíněné omezení pro b. Zároveň ze znalosti a a b jsme schopni dopočítat koeficient $c = {}^{(b^2-n)}/_{a}$. Danou kongruenci spočteme pomocí Tonelliho-Shanksova algoritmu (viz kap. 3.4). Ten ovšem vyžaduje, aby a bylo liché prvočíslo. Tato podmínka navíc již více neomezuje volbu a, protože výše bylo zmíněno, že požadujeme po a, aby bylo čtvercem B-hladkého čísla. Toho se obecně dosahuje tak, že zjistíme, která prvočísla se nacházejí kolem hodnoty ${}^{(2n)^{\frac{1}{4}}}/_{M^{\frac{1}{2}}}$, a vybereme jedno takové prvočíslo p, pro které $\left(\frac{n}{p}\right) = 1$, a a tedy zvolíme jako $a = p^2$. S každým zvoleným p jsme schopní vytvořit nový polynom. Pokud by se nám z nějakého důvodu nepodařilo nalézt vhodné prvočíslo, pro které platí $\left(\frac{n}{p}\right) = 1$, můžeme použít i jiné prvočíslo, ale musíme pamatovat na to, že zvolené prvočíslo nebude součástí faktorizační báze.

Jako poslední je nutné vyřešit velikost celého čísla M. Pokud zvolíme M příliš dlouhé, budeme muset pro každý polynom projít mnoho hodnot, a jak bylo zmíněno výše, s každou iterací se hodnota $Q_{a,b}(x)$ bude zvyšovat, a tak pravděpodobnost na získání B-hladkého čísla se bude snižovat, což povede k tomu, že prosívání bude trvat dlouho. Zvolení M jako velmi malé číslo také není vhodné, protože výpočet nového polynomu není triviální operace, a pokud budeme měnit polynomy často, pak výpočet nového polynomu bude značně ovlivňovat rychlost prosívání. Obvykle se tedy volí $M = B$.

6.8 Self-Initialization quadratic sieve (SIQS)

U metody MPQS (viz kap. 6.6.2) bylo řečeno, že počet provedených iterací M se obvykle volí jako $M = B$, protože pokud bychom zvolili M menší, pak by generování polynomu mohlo

ovlivnit celkový čas prosívání, jelikož generování nového polynomu není jednoduchou záležitostí. Problémem je, že čím vyšší M je, tím více rostou vypočtené hodnoty, pravděpodobnost tak na získání B-hladkého čísla klesá, a tedy celková doba prosívání bude delší. Začalo se tedy zkoumat, jaké jsou možnosti řešení tohoto problému. Jedním z řešení je metoda Large Prime Variation, která byla popsána v kapitole 6.6.3. Druhým řešením je Samoinicializující se kvadratické síto [4].

Polynomy, které slouží k prosívání relací, jsou u SIQS stejného tvaru jako u MPQS (viz 6.7). Mění se ovšem podmínka pro výběr hodnoty pro koeficient a. U MPQS bylo požadováno, aby a bylo čtverec B-hladkého čísla nebo aby bylo čtvercem jakéhokoli prvočísla, třeba i prvočísla, jež nespadá do faktorizační báze. Důležité je, aby a bylo čtvercem a neprodlužovalo tak vektor exponentů. V případě SIQS volíme však a jako produkt prvočísel z faktorizační báze, tedy $a = q_1 \dots q_s$, a zároveň q_l bude chápáno jako prvočíslo, jež patří mezi všechna prvočísla, pomocí kterých byla vytvořena hodnota koeficientu a. Hodnota $Q_{a,b}(x)$ je tak B-hladkým číslem tehdy a jen tehdy, je-li $ax^2 + 2bx + c$ číslo B-hladké. Je tak zároveň možné vytvořit různé polynomy, protože prvočísla z faktorizační báze je možno různě kombinovat. Jak bylo popsáno u metody MPQS, koeficient b získáme pomocí vztahu $b^2 \equiv N \ (mod\, a)$. Díky tomu, že pro a platí $a = q_1 \dots q_s$, pak počet hodnot koeficientu b, které můžeme získat, je 2^s. Jelikož ale výpočtem $Q_{a,b}(x)$ jsou získány stejné zbytky jako v případě použití $Q_{a,-b}(x)$, pak se počet použitelných hodnot koeficientu b sníží na 2^{s-1}, a tedy pro jednu hodnotu koeficientu a lze vygenerovat 2^{s-1} polynomů. Aby bylo možné získat hodnotu b, je nutné nejdříve spočítat všechna B_l $(1 \leq l \leq s)$ takto:

$$B_l^2 \equiv N \ (mod\, q_l) \wedge B_l \equiv 0 \ (mod\, q_j) \ pro\, 1 \leq j \leq s,\ j \neq l \tag{6.39}$$

Řešení výrazu $B_l^2 \equiv N \ (mod\, q_l)$ získáme použitím Tonelliho-Shanksova algoritmu (viz kap. 3.4). Ze všech takto spočtených B_l lze vybrat jakoukoli kombinaci $\pm B_1 \pm \ldots \pm B_s$ a čtverec každé vybrané kombinace bude kongruentní s N modulo a. Vybraná kombinace je tak zároveň hodnotou koeficientu b.

U metody SIQS se jako první vybere $b_1 = B_1 + \ldots + B_s \ (mod\, a)$ a v případě potřeby vygenerovat nový polynom se následující hodnota b_{i+1} spočte pomocí Grayova kódu (viz kap. 3.1) takto:

$$b_{i+1} = b_i + 2\,(-1)^{\left\lceil \frac{i}{2^v} \right\rceil} B_v, \tag{6.40}$$

kde $2^v \parallel 2i$ pro $1 \leq i \leq 2^{s-1} - 1$. Spočítat tak nový polynom v případě potřeby je oproti MPQS metodě jednodušší a zároveň i rychlejší. Tím, že spočítat nový polynom již není tak časově náročné, je možné snížit hodnotu M a obměňovat tak polynomy častěji. Zvýší se tak i pravděpodobnost, že bude nalezena relace s B-hladkou hodnotou, což celkově vede k rychlejšímu provedení fáze prosívání.

6.9 Obecné číselně teoretické síto (GNFS)

GNFS je dnes nejpoužívanější a nejrychlejší faktorizační metodou pro čísla s více jak 100 dekadickými číslicemi. Dlouhou dobu drželo prvenství ve faktorizaci kvadratické síto, kdy posledním rekordem bylo číslo RSA-129, tedy číslo o 129 dekadických číslicích. Prvenství následně převzalo právě GNFS, kdy bylo úspěšně faktorizováno číslo RSA-130. GNFS bylo

v té době poměrně nové a neozkoušené, brzy se však ukázalo, že se jedná o nástupce kvadratického síta. Pro čísla menší jak 100 dekadických číslic je ale kvadratické síto stále nejrychlejší [4]. Později bylo metodou faktorizováno i číslo o 155 dekadických číslicích, tedy číslo o velikosti 512bitů [3]. Posledním rekordem je úspěšné faktorizování čísla o 232 dekadických číslicích, tedy číslo velikosti 768 bitů. Tohoto úspěchu bylo dosaženo v roce 2009 [8]. Faktorizace tohoto čísla trvala přes dva roky a autoři metody GNFS, která faktorizovala toto číslo tvrdí, že faktorizovat číslo velikosti 1024 bitů by bylo asi tisíckrát náročnější.

Se samotnou metodou přišel roku 1988 pan Pollard, který je také autorem již představených metod Pollard ρ (viz kap. 6.2) a Pollard $p-1$ (viz kap. 6.3). Metoda byla původně určena jen k faktorizaci speciálních čísel, konkrétně čísel tvaru $2^{2^r}+1$. Roku 1989 pan Pomerance provedl analýzu, jak moc by se změnila časová složitost metody, pokud by se metoda upravila a použila pro faktorizaci obecných čísel. Zjistil, že složitost metody by se závratně nezhoršila a překvapivě by měla být lepší než v případě kvadratického síta. Samotnou úpravu metody pro použití k faktorizaci obecných čísel však provedli páni Joe Buhler, Henrik Lenstra a další [18].

Obdobně jako u kvadratického síta (viz kap. 6.6) se metoda snaží nalézt čísla x a y taková, aby platilo $x^2 \equiv y^2 \ (mod\, n)$. Tato čísla se však snaží GNFS nalézt jinak. Nejdříve je vybrán polynom $f(x)$ stupně $d > 1$ a tento polynom je dále nerozložitelný. Pro tento polynom zároveň existuje číslo m takové, že $f(m) \equiv 0 \ (mod\, n)$. V tomto případě tedy pracujeme v prostoru $\mathbb{Z}_n$. Dále nechť α je komplexním kořenem polynomu $f(x)$ a uvažujme těleso $\mathbb{Z}[\alpha]/f(\alpha)$ (viz kap. 2.11). Toto těleso je množinou všech polynomů, jež jsou kongruentní s 0 modulo $f(\alpha)$. Jelikož $f(m) \equiv 0 \ (mod\, n)$ v $\mathbb{Z}_n$ a $f(\alpha)$ je nulovým prvkem v $\mathbb{Z}[\alpha]/f(\alpha)$, můžeme mezi těmito okruhy definovat homomorfizmus φ.

Nyní mějme množinu S dvojic (a, b), kdy a a b jsou navzájem nesoudělná a disponují dvěma vlastnostmi:

- Součin čísel $\prod_{(a,b)\in S}(a+bm)$ je čtvercem v $\mathbb{Z}_n$, tedy v^2
- Součin polynomů $\prod_{(a,b)\in S}(a+b\alpha)$ je čtvercem v $\mathbb{Z}[\alpha]/f(\alpha)$, tedy γ^2

Jelikož nad okruhy máme definován homomorfizmus φ a γ lze napsat jako polynomiální výraz v α, získáme číslo u, pro které platí $\varphi(\gamma) \equiv u \ (mod\, n)$. Pak zcela jistě bude platit:

$$\begin{aligned} u^2 &\equiv \varphi(\gamma)^2 = \varphi\left(\gamma^2\right) = \varphi\left(\prod_{(a,b)\in S}(a-\alpha b)\right) = \\ &= \prod_{(a,b)\in S}\varphi(a-\alpha b) = \prod_{(a,b)\in S}(a-mb) \equiv v^2 \ (mod\, n), \end{aligned} \tag{6.41}$$

tím jsme získali potřebný vztah pro výpočet faktoru složeného čísla n.

Jak ale získat množinu S? Výběr a a b splňující první vlastnost je možné provést úplně stejným postupem jako při prosívání u kvadratického síta (viz kap. 6.6.1). Čísla a a b však musí mít i druhou vlastnost, jak postupovat v tomto případě je blíže popsáno v tomto článku [10]. Výsledkem prosívání je opět matice vektorů exponentů, a tak další postup je identický s postupem u metody kvadratického síta (viz kap. 6.6).

Kapitola 7

Zvolená metoda faktorizace a návrh implementace

7.1 Rozbor metod

I přes poměrně jednoduchou konstrukci a případně i možnosti paralelizace u faktorizačních metod s exponenciální složitostí jsou tyto metody z výběru vyřazeny. Zavedení paralelizace u těchto metod nepřinese dostatečné snížení složitosti, složitost by i tak zůstala exponenciální, a tedy paralelizovaná metoda by od určité hranice byla opět časově náročnější než metoda subexponenciální. Větší potenciál tak představují metody se subexponenciální složitostí. Paralelizace těchto metod také nepovede k významnému snížení složitosti, ale vzhledem k tomu, že metody s menší časovou náročností než subexponenciální nebyly zatím nalezeny, je každé urychlení žádoucí.

Kandidáti na zvolenou metodu jsou tedy *Kvadratické síto* a jeho varianty (viz kapitoly 6.6, 6.7, 6.8) a *Obecné číselně teoretické síto* (viz kapitola 6.9). Obecné číselně teoretické síto je metodou, která se používá pro faktorizaci čísel majících více jak 100 číslic. Doba faktorizace tak vysokého čísla však překračuje časové možnosti této práce a zvolenou metodou, která bude paralelizována a otestována v této práci, bude tedy metoda kvadratického síta. Tato práce však může mít přínos i pro metodu obecného číselně teoretického síta, protože po dokončení fáze prosívání tyto metody pracují shodně.

7.2 Zvolená faktorizační metoda a návrh paralelizace

Zvolenou metodou pro tuto práci je kvadratické síto. Tuto metodu je možno rozdělit do několika fází, a to do fáze *generování polynomu*, ta je u klasického kvadratického síta vynechána, protože pracuje jen s jedním polynomem, *prosívání* (viz kapitola 6.6.1), *hledání lineární závislosti* (viz kapitola 6.6.2) a *výpočet faktoru*.

Fáze generování polynomu, jak bylo zmíněno, probíhá pouze u metod MPQS (viz kapitola 6.7) a SIQS (viz kapitola 6.8). Při zvolení jedné z těchto metod je možné vygenerovat polynom pro každé jádro, které máme k dispozici. Fáze generování polynomu je tak dobře paralelizovatelná. V případě potřeby si jádro může vždy vygenerovat nový polynom. U metody SIQS si jádro může předpočítat některé hodnoty a generovat tak efektivněji nové polynomy.

Fáze prosívání pak může probíhat paralelně na každém jádru, kdy každé jádro zná hranici

M a faktorizační bázi F. Faktorizační bázi je ještě před generováním polynomu a samotným prosíváním možné zredukovat o ta prvočísla, která zcela jistě nikdy nebudou součástí B-hladkého čísla (viz kap. 7.4). Tím vznikne faktorizační báze, kterou budeme dále označovat B. Redukce je provedena buď sériově anebo paralelně tak, že každému jádru je přiřazeno kvantum prvočísel, které je následně jádrem prozkoumáno. Jak bylo zmíněno, prosívání probíhá paralelně tak, že každé jádro má svůj polynom, nad kterým generují v každé iteraci hodnotu $Q_{a,b}(x)$. Hodnota x je iterována v rozsahu $\langle -M; M \rangle$. Pokud jádro nalezne B-hladké číslo, je vektor exponentů tohoto čísla uložen do matice, která bude zpracována ve fázi hledání lineární závislosti. Celkový počet takto nalezených čísel je $\#(B)+1$, protože pak máme zaručeno, že v matici vektorů exponentů bude lineární závislost. Tento počet je možné podělit číslem jader, které jsou využity k prosívání, a každé jádro bude muset najít tento počet B-hladkých čísel. Jinou možností je mít počitadlo nalezených B-hladkých čísel ve sdílené paměti, a pokud jádro nalezne B-hladké číslo, opět uloží jeho vektor exponentů a zvýší počitadlo o 1. U této varianty budou jádra hledat B-hladké číslo, dokud nebude nasbírán dostatečný počet těchto B-hladkých čísel a dojde tak k lepšímu využití jader, kdežto u první varianty by mohlo dojít k situaci, kdy by bylo nutné čekat, než jednotlivá jádra naleznou přidělený počet B-hladkých čísel. Pokud proměnná x nabyla již všech hodnot z intervalu $\langle -M; M \rangle$, musí si jádro vygenerovat nový polynom. Po vygenerování nového polynomu prosívá jádro opět na celém intervalu $\langle -M; M \rangle$.

Pro fázi hledání lineární závislosti je v rámci paralelizace kvadratického síta vhodné použít blokovou variantu Lanczos metody popsané v kapitole 6.6.2. V takovém případě se totiž bude pracovat s podprostory, jejichž počet dimenzí bude roven počtu bitů ve slově na použitém systému. Práci s maticí vektorů exponentů pak bude možno rozdělit na úlohy a tyto úlohy pak řešit na jednotlivých jádrech.

Výpočet faktoru je pak triviální záležitostí, protože z předchozí fáze získáme vztah $x^2 \equiv y^2 \pmod{n}$ a stačí tedy spočítat $GCD(x-y,\, n)$. Výsledkem operace největšího společného dělitele může ovšem být i triviální dělitel, který pro nás není důležitý. V takovém případě je nutné se vrátit k fázi vygenerování polynomu, vypočítat nové polynomy a celý proces opakovat.

Implementována bude konkrétně metoda SIQS, protože se jedná o nejrychlejší variantu kvadratického síta. SIQS je zároveň nejrychlejší metodou pro faktorizaci čísel, která mají do 100 dekadických číslic, a druhou nejrychlejší faktorizační metodou vůbec. Způsob, jakým bude metoda implementována, je popsán v kapitole 7.4.

7.3 Možnosti paralelizace

7.3.1 Klasická vlákna

Klasická vlákna jsou zde vlákna z knihovny *pthread.h* na systému Unix anebo z *windows.h* na systému Windows. Ať už na systému Unix nebo Windows, knihovna vždy obsahuje metody pro práci s vlákny, jako je například vytváření, rušení vláken nebo komunikace mezi vlákny. Veškeré řízení a správa vláken je tak v režii programátora, což programátorovi ztěžuje práci, a kontrola správnosti kódu je náročnější, hlavně pokud je nutné mezi vlákny řešit komunikaci a synchronizaci. Na druhou stranu má nad vlákny úplnou kontrolu a může definovat, v kterou chvíli se mají operace nad vlákny vykonat.

7.3.2 nVidia CUDA

CUDA je zkratkou pro Compute Unified Device Architecture a jedná se o rozšíření jazyka C/C++, které poskytuje knihovní funkce pro využití GPU. Příkladem poskytovaných funkcí může být funkce pro přenos dat do paměti GPU, rozčlenění výpočtu do vláken a bloků, synchronizace vláken a také funkce pro spuštění výpočtu na GPU.

Program využívající GPU jako výpočetní síly pomocí CUDA obecně vypadá tak, že data ke zpracování a všechny přípravy před výpočtem jsou provedeny na CPU. Následně jsou připravená data odeslána do paměti GPU. Je spuštěn kernel, což je program, který se má vykonat na GPU a data zpracovává paralelně. Po dokončení výpočtu jsou data z paměti GPU přenesena zpět na CPU. Data mohou být následně ještě dodatečně dozpracována. Všechny výpočty, které jsou paralelizované, jsou počítány na GPU, naopak výpočty, které je nutné spočítat sériově, jsou počítány na CPU, protože výpočet takových úloh je na CPU efektivnější.

GPU je poměrně rozdílná architektura oproti CPU a programátor tak nemůže program pomocí CUDA programovat stejně jako na CPU. Je třeba změnit způsob myšlení, protože úlohy, které mají efektivně využívat GPU, by měly používat tisíce vláken, aby se překryla latence hlavní paměti GPU. Zároveň ale pokud chceme mít efektivní kód, který využívá tolik vláken, znamená to, že na každé vlákno bude připadat málo prostoru ve sdílené paměti. Kód tak musí být zároveň úsporný. Zmíněné problémy jsou pouze částí všeho, co musí programátor při programování na GPU řešit, aby paralelizace dosáhla požadovaného efektu. Pokud je ale kód napsán dobře, rychlost takového programu může značně předčit rychlost CPU varianty tohoto programu.

7.3.3 OpenMP

Open Multi-Processing neboli OpenMP[1] je API pro paralelní výpočty se sdílenou pamětí a je možno jej uplatnit v jazycích C/C++ nebo Fortran. OpenMP bylo vytvořeno a je vyvíjeno konsorciem mnoha SW a HW výrobců jako například Intel, AMD, Microsoft, nVidia, IBM atd. Základní komponenty toho API jsou direktivy překladače, runtime knihovní rutiny a proměnné prostředí. OpenMP se snaží docílit co možná největšího ulehčení práce s vlákny, kdy využívá klasická vlákna (viz kapitola 7.3.1), ale práci nad těmito vlákny před programátorem zakrývá například použitím direktiv překladače. OpenMP je podporováno na Windows například překladačem, který je standardně instalován s Visual Studiem, ale také na Linuxu, kde jej podporuje překladač gcc. Obecně je ale možné využít OpenMP kdekoli, kde je podporováno překladačem. Díky tomu, že OpenMP zastřešuje operace s vlákny, programátor má ušetřenou práci a zároveň je z tohoto pohledu zajištěna přenositelnost kódu, protože OpenMP pracuje s vlákny na dané platformě.

7.3.4 OpenMPI

Mechanizmus zasílání zpráv je běžně používán, pokud potřebujeme zajistit komunikaci mezi procesy, které mají oddělené adresové prostory, což znamená, že žádná data nejsou sdílená. Implementací takového mechanizmu je například Open Message Passing Interface neboli OpenMPI[2]. Jedná se o knihovnu, kterou můžeme použít v jazycích C++, Fortran, Java, Python a R. Knihovna zároveň zajišťuje přenositelnost námi implementovaných programů,

[1] http://openmp.org/wp/
[2] http://www.open-mpi.org/

protože ji můžeme používat na clusterech, síti PC, SMP apod. Knihovna kromě samotné komunikace poskytuje nástroje k dalším interakcím procesů, jako je agregace či synchronizace. Dále také umožňuje vytvářet a spravovat procesy či skupiny. Na rozdíl od OpenMP (viz kap. 7.3.3) OpenMPI potřebuje jako argument počet procesů, které se mají v rámci programu vytvořit, OpenMP se naopak vždy automaticky snaží použít všechna dostupná jádra (jeden proces pro jedno jádro procesoru), pokud nespecifikujeme jinak. Dalším rozdílem je fakt, že OpenMPI vytváří procesy při spuštění paralelního programu a tyto procesy existují až do konce běhu programu. Při použití OpenMP lze jednotlivé procesy dynamicky vytvářet a ukončovat dle potřeby. Je důležité si však uvědomit, že OpenMP se používá k paralelizaci na jedné pracovní stanici, protože OpenMP neumožňuje komunikovat mimo pracovní stanici, to naopak umožňuje OpenMPI, a proto OpenMPI používáme pro paralelizaci programu na síti počítačů. Dále je nutné poznamenat, že OpenMP i OpenMPI lze použít zároveň a vytvořit tak program, který bude paralelizovaný na síti stanic a zároveň úloha na každé stanici bude také paralelizována.

7.4 Návrh řešení a postup implementace

Projekt bude tvořen v jazyce C++ na architektuře x86 resp. x86-64. I přes to, že projekt bude rozsáhlý, bude se pracovat s jednoduchými strukturami, a tak se nebude využívat objektová orientace, kterou jazyk C++ poskytuje. Bylo by tak možné využít jazyka C, ten však neposkytuje některé standardizované knihovny, které má jazyk C++ k dispozici a budou se v rámci projektu používat. Ekvivalentní knihovny by tak musely být při použití jazyka C buď implementovány anebo převzaty. Tím, že jsou tyto knihovny v jazyce C++ již standardizované, je možné je jednoduše použít, ušetřit tak čas strávený jejich implementací a zaměřit se přímo na problém faktorizace. Jazyk C++ 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 (viz kap. 7.3.3). Bude tak možné jednoduše určovat místa, která mají být paralelizována, předávat vláknům data, určovat, jaká kvanta práce se budou jednotlivým vláknům přidělovat apod.

U faktorizace je však nutné řešit jeden důležitý problém. Tím problémem je fakt, že tradiční programovací jazyky ani architektury neposkytují datový typ, který by byl schopen pojmout čísla, která se mají faktorizovat. Vezměme v úvahu například datový typ *unsigned long int*, který bude mít délku 64 bitů. Do takového datového typu jsme schopni uložit dekadické číslo maximálně o 19 číslicích. Najít faktor takového složeného čísla je dnes velice rychlé, a tak není možné použít tak malá čísla pro šifrování. Asynchronní šifry dnes používají klíče o délce 1024 či 2048 bitů, kdy faktorizace čísel o takové délce je zatím velice náročná a doposud nebyla nikým provedena. Vytvořit si vlastní datový typ, který bude schopen uchovávat čísla takové velikosti, není nijak náročné. Problémem však je, že pro jejich další použití je nutné implementovat i operace nad nimi, jako je sčítání, odečítání, násobení, dělení, modulo apod. Efektivně naimplementovat některé z těchto operací již není triviální záležitostí. Neefektivní implementace těchto operací by měla následně veliký dopad na celkový čas faktorizace a použití takové faktorizace by postrádalo smysl. Proto bude pro práci s velkými čísly využitá knihovna *gmp*, která umožňuje velká čísla uchovávat a definuje nad nimi mnoho operací, které je možné využít. Příkladem může být i test prvočíselnosti, který je v této knihovně implementován. Test prvočíselnosti je nutné provést před každým spuštěním faktorizace, protože uživatel může programu předložit k faktorizaci prvočíslo a faktorizace

by tak vždy vedla k nalezení triviálního dělitele a navíc by uživatel musel čekat na výsledek přes celý proces faktorizace. Využitím této knihovny se tedy odstraní problém s velkými čísly a projekt tak může být zaměřen přímo na problém faktorizace.

Faktorizační metoda má za cíl většinou jednu ze dvou možných úloh. První možností je faktorizace, která se provádí tak dlouho, dokud nejsou nalezeny všechny faktory. Příkladem je číslo 30. Faktorizační metoda nalezne u tohoto čísla faktor hodnoty 2. Zbylé číslo 15 je následně opět faktorizováno. Dalším nalezeným faktorem je číslo 3. Jelikož zbylé číslo je hodnoty 5, což je prvočíslo, je metoda ukončena a výsledek je $2 * 3 * 5$. Druhou možností je metoda, která se ukončí při nalezení prvního faktoru. Metoda by se pro číslo uvedené v příkladu ukončila hned při nalezení faktoru 2, jež by byl i výsledkem této metody. Jelikož se tato práce zabývá faktorizací s ohledem RSA kryptoanalýzu, kde klíč RSA šifry je složen vždy ze dvou prvočísel, bude použita druhá metoda, tedy výsledkem metody bude nalezení jednoho faktoru. Program samozřejmě bude umět faktorizovat i čísla složená z více jak dvou prvočísel, ale pokud bude uživatel chtít získat všechny faktory, bude nucen v nejhorším případě spustit program vícekrát.

Pokud uživatel zadá k faktorizaci menší číslo, není nasazení kvadratického síta úplně efektivní. Jelikož bude kvadratické síto paralelizováno v maximální možné míře, bude zde režie s vytvářením vláken, také s prací nad nimi, komunikací a ukončením. Využití paralelní metody pro malá čísla tak bude neefektivní a zbytečné. Proto bude v rámci této práce také implementována pro faktorizaci takto malých čísel Pollard ρ metoda (viz kap. 6.2), která bude postačovat a bude dostatečně rychlá a její paměťová náročnost malá.

Pro faktorizaci velkých čísel bude v této práci implementována varianta kvadratického síta SIQS (viz kap. 6.8). Tato varianta byla zvolena z důvodu, že ji lze mnohem lépe paralelizovat než klasické kvadratické síto (viz kap. 6.6) a na rozdíl od varianty MPQS (viz kap. 6.7) neztrácí tolik času inicializací nového polynomu, pokud dojde k vyčerpání zadaného rozsahu. SIQS bude vylepšeno o myšlenku Single Large Prime Variation (viz kap. 6.6.3), protože zavedení této myšlenky nepřináší téměř žádné problémy navíc a naopak může přinést poměrně velké urychlení faktorizace.

Generování polynomu bude probíhat tak, jak bylo popsáno v kapitole 7.2. Program zjistí, kolik jader má k dispozici, a na základě toho získá stejný počet prvočísel a. Každé jádro získá tak své prvočíslo a, na jehož základě si vygeneruje polynom, s nímž bude pracovat. Dále si jádro předpočítá několik hodnot b dopředu, aby v případě nutnosti vygenerování nového polynomu byl čas strávený generováním minimální. Počet předpočítaných hodnot b bude v průběhu práce vyhodnocován a poté se stanoví hodnota, která bude optimální, protože každé předpočítání navíc stojí čas.

Když je dokončena fáze generování polynomu, přechází se k fázi prosívání. Každé jádro dostane hodnotu M, která bude určovat rozsah $\langle -M; M \rangle$, přes který se bude iterovat a hledat B-hladké hodnoty. Přesná hodnota M bude určena až na základě experimentálního měření. Každé jádro dále dostane shodnou faktorizační bázi, na základě které se bude určovat, zda je zkoumané číslo B-hladké. Před samotným předáním faktorizační báze budou z této báze odstraněna prvočísla, která nebudou splňovat Legendreův symbol, protože taková prvočísla zcela jistě nikdy nebudou obsažena v B-hladkém čísle. Obdobně jako u hodnoty M, bude optimální velikost faktorizační báze určena až na základě experimentálního měření. Jak bylo zmíněno, u každého vygenerovaného čísla při prosívání je nutné určit, zda se jedná o B-hladké číslo. Zde bude využito znalosti, že pro vygenerované číslo y, jež bude bude dělitelné nějakým prvočíslem p, bude platit $y \equiv 0 \ (mod\, p)$. Jelikož generu-

jící polynom s daným $x \in \langle -M; M \rangle$ reprezentuje toto y, můžeme vztah přepsat na tvar $(x+b)^2 - N \equiv 0 \ (mod\, p)$, kde $b = \sqrt{N}$. Pro jednodušší demonstraci je zde schválně uveden generující polynom základní varianty kvadratického síta. Řešením rovnice dostáváme dva kořeny r_1 a r_2. Tyto kořeny nám určují, na kterém indexu, respektive pro kterou iteraci začne platit, že číslo generované polynomem je dělitelné prvočíslem p. Dělitelnost tímto prvočíslem bude pak platit i pro všechny jeho násobky, tedy pro všechna $y = Q\,(r_1 + k * p)$ nebo $y = Q\,(r_2 + k * p)$. Lze tedy iterovat a pro každé prvočíslo si uchovávat následující index, pro který bude platit, že dané číslo vygenerované v této iteraci bude dělitelné prvočíslem p. Pro každé takto vygenerované číslo stačí projít pole indexů, a pokud pro dané prvočíslo p se shoduje hodnota indexu uchována v poli s aktuální hodnotou indexu, je pro dané vygenerované číslo zvýšen čítač o $log(p)$. Pokud po projití všech prvočísel bude hodnota čítače přibližně $log\,(Q\,(x_i))$, kde $x_i \in \langle -M; M \rangle$ a vyjadřuje aktuální iteraci, pak je číslo y velmi pravděpodobně B-hladké. Pro úplné potvrzení je nutné použít metodu zkusmého dělení. Předpokladem je, že sčítaní logaritmů je rychlejší než zkusmé dělení, a tak se ušetří čas, protože metoda bude provádět zkusmé dělení jen u některých vygenerovaných čísel a ne u všech.

Výstupem této fáze je matice vektorů exponentů. Na základě velikosti faktorizační báze je určeno, kolik vektorů, tedy B-hladkých čísel potřebujeme k sestavení matice. Každé jádro, které nalezne B-hladké číslo, uloží jeho vektor exponentů do matice a zvýší čítač nalezených B-hladkých čísel o 1. Teoreticky by mělo stačit naleznout $B + 1$ takovýchto čísel, protože máme zaručenu lineární závislost v matici. Přesný počet ovšem bude určen až experimentálním měřením, protože může dojít k situaci, kdy řešením takto sestavené matice bude triviální dělitel složeného čísla n. Je tedy vhodné prosít více hodnot a v případě vzniku takové situace lze některé řádky vyměnit za vektory, které byly prosety navíc a zkusit najít netriviálního dělitele znovu. Takovéto opakování hledání je obecně mnohem rychlejší než spuštění procesu prosívání úplně od znovu. Dále je také nutné brát v potaz fakt, že bude použito vylepšení Single Large Prime Variation, a tak k samotným B-hladkým číslům budeme získávat částečně B-hladká čísla, které lze dále mezi sebou vhodně skombinovat a získat tak další B-hladké číslo.

Po dokončení fáze prosívání je spuštěna fáze nalezení lineární závislosti. Pro nalezení závislosti bude použita Lanczos metoda, která vhodně používá informace naléza jící se v matici, a tak je výpočet rychlejší než v případě použití tradiční Gaussovy eliminační metody. Dále je možné metodu dobře paralelizovat, protože metoda pracuje s vektory v podprostorech tvořených nad maticí, které mají velikost podle velikosti slova dané architektury, na x86 resp. x86-64 CPU tedy 32 nebo 64bitů. Spočtením lineární závislosti získáme vztah $x^2 \equiv y^2 \ (mod\, N)$, pomocí $GCD\,(x - y,\ N)$ stačí tedy ověřit, že jsme získali netriviálního dělitele.

Kapitola 8

Praktický příklad faktorizace metodou SIQS

V této kapitole bude krok za krokem rozebrána faktorizace čísla o 20 dekadických číslicích metodou SIQS. Přibližný popis a postup implementace této metody je sice popsán například v [4], problémem však je, že metoda je zde popsána 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. Úkolem této kapitoly tak je pomoci zájemci o SIQS, aby v případě, že se rozhodne metodu SIQS implementovat, byl schopen si vždy ověřit, že jeho postup je správný. Pokud zájemce narazí na nějaký problém, s největší pravděpodobností nalezne na příslušném místě řešení tohoto problému. Zájemce tak není nucen procházet více článku týkajících se SIQS a hledat odpověď na jeho problém. Zároveň ale v této kapitole nebudou popsány implementační detaily SIQS tvořeného v rámci této práce, ty jsou popsány v kapitole 9.

8.1 Zadání

Číslo, na kterém bude metoda SIQS prezentována, je $n = 19326223710861634601$. Toto číslo bylo vytvořeno složením dvou prvočísel o 10 dekadických číslicích, *5915587277* a *3267000013*. Zadané číslo n, jenž má 20 dekadických číslic, by díky tomu, že je pro SIQS stále relativně malé, bylo vhodnější řešit nějakou jednoduchou faktorizační metodou, jako je například Pollard ρ metoda (viz kapitola 6.2). Pro demonstraci, jak SIQS pracuje, je naopak toto číslo již poměrně velké, protože jak bude dále ukázáno, je pro toto číslo nutné řešit operace nad maticí, která bude přes 200 řádků a 200 sloupců velká. Kontrola každé provedené operace nad touto maticí je tak již poměrně náročná.

8.2 Příprava SIQS

Nejdříve je nutné nastavit všechny parametry pro SIQS. To znamená nastavit velikost faktorizační báze, která nám určuje, kolika prvočísly se budeme pokoušet dělit kandidáty na B-hladké číslo. Dále musíme nastavit velikost prosévacího intervalu. V tomto příkladu nastavíme velikost faktorizační báze na prvních 450 prvočísel a prosévací interval nastavíme na $\langle -65535; 65535 \rangle$. Je vhodné podotknout, že nastavené hodnoty jak pro faktorizační bázi, tak

pro prosévací interval, jsou pro faktorizaci čísla o 20 dekadických číslicích zbytečně velké. Pro demonstraci, jak v jednoduchosti metoda SIQS funguje, jsou ale tyto hodnoty vhodnější.

Jak bylo zmíněno v kapitole 7.4, některá prvočísla z faktorizační báze nikdy nebudou děliteli B-hladkých čísel. Projdeme tedy všechna prvočísla z faktorizační báze, a pokud pro nějaké prvočíslo nebude Legendreův symbol (viz kap. 2.7) roven 1, pak zadané číslo n není kvadratickým zbytkem pro dané prvočíslo, a tedy toto prvočíslo nikdy nebude dělitelem B-hladkého čísla. Všechna taková prvočísla vyřadíme z faktorizační báze. V tomto příkladu tak bude faktorizační báze redukována na 219 prvočísel. Obecně platí, že v každé zvolené faktorizační bázi se bude nacházet zhruba polovina prvočísel, pro která je zadané číslo n kvadratickým zbytkem.

8.3 Generování polynomu

Před samotným vygenerováním polynomu je nutné nejdříve spočítat ideální hodnotu koeficientu a. Ideální hodnotu spočteme pomocí $a_{ideal} = \sqrt{2n}/M$. Pro náš příklad tak bude a_{ideal} nabývat hodnoty 94865. Platí, že s čím bližší hodnotou k a_{ideal} se nám podaří vygenerovat koeficient a, tím větší budeme mít šanci nalézt B-hladkou hodnotu při prosévání.

8.3.1 Generování koeficientu a

Jak bylo zmíněno výše, při generování koeficientu a se snažíme co nejvíce přiblížit jeho ideální hodnotě a_{ideal}. K tomuto účelu však můžeme využít pouze prvočísla z faktorizační báze. Zároveň je nutné zvolit vhodný počet prvočísel, kterými bude koeficient a dělitelný. Je nutné si uvědomit, že počet dělitelů a určuje, kolik polynomu pro dané a můžeme vygenerovat. Může se tak na první pohled zdát, že čím více dělitelů bude koeficient a mít, tím lépe, protože obměna koeficientů b a c je časově mnohem méně náročnější než generování nového koeficientu a. Pokud však použijeme většího počtu dělitelů, znamená to, že hodnoty všech dělitelů budou poměrně malé. Malí dělitelé však mají za následek, že při prosévání se budou hodnoty vygenerované polynomem pravděpodobně častěji opakovat. To ovšem znamená, že každé nalezené B-hladké číslo bude již pravděpodobně duplicitní s jiným nalezeným B-hladkým číslem. Duplicitní B-hladká čísla pro nás však nemají žádný užitek, je tedy nutné zajistit, aby duplicitní B-hladká čísla byla nalezena jen velmi vzácně. Platí tak, že dělitele volíme větších hodnot, aby byla pravděpodobnost opakování minimální.

Pro určení přesného počtu dělitelů neexistuje obecný výpočet, každý, kdo implementuje metodu SIQS, tak musí experimentálně zkoušet pro každé číslo různý počet dělitelů. Až najde počet, při kterém pracuje SIQS nejefektivněji, je vhodné si tento počet někam uložit, protože tento počet dělitelů pak bude efektivní pro každé číslo stejné velikosti. Pro náš příklad tak volíme počet dělitelů $s = 2$.

Když máme určen počet dělitelů koeficientu a, zbývá zvolit prvočísla z faktorizační báze, které budou děliteli a. Volíme tedy prvočísla 223 a 433, protože $\log(223 * 433)$ je nejblíže $\log(a_{ideal})$ ze všech možných kombinací ($11.48 \approx 14.46$). Je nutné si ale uvědomit, že koeficient a bude při faktorizaci mnohokrát obměněn. Není tak možné použít vždy kombinaci prvočísel, která bude nejblíže ideální hodnotě, ale budeme vybírat z kombinací, které budou přijatelně blízko. V rámci této práce bylo experimentálně zjištěno, že koeficient a je vhodný k použití, pokud je hodnota logaritmu pro toto a odlišná od logaritmu ideální hodnoty maximálně o 0.02. Samozřejmě platí, že bychom měli používat koeficienty a, u nichž je rozdíl od ideální hodnoty co možná nejmenší, a nespoléhat pouze na to, že koeficient a splňuje

podmínku tolerance. Pomocí zvolených prvočísel vytvoříme koeficient a jejich vynásobením. Získáváme tak $a = 96559$.

8.3.2 Získání koeficientů b a c

S vygenerovaným koeficientem a můžeme přejít k výpočtu zbylých koeficientů. Nejprve se počítá koeficient b. Musíme tedy spočíst jednotlivá B_l, tak aby splňovala podmínky uvedené v 6.39. Zde však nastává drobný problém. Pokud se pokusíme splnit první část podmínky, $B_l^2 \equiv N \ (mod\, q_l)$, kdy příslušné B_l hledáme pomocí Tonelli-Shanksova algoritmu (viz kap. 3.4), zjistíme, že získané B_l nesplňuje druhou část podmínky $B_l \equiv 0 \ (mod\, q_j)$. Pokud se naopak pokusíme nejdříve splnit druhou část podmínky, kterou můžeme vyřešit tak, že spočteme $B_l = {}^{a}/_{q_l}$, zjistíme, že v takovém případě nebude splněna první část podmínky. Oba výpočty však můžeme vhodně zkombinovat, a získáme tak požadované B_l. Postup je tento:

1. Řešíme $t^2 \equiv N \ (mod\, q_l)$ pomocí Tonelli-Shanksova algoritmu, tím získáme první řešení t_1
2. Jelikož má ale rovnice dvě řešení, druhé řešení získáme:
$$t_2 = q_l - t_1 \tag{8.1}$$
3. Dále spočteme:
$$a_l = \frac{a}{q_l} \tag{8.2}$$
4. Vyřešíme:
$$a_l * a_l^{-1} \equiv 1 \ (mod\, q_l) \tag{8.3}$$
5. Následně spočteme:
$$\begin{aligned} \gamma_1 &= t_1 * a_l^{-1} \ (mod\, q_l) \\ \gamma_2 &= t_2 * a_l^{-1} \ (mod\, q_l) \end{aligned} \tag{8.4}$$
6. Pokud $\gamma_1 < \gamma_2$ pak:
$$B_l = a_l * \gamma_1 \tag{8.5}$$
7. Jinak:
$$B_l = a_l * \gamma_2 \tag{8.6}$$

Z předvedeného postupu můžeme vysledovat, že spočtená inverzní hodnota a_l^{-1} zde slouží jako „propojovací“ prvek, a zajistí nám tak splnění obou částí podmínky. Tento postup aplikujeme pro všechna B_l, kde $l = 1 \ldots s$. V našem příkladu tak bude vypadat výpočet B_1 takto:

$$\begin{aligned} t_1 &= 199 \\ t_2 &= 24 \\ a_l &= 433 \\ a_l^{-1} &= 120 \\ \gamma_1 &= 19 \\ \gamma_2 &= 204 \\ B_1 &= 8227 \end{aligned} \tag{8.7}$$

Výsledná hodnota pro B_2 pak je 47053.

První hodnotu koeficientu získáme jako $b = \sum_{l=1}^{s} B_l$. Počet všech možných hodnot b pro daný koeficient a je 2^{s-1}. V našem případě tak pro dané a máme k dispozici 2 hodnoty b, kdy první hodnota bude $b = 55280$. Obecně, když je třeba vygenerovat nový polynom, pak další hodnotu b získáme pomocí Greyova kódu (viz 6.40). Koeficient c pak jednoduše spočteme pomocí $n = b^2 - ac$, v našem příkladě tak $c = -200149377145639$.

8.4 Fáze 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.

Na začátku prosévání novým polynomem je nejprve nutné si spočítat kořeny jednotlivých prvočísel. Kořeny nám určí, která hodnota z intervalu $\langle -M; M\rangle$ bude dělitelná daným prvočíslem p. Jednotlivé kořeny získáme výpočtem vztahu $x \equiv \pm t - b$, kde $t^2 \equiv N \ (mod\, p)$, a tedy t opět počítáme pomocí Tonelli-Shanksova algoritmu. Platí, že je-li pro danou hodnotu z intervalu polynom dělitelný daným prvočíslem p, $Q_{a,b}(x) \equiv 0 \ (mod\, p)$, pak je polynom dělitelný tímto prvočíslem p i pro $x + p$, $x + 2p$, $x + 3p$, ... Jelikož nepracujeme pouze v intervalu $\langle 0; M\rangle$, ale v intervalu $\langle -M; M\rangle$, pak musíme spočítat i kořeny pro záporné hodnoty. Ty získáme výpočtem $x_{neg} = x - p$. V našem případě tak například pro prvočíslo $p = 11$ budou kořeny pro kladnou část intervalu $x_1 = 7$ a $x_2 = 5$. Pro zápornou část intervalu pak budou kořeny $x_{neg1} = -4$ a $x_{neg2} = -6$. Výjimkou je prvočíslo 2, které má jen jeden kořen pro kladnou část intervalu, a tedy i jen jeden kořen pro zápornou část intervalu. Pro prvočísla, jež jsou děliteli koeficientu a se v této práci kořeny nepočítají a přes tato prvočísla se ani neprosívá. Prosévání přes tato prvočísla je ovšem také možné pouze za splnění určitých podmínek. Tyto podmínky jsou blíže popsány v článku [4].

Je vhodné zmínit, že jednotlivé fáze prosévání, tedy výpočet kořenů, hledání kandidátů a ověření kandidátů, se mohou navzájem prolínat vzhledem k optimalizaci rychlosti metody SIQS. V rámci této práce ale budou jednotlivé fáze, pro lepší demonstraci a lepší pochopení metody, probíhat v daném pořadí.

8.4.1 Hledání kandidátů B-hladkých čísel

Hledání kandidátů probíhá tak, že procházíme zvolený interval $\langle -M; M\rangle$, a pokud je polynom pro danou hodnotu z intervalu dělitelný prvočíslem p, což zjistíme ze spočtených kořenů, pak pro danou hodnotu přičteme do nějakého čítače hodnotu $\log(p)$. Až projdeme celý interval a všechny kořeny, začneme hledat kandidáty na B-hladké číslo. Kandidátem je určena každá hodnota polynomu, pro niž čítač přesáhl hodnotu $\log(2x\sqrt{n})$, kde $x \in \langle -M; M\rangle$. V našem příkladu bylo nalezeno 2085 kandidátů pro kladnou část intervalu a 2007 pro zápornou část intervalu. Počet kandidátů se může lišit od implementace. V této práci, jak bylo zmíněno, se neprosévá přes dělitele koeficientu a, a tak se u $\log(2x\sqrt{n})$ počítá s jistou chybou, nelze tak brát demonstrované počty kandidátů brát jako závazné. Obecně ale je vhodné hledat kandidáty co nejpřesněji, protože čím přesněji kandidáty vybíráme, tím rychlejší bude jejich následné ověření, a tedy i samotná faktorizace bude rychlejší.

V rámci této práce je hledání kandidátů provedeno tak, že se prochází faktorizační báze po prvočíslech a každý kořen příslušného prvočísla p je řešen zvlášť. Nejdříve se vezme první

kořen pro kladnou část intervalu a čítač pro příslušnou hodnotu intervalu, která je rovna kořenu, je navýšena o $\log(p)$. Hodnota kořene je navýšena o hodnotu prvočísla a příslušný čítač je opět navýšen. Tento proces je prováděn tak dlouho, dokud hodnota kořene nepřesáhne horní mez intervalu, v tomto příkladu tedy hodnotu 65536. Stejný proces je proveden i pro druhý kořen kladné části intervalu a obdobně pak pro oba kořeny záporné části intervalu. Pro demonstraci pro prvočíslo 11 tak nejprve procházíme kladnou část intervalu, kde kořen $x_1 = 7$. Na pozici 7 je tedy zvýšen čítač o $\log(11)$. Čítač pak navýšíme také o $\log(11)$ na pozicích 18, 29, ..., 65534.

8.4.2 Ověření kandidátů

Ověření, že kandidát je opravdu B-hladké číslo, provádíme tak, že spočteme hodnotu polynomu pro daného kandidáta a následně tuto hodnotu podělíme všemi prvočísly z faktorizační báze. Hodnotu polynomu spočítáme dle 6.36. Při dělení je nutné brát do úvahy, že číslo by mohlo být složeno z mocnin prvočísel. Pokud je výsledkem po dělení všemi prvočísly 1, nalezli jsme B-hladké číslo. Pro nalezené B-hladké číslo si uložíme jeho relaci, tedy vektor exponentů v prostoru modulo 2, a přejdeme na ověření dalšího kandidáta.

Ve skutečnosti není nutné počítat hodnotu dle 6.36, ale stačí spočíst hodnotu pomocí 6.35. To je možné, protože je-li hodnota dle 6.35 B-hladkým číslem, pak je hodnota dle 6.36 také B-hladká vzhledem k tomu, že koeficient a je složen z prvočísel z faktorizační báze. Dojde tak k ušetření několika aritmetických operací.

V našem příkladě bylo nalezeno 633 relací. Oproti počtu kandidátů, u kterého jsme řekli, že se může lišit, tak počet nalezených relací by se neměl příliš lišit i při úplně odlišné implementaci metody. Lišit se může maximálně v jednotkách, a to za předpokladu, že daná implementace prosévá i přes dělitele koeficientu a.

Prvním nalezeným kandidátem je $x = 2$. Spočtená hodnota pro tohoto kandidáta je -200149376538283. Toto číslo se rozloží na $-1 * 43 * 53 * 79^2 * 14071997$. Nejvyšší prvočíslo ve faktorizační bázi však je 3169, a tak toto číslo nemůže být považováno za B-hladké. První B-hladké číslo bylo nalezeno pro $x = 247$. Spočtená hodnota je -200143458869288 a je rozložitelná na $-1 * 2^3 * 11 * 23 * 47 * 683 * 1019 * 3023$. Výsledkem po dělení všemi prvočísly z faktorizační báze tak získáme -1. Kvůli těmto případům se do faktorizační báze ke všem prvočíslu přidává ještě -1, která bude fiktivním prvočíslem. Pro toto číslo tedy vytvoříme vektor exponentů:

$$\begin{array}{l} 10100100010000000000000000000000000000000000000000 \;\backslash\backslash \\ 00000000000001000000000000000000000000100000000000 \;\backslash\backslash \\ 00000000000000000000000000000000000000000000000000 \;\backslash\backslash \\ 00000000000000000000000000000000000000000000000000 \;\backslash\backslash \\ 00000000000100000001 \;\backslash\backslash \end{array} \tag{8.8}$$

kde poslední 1 ve vektoru exponentů zastupuje -1. Z vektoru exponentů lze vypozorovat, že i když hledáme B-hladká čísla pomocí 6.35, je nutné pak do vektoru exponentů uložit i všechny dělitele koeficientu a. Pokud bychom tak neudělali, nebylo by možné určit, ke kterému koeficientu a daná relace přísluší, což by vedlo k selhání výpočtu ve fázi lineární algebry.

8.5 Předzpracování matice

Aby bylo možné hledat kombinaci relací takových, že jejich součtem dostaneme nulový vektor, potřebujeme zajistit, aby ve vytvořené matici zcela jistě byla lineární závislost. Toho dosáhneme tak, že nasbíráme o jednu relaci více, než je počet prvočísel ve faktorizační bázi. To znamená, že vytvořená matice bude mít o jeden řádek více, než je počet sloupců. V našem příkladě obsahuje faktorizační báze 219 prvočísel, takže počet relací, který potřebujeme posbírat, je 220. Ve skutečnosti jsme ale do faktorizační báze zavedli i fiktivní prvočíslo -1, a tak počet relací, které je nutné nasbírat je 221. Jelikož se nám ale podařilo jedním polynomem nasbírat 633 relací, je počet nasbíraných relací více než dostatek.

Před samotným hledáním lineární závislosti je však nutné ještě relace projít a vyřadit ty, které by mohli vést k selhání ve fázi lineární algebry. Jedny z takových relací jsou duplikanty, tedy relace, které se shodují s nějakou dříve uloženou relací. Duplikant vytvoří se stejnou dříve uloženou relací nulový vektor, tento nulový vektor by však vedl k získání triviálního dělitele, který však pro nás není zajímavý. Další relace, které je nutné odstranit, jsou nulové vektory. Taková relace může být samotná řešením a vést k nalezení dělitele zadaného čísla n, před jejím odstraněním je tak vhodné ověřit, zda tomu tak není. Poslední typ relací, které se snažíme odstranit, jsou tzv. singletony. Singletony jsou relace obsahující prvočíslo, které není obsaženo v žádné jiné relaci. Tyto relace přímo nevedou k selhání metody při hledání lineární závislosti, ale jelikož neexistuje relace, která by obsahovala dané prvočíslo, tak relace nikdy nemůže byt součástí kombinace relací, které by tvořily nulový vektor. Odstraněním singletonů tak zvyšujeme pravděpodobnost na nalezení dělitele zadaného čísla n.

V našem příkladě tak lze nalézt 3 singletony, $r_{s1} = 126$, $r_{s2} = 547$ a $r_{s3} = 629$. Opět platí, že pozice singletonů se budou v různých implementacích lišit pouze o jednotky a to jen pokud bude zavedeno prosévání přes dělitele koeficientu a. I přes to, kolik bylo nalezeno relací, nebyl nalezen žádný duplikant a nulový vektor. Celkový počet relací, které budou použity ve fázi lineární algebry, tak je 630.

8.6 Fáze lineární algebry

Výše bylo řečeno, že pro fázi lineární algebry potřebujeme nasbírat minimálně 221 relací. Fakt, že jsme jich ve skutečnosti nasbírali více vůbec nevadí, naopak toho můžeme využít. Vždy když je dokončena fáze prosévání a předzpracování matice, sestavíme matici ze všech relací, které nám zůstaly k dispozici. Samozřejmě počet zbylých relací musí být po fázi předzpracování matice stále alespoň o 1 větší než je počet prvočísel faktorizační báze včetně čísla -1. Větší počet relací než je potřebný počet, nám však poskytuje větší pravděpodobnost nalezení nulového vektoru, respektive získáme více variant, jak dosáhnout nulového vektoru. Každá varianta by nám měla poskytnout 50% pravděpodobnost na získání netriviálního dělitele. Všech možných variant bude obvykle více, a tak je vhodné omezit počet zkoumaných variant na určitý počet, například 10, protože v takovém případě je už pravděpodobnost, že bychom vždy nalezli pouze triviálního dělitele zadaného čísla n, minimální.

Metod pro nalezení lineární závislosti je větší množství a každá metoda se poměrně dost odlišuje od metod ostatních. I když se v našem příkladě jedná o poměrně malé faktorizované číslo, matice již dosahuje větších rozměrů. Z těchto důvodu není možné prezentovat konkrétní čísla. Je vhodné upozornit, že fáze lineární algebry je částí SIQS, která je velmi neprůhledná a dost citlivá jak na vstupní data, tak na samotnou správnost implementace. Vzhledem k tomu, že v této fázi se bude pracovat s opravdu velkými maticemi, je manuální

kontrola velmi obtížná, a tak je nutné věnovat dostatek pozornosti při implementaci této části a provádět patřičné testy, aby byly všechny vzniklé chyby odstraněny co nejdříve.

8.7 Spočtení dělitele zadaného čísla

Pro každou variantu vedoucí k získaní nulového vektoru následně ověřujeme, zda platí $GCD\,(X - Y,\, n)$. Hodnotu X získáme takto:

$$X \quad = \quad (\sum (ax - b))\; mod\, n \tag{8.9}$$

a Y pak jako:

$$Y \quad = \quad \sqrt{\sum (ax - b)^2 - n}\; mod\, n \tag{8.10}$$

V našem příkladě varianta, která vedla k nalezení netriviálního dělitele, obsahovala 64 vektorů. Není tak možné zde vypsat celý výpočet hodnot X a Y. Na přiloženém DVD k této práci je však možné na uvedeném místě nalézt soubor s názvem *Example/result.txt*, ve kterém je daný výpočet proveden, a tak čtenář může na výpočet v případě zájmu nahlédnout.

8.8 Dodatek k příkladu

Na příkladu byl ukázán základní princip fungování metody SIQS. I přes to, že byly projity všechny fáze SIQS, nebylo pokryto ze SIQS vše. Jednou z věcí, která nebyla blíže probrána, je výměna polynomu. Jak bylo ukázáno, tak v tomto příkladě výměna polynomu nebyla nutná. Příklad, na kterém by mohla být demonstrována SIQS i s výměnou polynomu, by zahrnoval příliš velká čísla, a tak by metoda stejně nemohla být předvedena v plném rozsahu. Aby ale čtenář nebyl ochuzen, případně by si potřeboval zkontrolovat funkčnost své implementace i pro větší čísla, byly na přiložené DVD této práce uloženy i soubory (*Example/60dec)*, které obsahují všechny potřebné hodnoty získané při faktorizaci čísla o 40 dekadických číslicích.

Metoda SIQS zde byla předvedena v základní formě. Je však vhodné tuto metodu vylepšit o Knuth-Schroeppelův algoritmus, který vypočítává nejvhodnější koeficient, s nímž je následně zadané číslo n vynásobeno. Výsledkem je pak efektivnější faktorizace, protože do faktorizační báze jsou zahrnuty i malá prvočísla. Dále by metoda SIQS měla používat mechanizmus Large Prime Variation (viz kap. 6.6.3). Nutná úprava SIQS je v tomto případě minimální, ale naopak má velký dopad na rychlost faktorizace. Díky této úpravě jsme nuceni nasbírat jen o něco málo více než polovinu relací oproti původnímu počtu.

Kapitola 9

Implementace SIQS

9.1 Zadání čísla k faktorizaci

Číslo N, které má být faktorizováno, je předáno algoritmu přes jediný parametr, který algoritmus přijímá. Algoritmus nejdříve zjistí, zda zadané číslo není prvočíslem. V takovém případě zadané číslo nemá netriviálního dělitele, a výsledkem by tak bylo prvočíslo samotné. Nejedná-li se o prvočíslo, pak je zkoumáno, zda je zadané číslo dělitelné prvními 50 prvočísly. Pokud ano, je vypsáno řešení a algoritmus ukončen. V opačném případě algoritmus zjistí velikost tohoto čísla v dekadických číslicích. Pokud zadané číslo má 30 dekadických číslic či méně, pak je toto číslo faktorizováno Pollard ρ metodou (viz kap. 6.2), která je pro faktorizování čísel o této velikosti vhodnější. Je-li zadané číslo naopak větší, přistoupí se k použití metody SIQS.

9.2 Příprava SIQS

U každého zadaného čísla n je nejprve zjištěna jeho délka v bitech. Na základě této informace je vybráno nejvhodnější nastavení SIQS. Do nastavení je zařazena maximální velikost faktorizační báze F, velikost prosévacího intervalu M, kolik prvočísel budeme brát v potaz při uplatňování Large Prime Variation, a optimální počet dělitelů koeficientu a.

Následně se přechází k načítání prvočísel do faktorizační báze. U každého prvočísla se kontroluje jeho hodnota Legendreova symbolu (viz kap. 2.7). Je-li hodnota rovna 0, pak jsme přímo našli dělitele zadaného čísla, a tak stačí pouze vypsat výsledek a algoritmus je ukončen. Je-li hodnota rovna 1, pak toto prvočíslo ukládáme do faktorizační báze, jež je ve formě datového typu *vector*, a bude dále sloužit k identifikaci B-hladkých čísel. V opačném případě prvočíslo do faktorizační báze neukládáme. V obou případech je ale inkrementován čítač již prozkoumaných prvočísel. Dokud čítač nedosáhne hodnoty F, je proces opakován. Dokončením tohoto procesu získáme faktorizační bázi, kterou budeme dále označovat B. Obdobným způsobem načteme i prvočísla pro Large Prime Variation. Ta ale budou ukládána do datového typu *map*, aby se urychlilo hledání správného prvočísla ve fázi ověřování kandidátů.

Dalším krokem je výpočet logaritmů pro jednotlivá prvočísla z faktorizační báze B. Tyto logaritmy budou použity například při hledání dělitelů koeficientu a. Díky uložení těchto logaritmů nebude nutné je počítat pokaždé, když budou potřeba, a tak dojde k urychlení výpočtu.

Předposledním krokem v přípravě SIQS je výpočet ideální hodnoty koeficientu a, a_{ideal}. Ten provedeme dle vzorce $a_{ideal} = \sqrt{2n}/M$. Získaná hodnota s maximální definovanou odchylkou nám pak bude sloužit ke generování koeficientu a.

Generování koeficientu a je provedeno principem, který byl představen v tomto článku [2]. Je tak nutné rozdělit faktorizační bázi na dvě části. Do které části prvočíslo z faktorizační báze patří, určíme dle jeho indexu. Všechna prvočísla se sudým indexem budou v první části, s lichým pak v části druhé. Z první části vytvoříme všechny trojice. Jednotlivé trojice jsou uloženy v binárním vyhledávacím stromu, kde klíčem je suma logaritmů prvočísel tvořící danou trojici. Druhá část prvočísel je uložena do datového typu *vector*. Jak jsou tyto dvě části použity je vysvětleno v kapitole 9.3. Na dvě části však nerozdělujeme celou faktorizační bázi, ale pouze námi určenou část této faktorizační báze. Je tak spočtena ideální hodnota dělitele koeficientu a. Tuto hodnotu spočteme pomocí vzorce $d_{ideal} = \sqrt[s]{a_{ideal}}$. K této hodnotě nalezneme index i nejbližšího prvočísla z faktorizační báze. Interval $\langle i - 10; i + 200\rangle$ pak označuje část faktorizační báze, která bude dále rozdělena na zmíněné dvě menší části. Je použito pouze 10 prvočísel menších než prvočíslo na indexu i, protože použitá prvočísla by neměla být příliš malá. Došlo by totiž ke generování polynomů, jejichž hodnoty by se pravděpodobně častěji opakovaly, a docházelo by tak k nalezení duplicitních hodnot, což není žádoucí. Pan Contini uvádí ve svém článku [4], že prvočísla, která použili ke generování koeficientu a, nebyla nikdy menší než 2000. Díky použitému principu v této práci je možné, že jedním z dělitelů koeficientu a bude i prvočíslo menší než 2000, polynomy však budou generovány tak, aby pravděpodobnost duplicitních hodnot byla minimální.

Nevýhodou tohoto řešení je jeho poměrně větší časová náročnost. Čím větší číslo ale faktorizujeme, tím zanedbatelnější je čas strávený při této inicializaci.

9.3 Generování polynomu

Fáze generování polynomu má velký vliv na samotnou faktorizaci. Je vhodné tak vybrat postup generování jednotlivých koeficientů polynomu, aby byla pravděpodobnost nalezení B-hladké hodnoty co možná nejvyšší. Hlavní důraz je kladen na co nejpřesnější generování hodnot koeficientu a vzhledem k jeho ideální hodnotě a_{ideal}. Experimentálně bylo zjištěno, že polynom je možné považovat za kvalitní, pokud rozdíl logaritmu koeficientu a a logaritmu jeho ideální hodnoty a_{ideal} není větší než 0.02. V přiložených souborech na DVD, které slouží k demonstraci faktorizace čísla o 40 dekadických číslicích, je ukázáno, že implementace vytvořená v rámci této práce je schopna generovat koeficienty a s řádově větší přesností.

9.3.1 Generování koeficientu a

Jako první je nutné spočítat koeficient a generujícího polynomu, tak jak byl představen v kapitole 6.8. Koeficient a je u SIQS získán jako produkt vybraných prvočísel z faktorizační báze. Prvočísla, ze kterých bude koeficient a složen, nelze však vybírat náhodně či dle libosti. Je nutné, aby a bylo co možná nejblíže hodnotě ideálního a, kterou jsme stanovili jako $a_{ideal} = \sqrt{2n}/M$. Vytvořili jsme si tedy dvě části z faktorizační báze způsobem popsaným v kapitole 9.2. Část, která je uložena ve *vector*u, bude sloužit k výběru dělitelů koeficientu a algoritmem NEXKSB [13]. Algoritmus na základě lexikografického seřazení prvočísel vybírá vždy následující k-tici z celkového počtu prvočísel, který pro tento algoritmus použijeme. V tomto případě $k = s - 3$. Zbylá 3 prvočísla vybíráme ze všech trojic, které jsme vytvořili pomocí druhé části faktorizační báze.

Postup je takový, že při potřebě vygenerování nového koeficientu a je nejdříve použit algoritmus NEXKSB, kterým získáme $s-3$ dělitelů koeficientu a. Spočteme logaritmus pro hodnotu a_{ideal} a spočteme sumu logaritmů získaných dělitelů. Rozdíl těchto hodnot slouží k nalezení zbylých 3 dělitelů v binárním vyhledávacím stromu, který obsahuje všechny trojice prvočísel z první části faktorizační báze. Jak bylo řečeno výše, klíčem každého záznamu ve stromu je suma logaritmů prvočísel v daném záznamu, a tak se snažíme najít trojici, jejíž logaritmus je nejblíže spočtenému rozdílu.

Díky tomuto postupu si zajistíme, že dva vygenerované polynomy nikdy nebudou mít stejný koeficient a. Je sice možné, že pro dva koeficienty a bude vybrána stejná trojice z binárního vyhledávacího stromu, ale díky algoritmu NEXKSB nikdy nezískáme stejnou k-tici. Daný postup je tak vhodný pro paralelní použití, protože každé vlákno bude mít vždy jiný koeficient a, se kterým bude pracovat. Také díky tomu, že se poslední 3 dělitelé koeficientu a vybírají tak, aby se logaritmus a blížil co nejvíce logaritmu a_{ideal}, je zaručeno, že se budou generovat polynomy, u kterých bude vysoká pravděpodobnost nalezení B-hladkého čísla.

9.3.2 Generování koeficientu b a c

Následně je nutné spočíst koeficient b. Jak víme z kap. 6.8, pro každé a jsme schopni nalézt 2^{s-1} koeficientů b, a tedy vytvořit 2^{s-1} různých polynomů. Jednotlivé koeficienty b můžeme hledat pouze pomocí Tonelliho-Shanksova algoritmu (viz kap. 3.4). To by ale bylo příliš zdlouhavé, a tedy i neefektivní. Místo toho hledáme taková B_l, která splňují vztah uvedený v 6.39. Nalézt takové B_l, aby splňovalo $B_l \equiv N \ (mod\, q_l)$, je poměrně jednoduché, protože můžeme využít Tonelli-Shanksova algoritmu. Takto nalezené B_l však velice pravděpodobně nebude splňovat podmínku $B_l \equiv 0 \ (mod\, q_j)\ pro\ 1 \leq j \leq s,\ j \neq l$. To lze vyřešit tak, že B_l vynásobíme a_l, kde a_l je produktem všech q_j. Hodnotu B_l dále vynásobíme inverzním prvkem k a_l, a_l^{-1}, v rámci modulo q_l. Jelikož víme, že $a_l a_l^{-1} = 1\,(mod\, q_l)$ z 2.24, zajistíme si, že dané B_l bude dělitelné kterýmkoli prvočíslem, jež je dělitelem a a je různé od q_l, zároveň ale zůstane zachována vlastnost, že druhá mocnina B_l je kongruentní k N modulo q_l, a tedy jsou splněny obě podmínky pro B_l. Označíme-li hodnotu nalezenou Tonelli-Shanksovým algoritmem jako t_l, pak úplný výpočet B_l vypadá následovně:

$$B_l \quad = \quad t_l * a_l * a_l^{-1} \tag{9.1}$$

Součtem všech B_l získáme první koeficient b, který bude splňovat stanovenou podmínku $b^2 \equiv N \ (mod\, a)$. Další b nalezneme pomocí Greyova kódu (viz kap. 3.1) vztahem 6.40. Pokud jsou známy koeficienty a a b, spočtení koeficientu c je pak triviální záležitostí.

9.3.3 Zavedený paralelizmus

Paralelizmus v této implementaci vzniká požadavkem na první vygenerování polynomu. Paralelizmus je řešen pomocí OpenMP a počet vláken není nijak omezen. To má za následek, že každé vlákno pracuje se svým polynomem nezávisle na ostatních a je zpracováváno najednou tolik polynomů, kolik máme vláken k dispozici. Díky použití principu generování koeficientu a s algoritmem NEXKSB máme zajištěno, že každé vlákno bude mít svůj polynom vždy odlišný od ostatních. Aby to však byla úplná pravda, algoritmus NEXKSB musel

být vložen do kritické sekce, jedině tak lze originalitu polynomu zcela zaručit. Jelikož koeficienty b a c jsou zcela závislé na koeficientu a, bude mít každé vlákno tyto koeficienty také odlišné a není třeba je sdílet, komunikace tak mezi vlákny bude minimální.

9.4 Fáze prosévání

Prosévání je časově nejnáročnější částí algoritmu, kdy se snažíme nalézt potřebný počet relací k pozdějšímu hledání nulového vektoru.

9.4.1 Výpočet kořenů

Prvním krokem ve fázi prosévání je výpočet kořenů pro každé prvočíslo (viz kap 8.4 a 7.4). Kořeny se ukládají pouze pro kladnou část intervalu $\langle -M; M\rangle$, pro zápornou část jsou v případě potřeby dopočteny. Pokud je vygenerován nový polynom, kořeny tak budou jiné, a je tedy nutné je znovu spočíst. Kořeny jsou ukládány do struktury společně s příslušným prvočíslem, aby bylo jednoznačné, ke kterému prvočíslu dané kořeny náleží.

9.4.2 Hledání kandidátů B-hladkých čísel

Prosévání se provádí přes celou faktorizační bázi. Pro každé prvočíslo z faktorizační báze se načtou jeho kořeny. Pro každý kořen daného prvočísla p se inicializuje smyčka *for* na hodnotu kořene, nastaví se krok o velikosti daného prvočísla a iteruje se, dokud hodnota čítače nepřekročí horní mez intervalu $\langle 0; M\rangle$ resp. $\langle -M; 0\rangle$ pro záporné kořeny. Hodnota čítače nám v každé iteraci jednoznačně určuje, pro kterou hodnotu $x \in \langle 0; M\rangle$ resp. $x \in \langle -M; 0\rangle$ bude polynom dělitelný daným prvočíslem. Do vytvořeného *pole* na příslušnou pozici x přičteme $\log(p)$ a do *mapy* uložíme na pozici x prvočíslo p jakožto dělitele hodnoty polynomu pro dané x. Po projití všech prvočísel se prochází pole spočtených hodnot pro každé x. Pokud je v poli pro dané x uložená hodnota větší než $\log(2x\sqrt{n})$, pak jsme nalezli kandidáta, který bude blíže prozkoumán.

9.4.3 Ověření kandidátů

U všech nalezených kandidátů následně postupujeme tak, že nejdříve vypočteme hodnotu polynomu pro dané x. Ve skutečnosti nemusíme počítat hodnotu polynomu $a*(ax^2+2bx+c)$, ale stačí nám, když vypočteme $ax^2 + 2bx + c$, protože a je složeno z prvočísel z faktorizační báze, a tyto dělitele tedy známe. Spočtenou hodnotu vydělíme všemi prvočísly, které máme uložené v mapě na příslušné pozici. Jelikož spočtená hodnota může být dělitelná daným prvočíslem několikrát, dělí se každým prvočíslem tak dlouho, dokud vychází zbytek po dělení daným prvočíslem 0. Pokud je výsledkem po dělení všemi uloženými prvočísly 1, pak jsme nalezli B-hladké číslo. V takovém případě je vytvořen vektor exponentů v rámci modulo 2, kde 1 bude ve vektoru zastupovat všechny liché mocniny dělitelů a 0 bude zastupovat všechny sudé mocniny dělitelů nebo prvočísla, jež se dělení neúčastnila. Vektor exponentů se ukládá do *vector<bool>*, kde jednotlivé *bool* hodnoty zastupují jedno prvočíslo.

V mnoha případech nastane situace, kdy spočtením hodnoty dle výše zmíněného vzorce nám vyjde záporná hodnota. Z tohoto důvodu zavádíme do faktorizační báze i číslo -1, které slouží jako fiktivní prvočíslo a pomáhá nám tuto situaci řešit.

Do každého vektoru exponentů je však nutné ještě zařadit prvočísla, která jsou děliteli koeficientu a. Pokud by se na tento fakt zapomnělo, pak by každý vektor exponentů byl po-

važován za vytvořený ze stejného polynomu, což by vedlo k nárůstu duplikantů, ale hlavně při faktorizaci, kde by došlo k výměně koeficientu a, by taková chyba vedla k selhání ve fázi lineární algebry. Naopak tato chyba se nijak neprojeví, pokud nedojde při faktorizaci k výměně koeficientu a, což může být například v raných fázích implementace, kdy se snažíme ověřit, zda metoda SIQS funguje jako celek na číslech do 30 dekadických číslic. U této chyby je problém, že se projeví až ve fázi lineární algebry, kde sice budou nalezeny nulové vektory, ale téměř všechny povedou nalezení triviálních dělitelů a nalezení netriviálního dělitele je tak spíše „otázkou štěstí“. Jelikož se ve fázi lineární algebry pracuje s poměrně velkými maticemi, je hledání této chyby velice náročné, a navíc chyba nevzniká ve fázi lineární algebry, ale již ve fázi prosévání, kdy jsou špatně vytvářeny vektory exponentů. Ušetření jednoho násobení koeficientem a nám tak sice urychlí výpočet, ale nesmí být zapomenuto na zahrnutí dělitelů tohoto koeficientu do vektoru exponentů, protože následné hledání chyby může být velice obtížné.

Když máme vytvořen exponent vektorů, pak tento vektor uložíme do příslušné struktury. Do této struktury dále uložíme koeficienty a, b, c a proměnnou x. Naplněnou strukturu pak uložíme do *vektoru*. Vytvořený vektor exponentů budu v práci dále nazývat relací či úplnou relací.

Pokud výsledkem po dělení všemi prvočísly nebyla 1, pak je nutné ověřit, zda se nejedná o prvočíslo, které je sice tak velké, že jej nemáme ve faktorizační bázi, ale je v našem zvoleném intervalu větších prvočísel, se kterými budeme uplatňovat rozšíření Single Large Prime Variation (viz kap. 6.6.3). Pokud se jedná opravdu o takové prvočíslo, pak je následný postup velmi podobný tomu, jaký byl použit u úplných relací. Vytvoříme vektor exponentů v rámci modulo 2 a tento vektor uložíme do *multimapy*, kde klíčem bude právě výsledné prvočíslo. K vektoru exponentů jsou dále uloženy koeficienty a, b, c a proměnná x. Vytvořený vektor exponentů budu v tomto případě dále nazývat relací, pokud nebude nutné rozlišovat vektor exponentů pro velké prvočíslo a vektor exponentů pro B-hladké číslo, nebo částečnou relací, pokud bude nutné vektory exponentů rozlišovat.

Uložením všech kandidátů končí prosévání nad daným polynomem a polynom je považován za vyčerpaný. Pokud bylo nasbíráno méně relací než je potřeba, je vygenerován nový polynom a proces se opakuje. V opačném případě se přechází do fáze vytváření matice a následně do fáze lineární algebry. V této práci, díky použití Single Large Prime Variation, se sbírá pouze o 5% více relací než je polovina potřebného počtu. Zbytek potřebných úplných relací bude složeno z částečných relací. Doba faktorizace je tak značně urychlena. Může však nastat situace, kdy i přes složení úplných relací z těch částečných, nebude nasbírán dostatečný počet relací. V takovém případě je nutné vygenerovat nové polynomy a zbylé relace dosbírat. Díky použití algoritmu NEXKSB je zajištěno, že v případě nutnosti obnovy procesu prosévání, si každé vlákno vygeneruje jiný polynom, se kterým bude následně pracovat, a tak je obnova prosévání triviální.

9.5 Zpracování relací

Když máme nasbírán dostatek relací, je nutné z těchto relací vytvořit matici velikosti $n * m$, kde $n = m + 1$ a $m = B$. Ještě předtím, než je ale vytvořena matice, je nutné složit částečné relace do úplných relací. Jelikož byly částečné relace uloženy do multimapy, kde klíčem je velké prvočíslo, je hledání částečných relací, které lze společně sloučit, značně ulehčeno. Na začátku jsou vytvořeny dva iterátory nad touto multimapou, kdy jeden iterátor ukazuje

na první prvek multimapy a druhý ukazuje na následující prvek. Pokud se velká prvočísla shodují, pak jsou částečné relace společně XORovány, čímž získáme úplnou relaci. Tuto relaci uložíme mezi úplné relace a uložíme také příslušné koeficienty a proměnnou jednotlivých částečných relací. Příslušné částečné relace si poznačíme za použité a iterátory nastavíme na první dva prvky za těmi aktuálně zpracovanými. Pokud se velká prvočísla neshodují, jsou iterátory posunuty na následující prvek. Když je celá multimapa zpracována, měli bychom mít nasbíráno minimálně $B + 1$ úplných relací, které jsou nutné pro další práci v rámci metody SIQS. Pokud se nepodařilo nasbírat dostatečný počet relací, je nutné se vrátit do fáze prosévání a relace dosbírat. Ještě předtím jsou ale odstraněny všechny použité částečné relace. Pokud by tak nebylo učiněno, byly by po následném prosetí tyto částečné relace opět použity, což by vedlo k vytvoření duplikantů, a to je nežádoucí.

Po získání všech úplných relací z relací částečných se prochází všechny úplné relace a hledají se takové, které by mohly způsobit selhání ve fázi lineární algebry. Nejdříve se hledají nulové vektory, kde se prochází po jednotlivých prvcích relace, a pokud po projití všech prvků není nalezena ani jedna 1, pak se jedná o nulový vektor exponentů, a tím pádem je tento vektor odstraněn. Následně se hledají duplikanty. Zde se postupuje tak, že se aktuální relace porovnává se všemi následujícími (s předchozími již musela být porovnána) a pokud se nalezne nějaká relace, která je shodná s aktuální, pak je index této relace uložen do *mapy*, kde je zároveň i klíčem, a tato relace bude později odstraněna. Po projití všech relací je tak získán seznam duplicitních relací. Prvky v mapě jsou řazeny v sestupném pořadí, a tak při odstraňování duplicitních relací není nutné přepočítávat index odstraňované relace.

Poslední relace, které je vhodné odstranit, jsou takzvané singletony. Singletony jsou relace, které obsahují prvočíslo, jež se nenachází v žádné jiné relaci. Odstraněním singletonů zvyšujeme pravděpodobnost na nalezení netriviálního dělitele.

Při odstraňovaní nulových vektorů, duplicitních relací a singletonů může opět dojít k situaci, kdy budeme mít nedostatek potřebných relací. Pokud tato situace nastala, duplikanty a nulové vektory zůstanou odstraněny. Naopak singletony jsou vráceny zpět mezi úplné relace. To je provedeno z důvodu, že při opakování prosévací fáze je pravděpodobné, že nalezneme relace, které budou sdílet jedinečné prvočíslo s nějakým singletonem, a tak získáme relaci navíc resp. nebudeme ji muset odstraňovat.

Pokud jsou všechny nežádoucí relace odstraněny a je nasbírán dostatek relací, pak se z těchto relací vytvoří matice. Matice je reprezentována polem polí typu *bool*, která je řádkově orientována. Bylo-li na sbíráno více relací, než je $B + 1$, je to absolutně v pořádku a použijí se všechny nasbírané relace. Díky tomu bude nalezeno více nulových vektorů, a tak pravděpodobnost na získání netriviálního dělitele bude vyšší.

9.6 Fáze lineární algebry

Ve fázi lineární algebry je vytvořená matice zpracována pomocí algoritmu Gaussovy eliminační metody popsaném v článku [9]. Princip je takový, že se matice prochází po sloupcích. Každý sloupec se postupně prochází po řádcích, a pokud na nějakém řádku je nalezena 1, pak se jedná o tzv. pivota. Řádek pivota se opět prochází po sloupcích a hledají se sloupce, ve kterých se nachází 1. Pokud je takový sloupec nalezen, je k tomuto sloupci XORován sloupec s pivotem. Po projití všech sloupců pivota je daný řádek označen jako použitý. Kvůli tomu je nutné matici ještě před spuštěním algoritmu o jeden sloupec rozšířit. Může se stát,

že při hledání pivota v daném sloupci nebude nalezen žádný řádek, který by obsahoval 1. V takovém případě daný sloupec pivota prostě nemá a pokračuje se následujícím sloupcem.

Výsledkem algoritmu je zpracovaná matice s označenými řádky, které byly použity. Naopak každý nepoužitý řádek nám symbolizuje jeden nulový vektor. Nulový vektor získáme tak, že procházíme nepoužitý řádek po sloupcích, a pokud v daném sloupci nalezneme 1, pak v tomto sloupci hledáme všechny ostatní řádky, které mají také 1 v tomto sloupci. Vždy musí existovat alespoň jeden takový řádek. Indexy nalezených řádků včetně indexu neoznačeného společně uložíme do *vectoru*. Tento vektor uložíme do dalšího vektoru, který bude obsahovat všechny možnosti, jak získat nulový vektor. Tento vektor je výstupem fáze lineární algebry.

I když v návrhu (viz kap. 7.4) bylo zmíněno, že pro fázi lineární algebry bude zvolena Lanczos metoda, pro referenční implementaci ale nakonec byla zvolena Gaussova eliminační metoda. Metoda sice má horší časovou složitost než Lanczos metoda, ukázalo se ale, že při vhodné implementaci je tato metoda dostatečně rychlá i při faktorizaci čísel o 100 dekadických číslicích (viz kapitola 10.2.2). Je však zřejmé, že kvůli své kubické složitosti se při zvětšování faktorizovaného čísla dospěje do bodu, kdy zpracování matice touto metodou již bude trvat neúnosně dlouho. V takovém případě již bude nutné přejít na Lanczosovu či Wiedemannovu metodu, aby došlo ke snížení časové náročnosti této úlohy na kvadratickou.

9.7 Získání dělitele zadaného čísla

V této části jsou postupně zkoumány jednotlivé možnosti, jak získat nulový vektor. Zároveň v této části již relace nebudou třeba, ale naopak budou potřeba koeficienty a, b, c a proměnná x, které se k dané relaci váží. Pro každou možnost tedy procházíme uložené indexy ve vektoru. Tyto indexy odkazují do vektoru, kde je uložena příslušná relace, koeficienty a proměnná. Pomocí koeficientů a proměnné spočteme $u = ax + b$ a $v = (ax + b)^2 - N$. Hodnoty U a V pak získáme jako $U = \prod_{k=1}^{l} u_k \ mod\ N$ a $V = \sqrt{\prod_{k=1}^{l} v_k} \ mod\ N$, kde l je počet zúčastněných relací pro získání nulového vektoru. Když jsou spočteny U a V, ověřuje se, zda $U \neq V$. Pokud tato podmínka neplatí, získali jsme triviálního dělitele, a je nutné tak vyzkoušet jinou možnost. Platí-li podmínka, pak se provede výpočet největšího společného dělitele, $GCD\,(U - V,\ N)$, výsledek nás zajímá pouze v případě, že jsme získali něco jiného než 1 nebo N. Pokud jsme získali něco jiného, pak jsme nalezli netriviálního dělitele, a tedy faktorizace byla úspěšná. V takovém případě je vypsán výsledek na standardní výstup a algoritmus je ukončen. V opačném případě se zkoumají ostatní možnosti. Může nastat situace, že všechny možnosti, které jsme nalezli ve fázi lineární algebry, povedou pouze k nalezení triviálního dělitele. S každou nalezenou možností bychom měli mít asi 50% pravděpodobnost, že nalezneme netriviálního dělitele, a jelikož různých možností k zisku nulového vektoru bývá obvykle více než 10, měla by tato situace nastat velmi vzácně.

Kapitola 10

Měření a optimalizace

10.1 Referenční verze

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 o menší délce (a tedy s nižší časovou náročností) bylo ověřeno, že algoritmus byl správně pochopen a korektně naimplementován.

Konkrétně byla implementace nejdříve testována na čísle o 20 dekadických číslicích (65 bitů). Ve skutečnosti ale implementace takto malá čísla faktorizovat nebude, protože efektivnější je na tato čísla použít nějakou jednodušší metodu, jako například Pollard ρ metodu (viz kap. 6.2). Na tomto čísle však bylo možné otestovat, zda implementace správně provede tyto kroky:

- Vytvoří faktorizační bázi
- Vytvoří binární strom dvojic z faktorizační báze
- Spočte ideální hodnotu koeficientu a
- Vygeneruje koeficient a
- Na základě a získá koeficienty b, c a vytvoří tak polynom
- Provede fázi prosévání
  - Spočte kořeny pro daný polynom
  - Vybere kandidáty
  - Získá relace
- Provede fázi lineární algebry
- Získá netriviálního dělitele

Když byla implementace vyladěna a úspěšně splnila všechny výše uvedené body, bylo nutné se zaměřit na výměnu polynomů, protože u čísel s 20 dekadickými číslicemi nebylo nutné polynomy měnit a k nasbírání dostatečného počtu relací postačoval pouze jeden. Přešlo se tedy k faktorizaci čísel o 30 dekadických číslicích (97 bitů). U těchto čísel již došlo k obměně polynomů. Docházelo však pouze k obměně koeficientu b a c, vygenerovat nový koeficient a

zatím nebylo třeba. Ověřila se tak komplexnější činnost implementace. Posledním krokem bylo ověření kompletní implementace metody SIQS, kde se ověřovalo i nové generování koeficientu a, k tomu byla použita čísla o 40 dekadických číslicích (131 bitů).

Po úspěšné kontrole implementace byl zaveden paralelizmus do fáze generování polynomu a fáze prosévání. Takto provedený algoritmus posloužil jako referenční implementace s ověřenou funkčností. Bylo však zřejmé, že směrem k delším číslům jsou nutné jisté úpravy, zejména s ohledem na rychlosti provádění.

10.2 Optimalizace referenční verze

Optimalizace na rychlost byla druhou fází řešení implementace. Metodika iteračního provádění optimalizací se ukázala jako velmi účinná, přičemž je tato metodika použitelná obecně, nejenom v této úloze.

Jednotlivé kroky algoritmu SIQS mají různou třídu časové složitosti od lineární až po kubickou. Směrem k vyšším číslům tak byla v jednotlivých iteracích optimalizace odhalována vždy nová úzká hrdla provádění, která se u kratších čísel zatím jevila jako bezvýznamná.

10.2.1 Optimalizace pro rychlejší faktorizaci 60 dekadických číslic

Po dokončení implementace bylo provedeno měření a zkoumalo se, jak je implementace rychlá. Měření probíhalo na 10 číslech o délce 40 dekadických číslicích (131 bitů). Pro tato čísla bylo měření 10-krát opakováno. Dále se měřilo na 10 číslech o 50 dekadických číslicích (165 bitů). V tomto případě bylo měření opakováno 5-krát. Měření probíhalo jak pro sériové, tak pro paralelní řešení. Demonstračně byla provedena i faktorizace čísla o 60 dekadických číslicích (198 bitů). Jelikož ale faktorizace takto velkého čísla již trvala poměrně dlouho, bylo faktorizováno pouze jedno číslo, pouze jedenkrát a jen v paralelní variantě. Uvedené hodnoty v tabulce 10.1 pak vyjadřují průměrné zpracování jednoho čísla o dané velikosti. Měření byla provedená na notebooku vybaveném procesorem Intel i7 4700MQ se zapnutým Hyper-Threadingem a Turbo Boost technologií[1], k dispozici tak bylo 8 vláken, na kterých mohla metoda SIQS běžet. Notebook byl vybaven pamětí o velikosti 16GB a instalovaným operačním systémem byl Windows 8.1.

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

Tabulka 10.1: Naměřená doba faktorizace pro referenční verzi implementace

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. Profilace pomocí tohoto nástroje vždy probíhala na stroji vybaveném procesorem Intel Xeon E5 1650 v2 se zapnutým Hyper-Threadingem[2], k dispozici tak bylo 12 vláken. Stroj byl vybaven

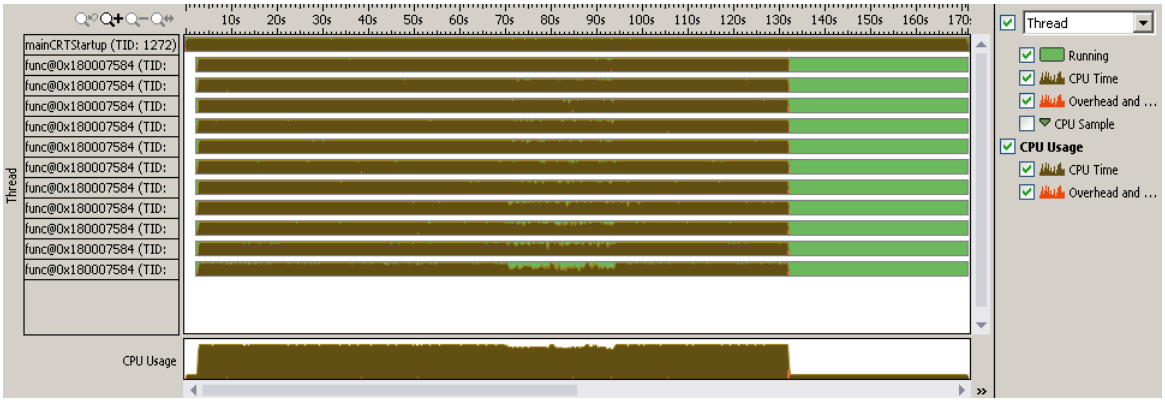
[1] Bližší informace o procesoru lze nalézt zde: http:// cpuboss.com/cpu/Intel-Core-i7-4700MQ
[2] http://cpuboss.com/cpu/ Intel-Xeon-E5-1650-v2

pamětí o velikosti 32GB a instalovaným operačním systémem byl Windows 7. Profilace byla provedena na čísle o 60 dekadických číslicích a výsledek profilace je vyobrazen na obrázku 10.1.

Function / Call Stack	CPU Time by Utilization (Idle Poor Ok Ideal Over)	Ov. an.	Module
SieveValues	684.245s	0s	DP_SIQS.exe
std::vector<unsigned __int64,class std::allocator<unsign	252.442s	0s	DP_SIQS.exe
std::_Tree<class std::_Tmap_traits<__int64,class std::v	141.619s	0s	DP_SIQS.exe
free	74.084s	0s	MSVCR120.dll
std::_Tree<class std::_Tmap_traits<__int64,class std::v	61.897s	0s	DP_SIQS.exe
std::_Tree<class std::_Tmap_traits<__int64,class std::v	53.140s	0s	DP_SIQS.exe
std::_Tree_buy<struct std::pair<__int64 const ,class st	39.197s	0s	DP_SIQS.exe
std::map<__int64,class std::vector<unsigned __int64,cl	31.118s	0s	DP_SIQS.exe
std::_Tree<class std::_Tmap_traits<__int64,class std::v	30.964s	0s	DP_SIQS.exe
FastGaussian	30.754s	0s	DP_SIQS.exe
log	12.849s	0s	MSVCR120.dll
std::map<unsigned __int64,double,struct std::less<unsig	12.776s	0s	DP_SIQS.exe
std::vector<unsigned __int64,class std::allocator<unsign	11.689s	0s	DP_SIQS.exe
_gmpn_get_d	11.045s	0s	DP_SIQS.exe
std::operator<<<struct std::char_traits<char> >	10.034s	0.051s	DP_SIQS.exe
DeleteDuplicants	8.136s	0s	DP_SIQS.exe
sqrt	6.784s	0s	MSVCR120.dll
_gmpn_divrem_1	6.714s	0s	DP_SIQS.exe
std::less<__int64>::operator()	4.907s	0s	DP_SIQS.exe
malloc	3.076s	0s	MSVCR120.dll

Obrázek 10.1: Profilace referenční verze

Zároveň byla provedena i profilace vytížení jednotlivých vláken, výsledek této profilace je vyobrazen na obrázku 10.2.



Obrázek 10.2: Vytížení vláken při faktorizaci

Z obrázku 10.2 lze vypozorovat, že vlákna při faktorizaci běží nezávisle, a každé vlákno tak vykonává svoji práci bez větší závislosti na ostatních. Ve chvíli, kdy je dokončena fáze prosévání, běží implementace sériově.

Z obrázku 10.1 lze vypozorovat, že nejvíce času se spotřebovává při prosévání, konkrétně ve funkci *SieveValues()*. Při hlubší analýze bylo zjištěno, že nejvíce času je spotřebováno v metodě *find()*. Jedná se o metodu mapy, která je ze standardní knihovny jazyka C++.

V tomto případě se metoda *find()* používá k ověření, zda pro dané x již máme uloženého nějakého dělitele. Pokud ne, je vytvořen nový vektor, do kterého je uložen daný dělitel a v budoucnu do něj budou ukládáni ostatní dělitelé. Problémem je, že metoda *find()* se volá pro každý násobek kořene každého prvočísla, a tak je tato metoda vyvolána opravdu často. Poměrně náročnou operací je také ukládání dělitele do vektoru, protože při ukládání dělitele je nutné nejdříve alokovat místo a pak teprve je možné dělitele uložit. Při dokončení prosévání nad daným polynomem jsou navíc vektory vyčištěny, protože hodnoty, které obsahují, již dále nejsou potřeba. Kromě vektorů jsou dealokována i některá pole, do kterých se průběžně přičítají hodnoty $\log(p)$ pro jednotlivá x a následně se podle uložených hodnot určují kandidáti. Při každém prosetí polynomu tak dochází k značnému volání funkce *free()*, což průběh samotné faktorizace také značně ovlivňuje.

Aby byla implementace urychlena, pak je nutné provést určité úpravy. Jednou z nich je, že zmíněná mapa z funkce *SieveValues()* bude nahrazena za vektor. Ukázalo se totiž, že téměř každé x z intervalu $\langle -M; M \rangle$ bude mít svého dělitele, a tak použití mapy v tomto případě postrádá smysl. Tím se odstraní i kontrola existence položky v mapě funkcí *find()*, která byla časově nejnáročnější. Vektor, který bude nyní uchovávat všechny vektory dělitelů, bude existovat po celou dobu fáze prosévání, a jelikož je jeho potřebná velikost známá hned na začátku této fáze, bude automatický zvětšen na tuto velikost.

K tomu se váže i další úprava. Všechna pole a vektory, jejichž velikost je známá hned na začátku fáze prosévání, budou alokována pouze jednou na příslušnou velikost a dealokována až ve chvíli, kdy zcela jistě již dále nebudou potřeba. Pokud bude nutné hodnoty v takovýchto polích či vektorech měnit, budou hodnoty již uložené v těchto polích nebo vektorech buď přepsány anebo nejdříve vynulovány a pak nahrazeny. Práce s pamětí tak bude efektivnější.

Dále budou hodnoty logaritmů pro jednotlivá prvočísla uložena do pole. Doposud se logaritmus prvočísla počítal vždy, když jej bylo potřeba, což je zbytečný výpočet, pokud tuto hodnotu je možno spočíst jednou a následně ji používat vždy, když je potřeba. Vytvořením pole ale dojde k větší paměťové konzumaci, než je ve skutečnosti nutné, protože se budou logaritmy z pole získávat dle samotného prvočísla, které bude indexem do pole. Problémem je, že reálně budou použity jen ta prvočísla z faktorizační báze, která se mohou podílet na získání B-hladkého čísla, a některé položky pole tak budou nevyužity. Jelikož se ale bude jednat o pole datového typu *double*, tak paměťová náročnost tohoto pole je vzhledem k celkové paměťové náročnosti metody SIQS zanedbatelná a navíc bude získán konstantní přístup k potřebnému logaritmu.

Poslední úpravou bude změna způsobu výpočtu prahu pro určení, zda je spočtená hodnota polynomu kandidátem. Jelikož se tato kontrola provádí poměrně často, bude konstantní část výpočtu spočtena pouze jednou a dopočítávat se k ní bude pouze ta 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 (235 bitů). Výsledky měření jsou zaneseny do tabulky 10.2. Hodnoty ve sloupci „Urychlení optimalizací“ jsou počítány pro paralelní verze implementací.

Úloha	Sériově [s]	Paralelně [s]	Urychlení paralelizmem	Urychlení optimalizací
40 dec	10.00	4.25	2.35	5.34
50 dec	157.86	59.24	2.66	5.20
60 dec	-	597.24	-	5.39
70 dec	-	5059.17	-	-

Tabulka 10.2: Měření SIQS po 1. provedené úpravě

10.2.2 Optimalizace pro rychlejší faktorizaci 70 dekadických číslic

Z tabulky 10.2 je možno vidět, že provedenými úpravami došlo k více než 5-násobnému urychlení. I když došlo ke značnému urychlení, stále je faktorizace poměrně pomalá, a tak se dále zkoumalo, kde jsou další možnosti urychlení. Ke stroji s profilačním nástrojem Intel VTune Amplifier nebyl přístup vždy zajištěn, a tak byla profilace v takovém případě prováděná přímo ve MS Visual Studio 2013, kde byla metoda SIQS vyvíjena. Profilace v tomto vývojovém prostředí postačuje k základní analýze, neposkytuje však takové vizuální a časové zobrazení jako Intel VTune Amplifier. Profilace byla provedena na čísle o 40 dekadických číslicích.

Obrázek 10.3: Profilace SIQS po 1. provedené úpravě

Obrázek 10.3 ukazuje, že nejvíce času metoda SIQS stále tráví ve funkci *SieveValues()*. To poukazuje na to, že aktuální přístup ještě stále není úplně správný, protože nejvíce času by mělo být spotřebováváno ve funkci *TrialDivision()*, kde dochází k masivnímu dělení velkých čísel, což je časově poměrné náročná operace. Byla tak provedena analýza, co výpočet nejvíce brzdí.

Bylo zjištěno, že ukládání dělitelů pro všechna x z intervalu $\langle -M; M \rangle$ 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 kdyby tato informace nebyla k dispozici, a bylo by nutné tak dělit všemi prvočísly z faktorizační báze. Problémem naopak 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 neuklá-

dala a každý kandidát by se dělil všemi prvočísly z faktorizační báze. Ukládání této informace tedy bylo odstraněno.

Dále byl doimplementován Knuth-Schroeppelův algoritmus [21]. Výstupem tohoto algoritmu je konstanta, kterou je nejvhodnější zadané číslo N vynásobit. U mnoha faktorizovaných 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 je větší pravděpodobnost, že nalezneme B-hladké číslo. Vynásobením zadaného čísla tímto násobitelem nám tedy může přinést značné urychlení.

Poslední provedenou úpravou bylo paralelizování odstranění duplicitních relací pomocí OpenMP direktivy *#pragma omp parallel for,* protože se procházejí všechny dvojice a kontroluje se, zda nejsou zkoumané relace shodné. Paralelizací se tak kontrola značně urychlí.

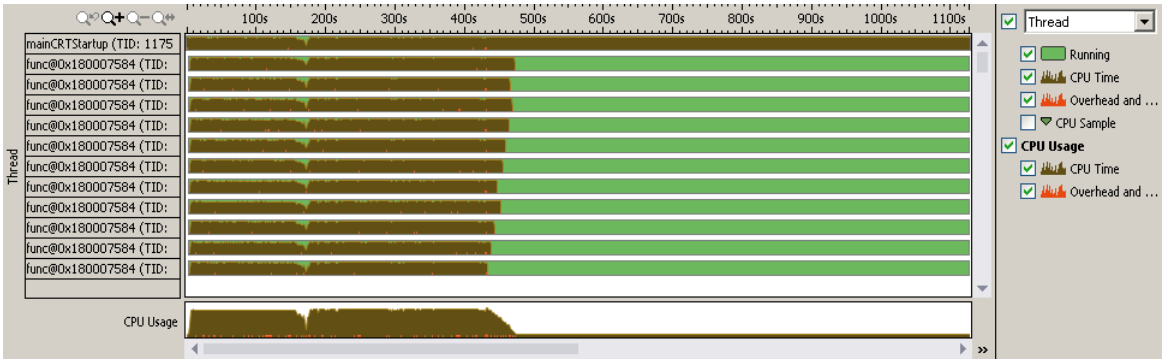
Po těchto úpravách byla opět provedena profilace. Výsledek je ukázán na obrázku 10.4. Profilace byla provedena na čísle o 70 dekadických číslicích, aby se projevilo co možná nejvíce slabých míst.

Function / Call Stack	CPU Time by Utilization Idle Poor Ok Ideal Over	Ov. an.	Module
SieveValues	731.829s	0s	DP_SIQS.exe
FastGaussian	644.813s	0s	DP_SIQS.exe
_gmpn_divrem_1	592.906s	0s	DP_SIQS.exe
log	334.865s	0s	MSVCR120.dll
DeleteDuplicantsomp1	308.109s	0s	DP_SIQS.exe
_gmpz_powm	253.242s	0s	DP_SIQS.exe
std::operator<<<struct std::char_traits<char> >	223.865s	0.530s	DP_SIQS.exe
malloc	185.066s	0s	MSVCR120.dll
_gmpn_addmul_1	174.017s	0s	DP_SIQS.exe
realloc	154.703s	0s	MSVCR120.dll
_gmpn_redc_1	137.273s	0s	DP_SIQS.exe
_gmpn_sqr_basecase	118.734s	0s	DP_SIQS.exe
free	102.190s	0s	MSVCR120.dll
_gmpn_jacobi_base	100.186s	0s	DP_SIQS.exe
_gmpn_add_n	86.221s	0s	DP_SIQS.exe
Sieve	85.391s	0s	DP_SIQS.exe
_gmpn_mul_1	81.529s	0s	DP_SIQS.exe
_gmpn_sqr	79.698s	0s	DP_SIQS.exe
_chkstk	76.414s	0s	DP_SIQS.exe
_gmpz_tdiv_r	70.798s	0s	DP_SIQS.exe

Obrázek 10.4: Profilace SIQS po 2. provedené úpravě

Jelikož byla provedena paralelizace i některých ostatních částí metody SIQS, byla opět provedena i profilace využití vláken. Výsledek profilace je vyobrazen na obrázku 10.5.

Z obrázku 10.4 lze vypozorovat, že doba strávená ve funkci *Sieve Value()* je velice podobná té z obrázku 10.1. Je nutné si ale uvědomit, že v prvním případě se prováděla profilace nad číslem s 60 dekadickými číslicemi, na obrázku 10.4 je však profilace s číslem o 70 dekadických číslicích, což poukazuje na značné urychlení faktorizace. Pro demonstraci, číslo o 60 dekadických číslicích, které bylo před provedenou úpravou faktorizováno za 597s, jak je možné vyčíst z tabulky 10.2, je nyní faktorizováno za 119s. To znamená, že provedenou úpravou byla faktorizace takto velkého čísla urychlena zhruba 5-krát.



Obrázek 10.5: Využití vláken po zavedení dalšího paralelizmu

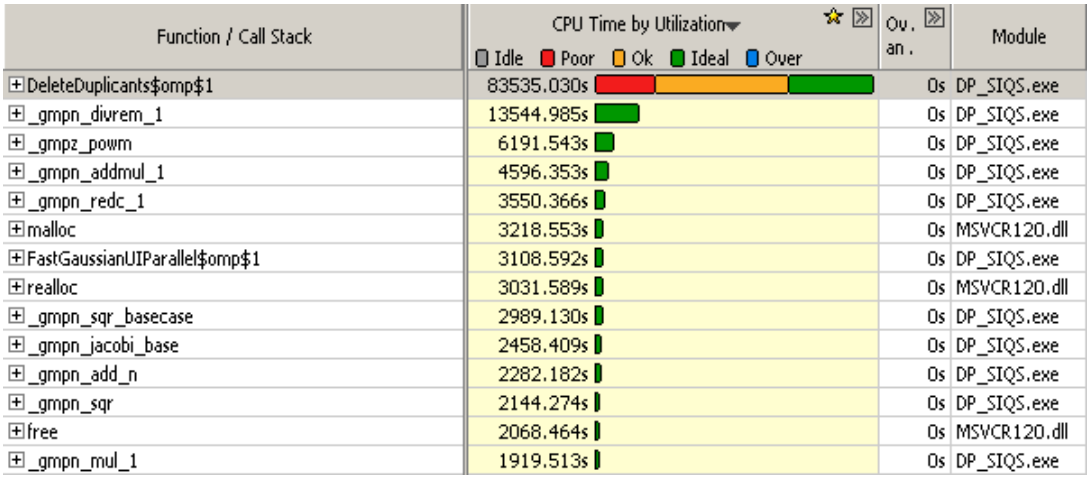
Obrázek 10.5 ukazuje, že při kontrole duplikantů si jednotlivá vlákna rozdělují kvanta dvojic, a tak některá vlákna mají více práce a některá méně. Pokud by rychlost v tomto místě byla kritická, bylo by možné tuto situaci zkusit vyřešit nastavením jiného přerozdělování práce, což OpenMP umožňuje. Co je ale nejdůležitější, jak obrázek 10.4, tak obrázek 10.5 ukazují, že stávající implementace Gaussovy eliminační metody se stala velkým problémem. Na obrázku 10.5 je možno pozorovat, že Gaussova eliminační metoda pro číslo o 70 dekadických číslicích dokonce trvá déle než fáze prosévání. Jelikož Gaussova eliminační metoda má kubickou časovou složitost, je tento problém s každým větším číslem horší. S touto implementací byl proveden pokus o faktorizaci čísla s 80 dekadickými číslicemi. Faktorizace ale byla předčasně ukončena, protože zpracování matice pomocí Gaussovy eliminační metody trvala déle jak 24 hodin.

Byla tedy provedena paralelizace této metody. Ukázalo se ale, že i přes paralelizaci metody byla doba zpracování matice příliš dlouhá. Důvodem je způsob uchování matice. Matice relací je ukládána do matice datového typu *bool*. Toto řešení je na jednu stranu velice pohodlné, protože jedna *bool* hodnota reprezentuje jedno prvočíslo z faktorizační báze, a tak se s tímto řešením dobře pracuje. Na druhou stranu ale pokud se musí provést větší kvantum operací nad maticí, což se v případě Gaussovy eliminační metody zcela jistě provádí, je zpracování matice velice neefektivní.

Navrženým řešením tedy je, aby se ukládání matice provádělo do matice 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. Bylo rozhodnuto, že se použije integer o velikosti 64 bitů. Dále se provede paralelizace odstranění singletonů, aby částí, které běží v SIQS sériově, bylo minimum. Po provedených úpravách byla provedena další profilace, která byla provedena pro faktorizaci čísla o 80 dekadických číslicích (266 bitů). Výsledek provedené profilace je vyobrazen v obrázku 10.6.

Profilace ukázala, že provedená změna měla razantní vliv na rychlost zpracování matice Gaussovy eliminační metody. Zatímco před úpravou musela být faktorizace čísla o 80 dekadických číslicích předčasně ukončena, po úpravě byla faktorizace tohoto čísla úspěšně dokončena a Gaussova eliminační metoda trvala jen několik málo minut.

Profilace zároveň ale také ukázala, že nyní se stala problémem funkce pro odstranění duplicitních relací. I přes to, že funkce běží paralelně, nastává stejný problém jako v případě Gaussovy eliminační metody. Relace jsou ukládány ve vektoru typu *bool*, a tak *xor* operace,



Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Ov, an.	Module
DeleteDuplicantsomp1	83535.030s	0s	DP_SIQS.exe
_gmpn_divrem_1	13544.985s	0s	DP_SIQS.exe
_gmpz_powm	6191.543s	0s	DP_SIQS.exe
_gmpn_addmul_1	4596.353s	0s	DP_SIQS.exe
_gmpn_redc_1	3550.366s	0s	DP_SIQS.exe
malloc	3218.553s	0s	MSVCR120.dll
FastGaussianUIParallelomp1	3108.592s	0s	DP_SIQS.exe
realloc	3031.589s	0s	MSVCR120.dll
_gmpn_sqr_basecase	2989.130s	0s	DP_SIQS.exe
_gmpn_jacobi_base	2458.409s	0s	DP_SIQS.exe
_gmpn_add_n	2282.182s	0s	DP_SIQS.exe
_gmpn_sqr	2144.274s	0s	DP_SIQS.exe
free	2068.464s	0s	MSVCR120.dll
_gmpn_mul_1	1919.513s	0s	DP_SIQS.exe

Obrázek 10.6: Profilace po úpravě Gaussovy eliminační metody

které je nutné provádět při kontrole duplicitních relací, jsou časově velice náročné. Bylo rozhodnuto, že i relace samotné budou ukládány v poli typu *integer*. Ve fázi prosévání nebude mít tato změna téměř žádný vliv na rychlost, naopak ale bude mít velký vliv právě při kontrole duplicitních relací, kdy bude možno zkoumat v dané dvojici více prvočísel najednou. Daná úprava byla tedy provedena. Následně se provedlo nové měření rychlosti a profilace. Jelikož rychlost implementace se rozumně zvýšila, testovalo se vždy 30 čísel o 40, 50 a 60 dekadických číslicích a to jak sériově, tak paralelně. Číslo o 70 dekadických číslicích však stále bylo měřeno jen jedno.

Úloha	Sériově [s]	Paralelně [s]	Urychlení paralelizmem	Urychlení optimalizací	Urychlení vůči referenční variantě
40 dec	2.12	1.32	1.61	3.22	14.11
50 dec	18.53	10.72	1.73	5.53	28.73
60 dec	102.19	32.23	3.17	18.53	99.83
70 dec	-	583.67	-	8.67	-

Tabulka 10.3: Měření rychlosti po přechodu z *bool* na *integer*

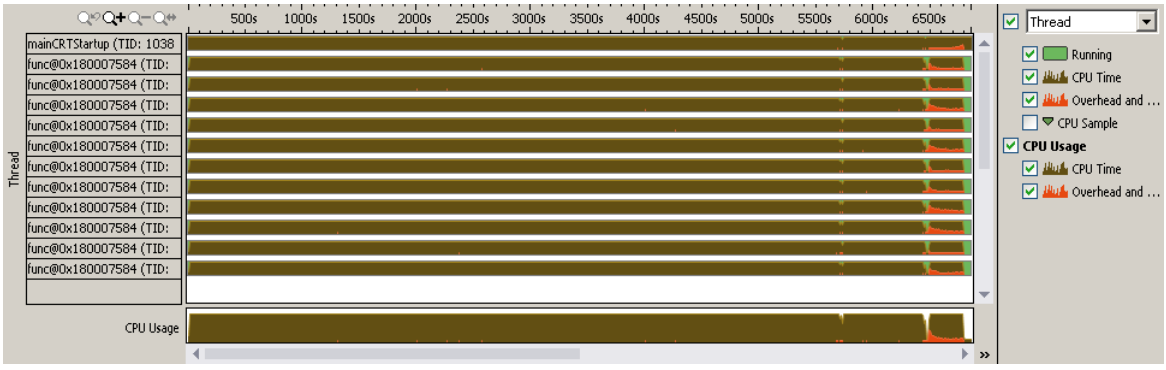
Provedené úpravy přinesly opět značné urychlení. Například u čísla s 60 dekadickými číslicemi došlo k téměř 100-násobnému urychlení vzhledem k referenční implementaci. Při prováděné profilaci bylo faktorizováno číslo o 80 dekadických číslicích. Faktorizace byla úspěšně dokončena za 1h 54min a 25s. Je vhodné zmínit, že veškeré faktorizace, při kterých zadané číslo mělo 80 dekadických číslic a více, byly z časových důvodu prováděny na zmíněném stroji s procesorem Intel Xeon E5 1650 v2. Výsledek profilace je zobrazen na obrázku 10.7.

Jelikož byla některá další místa v algoritmu optimalizována, byla provedena i profilace využití vláken. Výsledek této profilace zachycuje obrázek 10.8.

Z výsledků profilace na obrázku 10.7 lze vypozorovat, že nyní se největší spotřeba času nachází v aritmetických operacích MPIR knihovny. Další úpravy by tak musely vést k redukci počtu těchto operací. Na obrázku 10.8 pak lze pozorovat, že metoda SIQS pracuje již skoro ve všech místech paralelně. Jediná místa, kde nyní běží implementace sériově, je

Function / Call Stack	CPU Time by Utilization Idle Poor Ok Ideal Over	Ov. an.	Module
_gmpn_divrem_1	13694.545s	0s	DP_SIQS.exe
_gmpn_tdiv_qr	9908.148s	0s	DP_SIQS.exe
_gmpz_tdiv_r ← _gmpz_mod	5059.989s	0s	DP_SIQS.exe
_gmpz_powm	2999.607s	0s	DP_SIQS.exe
_gmpz_powm_ui ← quadratic_residue ← ComputeRoots ←	1848.552s	0s	DP_SIQS.exe
_gmpz_tdiv_qr_ui ← TrialDivision ← Sieve ← SIQSomp1	3101.516s	0s	DP_SIQS.exe
RtlUserThreadStart	2837.441s	0s	ntdll.dll
func@0x180007bc8 ← [OpenMP fork] ← SIQS ← main ←	264.075s	0s	VCOMP120.DLL
_gmpn_tdiv_q ← _gmpz_tdiv_q ← _gmpz_gcdext ← _gmpz	684.881s	0s	DP_SIQS.exe
RtlUserThreadStart	622.336s	0s	ntdll.dll
func@0x180007bc8 ← [OpenMP fork] ← SIQS ← main ←	62.545s	0s	VCOMP120.DLL
_gmpz_powm	6268.052s	0s	DP_SIQS.exe
quadratic_residue ← ComputeRoots ← Sieve ← SIQSomp1	6066.424s	0s	DP_SIQS.exe
RtlUserThreadStart	5555.698s	0s	ntdll.dll
func@0x180007bc8 ← [OpenMP fork] ← SIQS ← main ←	510.727s	0s	VCOMP120.DLL
_gmpz_powm ← quadratic_residue ← ComputeRoots ← Sieve	201.627s	0s	DP_SIQS.exe
RtlUserThreadStart	185.277s	0s	ntdll.dll
func@0x180007bc8 ← [OpenMP fork] ← SIQS ← main ←	16.350s	0s	VCOMP120.DLL

Obrázek 10.7: Profilace po přechodu z *bool* na *integer*



Obrázek 10.8: Využití vláken s rozšířeným paralelizmem

příprava SIQS, samotný výpočet faktoru zadaného čísla a transformace relací do matice. I když jsou relace ukládány do polí integerů a matice jsou opět pole integerů, je zde rozdíl, který brání polím relací k přímému použití pro matici. Ukládání relací je totiž řádkově orientované, protože jedna relace tvoří jeden řádek. Naopak matice, která se používá v Gaussově eliminační metodě, je sloupcově orientovaná, protože se pracuje nad pivotem, který se ve sloupcích vyhledává. Když byly relace ukládány do vektoru boolů a matice v Gaussově eliminační metodě byla taky typu bool, tak tento problém neexistoval, přechodem na datový integer je ale nutné tento problém řešit. Vzhledem k získanému urychlení se ale jedná o triviální záležitost.

Faktorizace čísla s 80 dekadickými číslicemi proběhla nad očekávání rychle, a tak byla rovnou provedena faktorizace čísla o 90 dekadických číslicích (299 bitů). 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 pravděpodobně bylo možné faktorizovat i číslo se 100 dekadickými číslicemi (332 bitů). Byl tedy učiněn pokus o faktorizování takto velkého čísla. Faktorizace takového čísla je však výrazně náročnější než faktorizace čísla o 90 dekadických číslicích 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, bylo tak ukázáno, že šifrování pomocí RSA-332 by bylo velkým bezpečnostním rizikem, protože klíč této velikosti by byl nalezen do 3 dnů.

Z tabulky 10.3 si je možné povšimnout, 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 naměřené časy tak ovlivňuje příprava SIQS, která běží vždy sériově. Příprava čísel od 40 dekadických čísel výše je vždy stejně časově náročná, a tak čím větší číslo se snažíme faktorizovat, tím více je čas strávený přípravou SIQS zanedbatelnější. Tabulka 10.4 tak ukazuje časy naměřené při provádění samotné faktorizace.

Úloha	Sériově [s]	Paralelně [s]	Urychlení paralelizmem	Rozdíl urychlení
40 dec	1.05	0.26	4.04	2.43
50 dec	11.60	2.99	3.88	2.15
60 dec	100.25	29.65	3.38	0.21

Tabulka 10.4: Naměřená doba pouze při provádění paralelizmem

Tabulka 10.4 prozrazuje, že v případě faktorizace čísel o 50 dekadických číslicích je příprava SIQS značně dlouhá. Byla tedy provedena hlubší analýza. Zjistilo se, že příčinou takto dlouhé přípravy je vytváření binárního stromu trojic. Pomocí tabulky 10.5 bude situace lépe popsána.

Jak bylo popsáno v kapitole 9.3.1, koeficient a polynomu se získá výběrem dělitelů pomocí algoritmu NEXKSB a výběrem nejlepší trojice či dvojice z binárního stromu. V kapitole 9.2 pak bylo napsáno, že vhodná prvočísla pro vytváření dvojic či trojic se získávají z určitého intervalu faktorizační báze na základě spočteného ideálního dělitele koeficientu a. Pro čísla o 50 dekadických číslicích vycházel ideální dělitel poměrně větší, než u čísel se 40 nebo 60 dekadickými číslicemi, což se projevilo v poměrně velkém intervalu, který byl pro čísla o 50

Úloha	Spodní hranice	Horní hranice	Celkem prvočísel v intervalu
40 dec	55	289	234
50 dec	55	374	319
50 dec*	55	276	221
60 dec	55	339	284

Tabulka 10.5: Tabulka hranic pro tvorbu binárního stromu

dekadických číslicích vytvořen. Čím větší interval je vytvořen, tím více trojic vznikne, což se projeví i v časové náročnosti tvorby binárního stromu. Byl tak učiněn pokus, kdy se zvětší počet dělitelů o 1. Díky tomu se interval poměrně zmenší, a tím pádem bude tvorba binárního stromu rychlejší. Informace o intervalu jsou v tabulce 10.5 zaznamenány na řádku s hvězdičkou.

Provedená úprava opravdu vedla k urychlení faktorizace čísel o 50 dekadických číslicích. Změna doby faktorizace je zaznamenaná v tabulce 10.6.

Stav	Sériově	Paralelně	Pouze faktorizace sér.	Pouze faktorizace par.
Před úpravou	18.53	10.72	11.60	2.99
Po úpravě	12.49	3.87	11.90	3.14

Tabulka 10.6: Porovnání rychlosti faktorizace čísla o 50 dekadických číslicích

Doba přípravy SIQS pro 50 dekadických číslic byla tedy poměrně redukována. Naopak doba samotné faktorizace lehce stoupla. Důvodem může být chyba měření, ale také fakt, že díky posunu hranice se vybírají pro koeficient a i již poměrně malá prvočísla, což může vést ke vzniku duplicitních relací. Výsledek úpravy však jasně ukazuje, že mnoho hodnot, jako je počet dělitelů, velikost intervalu, velikost faktorizační báze, atd. není možné jednoduše spočítat, ale je nutné tyto hodnoty experimentálně zkoušet a nalézt tak nejvhodnější variantu.

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 se opakujících 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 setrváním v rychlé vyrovnávací paměti. Navíc je nutné tato pole mezi vlákny sdílet. Problém se stupňuje s většími čísly, jež chceme faktorizovat, protože s každým větším číslem narůstá faktorizační báze i prosévací interval, podle čehož se úměrně zvětšují i zmíněná pole. Jelikož faktorizace běží paralelně, je tento problém o to větší. Nastává tak situace, kdy další úpravy podobného typu 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 nebyly.

10.3 Paměťové náročnosti

Pro poslední optimalizovanou verzi implementace SIQS byla měřena i paměťová náročnost. Výsledek měření byl zanesen do tabulky 10.7.

Úloha	Sériově [MB]	Paralelně [MB]
40 dec	33.2	42.0
50 dec	140.0	152.0
50 dec*	35.1	47.0
60 dec	109.0	127.8

Tabulka 10.7: Paměťová náročnost

V tabulce 10.7 je zaznačena i změna paměťové náročnosti u čísla o 50 dekadických číslicích po úpravě počtu dělitelů. Tato varianta je opět označena hvězdičkou. Zároveň toto srovnání poměrně jednoznačně ukazuje, že největší vliv na paměťovou náročnost má zatím binární strom trojic. S každým větším faktorizovaným číslem se ale bude velikost toho stromu zvětšovat jen minimálně a naopak ukládání relací bude mít stále větší vliv. Nakonec bude dosaženo situace, kdy tento strom bude mít téměř zanedbatelnou paměťovou náročnost vůči relacím, které je nutné při faktorizaci uchovávat.

10.4 Porovnání procesorů

Při vývoji byly použity dva stroje s různými procesory, a tedy i rozdílným výkonem. Stroj s procesorem Intel Core i7 4700MQ byl notebook, na kterém byla metoda SIQS implementována a obvykle zde probíhala faktorizace čísel do 60 dekadických čísel. Na druhém stroji s procesorem Intel Xeon E5 1650 v2 pak probíhala profilace a faktorizace čísel, které měla více než 60 dekadických číslic.

Úloha	Intel i7 Core 4700MQ			Intel Xeon E5 1650 v2		
	Sériově [s]	Paralelně [s]	Urychlení	Sériově [s]	Paralelně [s]	Urychlení
40 dec	2.12	1.32	1.61	1.73	0.96	1.80
50 dec	12.49	3.87	3.23	10.14	2.05	4.95
60 dec	102.19	32.23	3.17	91.32	14.83	6.16

Tabulka 10.8: Srovnání rychlosti použitých procesorů

Procesor Intel Core i7 4700MQ i Intel Xeon E5 1650 v2 v sériovém režimu těží hlavně z technologie TurboBoost. Jelikož faktorizace běží jen v jednom vlákně, je TDP (tepelný výkon) na přijatelné úrovni, a tak procesory mohou pracovat na maximální frekvenci $3.4GHz$ resp. $3.9GHz$. Rozdíl časů faktorizace je tak dán rozdílem pracovní frekvence a architektury procesoru (Haswell vs. Ivy-Bridge). V paralelním režimu je doba faktorizace již ale značně rozdílná. Jedním z důvodů je, že u procesoru Intel Xeon E5 1650 v2 máme k dispozici 12 vláken, které může při paralelizaci použít, oproti 8 vláknům, které poskytuje procesor Intel Core i7 4700MQ.

Neméně významným faktorem rychlosti faktorizace je však i pracovní frekvence. Jelikož v paralelním režimu faktorizuje pomocí všech vláken, které jsou k dispozici, je procesor vytížen na maximum a tím pádem je i TDP výrazně vyšší. Kvůli tomu již procesory

nevyužívají technologii TurboBoost, a pracují tak na normální frekvenci. Normální pracovní frekvence procesoru Intel Core i7 4700MQ je $2.4GHz$, u procesoru Intel Xeon E5 1650 v2 je to $3.5GHz$. Rozdíl pracovních frekvencí je tedy výraznější, což se i úměrně projevuje na době faktorizace. Z tabulky 10.8 je tak možno vidět, že díky těmto rozdílům je paralelní faktorizace čísla o 60 dekadických číslicích na procesoru Intel Xeon E5 1650 v2 dvakrát rychlejší.

Čisté porovnání procesorů v paralelním režimu však ovlivňuje příprava SIQS, která je vykonávaná sériově. Jelikož paralelní faktorizace čísel do 60 dekadických číslic je již poměrně rychlá, tak příprava SIQS může mít nemalý vliv na celkový čas faktorizace. Proto byla vytvořena tabulka 10.9, která porovnává časy samotné faktorizace. Samotná faktorizace je vykonávaná téměř celá paralelně, sériová část je však vykonána tak rychle, že je možné ji zanedbat.

Úloha	Intel i7 Core 4700MQ			Intel Xeon E5 1650 v2		
	Sériově [s]	Paralelně [s]	Urychlení	Sériově [s]	Paralelně [s]	Urychlení
40 dec	1.05	0.26	4.04	0.98	0.22	4.45
50 dec	11.90	3.14	3.80	9.73	1.64	5.93
60 dec	100.25	29.65	3.38	89.92	13.42	6.70

Tabulka 10.9: Srovnání rychlosti pouze faktorizace

10.5 Porovnání s Msieve

Pro srovnání SIQS implementace této práce s nejznámější implementací SIQS, Msieve, byla vytvořena tabulka 10.10. Implementace SIQS této práce je prostě pojmenována jako 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 10.10: Srovnání SIQS této práce a Msieve

Jak je z tabulky 10.10 vidět, implementace vytvořená v rámci této práce nedosahuje rychlosti Msieve. Je nutné si ale uvědomit, že čas investovaný do vývoje implementace této práce, 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. Díky tomu může implementace sloužit k demonstračním a výukovým účelům.

10.6 Porovnání Pollard ρ metody a SIQS

V rámci této práce byla implementována kromě metody SIQS ještě Pollard ρ metoda. Jelikož metoda má exponenciální časovou složitost, není vhodné ji používat pro větší čísla. Většími

čísly jsou zde myšlena čísla, která mají více jak 30 dekadických číslic. Pro čísla o 30 dekadických číslicích a menších by ale naopak měla být tato metoda efektivnější. Bylo tak provedeno měření, které mělo prokázat toto tvrzení.

Úloha	SIQS sériově [s]	SIQS Paralelně [s]	Pollard ρ metoda [s]
20 dec	0.14	-	0.03
30 dec	0.47	0.26	0.06
40 dec	2.12	1.32	-

Tabulka 10.11: Porovnání SIQS a Pollard ρ metody

Tabulka 10.11 ukazuje, že implementovaná Pollard ρ metoda je pro čísla do 30 dekadických číslic opravdu rychlejší než SIQS a to jak v sériovém, tak paralelním režimu. Metoda SIQS nemohla být měřena v paralelním režimu pro čísla o 20 dekadických číslicích, protože implementace generování polynomu nepočítá s tak malými čísly a tak nízkým počtem dělitelů koeficientu a. Naopak byl proveden pokus o faktorizaci čísel se 40 dekadickými číslicemi Pollard ρ metodou. Jelikož se ale ani po 10 minutách nepodařilo získat výsledek, bylo měření ukončeno.

10.7 Návrhy na další vylepšení

Kromě navržené úpravy na více „cache-friendly“ kód se dále 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. Také by bylo vhodné nahradit aktuální řešení tvorby stromu všech dvojic či trojic pro generování koeficientu a, protože aktuální řešení má velký vliv na výsledný čas při faktorizaci čísel do 60 dekadických číslic. Dalším vylepšením by také bylo přepočítávání kořenů na základě aktuálních kořenů při změně polynomů pouze výměnou koeficientů b, c. Aktuálně se kořeny vždy vypočítávají znovu.

Urychlení faktorizace by také mohl přinést přechod ze Single Large Prime Variation na Double Large Prime Variation. Díky tomu by stačilo nasbírat méně úplných relací a faktorizace by tak byla rychlejší. Poslední navrženou úpravou je sofistikovanější přístup k přípravě metody SIQS. Efektivnější faktorizace by bylo dosaženo, pokud by zvolené parametry jako velikost faktorizační báze byly nastavovány pro konkrétní faktorizované číslo a ne pouze pro čísla stejné velikosti.

Kapitola 11

Závěr

V práci byly popsány nejznámější faktorizační metody. Ke každé metodě byly blíže popsány i principy, na kterých je metoda založena. Čtenář tak v případě neznalosti těchto principů není nucen vyhledávat jinou literaturu, kde jsou tyto principy také popsány, ale může si je přečíst přímo v této práci. Studium dané metody je tak maximálně efektivní. Popis metody je v některých případech doplněn i o příklad, který metodu názorně předvádí a případně na něm vysvětluje její přednosti či nedostatky.

Všechny metody byly následně probrány z pohledu jejich paralelizace a smyslu provedení takovéto paralelizace. Ukázalo se, že metody s exponenciální časovou složitostí nemá smysl vůbec uvažovat, protože i paralelizovaná exponenciální metoda nebude rychlejší než metoda subexponenciální. Z metod se subexponenciální časovou složitostí byla nakonec vybrána metoda kvadratického síta ve variantě SIQS (viz kap. 6.8).

Metoda SIQS byla implementována v jazyce C++ a jako architektura byla zvolena x86 (x86-64) CPU. Architektura x86 (x86-64) byla zvolena z důvodu, že i jedna výpočetní stanice je v dnešní době schopna poskytnout dostatek paměti například pro výpočet faktorizace RSA-174, pro jehož úspěšnou faktorizaci je potřeba kolem 7GB paměti. Jazyk C++ nabízí v rámci svých standardních knihoven možnost použití konstrukcí, které sice velice usnadňují programování, ale jak bylo ukázáno (viz kap. 10), tyto konstrukce nepřispívají k rychlosti faktorizace, a tak byly většinou nahrazeny jinými konstrukcemi. Dále jazyk C++ podporuje využití API OpenMP, a v této práci bylo tedy OpenMP využito k paralelizaci SIQS. Přestože je práce s vlákny při použití OpenMP výrazně jednodušší, může nastat situace, kdy OpenMP není schopno poskytnout všechny logické procesory, které jsou k dispozici, a tak faktorizace není někdy úplně efektivní. V práci byla použita knihovna pro práci s velkými čísly, MPIR, jež je upravenou knihovnou GMP pro operační systém Windows. Díky této knihovně bylo možno pracovat s optimalizovanými aritmetickými operacemi nad velkými čísly.

V první fázi implementace byla vytvořena referenční verze. Referenční verze byla implementována nejjednodušším způsobem bez přihlédnutí k požadavkům na rychlost. Pomocí referenční verze bylo ověřeno, že algoritmus byl správně pochopen a korektně naimplementován. Bylo zřejmé, že rychlost takto naimplementované metody nebude ideální, a tak v druhé fázi byla provedena optimalizace na rychlost. Za tímto účelem byla zvolena metodika iteračního provádění optimalizací, která se ukázala jako účinná a je použitelná obecně, nejenom v této úloze. V každé iteraci optimalizací bylo obvykle faktorizováno číslo větší než v předešlé iteraci. Jelikož se algoritmus SIQS skládá z několika kroků, které mají různou časovou složitost, byla tímto postupem odhalována další úzká hrdla, která se u kratších čísel jevila jako bezvýznamná. Jednou z částí optimalizace bylo i důsledné zavádění paralelizace

všude tam, kde to bylo možné z podstaty algoritmu. Aktuálně tak není paralelizována jen malá část algoritmu.

Díky použití metodiky iteračního provádění iterací, jak ukazuje tabulka 10.3, bylo dosaženo až 100-násobného urychlení faktorizace čísel s 60 dekadickými číslicemi. Optimalizovanou implementací SIQS se zároveň podařilo faktorizovat číslo o 100 dekadických číslicích, což odpovídá 332 bitovému RSA. Faktorizace tohoto čísla byla úspěšně provedena za 2 dny 16 hodin a 13 minut, a práce tak ukazuje, že použití šifry RSA o této délce by bylo velice nebezpečné.

V kapitole 10.7 byly navrženy úpravy, které povedou k dalšímu urychlení. Výpis není vyčerpávající, protože jak bylo ukázáno, po každých optimalizacích je vhodné opět provést profilaci a hledat místa, která nově vytvářejí úzké hrdlo anebo by bylo možné je ještě vylepšit. Je odhadováno, že díky dalším optimalizacím, které by však nutně znamenaly výraznější úpravu kódu, by bylo možné faktorizovat čísla dlouhá až 512 bitů.

Literatura

[1] Brown, E.: Square Roots From 1: 24.51. 10 to Dan Shanks. *College Mathematics Journal*, ročník 30, 1999: s. 82–95.

[2] Carrier, B.; Wagstaff Jr, S. S.: Implementing the hypercube quadratic sieve with two large primes. In *International Conference on Number Theory for Secure Communications*, 2003, s. 51–64.

[3] Cavallar, S.; Dodson, B.; Lenstra, A. K.; aj.: Factorization of a 512-bit RSA modulus. In *Advances in Cryptology—EUROCRYPT 2000*, Springer, 2000, s. 1–18.

[4] Contini, S. P.: Factoring integers with the self-initializing quadratic sieve. 1997.

[5] Coppersmith, D.: Solving linear equations over GF (2): block Lanczos algorithm. *Linear algebra and its applications*, ročník 192, 1993: s. 33–60.

[6] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to algorithms*, ročník 2. MIT press Cambridge, 2001.

[7] Guy, R. K.: How to factor a number. In *Fifth Manitoba Conference on Numeral Mathematics Congressus Numerantium*, ročník 16, 1976, s. 49–89.

[8] Kleinjung, T.; Aoki, K.; Franke, J.; aj.: Factorization of a 768-bit RSA modulus. In *Advances in Cryptology–CRYPTO 2010*, Springer, 2010, s. 333–350.

[9] Koç, Ç. K.; Arachchige, S. N.: A Fast Algorithm for Gaussian Elimination over GF (2) and its Implementation on the GAPP. *Journal of Parallel and Distributed Computing*, ročník 13, č. 1, 1991: s. 118–122.

[10] Lenstra, A. K.; Lenstra Jr, H. W.; Manasse, M. S.; aj.: *The number field sieve*. Springer, 1993.

[11] Montgomery, P. L.: A block Lanczos algorithm for finding dependencies over GF (2). In *Advances in cryptology—EUROCRYPT'95*, Springer, 1995, s. 106–120.

[12] Morrison, M. A.; Brillhart, J.: A method of factoring and the factorization of F_7. *Mathematics of Computation*, ročník 29, č. 129, 1975: s. 183–205.

[13] Nijenhuis, A.; Wilf, H. S.: *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014.

[14] Peterson, M.: *Parallel block lanczos for solving large binary systems*. Dizertační práce, Texas Tech University, 2006.

[15] Pollard, J. M.: Theorems on factorization and primality testing. In *Mathematical Proceedings of the Cambridge Philosophical Society*, ročník 76, Cambridge Univ Press, 1974, s. 521–528.

[16] Pollard, J. M.: A Monte Carlo method for factorization. *BIT Numerical Mathematics*, ročník 15, č. 3, 1975: s. 331–334.

[17] Pomerance, C.: The quadratic sieve factoring algorithm. In *Advances in cryptology*, Springer, 1985, s. 169–182.

[18] Pomerance, C.: A tale of two sieves. *Biscuits of Number Theory*, ročník 85, 2008.

[19] Rabin, M. O.: Probabilistic algorithm for testing primality. *Journal of number theory*, ročník 12, č. 1, 1980: s. 128–138.

[20] Rivest, R. L.; Shamir, A.; Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, ročník 21, č. 2, 1978: s. 120–126.

[21] Silverman, R. D.: The multiple polynomial quadratic sieve. *Mathematics of Computation*, ročník 48, č. 177, 1987: s. 329–339.

[22] Solovay, R.; Strassen, V.: A fast Monte-Carlo test for primality. *SIAM journal on Computing*, ročník 6, č. 1, 1977: s. 84–85.

Příloha A

Obsah DVD

- **Tex** - zdrojové kódy pro LaTeX a LyX
- **Doc** - obsahuje vygenerovaný pdf dokument
- **Src** - zdrojové kódy implementované faktorizační metody
- **Example** - přiložený konečný výpočet praktického příkladu z kapitoly 8
  - **60dec** - log soubory z faktorizace čísla o 60 dekadických číslic
- **Utils** - použité nestandardní knihovny použité v této práci