FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Porovnání algoritmů pro faktorizaci velkých celých čísel
Student:	Bc. Jakub Dvořák
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

1. Nastudujte vybrané faktorizační algoritmy (Pollard p - 1, Pollard-Rho, ECM, kvadratické síto, GNFS) [1-6]
2. Popište princip vybraných algoritmů společně s jejich časovými a paměťovými složitostmi.
3. Implementujte vybrané algoritmy pomocí knihoven GMP a FLINT.
4. Diskutujte jejich možnost paralelizace, možnou paralelizaci uskutečněte pomocí knihovny OpenMP nebo OpenMPI.
5. Implementované algoritmy vzájemně porovnejte z hlediska časové a paměťové složitosti. Diskutujte naměřené výsledky.

Seznam odborné literatury

[1] Landquist, E.: The quadratic sieve factoring algorithm. University of Virginia, 2001.
[2] Pollard, J. M.: A Monte Carlo method for factorization. BIT, 1975.
[3] Charest, AS.: Pollard's p-1 and Lenstra's factoring algorithms. 2005.
[4] Pomerance, C.: The Quadratic Sieve Factoring Algorithm. EUROCRYPT, 1985.
[5] Parker, D.: Elliptic Curves and Lenstra's Factorization Algorithm. University of Chicago, 2004.
[6] Briggs, M. E.: An Introduction to the General Number Field Sieve.Virginia Polytechnic Institute and State University, 1998

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 24. října 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Porovnání algoritmů pro faktorizaci velkých celých čísel

Bc. Jakub Dvořák

Katedra informační bezpečnosti
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

28. května 2020

Poděkování

Na tomto místě bych rád poděkoval doc. Ing. Ivanovi Šimečekovi, Ph.D. za vedení této práce a věcné rady při zpracovávání této práce. Také bych rád poděkoval své rodině za velkou podporu během celého mého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 28. května 2020 . . . . . .

Odkaz na tuto práci

Dvořák, Jakub. *Porovnání algoritmů pro faktorizaci velkých celých čísel.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

V této práci se zabývám faktorizačními algoritmy. V první části zavedu matematické pojmy, které jsou nutné pro pochopení popsaných algoritmů. V další části popíši kryptosystém RSA založený na problému faktorizace. Následně se zaměřuji na analýzu časové složitosti a paměťové náročnosti vybraných algoritmů. Další část popisuje implementaci algoritmů a jejich paralelizace. Na závěr jsou ukázány výsledky měření časů a spotřebované paměti implementací algoritmů.

Klíčová slova Pollard, Pollard–ρ, Pollard $p-1$, ECM, Lenstrův algoritmus, kvadratické síto, obecné číselné síto, faktorizace

Abstract

In this thesis I deal with factorization algorithms. In the first part I define mathematical concepts that are necessary for understanding the described algorithms. In the next part I describe RSA cryptosystem that is based on factorization problem. Then I focus on the analysis of time and space complexity of selected algorithms. Next part describes implementation of algorithms and their parallelization. At the end are shown the results of time and space measurement of implemented algorithms.

Keywords Pollard, Pollard–ρ, Pollard $p - 1$, ECM, Lenstra's algorithm, quadratic sieve, general number field sieve, factorization

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Šifrovaná komunikace na Internetu je dnes téměř běžnou činností. Některé šifrovací kryptosystémy bývají založeny na matematických problémech, které jsou pro nás dnes obtížně řešitelné. Mezi takové problémy patří například problém diskrétního logaritmu nebo problém faktorizace celých čísel. Pokud by se nám tedy podařilo některý z těchto problémů efektivně řešit, tak bychom tím mohli narušit kryptosystém, který je na daném problému založen, a tím dešifrovat tajná data.

Faktorizace celých čísel je dnes stále diskutované téma. Tento problém není pouze definován v naší době. Tímto problémem se zabývali matematici ještě v dobách před naším letopočtem jako například Eukleidés, avšak pevné základy tohoto problému se začaly definovat až v období kolem 17. století. V této době se daným problémem začali zabývat známí matematici jako byli Leonhard Euler nebo Pierre de Fermat. Počítače sice výpočet urychlily, avšak dnes známe algoritmy jsou stále pomalé. Od 70. let 20. století se začínají objevovat algoritmy, jejichž časová složitost je subexponenciální. Některé z těchto algoritmů v této práci si ukážeme.

Problém faktorizace celých čísel řadíme do třídy takzvaných NP–problémů. To znamená, že nedokážeme faktorizovat čísla v polynomiálním čase, tedy výpočet může trvat i několik let od nějaké velikosti faktorizovaného čísla. Díky tomu se na tomto problému zakládají některé šifrovací algoritmy jako je například RSA.

I když se dnes zdá být tento problém neřešitelný, algoritmus pro kvantové počítače již byl vyvinut, který tento problém dokáže řešit v polynomiálním čase. Tento algoritmus je pojmenovám podle svého objevitele, amerického profesora na MIT, Petera Shora, Shorův algoritmus. Společnosti IBM se sice již povedlo implementovat tento algoritmus na jejich kvantovém systému (více viz `https://www.ibm.com/quantum-computing/`), avšak jak je uvedeno v [1], tak stále tento systém není v takové fázi vývoje, aby ohrozil dnešní kryptosystémy.

Kapitola 1

Teoretický základ pro metody faktorizace

V této kapitole se seznámíme se základní teorií. Následující definice jsou důležité pro pochopení fungování použitých faktorizačních algoritmů. Dané pojmy budou použity v další kapitole s popisem algoritmů. Budeme zde vycházet zejména ze studijního textu k předmětu MI-MKY [2] a skript [3].

1.1 Základní algebraické struktury

Definice 1. (Grupa) **Grupou** nazýváme uspořádanou dvojici $(M, \circ)$, kde M je neprázdná množina (*nosič*) a $\circ$ je binární operace, pokud splňuje následující podmínky:

- asociativity, tedy $a \circ (b \circ c) = (a \circ b) \circ c$, platí pro každé $a, b, c \in M$,
- existence neutrálního prvku e vůči binární operaci $\circ$, tedy $a \circ e = e \circ a$ pro každé $a \in M$,
- existence inverzních prvků – ke každému $a \in M$ existuje prvek $a^{-1} \in M$ takový, že $a \circ a^{-1} = e = a^{-1} \circ a$.

Pokud navíc splňuje podmínku komutativity, tj. $a \circ b = b \circ a$, pro každé $a, b \in M$, nazýváme grupu **abelovskou** (komutativní). Pokud se bavíme o prvku grupy $g \in G$, tak myslíme prvek *nosiče*, tedy $g \in M$. Pro abelovské grupy značíme $\circ$ symbolem $+$, pokud se bavíme o aditivní notaci. Pro multiplikativní notaci budeme $\circ$ značit symbolem $\cdot$.

Definice 2. (Konečná grupa) Nechť $G = (M, \circ)$ je grupa. Grupu nazýváme konečnou, má-li množina M konečný počet prvků. **Řádem** konečné grupy názýváme počet prvků množiny M a značíme jej $\#G$. Pokud má množina M nekonečný počet prvků, pak říkáme, že řád grupy je nekonečný.

Definice 3. (Řád prvku) Nechť $G = (M, \circ)$ je grupa a $a \in M$ je prvkem této grupy. Nejmenší $x \in \mathbb{N}$, takové že $a^x = e$ nazýváme **řádem prvku**. Pokud takové x neexistuje, říkáme, že prvek má nekonečný řád.

Nyní si připomeňme pár notačních zvyklostí. Pokud se budeme bavit o multiplikativní notaci, tak místo $a \cdot b$ budeme psát pouze ab. Pro aditivní notaci budeme inverzní prvek k $a \in G$ (viz definici 1) místo a^{-1} značit $-a$. Neutrální prvek v aditivní notaci budeme značit symbolem 0. V multiplikativní notaci jej budeme značit symbolem 1.

V grupě si zavedeme **násobení** pro aditivní notaci takto:

$$\forall a \in G, \forall k \in \mathbb{N} : a^k = ka = \underbrace{a + a + \cdots + a}_{\text{k–krát}}$$

Pro umocňování v multiplikativní notaci pak takto:

$$\forall a \in G, \forall k \in \mathbb{N} : a^k = \underbrace{a \cdot a \cdot \ldots \cdot a}_{\text{k–krát}}$$

Uvedeme si pro příklad pár grup:

1. Množina celých čísel $\mathbb{Z}$ vybavená operací sčítání je prvků z této množiny tvoří grupu. Neutrálním prvkem je 0 a inverzním k $n \in \mathbb{Z}$ je $-n \in \mathbb{Z}$
2. Množina $\{0, 1, \ldots, n-1\}$ vybavená operací sčítání modulo n tvoří modulární aditivní grupu, kterou značíme $\mathbb{Z}_n^+$.
3. Množina $\{k \in \mathbb{Z} | 1 \leq k \leq n-1, \gcd(k, n) = 1\}$ vybavená operací násobení modulo n tvoří modulární multiplikativní grupu, kterou značíme $\mathbb{Z}_n^\times$.

Definice 4. (Podgrupa) Nechť je dána grupa $G = (M, \circ)$ a podmnožina N množiny M. O dvojici $H = (N, \circ)$ říkáme, že je **podgrupou** grupy G, pokud H tvoří grupu.

Definice 5. (Homomorfizmus grup) Zobrazení $f : G \to H$ grupy $(G, \cdot)$ do grupy $(H, \circ)$ splňující $\forall a, b \in G : f(a \cdot b) = f(a) \circ f(b)$ nazýváme **homomorfizmem** grup G a H. Pokud navíc:

- f bijektivní (prosté a na), pak f nazýváme **izomorfizmem** G a H,
- $G = H$ a f je bijektivní, pak f nazýváme automorfizmem G.

Definice 6. (Ekvivalence) **Relací** R na množině M nazýváme libovolnou podmnožinu kartézského součinu $M \times M$. Relaci R na množině M nazýváme ekvivalencí na množině M, právě když platí následující:

- reflexivita: $\forall x \in M : (x, x) \in R$,
- symetrie: $\forall x, y \in M : (x, y) \in R \Rightarrow (y, x) \in R$,

- tranzitivita: $\forall x, y, z \in M : (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$.

Množinu všech prvků ekvivalentních s $x \in M$ značíme $[x] = y \in M|(x, y) \in R$.

Tvrzení 1.1.1. Buď H podgrupou grupy G. Relace $\mathcal{L}_H$ na G definovaná jako

$$(a, b) \in \mathcal{L}_H \Leftrightarrow (\exists h \in H)(a = bh)$$

je relací ekvivalence na G. Relace ekvivalence $\mathcal{L}_H$ rozkládá množinu G na disjunktní třídy ekvivalentních prvků $[a]_{\mathcal{L}_H}, a \in G$, pro které platí

$$[a]_{\mathcal{L}_H} = \{ah|h \in H\}$$

a které nazýváme **levé třídy rozkladu** G vzhledem k H. Třídy ekvivalentních prvků někdy značíme jako $aH := [a]_{\mathcal{L}_H}$. Pravé třídy rozkladu bychom definovali analogicky k levým.

Definice 7. (Normální podgrupa) Podgrupu H grupy G nazýváme **normální podgrupou**, pokud pro každé $x \in G$ platí:

$$xHx^{-1} = H$$

Definice 8. (Faktorgrupa) Je-li H normální podgrupa grupy G, pak třídy rozkladu G vzhledem k H spolu s operací $[a] \cdot [b] = [a \cdot b], a, b \in G$ tvoří grupu. Takto zavedenou grupu nazýváme **faktorgrupou** G vzhledem k H a značíme ji G/H.

Příkladem takové faktorgrupy může být tento. Mějme grupu reálných čísel $\mathbb{R}$ se standardním sčítáním a její podgrupu celých čísel $\mathbb{Z}$. Díky komutativitě operace je $\mathbb{Z}$ normální podgrupou. Podle definice potom platí:

$$[x] = \{x + k|k \in \mathbb{Z}\}, x \in \mathbb{R}$$

Definice 9. (Cyklická grupa) Grupu G nazýváme cyklickou, pokud existuje prvek $a \in G$ takový, že pro libovolné $b \in G$ existuje celé číslo n splňující b = a^n. Tento prvek a nazýváme **generátorem** grupy G.

1.2 Okruhy a tělesa

Definice 10. (Okruh) **Okruhem** R nazýváme množinu R se dvěma binárními operacemi $+ : R \times R \to R$ a $\cdot : R \times R \to R$ splňujícími:

- R je abelovská grupa vůči +,
- operace $\cdot$ je asociativní,
- platí levý i pravý distributivní zákon.

Pokud navíc:

- existuje neutrální prvek vůči násobení $\cdot$, nazýváme okruh **okruhem s jedničkou**,
- násobení $\cdot$ je komutativní, nazýváme okruh **komutativní**,
- je okruh komutativním a s jedničkou, a z rovnosti $ab = 0$ plyne $a = 0 \vee b = 0$, nazýváme okruh **oborem integrity**.
- $(R \setminus \{0\}, \cdot)$ tvoří grupu, nazýváme okruh **okruhem s dělením**.

Definice 11. (Těleso) Komutativní okruh s dělením nazýváme **tělesem**.

Definice 12. (Podokruh, podtěleso) Podmnožinu S okruhu R nazýváme **podokruhem** okruhu R, pokud trojice $(S, +, \cdot)$ tvoří okruh. Analogicky definujeme i pro **podtěleso**.

Definice 13. (Ideál) Podmnožinu J množiny R nazýváme **ideálem** okruhu R, pokud je J podokruhem okruhu R a $\forall a \in J, b \in R : ab \in J \wedge ba \in J$.

Definice 14. (Faktorokruh) Množina rozkladových tříd okruhu R vzhledem k ideálu J se nazývá **faktorokruh** okruhu R vzhledem k ideálu J a značíme jej R/J.

Definice 15. (Polynom) **Polynom** $p(x)$ nad libovolným okruhem R definujeme jako:

$$p(x) = \sum_{k=0}^{n} \alpha_k x^k,$$

kde $n \in \mathbb{N}_0, \alpha_k \in R$, α_k nazýváme **koeficienty polynomu**. Dále zkusme předpokládat $\alpha_n \neq 0$. Tento prvek nazýváme **vedoucím koeficientem** polynomu $p(x)$, $st(f) = n$ nazýváme **stupněm polynomu** a α_0 **konstantním členem**. Pokud okruh R je okruh s jedničkou a pokud $\alpha_n = 1$, pak polynom p nazýváme **monickým**. Polynom, jehož stupeň je menší nebo roven 0 nazýváme **konstantním polynomem**.

Definice 16. (Okruh polynomů) Okruh polynomů nad okruhem R značíme jako $R[x]$.

Definice 17. (Ireducibilní polynom) Polynom $p \in T[x]$ se nazývá **ireducibilní** nad tělesem T právě když má kladný stupneň a pokud $p = bc, b, c \in T[x]$, pak b nebo c jsou konstantní polynomy.

Tvrzení 1.2.1. Je-li $p \in T[x]$, potom okruh rozkladových tříd $T[x]/f$ je tělesem, práve když polynom p je ireducibilní polynom nad tělesem T.

Definice 18. (Kořen polynomu) Prvek $r \in T$, T je těleso, nazýváme kořenem polynomu $p \in T[x]$, pokud $p(r) = 0$.

Definice 19. (Derivace polynomu) Nechť $p(x) = \alpha_0+\alpha_1 x+\alpha_2 x^2+\ldots+\alpha_n x^n \in T[x]$, pak polynom $p'(x) = \alpha_1 + 2\alpha_2 x + \cdots + n\alpha_n x^{n-1} \in T[x]$ nazýváme **derivací polynomu** p.

Definice 20. (Rozšíření) Nechť je dáno těleso T a jeho podtěleso K. O tělese T pak mluvíme jako o **rozšíření** tělesa K.

Definice 21. Nechť je K podtěleso tělesa T a M libovolná podmnožina T. Potom definujeme jako těleso $K(M)$ jako průnik všech podtěles obsahujících K i M a nazýváme ho **rozšířením** K **připojením** M. Pro konečnou množinu $M = \{\theta_0, \ldots, \theta_n\} \subset T$ píšeme zkráceně $K(M) = K(\theta_0, \ldots, \theta_n)$. Pokud M obsahuje pouze jeden prvek $\theta \in T$, pak $K(\theta)$ nazýváme **jednoduchým rozšířením** K a θ se nazývá **definujícím prvkem** $K(\theta)$ nad K.

Definice 22. (Algebraický prvek, algebraické rozšíření) Mějme K podtěleso tělesa T a $\theta \in T$. Pokud θ splňuje netriviální polynomiální rovnici:

$$\alpha_n\theta^n + \cdots + \alpha_1\theta + \alpha_0 = 0, \exists i \in \{0, \ldots, n\} : \alpha_i \neq 0, \alpha_i \in K,$$

potom nazýváme θ **algebraickým prvkem** nad K. Rozšíření L podtělesa K nazýváme **algebraickým**, pokud každý prvek L je algebraický nad K.

Definice 23. (Minimální polynom) Je-li $\theta \in T$ algebraický prvek nad K, potom jednoznačně určený monický polynom $g \in K[x]$ se nazývá **minimální polynom** θ nad K. **Stupněm** θ nad K nazýváme stupeň polynomu $g \in K[x]$.

Tvrzení 1.2.2. Je-li $\theta \in T$ algebraický prvek nad K, potom jemu příslušný minimální polynom g nad K má následující vlastnosti:

1. g je ireducibilní v $K[x]$,
2. pro $f \in K[x]$ platí rovnice $f(\theta) = 0$ tehdy a jen tehdy, když g dělí f,
3. g je monický polynom z $K[x]$ nejmenšího stupně mající θ jako kořen,
4. $K(\theta)$ je izomorfní s $K[x]/g$.

1.3 Eliptické křivky

Další částí, které potřebujeme definovat jsou operace nad množinou prvků eliptické křivky.

Definice 24. (Eliptická křivka) **Eliptickou křivkou** nad tělesem $\mathbb{R}$ nazýváme množinu $E \subset \mathbb{R}^2$ všech řešení rovnice

$$y^2 = x^3 + ax + b, (x, y) \in \mathbb{R}^2, a, b \in \mathbb{R}, 4a^3 + 27b^2 \neq 0$$

Takto definované podmínky nám zaručí korektní definici potřebných operací nad touto množinou. Do takovéto množiny se navíc přidává ještě bod O, který označuje neutrální prvek. Nyní si zavedem sčítání nad takto definované množině.

Definice 25. (Operace $\oplus$) Mějme eliptickou křivku $E \cup \{O\}$ nad $\mathbb{R}$ danou rovnicí

$$y^2 = x^3 + ax + b, 4a^3 + 27b^2 \neq 0,$$

potom operaci $\oplus$ bodů $P, Q \in E$ definujeme takto:

1. $P = O \Rightarrow P + Q = Q$,
2. $P = (p_1, p_2), Q = (q_1, q_2), p_1 = q_1, p_2 = -q_2$, potom $P \oplus Q = O$,
3. pokud nenastal ani jeden z bodů předtím, pak pro $P = (p_1, p_2), Q = (q_1, q_2)$

$$\lambda = \begin{cases} \frac{q_2 - p_2}{q_1 - p_1} & , P \neq Q \\ \frac{3p_1^2 + a}{2p_2} & , P = Q \end{cases}$$

$P \oplus Q = (r_1, r_2)$, kde $r_1 = \lambda^2 - p_1 - q_1$ a $r_2 = \lambda(p_1 - r_1) - p_2$.

Takto definovaná operace a k ní daná množina potom splňuje podmínky pro abelovskou grupu.

Kapitola 2

RSA

V této kapitole si ukážeme příklad jedné šifry, která je založena na problému faktorizace a je používána dodnes. Tato šifra se jmenuje RSA. V 70. letech ji navrhli Ron Rivest, Adi Shamir a Leonard Adelman a šifra nese jméno právě podle nich, respektive název se skládá z iniciálů autorů. Je řazena do kategorie asymetrických šifer, tedy používá dva navzájem různé klíče. Jeden klíč se používá pro šifrování (veřejný) a druhý klíč (soukromý) pro dešifrování. Tato šifra je dnes stále využívána a při dostatečně dlouhém klíči, jak se můžeme dočíst v [4], je stále bezpečná.

2.1 Symetrické a asymetrické šifry

Dříve než začneme s popisem konkrétního algoritmu, tak si uvedeme základní rozdělení šifer.

2.1.1 Symetrické šifry

Symetrické šifry pracují pouze s jedním klíčem, který je určen pro šifrování a dešifrování. Pokud vyžadujeme šifrovanou komunikaci například mezi dvěma zařízeními, tak je nutné si nejdříve domluvit společný klíč, který se bude právě používat pro šifrování a dešifrování. K tomu můžeme využít například algoritmus Diffieho–Hellmana pro výměnu klíčů, který zajistí, že po nešifrovaném kanále si můžeme domluvit tajný klíč.

Výhoda těchto šifer spočívá v tom, že nejsou příliš náročné na různé výpočty. Nevýhoda naopak je právě domluva tajného klíče, kdy může být tajný klíč odposlechnut a zneužit k dešifrování tajných zpráv.

Mezi známé symetrické šifry například patří:

- **AES** – dnes jeden z nejvíce využívaných a nejbezpečnějších algoritmů,
- **3DES** – dříve používaná šifra, kterou i dnes je možné nalézt ve starších systémech, avšak už se od ní ustupuje,

2.1.2 Asymetrické šifry

Asymetrické šifry naopak využívají různých klíčů pro šifrování a dešifrování. Klíč pro šifrování většinou nazýváme veřejným klíčem, pro dešifrování pak nazýváme jako soukromý. Není tedy potřeba využívat různých technik, jak si domluvit tajný klíč, ale můžeme jednoduše zaslat svůj veřejný klíč.

Výhodou těchto šifer je, že soukromý klíč nikde nedistribuujeme, takže útočník pak hůř bude získávat klíč k dešifrování. Nevýhodou však je, že je potřeba provést více výpočtů a tím se stávají tyto šifry pomalejší oproti symetrickým šifrám.

Mezi známými zástupci asymetrických šifer jsou:

- **RSA** – dnes běžná šifra, která se využívá společně se symetrickými šiframi, níže si jí popíšeme více podrobněji. Tato šifra se zakládá na problému faktorizace,
- **ElGamal** – je méně využívaná šifra oproti RSA, protože šifrovaná data jsou dvakrát delší než data šifrovaná. Tato šifra se zakládá na problému výpočtu diskrétního logarimu.

2.2 Princip RSA

Pro úplnost si nejdříve zavedeme definici Eulerovy funkce, která je nutná pro princip algoritmu RSA.

Definice 26. (Eulerova funkce) Eulerova funkce $\varphi(n) : \mathbb{N} \to \mathbb{N}$, kde $n \in \mathbb{N}$, udává počet kladných celých čísel menších nebo rovných n, která jsou nesoudělná s n.

Definice 27. (Eulerova věta) Nechť je dáno $n \in \mathbb{N}$ a $a \in \mathbb{N}$ takové, že $\gcd(a, n) = 1$, potom platí:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Jedná se o zobecnění Malé Fermatovy věty[1].

Nyní si ukážeme, jak se generují klíče pro šifrování a dešifrování:

1. Zvolíme náhodně dvě prvočísla p a q.
2. Následně vypočítáme $n = pq$.
3. Vypočítáme hodnotu Eulerovy funkce $\varphi(n)$. Tato hodnota bude jistě rovna $(p-1)(q-1)$.
4. Zvolíme číslo e, pro které platí: $1 < e < n$ a $\gcd(\varphi(n), e) = 1$.

[1] $a^{p-1} \equiv 1 \pmod{p}$, pro prvočíslo p a $\gcd(p, a) = 1$

5. Vypočítáme číslo d, pro které platí $de \equiv 1 \pmod{\varphi(n)}$

Veřejným klíčem nazýváme dvojici $V_k = (n, e)$ a soukromý klíč nazýváme dvojici $S_k = (n, d)$. Podle [4] je vhodné, aby číslo n mělo nejméně 2048 bitů.

Veřejný klíč může být rozdistribuován komukoliv po otevřeném kanále, neboť veřejným klíčem můžeme zprávu zašifrovat, ale již není možné jím zprávu dešifrovat. Z toho tedy vyplývá, že soukromý klíč musíme mít uchován pouze u sebe a nikam ho nedistribuovat.

2.3 Šifrování a dešifrování komunikace pomocí RSA

Nyní si ukážeme, jak pomocí RSA lze zprávu šifrovat. Tuto operaci ukážeme na běžném příkladu, kdy Alice a Bob si chtějí vyměnit zprávu po otevřeném kanále.

Nejdříve si Bob vygeneruje svojí dvojici veřejného a soukromého klíče, nazvěme si tuto dvojici:

$$V_B = (n_B, e_B) \text{ a } S_B = (n_B, d_B)$$

Alice a Bob si vymění své veřejné klíče, tedy Alice zná pouze V_B. Uvažujme konkrétní hodnoty:

$$V_B = (65, 5), S_B = (65, 29)$$

Alice bude chtít Bobovi poslat zprávu O_T. Pro jednoduchost chceme poslat $O_T = 10$. Tuto zprávu tedy zašifruje pomocí veřejného klíče Boba V_B. Šifrovanou zprávu získáme jako

$$C_T = O_T^{e_B} \bmod n_B$$

$$C_T = 10^5 \bmod 65 = 30$$

Bob přijme C_T a začně zprávu dešifrovat.

$$O_T = C_T^{d_B} \bmod n_B$$

$$O_T = 30^{29} \bmod 65 = 10$$

. Zprávu jsme úspěšně tedy dešifrovali, analogicky pokud bude chtít Bob poslat stejnou zprávu Alici, tak využije jejího veřejného klíče.

2.4 Využití RSA

RSA se dneska využívá společně s ostatními symetrickými šiframi. Využívá se například pro digitální podpisy zpráv. Digitální podpis je vytvořen tak, že ke zprávě je přidána takzvaná haš, která se pomocí soukromého klíče *zašifruje*.

Na druhé straně je potom možné právě veřejným klíčem haš opět dešifrovat a můžeme si tak porovnat, že jsme přijali nepodvrženou zprávu.

Dalším využitím může být také již zmíněná výměna klíčů, kde zúčastněné strany pro šifrovanou komunikaci se drží těchto kroků:

1. Zúčastněné strany si vygenerují své dvojice klíčů (veřejný a soukromý).
2. Zúčastněné strany si rozešlou navzájem veřejné klíče.
3. Každý jedinec si vygeneruje klíč, který chce sdílet s ostatními.
4. Každý vygenerovaný klíč zašifruje veřejným klíčem příjemce a zašle ho takto všem.
5. Jakmile mají všichni zúčastnění rozeslané všechny tajné klíče, tak si je podle předem daného pravidla zřetězí a vytvoří tím společný tajný klíč.

2.5 Útoky vedené na RSA

Na tento kryptosystém jsou vedené i jiné útoky než pomocí faktorizace čísel. Ty si alespoň z části nyní nastíníme.

2.5.1 Časový útok

Tento útok je založen na měření času určitých výpočetních operací, které jsou potřeba při šifrování nebo dešifrování. Při těchto úkonech počítáme modulární mocninu. K tomu může být využit například algoritmus *square and multiply* popsaný v algoritmu 1.

Algorithm 1 Pseudokód SquareAndMultiply

```
1:  function SquareAndMultiply(exponent e, number m, modulus n)
2:                                          ▷ Chceme vypočíst m^e (mod n)
3:      l = e
4:      tmp = m
5:      ret = 1
6:      while l ≠ 0 do
7:          if d_i = 1 then ret = |tmp · ret|_n
8:          end if
9:          tmp = |tmp^2|_n
10:         l = ⌊l/2⌋
11:     end while
12:     return ret
13: end function
```

Měříme časové odchylky v každém cyklu. Vždy když časová odchylka bude delší, signalizuje nám to, že daný bit je nastaven na hodnotu 1. V opačném

případě je nastaven na 0. Je však také nutné odhadnout, v jakých intervalech daná smyčka probíhá.

2.5.2 Odběrová analýza

Další možností jak získat utajovaný klíč je pomocí odběrové analýzy. Zde máme nejméně dva způsoby, jak odběrovou analýzu provést. Prvním způsobem je jednoduchá odběrová analýza, kde útok probíhá tak, že si změříme spotřebu při dešifrování. Následně zanalyzujeme spotřebu. Tam, kde modul počítal nejvíce, bude v přijatém signálu mít velkou spotřebu, kde méně, tam naopak menší. Je opět důležité si nalézt v odebraném signálu, kde začíná cyklus při výpočtu. Druhým způsobem je diferenciální odběrová analýza. Zde útok probíhá tak, že změříme více průběhů šifrování nebo dešifrování a zanalyzujeme signál ve stejných okamžicích v čase napříč všemi průběhy.

Kapitola 3

Popis faktorizačních algoritmů

Nyní si popíšeme, jak použité faktorizační algoritmy fungují. V této kapitole se nachází jejich obecný popis a přesná implementace bude popsána později v 6. kapitole.

3.1 Pollardova ρ–metoda

Nejdříve začneme s jednou z jednodušších metod. Tato metoda byla publikována již v 70. letech 20. století jako velmi efektivní pro faktorizaci čísel v [5]. Efektivní z toho důvodu, jakou má časovou složitost, kterou si níže ukážeme.

Mějme číslo n, které chceme faktorizovat. Dále mějme množinu $M = \{0, 1, \ldots, n-1\}$ a zobrazení $f : M \to M$. Snažíme se generovat body posloupnosti pomocí zobrazení f tak, že si zvolíme počáteční bod posloupnosti libovolné $x_0 \in M$ a další body definujeme jako $x_i = f(x_{i-1})$ pro $i \in \mathbb{N}$. Po získání těchto hodnot hledáme největšího společného dělitele absolutní hodnoty rozdílu těchto bodů a faktorizovaného čísla n, tedy $d = \gcd(|x_{2i} - x_i|, \mathrm{n})$. Zde nám mohou nastat tři možnosti:

1. $d = 1$, potom pokračujeme k výpočtu další dvojice bodů posloupnosti,
2. $d = n$, potom číslo d je násobkem faktorizovaného čísla n a musíme zvolit nové x_0 a začít s výpočtem od začátku,
3. $1 < d < n$, potom jsme našli faktor čísla n.

Algoritmus 2 popisuje pseudokódem, jak tato metoda lze realizovat. Vstupem je číslo n, které chceme faktorizovat a zobrazení f, pro generování bodů posloupnosti.

Algorithm 2 Pseudokód Pollardovy ρ–metody

```
1: function PollardRho(n, f)
2:     Vyber libovolné x ∈ {0, 1, ..., n − 1} ▷ Zde si uchováváme hodnotu x_i
3:     y = x                                  ▷ Zde si uchováváme hodnotu x_2i
4:     divisor = 1
5:     while divisor = 1 do
6:         x = f(x) (mod n)
7:         y = f(f(y)) (mod n)
8:         divisor = gcd(|y − x|, n)
9:     end while
10:    if divisor ≠ n then return divisor
11:    end if
12:    goto 2
13: end function
```

3.1.1 Příklad použití

Uveďme příklad. Snažíme se faktorizovat číslo $n = 1517 = 37 \cdot 41$. Jako zobrazení f si zvolíme polynom $f(x) = x^2 + 1$ (mod n). Dále zvolíme počáteční hodnotu $x_0 = 2$. V tabulce 3.1 můžeme vidět průběh výpočtu.

Číslo kroku i	x_i	x_{2i}	$\lvert x_{2i} - x_i \rvert$	gcd($\lvert x_{2i} - x_i \rvert$, n)
1	5	26	21	1
2	26	196	170	1
3	677	862	185	37

Tabulka 3.1: Průběh výpočtu Pollardovy ρ – metody

3.1.2 Časová a paměťová složitost

Časová složitost tohoto algoritmu je $O(n^{1/4})$. Odvození této složitosti můžeme najít v [2]. Paměťovou složitost změříme v potřebných bitech na uložení. Uchováváme si 2 hodnoty x, y, respektive, x_i, x_{2i}. Dále potřebujeme uchovávát výsledek největšího společného dělitele v proměnné *divisor*. Paměťová složitost tedy je $O(\lceil \log_2 x \rceil + \lceil \log_2 y \rceil + \lceil \log_2 divisor \rceil)$. Logaritmy jsou zde z důvodu udání velikosti čísla v bitech.

3.2 Pollardova $p - 1$ metoda

Další metodou, kterou také publikoval J. M. Polard v [6], se zakládá na Malé Fermatově větě. Snažíme se využít znalost toho, co víme o řádu nějakého prvku z multiplikativní grupy $\mathbb{Z}_p$, abychom našli faktor p našeho faktorizovaného čísla $n = pq$. Hlavní myšlenka této metody tedy spočívá v tom, že

pokud $p \mid n$, číslo p je prvočíslo a $a \in \mathbb{N}$, potom platí, že pro $a^{p-1} \equiv 1 \pmod{p}$. V důsledku předešlého tvrzení také platí $\gcd(n, a^{p-1} - 1 \pmod{p}) = p$.

Nyní se potřebujeme zaměřit na to, jak takové $p-1$ najít. Předpokládejme nějaké $L \in \mathbb{N}$, které splňuje $(p-1) \mid L$ a $(q-1) \nmid L$. Potom existují nezáporná celá čísla $i, j, k, k \neq 0$ taková, že

$$L = i(p-1), L = j(q-1) + k$$

J. M. Pollard navrhuje, že pro $m = 2, 3, \ldots$ je vhodné volit $a \in \mathbb{N}$ a spočíst $d = \gcd(a^{m!} - 1, n)$, protože pokud $p-1$ je součinem malých prvočísel, potom bude $p-1$ dělit $m!$ již pro relativně malé m. Pro d nám opět vyvstávají tři možnosti. Pokud $d = 1$, pokračujeme k vyššímu m, pokud $d = n$, d je násobkem n a volíme jiné a, pokud $1 < d < n$, potom d je faktorem n.

Algoritmus 3 nám opět popisuje, jak se celá tato metoda dá přepsat do pseudokódu. Existují i jiné pseudokódy, jak tento algoritmus zapsat a získat stejný výsledek. K vidění může být například v [7].

Algorithm 3 Pseudokód Pollardovy $p-1$ metody

```
1:  function PollardP(n)
2:      Vyber libovolné a ∈ {2, 3, ..., n − 1}
3:      d = 1
4:      r = 1
5:      while d = 1 do
6:          d = gcd(n, a^r! − 1 (mod n))
7:          r = r + 1
8:      end while
9:      if d ≠ n then
10:         return d
11:     end if
12:     goto 2
13: end function
```

3.2.1 Příklad použití

r	$a^{r!} - 1 \pmod{n}$	$\gcd(n, a^{r!} - 1 \pmod{n})$
1	1	1
2	3	1
3	63	1
4	26	13

Tabulka 3.2: Průběh výpočtu Pollardovy $p-1$ metody pro $a = 2$

Uveďme opět příklad, kde budeme faktorizovat číslo $n = 143 = 11 \cdot 13$. Zvolme si proměnnou $a = 2$. V tabulce 3.2 můžeme vidět, jak bude vypa-

dat průběh výpočtu. Výpočet hodnoty $a^{r!} - 1$ můžeme počítat v modulu n. Takovýto způsob zajistí lepší paměťové nároky.

3.2.2 Časová a paměťová složitost

Nyní si odvodíme časovou a paměťovou složitost tohoto algoritmu. Rozebereme si jednotlivé kroky algoritmu 3. Začneme cyklem, který proběhne r-krát. V tomto cyklu počítáme jednak faktoriál hodnoty r a poté největšího společného dělitele. Pro výpočet faktoriálu si vždy můžeme uchovat dosavadní vypočtený faktoriál a následně ho jen vynásobit hodnotou o 1 vetší, tedy $(n+1)! = (n+1)n!$. To je opět konstantní operace vynásobit dvě čísla mezi sebou. Dále potřebujeme vypočítat hodnotu $a^{r!} - 1$ (mod n), to můžeme vypočíst pomocí algoritmu *square and multiply*, jehož časová složitost je $O(\log exponent)$. Složitost výpočtu $\gcd(a, b)$ je $O(\ln^2(max(a, b)))$, v popsaném algoritmu to tedy bude vždy $O(\ln^2 n)$, protože nám platí $a^{r!} - 1 < n$. Toto se nám provede tedy celé r-krát, takže celková složitost je $O(r(\log(r!) + \ln^2(n)))$. Takto by se mohlo zdát, že časová složitost je polynomiální, avšak je nutné si uvědomit, že pod hodnotou r se skrývá 2^m bitových operacích, kde m je počet bitů dané hodnoty. Paměťová složitost zde bude opět $O(1)$, protože nám opět stačí pouze ukládat jednotlivé mezivýsledky do proměnných.

3.3 Lenstrův algoritmus

Tento algoritmus se odvíji od Pollardovy $p-1$ metody, avšak s tím rozdílem, že využívá eliptických křivek a operace sčítání místo postupného mocnění. Nyní budeme také pracovat v okruhu $R := \mathbb{Z}/(n)$, kde n je faktorizované číslo.

Uvažujme eliptickou křivku $E : y^2 = x^3 + ax + b$, kde $(x, y), (a, b) \in R^2$. Nemáme zaručeno, že operace sčítání nad daným okruhem je dobře definovaná. Pro $P, Q \in E$ se může stát, že $P \oplus Q$ nelze spočítat, neboť nemusí existovat inverze. To však bude v našem algoritmu klíčová chvíle, neboť je jistá šance, že jsme nalezli hledaný faktor. Objevení faktoru jednoduše ověříme vypočtením největšího společného dělitele invertovaného čísla d a čísla n.

Rozeberme si algoritmus 4. Nejdříve potřebujeme zvolit náhodně eliptickou křivku E a bod P. Dále vypočítáváme v cyklu násobek i-tý násobek bodu P. Tímto způsobem postupně počítáme $i!P$, protože zde opět využíváme Pollardova návrhu, který byl zmíněn již v předešlé metodě. Následně při výpočtu inverze čísla d v R mohly nastat tři možnosti. Inverze byla nalezena a pokračujeme tedy k vyššímu i. Další možností je, že d je násobkem čísla n, potom tedy volíme novou eliptickou křivku E a bod P a začínáme s cyklem od začátku. Poslední možností je, že $1 < d < n$, potom tedy $\gcd(d, n)$ je hledaným faktorem.

Algorithm 4 Pseudokód Lenstrova algoritmu

```
1: Zvolme eliptickou křivku E a její bod P.
2: for i = 2, 3, ... do
3:     Q = iP
4:     if Výpočet bodu Q selhal then
5:         if d = n then
6:             goto 1
7:         else if d > 1 then
8:             gcd(d, n) je hledaným faktorem.
9:         end if
10:    end if
11:    P = Q
12: end for
```

3.3.1 Příklad použití

Nyní si opět uvedeme příklad, jak vypadá průběh výpočtu. Kroky můžeme vidět v tabulce 3.3. Zvolme si jako číslo $n = 221 = 13 \cdot 17$.

i	$P(x, y)$	$\gcd(d, n)$
1	$(116, 75)$	1
2	$(44, 53)$	1
3	$(157, 100)$	1
4	$(89, 36)$	1
5	SELHALO	17

Tabulka 3.3: Faktorizace čísla $n = 221$ pomocí Lenstrova algoritmu pro eliptickou křivku s $a = 47, b = 200$

3.3.2 Časová a pamětová složitost

Časová složitost tohoto algoritmu je $O(e^{\sqrt{2((\ln n)(\ln \ln n))}})$. Její odvození můžeme nalézt v [7]. Využitá paměť paměť nebude opět příliš velká, jelikož si pouze držíme aktuální bod (roznásobený), což jsou 3 hodnoty, respektive souřadnice, nazvěme si je P_x, Py, Pz. Dále si potřebujeme držet informaci o eliptické křivce. Pro tu si uchováváme 2 parametry a a b. Tedy výsledná paměťová složitost v bitech je $O(\lceil \log_2 P_x \rceil + \lceil \log_2 P_y \rceil + \lceil \log_2 P_z \rceil + \lceil \log_2 a \rceil + \lceil \log_2 b \rceil)$.

3.4 Kraitchikovo schéma

Než začneme s popisem konkrétních číselných sít, ukážeme si, jak obecně fungují. Jak se můžeme dozvědět z [8], tak tyto algoritmy vycházejí z takzvaného

Kraitchikova schématu. Idea spočívá v tom, že se budeme snažit násobit kongruence typu $U \equiv V \pmod{n}$, kde $U \neq V$ a číslo n je to, které chceme faktorizovat. Jejich produktem získáme speciální kongruenci $X^2 \equiv Y^2 \pmod{n}$ a z této kongruence je jistá šance, že největší společný dělitel čísel $(X \pm Y, n)$ bude netriviálním faktorem čísla n. V bodech májí algoritmy těchto pár částí:

1. Vygeneruj kongruence $U \equiv V \pmod{n}$,
2. Stanovení úplné nebo částečné faktorizace U a V pro některé z kongruencí,
3. Stanovení podmnožiny faktorovaných kongruencí, které mohou být roznásobeny k získání speciální kongruence $X^2 \equiv Y^2 \pmod{n}$,
4. Výpočet největšího společného dělitele dvojice čísel $(X - Y, n)$.

Pro kompletnost si uvedeme příklad. Chceme faktorizovat číslo $n = 33 = 3 \cdot 11$. Budeme postupovat tedy podle popsaných vygenerujeme si kongruence splňující bod 1. Z nich se zaměříme na tyto kongruence[2]:

$$32 \equiv -1 \pmod{n},\ 8 \equiv -25 \pmod{n}$$

Nyní přejdeme k bodu 2. Tyto kongruence můžeme zapsat jako:

$$2^5 \equiv -1 \pmod{n},\ 2^3 \equiv -5^2 \pmod{n}$$

Pokud tyto dvě kongruence roznásobíme (bod 3), získáme:

$$2^5 \cdot 2^3 \equiv -1(-5^2) \pmod{n} \Rightarrow 2^8 \equiv 5^2 \pmod{n}$$

A na závěr z bodu 4 nám potom plyne $11 = \gcd(2^4 - 5, 33)$, tedy nalezli jsme faktor, kterým je číslo 11.

Pokud chceme používat toto schéma, je nutné, aby faktorizované číslo nebyl perfektním čtvercem nebo prvočíslo. Důležité je také si uvědomit, že generování potřebných kongruencí může být výpočetně náročné. To se nám ovšem snaží zjednodušit číselná síta, která si nyní popíšeme.

3.5 Kvadratické síto

Na úvod si zde uvedeme definice pro kompletnost.

Definice 28. (Hladké čísla, hladké polynomy, faktorová báze) Mějme $n \in \mathbb{Z}$. Číslo n nazveme B–hladké, jestliže má všechny prvočíselné dělitele menší nebo rovny B. Mějme dán polynom f nad konečným tělesem. Polynom f nazveme B–hladký, jestliže jej lze faktorizovat na ireducibilní polynomy stupně maximálně B. Mějme dán okruh R a prvek $x \in R$. Říkáme, že prvek x je hladký nad faktorovu bází $F = \{p_1, p_2, \ldots, p_t\} \subset R$, pokud prvek x lze vyjádřit jako součin prvků z množiny F.

[2]Mohli bychom vybrat i jiné vyhovující kongruence, avšak z těchto získáme výsledek

Kvadratické síto popsal Carl Pomerance v [8]. Rozšířil tím výše zmíněné Kraitchikovo schéma a využil Dixonova algoritmu. Tento algoritmus se považoval za nejrychlejší velmi dlouhou dobu, protože nepoužívá výpočetně náročné operace jako je tomu v případě Lenstrova algoritmu při násobení bodu.

Kvadratické síto se snaží hledat kongruence v určitém tvaru a pro nějakou hranici. Popišme si tedy jak to funguje. Jak již bylo zmíněno tento algoritmus vychází z Kraitchikova schématu, pro které je důležité, aby faktorizované číslo nebylo perfektním čtvercem a prvočíslo. Kvadratické síto používá pro hledání kongruencí kvadratický polynom:

$$Q(x) = (x + \lfloor\sqrt{n}\rfloor)^2 - n$$

Našim cílem je získat tuto kongruenci:

$$Q(x_{i_1})Q(x_{i_2})\cdots Q(x_{i_r}) \equiv (x_{i_1}x_{i_2}\cdots x_{i_r})^2 \pmod{n}$$

Nyní si ukážeme, jak takový algoritmus celý vypadá a na něm si popíšeme, co z výše uvedené kongruence znamenají nepopsané členy x_i a r.

Algorithm 5 Pseudokód kvadratického síta

```
1:  function QuadraticSieve(n, B)
2:      faktor_báze = {p_i | LegendreSymbol(n, p_i) = 1, p_i < B, i ∈ ℕ}
3:      relace = ∅
4:      definuj interval síta = [−M, M]          ▷ Zde začíná prosévací část
5:      while #relace< #faktor_báze do
6:          Vypočítej Q(x_i) pro x_i ∈ [−M, M]
7:          if Q(x_i) je hladká nad faktorovou bází then
8:              ulož Q(x_i) mezi relace
9:          end if
10:     end while                                ▷ Konec prosévací části
11:     Sestav matici 𝔸, kde každá řádka je sestavená z exponentů faktorů
        r ∈ relace
12:     Najdi množinu řešení S homogenní soustavy lineárních kongruencí
        z⃗𝔸 ≡ 0⃗ (mod 2)
13:     for s ∈ S do
14:         x^2 = ∏_{Q(x_i)∈s} Q(x_i)
15:         y^2 = ∏_{x_i∈s} x_i
16:         if 1 < gcd(x ± y, n) < n then return gcd(x ± y, n)
17:         end if
18:     end for
19:     return Nepovedlo se faktorizovat
20: end function
```

V pseudokódu 5 můžeme vidět fungování tohoto algoritmu. Jsou zde dvě hodnoty B a M. Jejich přesné hodnoty si určíme níže, v popisu složitosti časové

a paměťové. Berme je nyní pouze jako nějaké hranice. V kroku 2 vidíme, že nejdříve je nutné si vygenerovat faktor bázi. Ta je sestavena z prvočísel, která jsou menší než nějaká hranice B a zároveň Legendreův symbol čísel n a prvočísla je roven 1. Následně hledáme takové relace $Q(x_i)$, které jsou hladké nad faktor bází neboli $Q(x_i) = p_1^{k_1} p_2^{k_2} \cdots p_f^{k_f}$ pro $p_f \in$ faktor báze, $k \in \mathbb{N}$. Následně si sestavíme matici z relací. V každém řádku budeme mít vektor, který reprezentuje exponenty prvočísel z faktor báze pro rozklad daného $Q(x)$, tyto exponenty jsou nad $\mathbb{Z}_2$. Například, mějme faktor bázi rovnu množině $\{-1, 2, 5, 11, 19\}$, a rozklad $Q(x) = 2 \cdot 11^2 \cdot 19$. Potom v řádce máme vektor, který vypadá následovně $(0, 1, 0, 0, 1)$. Do faktorové báze přidáváme ještě číslo -1, kvůli možným zaporným hodnotám polynomu $Q(x)$. Ve sloupcích potom této matice máme exponenty prvočísel z faktor báze. Dále nalezneme množinu řešení S homogenní soustavy rovnic z kroku 12. Každé řešení představuje množinu vybraných $Q(x_i)$, jejichž produktem je hledaný čtverec. Nakonec postupně procházíme všechna řešení a utváříme z nich hledané čtverce, z jejichž rozdílů kořenů hledáme největšího společného dělitele.

Takto funguje ve vší obecnosti tento algoritmus. Pojďme se na chvíli zaměřit na řešení rovnice. Může se zde naskýtat otázka, proč zrovna musíme takto řešit požadovanou homogenní rovnici. Je to z toho důvodu, že chceme nalézt součin podmnožiny nějakých $Q(x_i)$, jejichž souřadnice (exponenty) jsou liché. Tímto v podstatě říkáme, že hledáme takovou podmnožinu vektorů, jejichž součet všech souřadnic je sudý.

Mějme danou množinu všech $Q(x)$. Snažíme se nalézt řešení kongruence

$$Q(x_1)a_1 + Q(x_2)a_2 + \cdots + Q(x_n)a_n \equiv 0 \pmod 2, a_i \in \mathbb{Z}_2$$

Pokud přepíšeme všechna $Q(x_i)$ na dané vektory (nazvěme si je třeba $\vec{z}_i$), jak je výše uvedeno, získáme z původní konruence

$$\vec{z}_1 a_1 + \vec{z}_2 a_2 + \cdots + \vec{z}_n a_n \equiv 0 \pmod 2$$

To odpovídá homogenní soustavě lineárních kongruencí z algoritmu 5, kroku 12

$$\vec{z}\mathbb{A} \equiv \vec{0} \pmod 2$$

Algoritmus existuje i v jiných variantách, kdy se snažíme optimalizovat prosívací část, zejména prosívací interval. Například existuje další varianta zvaná MPQS(*Multiple Polynomial Quadratic Sieve*), která nepracuje pouze s jedním polynomem, ale s více polynomy najednou. Těmito optimalizacemi se tu však zabývat nebudeme, postačí nám pouze základní varianta tohoto algoritmu, ale je vhodné je alespoň zmínit.

Tyto algoritmy dokážou nalézt netriviální faktor s pravděpodobností 2/3. Vycházejme z posledního bodu algoritmu tedy $x^2 \equiv y^2 \pmod n$. Jestliže $n = pq$, potom platí:

$$x^2 \equiv y^2 \pmod{pq} \Leftrightarrow pq|(x^2 - y^2) \Leftrightarrow pq|(x + y)(x - y) \Leftrightarrow$$

$p\|(x+y)$	$p\|(x-y)$	$q\|(x+y)$	$q\|(x-y)$	gcd $x+y, n$	gcd $x-y, n$	Úspěch
lze	lze	lze	lze	n	n	ne
lze	lze	lze	nelze	n	p	ano
lze	lze	nelze	lze	p	n	ano
lze	nelze	lze	lze	n	q	ano
lze	nelze	lze	nelze	n	1	ne
lze	nelze	nelze	lze	p	q	ano
nelze	lze	lze	lze	q	n	ano
nelze	lze	lze	nelze	q	p	ano
nelze	lze	nelze	lze	1	n	ne

Tabulka 3.4: Možné výstupy pro kongruenci $x^2 \equiv y^2 \pmod{N}$, tabulka převzata z [9]

.

$$(p|(x+y) \vee p|(x-y)) \wedge (q|(x+y) \vee q|(x-y))$$

Z tabulky 3.4 můžeme vidět, že v šesti z devíti případů nalezneme hledaný faktor. Proto tedy je pravděpodobnost nalezení určitého faktoru 2/3.

3.5.1 Příklad použití

Mějme číslo $n = 221 = 13 \cdot 17$, které chceme faktorizovat. Nejdříve si určíme náš polynom, kterým bude $Q(x) = (x + \lfloor\sqrt{n}\rfloor)^2 - n$, tedy pro $n = 221$ bude nasledující $Q(x) = (x + 14)^2 - 221$. Zvolme si naší faktorovu bázi do čísla 37 $F = \{-1, 5, 7, 11, 31, 37\}$. Dále vypočítáme některé hodnoty:

i	x_i	$Q(x_i)$
1	-8	-185
2	-7	-172
5	-4	-121
11	2	35

Tabulka 3.5: Zvolené hodnoty z prosívacího intervalu

První hodnota v tabulce je hladká na faktorové bázi. Ověříme si to rozepsáním na prvočísla

$$-185 = -1 \cdot 5 \cdot 37$$

Další hodnota již ale hladká na faktor bázi není. Pokud si jej opět rozepíšeme, získáme

$$-172 = -1 \cdot 2^2 \cdot 43$$

Zbývající hodnoty hladké jsou. Jejich roznásobením získáme kongruenci

$$Q(x_1)Q(x_5)Q(x_{11}) \equiv (-8 \cdot (-4) \cdot 2) \pmod{n}$$

což jsou naše $x^2 \equiv y^2 \pmod{n}$. Kořen hodnoty $y^2 = 30$ není na první pohled zřejmý. Musíme nalézt kvadratické reziduum q, které splňuje $y^2 \equiv q \pmod{n}$. Jeho nalezení můžeme učinit například postupným zkoušením hodnot z $\mathbb{Z}_{221}$. Nalezené $q = 76$, kořen čtverce x^2 je na první pohled zřejmý, neboť $x^2 = 64 = 2^6 \Rightarrow x = 2^3$. Získáním největšího společného dělitele $\gcd(76-8, 221)$ získáme faktor 17. Takovýto způsob je samozřejmě neefektivní. V programu se snažíme nacházet vždy pouze perfektní čtverce.

3.5.2 Časová a pamětová složitost

Časová a paměťová složitost algoritmu se odvíjí hlavně od velikostí faktorové báze a prosévacího intervalu. Uveďme si tedy jaké by měli být již zmiňované hodnoty velikosti faktorové báze B a hodnoty krajů prosévacího intervalu M. Ty jsou již popsané v některých pracích (viz [10]), avšak mnoho těchto prací se shodují na těchto hodnotách:

$$B = (e^{\sqrt{\ln(n)\ln\ln(n)}})^{\sqrt{2}/4}$$

$$M = B^3 = (e^{\sqrt{\ln(n)\ln\ln(n)}})^{3\sqrt{2}/4}$$

Paměťová složitost kvadratického síta potom bude $O(B)$, protože hodnoty faktorové báze si uchováváme. Hodnoty z intervalu není nutné uchovávat všechny, ale postupně je vybírat.

Základní časová složitost kvadratického síta bez různých optimalizací se odhaduje jako $O(e^{\sqrt{1.125\ln(n)\ln\ln(n)}})$. Složitost je skoro stejná jako tomu bylo u Lenstrova algoritmu, avšak jak již bylo řečeno, kvadratické síto používá méně náročné operace a proto je považováno za rychlejší.

3.6 Obecné číselné síto

Posledním vybraným algoritmem a taky doposud nejrychlejším známým, je číselné síto nad obecným tělesem[3]. V mnoha krocích je podobný kvadratickému sítu, avšak s tím rozdílem, že pracuje nad obecným tělesem. Dokonce má i stejnou pravděpodobnost pro nalezení hledaného faktoru.

Uveďme si na začátek pár důležitých definic, kterých následně využijeme.

Tvrzení 3.6.1. Nechť je dán polynom $f(x) \in \mathbb{Z}[x]$ a jeho komplexní kořen $\alpha \in \mathbb{C}$ a $m \in \mathbb{Z}/n\mathbb{Z}$ takové, že $f(m) \equiv 0 \pmod{n}$, pro $n \in \mathbb{N}, n \geq 2$, potom existuje jednoznačné zobrazení $\phi : \mathbb{Z}[\alpha] \to \mathbb{Z}n\mathbb{Z}$, pro které platí:

1. $\phi(ab) = \phi(a)\phi(b) \forall a, b \in \mathbb{Z}[\alpha]$
2. $\phi(a+b) = \phi(a) + \phi(b) \forall a, b \in \mathbb{Z}[\alpha]$

[3] zkráceně jej budeme nazývat GNFS z anglického General Number Field Sieve

3. $\phi(1) \equiv 1 \pmod{n}$

4. $\phi(\alpha) \equiv m \pmod{n}$

Začněme opět pseudokódem, kterým si popíšeme, jak funguje tento algoritmus.

Algorithm 6 Pseudokód GNFS

1: **function** GNFS(n)
2: Zvol ireducibilní polynom $f(x)$, jehož kořenem je $m \in \mathbb{Z}$ takový, že $f(m) \equiv 0 \pmod{n}, f(x) \in \mathbb{Z}[x]$
3: Zvol velikost faktorové báze a sestav racionální a algebraickou faktorovou bázi a kvadratickou bázi
4: Nalezni relace (dvojice $(a,b)a, b \in \mathbb{Z}$), pro které platí: $\gcd(a,b) = 1$, $a + bm$ je hladké na racionální faktorové bázi a $b^{st(f)}f(a/b)$ je hladké na algebraické bázi.
5: Z relací utvoř matici $\mathbb{A}$
6: Nalezni množinu řešení S homogenní soustavy rovnice $\vec{x}\mathbb{A} \equiv 0 \pmod{2}$
7: Nalezni racionální kořen hodnoty y, kde $y^2 = \prod_{(a,b)\in S}(a + bm)$
8: Nalezni algebraický kořen hodnoty x, kde $x^2 = \prod_{(a,b)\in S}(a + b\alpha)$, kde α je komplexním kořenem polynomu $f(x)$.
9: **return** $\gcd(x - y, n)$, $\gcd(x + y, n)$
10: **end function**

Algoritmus 6 si nyní detailněji rozebereme. V prvním kroce vybíráme polynom $f(x)$ takový, jehož kořenem je $m \in \mathbb{Z}$. Vybrat takový polynom není nijak obtížné. Postup funguje podobně jako zápis čísla v jiné reprezentaci (například binární). Vybereme si nejdříve nějaké m a následně zvolíme polynom tak, že $n = \sum_{i=0}^{d} a_i m^i$, kde $0 < a_i < m$. Tímto získáme polynom $f(x) = a_d x^d + a_{d-1}x^{d-1} + \cdots + a_0$ a zároveň nám bude platit $f(m) \equiv 0 \pmod{n}$. Koeficienty polynomu můžeme navíc upravit takto:

$$a_i = a_i - m$$

$$a_{i+1} = a_{i+1} + 1$$

Jak vybírat takový polynom se zabývají například v práci [11].

Dalším krokem je zvolit vhodnou hranici pro faktorové báze (označíme si je B_r pro racionální faktorovu bázi a B_a pro algebraickou). Velikosti těchto bází jsou dány empiricky a jsou také mnohdy závislé na prosévací části algoritmu. Racionální báze obsahuje všechny prvočísla p_i do námi zvolené hranice. K tomuto prvku ukládáme také $p_i \pmod{m}$, tedy racionální báze je množina:

$$RFB = \{(p_i, p_i \pmod{m}) | p_i < B_r\}$$

Další bázi, kterou vybíráme je algebraická faktorová báze. To je pro nás množina:

$$AFB = \{(p_i, r) | p_i < B_a, f(r) \equiv 0 \pmod{p}\}$$

Nakonec vybíráme kvadratickou bázi. Ta splňuje stejné požadavky, jako algebraická faktorová báze, avšak prvočísla zde jsou větší, než největší prvočíslo v algebraické faktorové bázi.

Navazuje prosévací část. Nejdříve si zvolíme inteval $I = [-C; C]$. Následně si zvolíme fixní hodnotu b, většinou $b = 0$ a v každé iteraci následně tuto hodnotu zvýšíme o 1. Hodnotu a vypočítáme podle uvedeného postupu v algoritmu 7. Tuto část si popíšeme pseudokódem, aby bylo zřetelnější, jak tento postup funguje.

Algorithm 7 Prosévací část GNFS

function síto
    relace $= \emptyset$
    $b = 0$
    while #relace< #RFB+#AFB+#KB+1 **do**
        $a_i = i + bm, e_i = b^{st(f)} f(i/b) \forall i \in [-C; C]$
        for all $(p, r) \in$ RFB **do**
            Odstraň největší mocninu prvočísla p z $a_i, \forall i \in [-C; C]$.
        end for
        for all $(p, r) \in$ AFB **do**
            Odstraň největší mocninu prvočísla p z $e_i, \forall i \in [-C; C]$.
        end for
        Přidej do množiny relací množinu $M = \{(i, b) | a_i = e_i = 1, \gcd(i, b) = 1, i \in [-C; C]\}$
        Zvedni číslo b o 1.
    end while
end function

Z algoritmu 7 můžeme tedy vidět, jak prosévací část funguje. Je velmi podobná té části, která již byla v kvadratickém sítu, avšak zde neoperujeme pouze nad jednou bází, ale nad dvěma bázemi. Tato část v algoritmu je právě ta, která je nejvíce časově náročná, jelikož se prochází celkem široký interval a to vždy do té doby, než se nám naplní množina relací.

Dalším krokem v algoritmu 6 je vytvoření matice $\mathbb{A}$ a následné získání řešení homogenní soustavy rovnic $\vec{x}\mathbb{A} = 0 \pmod 2$. Tuto rovnici jsme si již popsali v kapitole 3.5. Její princip spočívá v tom samém, jako tomu bylo u kvadratického síta. Ukážeme si ale, jak se taková matice z těchto relací skládá opět pseudokódem.

Sestavení matice můžeme vidět v algoritmu 8. Opět vkládáme do řádku exponenty tak, abychom vždy získali požadovaný perfektní čtverec. Tato část není sice až tak náročná časově, ale naopak je velmi náročná paměťově, neboť

Algorithm 8 Sestavení matice GNFS

function SÍTO(Polynom $f(x)$, kořen m polynomu $f(x)$, relace)
  Připravme si nulovou matici $\mathbb{A} \in \mathbb{Z}_2^{\#\text{relace},\#\text{RFB}+\#\text{AFB}+\#\text{KB}+1}$
  for all $(a_i, b_i) \in$ relace **do**
    if $a_i + b_i m < 0$ **then** $\mathbb{A}_{i,0} = 1$
    end if
    for all $(p_j, r_j) \in$ RFB **do**
      exponent = nejvyšší mocnina p_j, která dělí $a_i + b_i m$
      if exponent je lichý **then** $\mathbb{A}_{i,1+j} = 1$
      end if
    end for
    for all $(p_j, r_j) \in$ AFB **do**
      exponent = nejvyšší mocnina p_j, která dělí $(-b_i)^{st(f)} f(-a_i/b_i)$
      if exponent je lichý **then** $\mathbb{A}_{i,1+\#\text{RFB}+j} = 1$
      end if
    end for
    for all $(p_j, r_j) \in$ KB **do**
      if Legendreův symbol dvojice $(a_i + b_i p_j, r_j) \neq 1$ **then** $\mathbb{A}_{i,1+\#\text{RFB}+\#\text{AFB}+j} = 1$
      end if
    end for
  end for
end function

tato matice může mít příliš velké rozměry. Zde by mohlo být spíše paměťově rozumnější využít řídkých matic.

Posledním krokem je určení čtverců hodnot x a y. Množina řešení nám opět říká, které a_i a b_i z relací máme použít. Tím získáme jejich produkt a následně otestujeme, zda rozdíl kořenů těchto čtverců je naším hledaným faktorem.

Tímto máme určený obecný postup, jak tento algoritmus funguje. Nyní následuje příklad, na kterém si objasníme, jak to vlastně celé funguje a jak to použít.

3.6.1 Příklad použití

Nyní použijeme příklad z [12]. Budeme se snažit faktorizovat číslo $n = 45113 = 197 \cdot 229$. První bod algoritmu nám říká, abychom vybrali nějaký vhodný polynom $f(x)$. Zvolme si nejdříve náš kořen polynomu $m = 31$ a rozepišme číslo n v reprezentaci o základu 31.

$$45113 = 31^3 + 1531^2 + 29 \cdot 31 + 8$$

náš polynom bude mít následující tvar:

$$f(x) = x^3 + 15x^2 + 29x + 8$$

Nyní máme náš polynom vybraný a k němu i přiřazen kořen. Dalším krokem je určení našich faktorových bází [4]. Pro RFB si zvolíme hranici prvočísel $B_r = 30$ a pro AFB si zvolíme $B_a = 90$. Báze budou mít následující podobu:

$$RFB = \{(2,2),(3,3),(5,5),(7,7),(11,11),(13,13),(17,17),(19,19),$$
$$(23,23),(29,29)\}$$
$$AFB = \{(2,0),(7,6),(17,13),(23,11),(29,26),(31,18),(41,19),(43,13),$$
$$(53,1),(61,46),(67,2),(67,6),(67,44),(73,50),(79,23),(79,47),(79,73),$$
$$(89,28),(89,62),(89,73)\}$$
$$KB = \{(97,28),(101,87),(103,47),(107,4),(107,8),(107,80)\}$$

Všechny báze máme nastavené, nyní tedy přistoupíme k prosévací části. Nastavíme si interval $I = [-400; 400]$ a budeme hledat takové dvojice (a_i, b_i), které jsou hladké na RFB a AFB. Naše množina relací (dvojice (a, b)) je následující:

$$\{(-73,1),(-13,1),(-6,1),(-2,1),(-1,1),(1,1),(2,1),(3,1),(13,1),(15,1),$$
$$(23,1),(61,1),(1,2),(3,2),(33,2),(2,3),(5,3),(19,4),(14,5),(37,5),(313,5),$$
$$(11,7),(15,7),(-7,9),(119,11),(-247,12),(175,13),(5,17),(-1,19),(35,19),$$
$$(17,25),(49,26),(375,29),(9,32),(1,33),(78,37),(5,41),(9,41)\}$$

Nyní musíme sestavit matici $\mathbb{A}$. Uvedeme si příklad zde jednoho řádku, jak v této matici vypadá. Řádka matice je opět vektor. Rozepišme si, co se v tomto vektoru nachází například pro relaci $(119, 11)$:

$$(\overbrace{0}^{|\operatorname{sgn} a+bm|}, \underbrace{0,0,1,0,0,0,0,0,1,0}_{\text{exponenty RFB}}, \overbrace{0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,}^{\text{exponenty AFB}}$$
$$\underbrace{1,1,0,0,0,0}_{\text{KB Legendre}})$$

Po sestavení matice $\mathbb{A}$ a nalezení řešení homogenní soustavy rovnic uvedené v algoritmu 6 kroku 6, získáme následující řešení:

$$S_0 = \{(-2,1),(1,1),(13,1),(15,1),(23,1),(3,2),(33,2),(5,3),(19,4),(14,5),$$

[4] Budeme zde používat zkratky jejich názvů, jak jsou uvedeny v algoritmech

$$(15,7),(119,11),(175,13),(-1,19),(49,26)\}$$

Nyní vypočítáme požadované x^2 a y^2:

$$y^2 = \prod_{(a,b)\in S_0}(a+bm) = 459997127517951955826063769600 00$$

$$x^2 = (\prod_{(a,b)\in S_0}(a+b\alpha)) \mod f(x) =$$

$$58251363820606365\alpha^2 + 149816899035790332\alpha + 75158930297695972$$

Tedy našimi hledanými kořeny jsou:

$$x^2 = 2553045317222400^2$$

$$y^2 = (108141021\alpha^2 + 235698019\alpha + 62585630)^2$$

Pro hodnotu y použijeme větu 3.6.1, z čehož získáme

$$y^2 = (\phi(108141021\alpha^2 + 235698019\alpha + 62585630)^2)^2 = 111292745400^2$$

Faktory potom získáme pomocí

$$\gcd(45113, 111292745400 + 2553045317222400) = 197$$

a

$$\gcd(45113, 111292745400 - 2553045317222400) = 229$$

3.6.2 Časová a paměťová složitost

Zde budeme vycházet z popisu v [13]. Časová složitost vychází z optimálního výběru stupně generovaného polynomu a v závisloti na stupni polynomu určení velikosti bází. Složitost definujeme jako

$$O(\exp(\sqrt[3]{\frac{64}{9}}(\ln n)^{1/3}(\ln\ln n)^{2/3}))$$

Paměťová složitost je složena z uložení všech faktorových bází a kvadratické báze. K tomu potřebujeme uložit všechny nalezené relace a nakonec potřebujeme uložit matici, kterou skládáme z relací. Výsledná paměťová složitost tedy je:

$$O(\#RFB+\#AFB+\#KB+\#relace+\#relace\cdot(\#RFB+\#AFB+\#KB)) =$$

$$O((\#RFB + \#AFB + \#KB)(\#relace + 1) + \#relace)$$

#cifer	#bitů	P	R	L	Q	G
20	67	10^5	$3.164{\cdot}10^2$	$8.649{\cdot}10^2$	$1.595{\cdot}10^2$	$1.723{\cdot}10^2$
30	100	10^{10}	10^5	$1.930{\cdot}10^5$	$9.210{\cdot}10^3$	$7.567{\cdot}10^3$
40	133	10^{15}	$3.164{\cdot}10^7$	$2.070{\cdot}10^7$	$3.069{\cdot}10^5$	$1.674{\cdot}10^5$
50	167	10^{20}	10^{10}	$1.371{\cdot}10^9$	$7.127{\cdot}10^6$	$2.414{\cdot}10^6$
60	200	10^{25}	$3.164{\cdot}10^{12}$	$6.408{\cdot}10^{10}$	$1.273{\cdot}10^8$	$2.583{\cdot}10^7$
70	233	10^{30}	10^{15}	$2.288{\cdot}10^{12}$	$1.860{\cdot}10^9$	$2.210{\cdot}10^8$
80	266	10^{35}	$3.164{\cdot}10^{17}$	$6.583{\cdot}10^{13}$	$2.311{\cdot}10^{10}$	$1.588{\cdot}10^9$
90	299	10^{40}	10^{20}	$1.585{\cdot}10^{15}$	$2.512{\cdot}10^{11}$	$9.911{\cdot}10^9$
100	333	10^{45}	$3.164{\cdot}10^{22}$	$3.282{\cdot}10^{16}$	$2.438{\cdot}10^{12}$	$5.502{\cdot}10^{10}$
110	366	10^{50}	10^{25}	$5.969{\cdot}10^{17}$	$2.147{\cdot}10^{13}$	$2.766{\cdot}10^{11}$
120	399	10^{55}	$3.164{\cdot}10^{27}$	$9.693{\cdot}10^{18}$	$1.737{\cdot}10^{14}$	$1.277{\cdot}10^{12}$
130	432	10^{60}	10^{30}	$1.424{\cdot}10^{20}$	$1.304{\cdot}10^{15}$	$5.479{\cdot}10^{12}$
140	466	10^{65}	$3.164{\cdot}10^{32}$	$1.914{\cdot}10^{21}$	$9.152{\cdot}10^{15}$	$2.201{\cdot}10^{13}$
150	499	10^{70}	10^{35}	$2.374{\cdot}10^{22}$	$6.049{\cdot}10^{16}$	$8.344{\cdot}10^{13}$

Tabulka 3.6: Přibližný nárůst výpočetního času pro vybrané faktorizační algoritmy

3.7 Předpokládaný běh vybraných faktorizačních metod

Tabulka 3.6 představuje přibližný nárůst všech vybraných faktorizačních algoritmů. Vycházíme zde z asymptotických složitostí těchto metod, které jsme si definovali výše. Nárůst pak počítáme jako:

$$\frac{f(10^{\#cifer}-1)}{f(10^{10}-1)},$$

kde f je funkce definující časovou složitost dané faktorizační metody.

První sloupec tabulky odpovídá počtu cifer v dekadickém zápise složeného čísla, druhý sloupec pak délku v binární soustavě. Další sloupce jsou pak zkratkami pro jednotlivé algoritmy:

- **P** pro Pollardovu $p-1$ metodu,
- **R** pro Pollardovu ρ–metodu,
- **L** pro Lenstrův algoritmus,
- **Q** pro kvadratické síto,
- **G** pro obecné číselné síto.

Funkci pro Pollardovu $p-1$ metodu používáme $\sqrt{n}$, protože v nejhorším případě nám tato metoda udělá právě $\sqrt{n}$ kroků. Například pokud bychom faktorizovali perfektní čtverec, tak bude tento počet kroků potřeba.

Kapitola 4

Paralelizace výpočtů

Nyní si rozebereme obecný postup při paralelizaci vybraných algoritmů. Nejdříve začneme obecným postupem.

4.1 Návrh paralelizace

Algorithm 9 Pseudokód Master–slave algoritmu

```
1: function FACTORIZE_PARALLEL(composite_number)
2:     MPI_Init() // Inicializace procesu
3:     MPI_Comm_rank(MPI_COMM_WORLD, rank) // číslo procesu
4:     if rank = 0 then
5:         _master_part(composite_number) // Rozděl práci
6:         _process_results(composite_number, _result) // Zpracuj výsledky
7:         MPI_Comm_size(MPI_COMM_WORLD, processNumber) // Počet
   procesů
8:         for i ∈ {1, ..., processNumber − 1} do
9:             MPI_Isend(dummy_message, ..., i, TAG_STOP, ...) // Pošli
   zprávu o ukončení výpočtu
10:        end for
11:    else
12:        _slave_part(composite_number) // Početní část
13:    end if
14:    MPI_Finalize()
15: end function
```

Pro paralelizaci jednotlivých výpočtů je zvolen algoritmus *Master–slave* pro práci nad distribuovanou pamětí. Tento algoritmus spočívá v tom, že máme jeden hlavní proces (**master**), který rozděluje úkoly a přijímá výsledky, které mu zašlou pracovní procesy (**slaves**). Hlavní proces většinou určujeme

tak, že jeho hodnota *rank* je 0. Algoritmus 9 popisuje pseudokódem jeho podobu.

Aby nedocházelo ke zbytečnému čekání na odpovědi od ostatních pracujících procesů, hlavní proces také počítá a v průběhu svého výpočtu vždy zkouší, jestli mu nepřišla nějaká řešení od pracujících procesů.

Specifické výpočty algoritmů budou ještě více rozmělněny. To provedeme tak, že využijeme vlákna, které si výpočty také rozdělí.

4.2 Pollardova $p-1$ a ρ–metoda

Pollardovu $p-1$ a ρ–metodu popíšeme dohromady, protože jsou si algoritmy velmi podobné a liší se pouze ve výpočtu. Jak je uvedeno v předešlé kapitole, obě metody v podstatě vyberou hodnotu z předem dané množiny a začnou výpočet.

4.2.1 Paralelizace nad distribuovanou pamětí

Obě metody využijí pracovní procesy tak, že rozdělí interval, ze kterého jsou hodnoty generovány na tolik částí, kolik je procesů. Každý proces tedy pracuje nad určitým intervalem. Postup v krocích je takový, že:

1. Každý proces si určí svůj interval, ze kterého generuje.

2. Každý proces uchovává množinu již vygenerovaných hodnot, aby nedocházelo k duplikátním výpočtům.

3. Každý pracovní proces, zašle nalezený faktor čísla hlavnímu procesu. Hlavní proces naopak přijme od pracovních procesů nalezené faktory.

4.2.2 Paralelizace nad sdílenou pamětí

Každý proces má svůj určitý počet dostupných vláken. Každé toto vlákno si vybere unikátní hodnotu z daného intervalu a začne výpočet. V krocích to vypadá takto:

1. Vlákno si vybere unikátní hodnotu z intervalu.

2. Vlákno spustí výpočet nad unikátní hodnotou.

3. Po dokončení výpočtu zašle faktor čísla. Pokud ho nenalezl, vygeneruje novou hodnotu.

4.3 Lenstrův algoritmus

Paralelizace této metody se podobá předešlým dvoum metodám, avšak s patrným rozdílem. Jelikož bychom nedokázali zamezit procesům, aby si negenerovaly duplikátní eliptické křivky, potřebujeme nějakého správce pro generování.

4.3.1 Paralelizace nad distribuovanou pamětí

Hlavní proces se stará o generování eliptických křivek a bodech na nich. Eliptickou křivku vždy zašle všem pracovním procesům a následně vygeneruje i pro sebe, aby mohl dále počítat. Rozepišme si opět kroky do bodů:

1. Rozešleme eliptické křivky a body na nich.
   a) Pracovní proces si zažádá o vygenerování eliptické křivky a bodu.
   b) Hlavní proces vygeneruje eliptickou křivku pro pracovní proces, který o ní žádá.
2. Pracovní procesy spustí výpočet nad eliptickou křivkou.
3. Zpracování výsledků.
   a) Pracovní proces zašle nalezený faktor. Pokud při hledání byl proces neúspěšný, zažádá si o novou eliptickou křivku a bod.
   b) Hlavní proces přijme nalezený faktor od pracovních procesů, pokud nějaký je. Pokud hlavní proces získal faktor čísla ať už vlastním výpočtem nebo přijetím faktoru od pracovního procesu, rozešle všem pracovním procesům zprávu o ukončení výpočtu.

4.3.2 Paralelizace nad sdílenou pamětí

Vlákna zde využijeme pro práci nad více eliptickýma křivkama a body zároveň. Každé vlákno pracovního procesu si zažádá o unikátní eliptickou křivku a bod. Kroky tohoto algoritmu jsou:

1. Hlavní proces určí vlákno, které se stará o generování eliptických křivek a bodů a toto vlákno čeká na zprávu o vygenerování.
2. Každé vlákno pracovního procesu si zažádá o přidělení unikátní eliptický křivky a bodu a spustí nad ním výpočet.
3. Zpracování výsledků.
   a) Vlákna hlavního procesu během výpočtů kontrolují příchozí zprávu o nalezeném faktoru a případně jej přijímají.

b) Vlákna pracovních procesů kontrolují příchozí zprávu o ukončení výpočtu. Pokud vlákno nalezne faktor čísla, zašle ho hlavnímu procesu.

4.4 Kvadratické síto

Paralelní výpočet se zde liší od ostatních zmíněných algoritmů. Paralelismus zde spočívá v tom, že každý procesor vždy počítá určitou část. Faktorová báze (FB) je získána tak, že hranice nejvyššího prvočísla je rozdělena na tolik částí, kolik máme procesorů a každý procesor začíná od určité části. Uveďme si příklad:

$$FB = \{ \overbrace{2,3}^{\text{Procesor 1}} , \underbrace{5,7}_{Procesor2} , \overbrace{11,13,23}^{Procesor3} \}$$

Další paralelizovanou částí je prosévací část, kde každý proces si vybere pouze část intervalu, ze kterého čerpá hodnoty pro získání relací. Tato část je ještě podrobněji rozmělněna a každé vlákno každého procesu vždy vybírá jednu hodnotu z intervalu.

4.4.1 Paralelizace nad distribuovanou pamětí

Každý proces získá prvky faktorové báze nad pevně daným intervalem. Nalezené prvky si procesy rozešlou a uloží do své paměti, respektive do množiny faktorové báze. Následně spustí prosévací část, kde pracovní vlákna zasílají nalezenou relaci hlavnímu procesu. Pracovní proces končí s výpočtem pokud nalezených relací je dostatek. O tom informuje hlavní proces pomocí zprávy.

Hlavní proces dále pokračuje ve výpočtu homogenní soustavy rovnic a hledá požadovaný faktor čísla. Pracovní procesy v této činnosti nejsou nijak zakomponovány, i když by bylo možné jich využít například pro paralelní výpočet řešení dané soustavy

4.4.2 Paralelizace nad sdílenou pamětí

Vlákna jsou použita pouze pro prosévací část. V prosévací části si každé vlákno vybere jeden člen z intervalu a zkusí jeho hladkost na faktorové bázi. Pokud je vybrané číslo hladké na faktorové bázi, tak v případě, že se jedná o pracovní proces ho zašle hlavnímu procesu. Pokud hlavní proces nalezne takové číslo, tak ho uloží mezi nalezené relace.

4.5 GNFS

U tohoto algoritmu můžeme volit více způsobů, jak jej paralelizovat. Můžeme vygenerovat hlavním procesem pro každé pracovní vlastní polynom společně

s vlastním kořenem. Další možností je také přistupovat k rozmělnění podobně jako tomu bylo u kvadratického síta u jeho prosévací části.

V kapitole 3.6 jsme zmiňovali práci, která se zabývá výběrem polynomu. Daný polynom je pro tento algoritmus velmi důležitý, protože na jeho základě probíhá generace relací atd. Zvolíme tedy možnost vygenerování unikátního polynomu pro každý procesor zvlášť společně s jeho kořenem. To by nám mohlo zajistit větší pravděpodobnost, že máme vybrán správný polynom a tím i získat rozklad čísla.

4.5.1 Paralelizace nad distribuovanou pamětí

Kroky procesů si rozepíšeme:

1. Hlavní proces rozešle všem pracovním procesům polynom a kořen tohoto polynomu. Pracovní vlákna tuto zprávu zpracují.
2. Další krok je podobný sekvenčnímu. Všechny procesy si nastaví hodnoty hranic pro faktorové báze a naleznou tyto faktorové báze.
3. Následuje vyhledání potřebných relací. Všechny procesy si své relace naleznou.
4. Každý proces nalezne řešení homogenní soustavy rovnic a hledají faktor.
5. Pracovní procesy zašlou nalezený faktor hlavnímu procesu a ukončí se. Hlavní proces nalezené faktory zpracuje a ukončí se.

V těchto pěti jednoduchých krocích probíhá celá faktorizace. Jak už bylo zmíněno je to z důvodu pracováním nad více polynomy.

4.5.2 Paralelizace nad sdílenou pamětí

Každý proces využije dostupný počet vláken pro prosévací část, tedy hledání relací. Z algoritmu 7 si každé vlákno vybere jednu hodnotu b a začne na něm výpočet. Vlákna si udržují společnou množinu všech nalezených relací, do které postupně všechna vlákna přidávají nalezené relace. Pokud je jich dostatek, je výpočet vláken ukončen.

Kapitola 5

Vybrané technologie pro implementaci

Dříve než začneme s popisem implementací jednotlivých algoritmů, bude na místě uvést použité technologie, které napomáhají implementaci.

5.1 Výběr programovacího jazyka

Nejdůležitější technologií použitou pro implementaci je jistě vybraný programovací jazyk. Nejdříve jsem se rozhodoval mezi jazyky C++ a Python. Pro oba tyto jazyky jsou implementovány knihovny, respektive moduly, které obsahují efektivní implementace různých matematických operací. Pro Python navíc existuje kompilátor jménem Cython, který je i sám o sobě programovacím jazykem. Dokáže převést Pythonní kód do jazyka C, stejně tak i kód napsaný v jazyce Cython a tím urychlit celý průběh, neboť bude kód zkompilovaný. Ovšem stále zde používá struktury z jazyka Python, které průběh zpomalují. Pro jazyky C++ a Python existuje implementace stejné technologie pro paralelizaci výpočtů MPI, jež byl vybrán a je popsán níže.

Nakonec jsem se rozhodl pro jazyk C++, respektive standard C++17 z dvou důvodů. První důvodem je fakt, že výpočet probíhal na školním klastru STAR (viz [14]), na kterém je připravené prostředí pro použití C++ a MPI. Druhým důvodem je ten, že s použitím MPI a C++ mám větší zkušenosti.

V implementacích algoritmů je nutné pracovat s velkými čísly. Toto nám zajišťuje knihovna GMP, jejíž popis je uveden níže. Kvadratické síto a číselné síto nad obecným tělesem vyžadují práci s maticemi a hledání netriviálního řešení homogenní soustavy $\mathbb{A}x = \theta$ nad $\mathbb{Z}_2$. GNFS algoritmus také navíc vyžaduje práci s prvky nad tělesem $\mathbb{Z}[x]$. Pro toto nám slouží knihovna FLINT. Další knihovna, která pro implementaci algoritmů byla využita je Open MPI. Tato knihovna napomáhá paralelizaci výpočtů.

5.2 GMP

GNU Multiple Precision[15] neboli zkráceně GMP, je knihovna, která nám umožňuje reprezentovat celá čísla s neomezenou přesností a práci s těmito čísly.

Tato knihovna nabízí struktury pro práci s celými čísly, racionálními čísly a také s reálnými čísly. V implementacích budeme používat z této knihovny dvě struktury. První strukturou je **mpz_t**, pro reprezentaci celých čísel a druhou strukturou je **gmp_randstate_t**, která vybraný algoritmus generující náhodná čísla a jeho stav.

Abychom mohli s těmito strukturami pracovat je nutné je v první řadě inicializovat a po skončení práce opět pomocí jisté funkce paměť, kterou si struktura inicializovala, uvolnit. Pro tyto účely jsou zde funkce:

- **mpz_init** – Funkce pro inicializaci struktury **mpz_t**,
- **mpz_init_set** – Funkce pro inicializaci struktury **mpz_t** a následné přiřazení hodnoty z jiné struktury **mpz_t**,
- **gmp_randinit_default** – Funkce pro inicializaci struktury, která drží poslední stav a výchozí algoritmus (definovaný knihovnou GMP), pro generování náhodných čísel,
- **mpz_clear** – Funkce pro uvolnění paměti, která byla alokována pro strukturu **mpz_t**,
- **gmp_randclear** – Funkce pro uvolnění paměti, která byla alokována pro strukturu **gmp_randstate_t**.
- **mpz_inits** – Funkce pro inicializaci více struktur **mpz_t** najednou,
- **mpz_clears** – Funkce pro uvolnění paměti více struktur **mpz_t** najednou.

Knihovna nabízí také kombinaci operací mezi strukturou **mpz_t** a klasickými datovými typy, které jazyk C++ nabízí, jakou jsou **int** nebo **unsigned int**. Tyto funkce vždy končí trojicí **_si** pro práci s **int**, respektive **_ui** pro práci s **unsigned int**. Tyto funkce nebudeme rozepisovat, vždy pouze uvedeme, jestli má daná funkce i takovouto formu.

Pro různá přiřazení jsou tyto použity tyto funkce:

- **mpz_set** – Přiřadí hodnotu do proměnné z jiné struktury **mpz_t**, zde existují i formy pro **unsigned int**, **int**.
- **mpz_set_str** – Nastaví hodnotu proměnné na číslo uvedené pomocí textového řetězce, tato funkce přijímá jakékoliv číslo, které lze zapsat v bázi 2 – 62.

- **mpz_get_str** – Nastaví, případně vrátí (záleží na argumentu předaném funkci), textový řetězec reprezentující číslo v požadované bázi.

Dalšími důležitými funkcemi jsou ty, které nám umožňují provést různé aritmetické operace nebo porovnání dvou čísel:

- **mpz_add** – Funkce pro součet dvou čísel. Existuje také ve variantě pro **unsigned int** a **int**,
- **mpz_sub** – Funkce pro rozdíl dvou čísel. Existuje také ve variantě pro **unsigned int** a **int**,
- **mpz_mul** – Funkce pro násobení dvou čísel. Existuje také ve variantě pro **unsigned int** a **int**,
- **mpz_div_2exp** – Funkce pro bitový posun doprava, tzn. $result = \lfloor \frac{a}{2^b} \rfloor$, kde b je počet bitů,
- **mpz_tdiv_qr** – Funkce pro získání kvocientu q po celočíselném dělení $\frac{n}{d}$ a jeho zbytku r, tedy hodnoty splňující $n = qd + r$, $0 \leq |r| < |d|$.
- **mpz_mod** – Funkce pro výpočet hodnoty $result = b \bmod n$. Existuje také ve variantě pro **unsigned int** a **int**,
- **mpz_powm** – Funkce, jež vypočítá mocninu čísla a výsledek se nachází v modulu jiného jistého čísla,
- **mpz_sqrt** – Funkce pro výpočet odmocniny z daného čísla,
- **mpz_cmp** – Funkce pro porovnání dvou čísel. Návratová hodnota je záporná, pokud číslo na levé straně je menší než na pravé. Pro stejná čísla vrátí hodnotu 0. Ve zbylém případě vrátí kladné číslo. Existuje také ve variantě pro **unsigned int** a **int**,
- **mpz_tstbit** – Funkce pro získání hodnoty bitu na pozici i v čísle ve dvojkové soustavě.

V neposlední řadě jsou použity funkce, které napomáhají například k rychlejšímu určení vlastností čísla nebo určení důležité hodnoty pro kvadratické síto, Legendrův symbol.

- **mpz_gcd** – Tato funkce, jak již vypovídá název, slouží k výpočtu největšího společného dělitele,
- **mpz_invert** – Funkce která, je určena k výpočtu inverze v jistém modulu. Pokud inverze neexistuje, tato funkce nám vrací hodnotu 0,
- **mpz_legendre** – Funkce pro výpočet Legendrova symbolu,

- **mpz_urandomm** – Funkce pro generování náhodného čísla,
- **mpz_nextprime** – Funkce pro vygenerování následujícího prvočísla,
- **mpz_remove** – Funkce pro odstranění všech výskytů nějakého faktoru daného čísla, návratová hodnota obsahuje počet výskytů odstraněného faktoru,
- **mpz_probab_prime_p** – Tato funkce s vysokou pravděpodobností, zda předané číslo je prvočíslo, či nikoliv,
- **mpz_perfect_square_p** – Funkce pro určení perfektního čtverce, tedy perfektní čtverec splňuje následující: $\sqrt{n} \in \mathbb{N}$, $n \in \mathbb{N}$,
- **mpz_abs** – Funkce pro získání absolutní hodnoty čísla.

Knihovna GMP samozřejmě obsahuje mnohem více funkcí, které pomáhají pracovat s danými strukturami. Tuto knihovnu využívají i jiné knihovny jako základ, který dovoluje jiným knihovnám pracovat s čísly s neomezenou přesností. Tuto knihovnu využívá například matematický software SageMath (viz [16]) nebo knihovna FLINT, kterou si nyní popíšeme.

5.3 FLINT

Knihovna FLINT napsaná v jazyce C [17] (*Fast Library for Number Theory*) je určená pro výpočty z oboru teorie čísel. Díky této knihovně je nám umožněno v implementaci používat aritmetiku například nad $\mathbb{Z}$, $\mathbb{Z}/n\mathbb{Z}$ nebo $\mathbb{Z}[x]$. Výhodou této knihovny je také ta, že nám dokáže reprezentovat matice a provádět různé operace s nimi jako nalezení řešení homogenní soustavy rovnic.

Jak již bylo zmíněno knihovna využívá jako základ knihovnu GMP, mnoho funkcí je tedy velmi podobných jako u této knihovny, avšak pro reprezentaci čísel používá knihovna FLINT vlastní pojmenování, například pro celá čísla **fmpz_t**. Je to z toho důvodu, že FLINT si řídí sám velikost alokované paměti pro strukturu reprezentující celá čísla. Stejně tak používá strukturu **fmpq_t** pro reprezentaci racionálních čísel. Ty si ukládá jako dvě čísla, aby je mohl reprezentovat v kanonickém tvaru.

Nebudeme již zde zmiňovat znovu funkce pro inicializaci a uvolňování paměti, protože jsou velmi podobné těm, které již známe z knihovny GMP. Jedním z rozdílu je ten, že jejich název začíná písmenem *f*, tedy funkce pro inicializaci je **fmpz_init** nebo pro uvolnění paměti **fmpz_clear**.

Uvedeme si nejdříve funkce, které slouží pro přiřazení hodnot ze struktur definované knihovnou GMP a jaké operace se dají pomocí knihovny FLINT vykonávat:

- **fmpz_set_mpz** – Tato funkce nastaví hodnotu struktury **fmpz_t** na hodnotu ze struktury **mpz_t**,
- **fmpz_get_mpz** – Tato funkce nastaví hodnotu struktury **mpz_t** na hodnotu ze struktury **fmpz_t**,
- **fmpz_dlog** – Výstupem této funkce je aproximace přirozeného logaritmu zadaného čísla. Návratová hodnota je typu **double**. Přesto, že se jedná o aproximaci, chyba této funkce nepřesahuje 2 bity.
- **fmpz_sqrtmod** – Výstupem této funkce je kořen kongruence ve tvaru $x^2 \equiv n \pmod{p}$, pro $n \in \mathbb{Z}$ a prvočíslo p.
- **fmpz_root** – Funkce, která vypočítává n–tý kořen celého čísla a, tedy $\lfloor \sqrt[n]{a} \rfloor$.

Další funkce slouží pro operace s maticemi. Struktura reprezentující matice operující nad $\mathbb{Z}_n$ se nazývá **nmod_mat_t**. Opět je nutné pro ni alokovat paměť a následně také uvolnit, pokud ji již nebudeme používat. Pro inicializační funkci navíc je nutné uvést počet řádků, počet sloupců a modulus. Nutno zdůraznit, že tato matice operuje pouze s prvky, které knihovna definuje jako **mp_limb_t**, ovšem pod touto definicí se skrývá klasický typ **unsigned int**, takže není nutné používat nějaké speciální struktury pro to. Nyní si ukážeme jak s takovou maticí pracovat, respektive které funkce jsou pro to využity:

- **nmod_mat_get_entry** – Funkce, která slouží k získání prvku matice A na souřadnicích A_{ij}, kde i je číslo řádku a j je číslo sloupce.
- **nmod_mat_entry_ptr** – Funkce, která slouží k získání ukazatele na prvek matice A na souřadnicích A_{ij}. Zde může být název lehce matoucí, ale skrze tento ukazatel můžeme nastavovat hodnotu prvku.
- **nmod_mat_nullspace** – Tato funkce nám uloží do matice, která má alespoň řádků jako matice, ze které je výpočet vykonávan. Každý řádek je jedno řešení. Návratová hodnota udává hodnost matice, ve které jsou uložené řešení.

Následující typ, který využijeme, je polynom, respektive struktura **fmpz_poly_t**. Ta nám přesněji reprezentuje prvky $\mathbb{Z}[x]$. Inicializační a dealokační funkce jsou opět analogicky k ostatním strukturám definované stejně. Žádne argumenty navíc se zde nepředávají. V implementaci využíváme funkce:

- **fmpz_poly_zero_coeffs** – Vynuluje všechny koeficienty polynomu do určeného stupně polynomu,
- **fmpz_poly_set_coeff_fmpz** – Nastaví koeficient na hodnotu definovanou stukturou **fmpz_t**,

- **fmpz_poly_get_coeff_fmpz** – Získá hodnotu koeficientu a uloží do stuktury **fmpz_t**,

Ukázali jsme si všechny podpůrné funkce, které jsou důležité pro fungování našich algoritmů. Nyní se dostaneme k poslední použité technologii, která je využita pro paralelizaci výpočtů.

5.4 OpenMP

Náš program budeme spouštět na školním klastru STAR, kde budeme chtít využít všechna dostupná vlákna, protože výše zmíněná knihovna by je všechna nevyužila kvůli konfiguraci. Zde se nám nabízejí dvě možnosti. Jednou takovou možností je využít klasická POSIXová vlákna, respektive POSIX API, na úkor složitější implementace a manipulace s vlákny. Druhou možností je, a kterou i volíme, využít knihovnu OpenMP [18]. Tato knihovna nám zjednodušuje práci s vlákny, protože se vývojáři této technologie snaží standardizovat vysokoúrovňovou paralelizaci, která je i přenosná, což by v případě POSIX API nebylo vždy možné, protože například operační systém Windows jich nevyužívá a využívají vlastní implementace vláken, kterou můžeme nalézt v [19].

Narozdíl od knihovny OpenMPI, tato knihovna pracuje nad sdílenou pamětí, je tedy na programátorovi, aby si hlídal různé datové závislosti. Abychom mohli určit v kódu, kde požadujeme paralelizaci, tak k tomu slouží direktivy pro překladač.

Uvedeme si nyní dvě základní direktivy, které jsou pro práci velmi důležité:

parallel direktiva definuje blok kódu, tzv. **paralelní region**, který bude spuštěn každým vláknem zvlášť. Na konci tohoto bloku je vytvořena bariéra, která čeká na všechna vlákna, než dokončí výpočet.

critical direktiva určuje kritickou sekci kódu. To nám zajistí, abychom byli schopni předcházet datovým závislostem a nepřepisovali si tak hodnoty sdílených proměnných.

V této knihovně je také možné určit v paralelním bloku, která vlákna jsou pracovní a které vlákno je hlavní.

Direktiv samozřejmě tato knihovna obsahuje mnohem více(asi přes 50), jak se můžeme dočíst například v [20]. K těmto direktivám je možné také použít parametry, které specifikují vlastnosti například pro paralelní blok. Opět si jich pár uvedeme:

private parametr přebírá seznam proměnných, které mají být pro každé vlákno unikátní. Vlákna si v paralelním bloku vytvoří kopii proměnných předané v seznamu a nedochází tak k datové závislosti.

shared parametr naopak značí, které proměnné (opět předané seznamem) jsou pro všechna vlákna společné. Je tedy nutné hlídat, aby proměnnou vždy upravovalo pouze jedno vlákno a zbylé vlákna k němu v té chvíli neměla přístup.

5.5 MPI

Nejdříve než začneme s popisem knihovny jako takové, je vhodné na začátek uvést fakt, že MPI je pouze standard, který implementujou některé knihovny, jako například OpenMPI knihovna, kterou si popíšeme níže. Dokumentaci k tomuto standardu můžeme nalézt v [21]. Tento standard nám definuje rozhraní, podle kterého se implementace drží. Implementace se snaží zachovávat stejné názvosloví, jako má právě daný standard.

Další důležitou použitou technologií je knihovna projektu OpenMPI [22] (*Message Passing Interface*). Je důležitým faktorem pro komunikaci mezi procesy v reálném čase. Pokud chceme tuto technologii využít, je nutné využít i jejich aplikaci **mpiexec**, případně **mpirun**, které spustí naši aplikaci na všech procesorech ve stejnou dobu. Využívají se veškerá jádra procesorů, tedy pokud bychom tuto aplikaci spouštěli na systému se dvěma procesory, kde každý procesor má 4 jádra, tak chování programu bude takové, že celá komunikace bude probíhat mezi všemi 8 jádry. Zde také samozřejmě záleží i na tom, s jakou konfigurací aplikace **mpirun** nebo **mpiexec** program spouštíme.

Každý proces operuje se svojí přidělenou pamětí a komunikace mezi procesy může probíhat pouze přes funkce, které si níže popíšeme, tedy veškeré proměnné si každý proces udržuje ve vlastní paměti

Abychom v naší aplikaci mohli používat komunikaci mezi těmito procesy, je nutné si nejdříve zařadit daný procse do komunikačního prostředí pomocí funkce **MPI_Init**. Po této inicializaci jsou všechny procesy v komunikačním prostředí zvaném **MPI_COMM_WOLRD**. Každý proces má svůj takzvaný *rank* neboli číslo procesu. Číslování je vždy od 0 do p - 1, kde p je počet spuštěných procesů.

Komunikace mezi procesy může probíhat ve dvou variantách, které jsou blokující a neblokující. Blokující komunikace znamená, že proces, který odesílá nějakou zprávu jinému procesu, čeká dokud ji nepřijme. Přijetí můžeme zde chápat jako splnění určité podmínky. Neblokující naopak pošle zprávu a může pracovat dál a nečeká na potvrzení, že si ji proces vyzvedl. Pro tuto komunikaci nabízí tyto funkce:

- **MPI_Send** – Blokující zaslání zprávy,
- **MPI_Recv** – Blokující přijetí zprávy,
- **MPI_Isend** – Neblokující zaslání zprávy,
- **MPI_Irecv** – Neblokující přijetí zprávy,

- **MPI_Iprobe** – Neblokující ověření zda byla procesu zaslána zpráva,

Pomocí těchto funkcí může být uskutečněna komunikace. Tyto funkce také umožňují zasílat zprávy se specifickou značkou (**TAG**), která nám může například napomoci tomu, abychom věděli o co se ve zprávě jedná. Značky jsou určeny programátorem, jež za ně odpovídá. Použité značky si popíšeme v kapitole s implementací. Knihovna nabízí i další podpůrné funkce pro komunikaci mezi procesy. Komunikaci mezi procesy ještě dělíme navíc na další dvě varianty, kterými jsou:

- **2-bodové** (point-to-point) operace se uskutečňují mezi dvěma procesy,
- **kolektivní** operace naopak probíhá mezi všemi procesy v komunikačním prostředí.

Mezi 2-bodové patří právě například funkce pro zasílání zpráv uvedené výše. Kolektivní komunikaci zaštiťují funkce jako jsou například **MPI_Bcast**, která rozdistribuuje všem procesům v komunikačním prostředí zprávu. Příjem této zprávy se musí vykonat pomocí stejné funkce. Dalším takovým příkladem může být i funkce **MPI_Reduce**, která dokáže vypočíst sumu, produkt, získat maximum apod. z předaných hodnot jako parametr. Funkce automaticky rozdělí hodnoty mezi procesy a provede požadovanou operaci a následně vrátí procesu, který je považován za hlavní.

Je vždy ale nutné také uvádět, o co se ve zprávě jedná. Vždy musíme určit, jaký datový typ zasíláme. K tomu nám slouží předem definované datové typy. Tyto typy odpovídají těm, které jsou v jazyce C běžné. Pro příklad si některé uvedeme:

- **MPI_CHAR** – Datový typ **char**,
- **MPI_UNSIGNED** – Datový typ **unsigned**,
- **MPI_DOUBLE** – Datový typ **double**,
- **MPI_LONG** – Datový typ **long**.

Další typ, který mezi nabízenými máme k dispozici a není odvozen od klasických typů, se nazývá **MPI_PACKED**. Ten v naší implementaci je hojně využíván, jelikož si potřebujeme v jedné zprávě poslat více věcí najednou, zejména pokud si posíláme nějaké číslo, je nutné vědět jak je dlouhé (kolik má cifer). K tomu je nám i nabízena funkce **MPI_Pack**, která nám uloží danou hodnotu (jistého datového typu) do připravované zprávy k odeslání. Na druhé straně musí proces rozbalit zprávu přesně v tom pořadí, ve kterém byla zabalena se správnými datovými typy. K rozbalení slouží analogicky funkce **MPI_Unpack**.

Knihovna nám také nabízí možnost synchronizace procesů pomocí funkce **MPI_Barrier**. Tato funkce blokuje volajícího, dokud všechny procesy z komunikačního prostředí nezavolej tuto funkci.

Na závěr je nutné zavolat funkci **MPI_Finalize**, která odpojí proces od komunikačního prostředí.

Nyní máme popsané veškeré technologie 3. stran, které jsme pro implementaci využili. Dále je již příliš rozebírat nebudeme a případně se na ně pouze odkážeme.

Kapitola 6

Popis implementace

Nyní známe všechny potřebné informace o použitých technologiích a máme i dostatečný teoretický základ pro implementaci jednotlivých algoritmů. Pro sekvenční řešení algoritmů budeme vycházet z popsaných pseudokódů v kapitole 3. Každá implementace je napsána jako samostatná třída a nachází se ve složce s implementací na přiložené SD kartě v jednotlivých složkách.

6.1 AbstractFactoringMethod

Každá třída dědí od nadřazené abstraktní třídy, kterou jsme si pojmenovali jako **AbstractFactoringMethod**. Ta nám definuje předpis jednotlivých metod. Zejména pak právě ty abstraktní metody, které musí být implementovány pro sekvenční a paralelní část. Členské proměnné této třídy jsou v chráněné (**protected**) části třídy jsou:

_randstastate, která je určena pro držení si stavu generování čísel,

_result, která je určen pro zachování výsledku.

Konstruktor této rodičovské třídy vždy inicializuje tyto proměnné. Destruktor je pak naopak dealokuje. Další členské proměnné si případně každá poděděná třída doplní sama.

Ve veřejném (**public**) rozhraní třídy se nachází metody:

void factorize(mpz_t number), která je určena pro spuštění sekvenčního faktorizování čísla,

void factorize_parallel(int argc, char argv, mpz_t number)** pro spuštění paralelního výpočtu,

void get_result(mpz_t result) pro uložení výsledku do parametru result.

Metody určené pro faktorizování jsou abstraktní a musí si je každá implementace definovat. Metoda **get_result** je pro všechny stejná.

Dalšími metodami této třídy jsou ve chráněné části (**protected**):

void _master_part(mpz_t number) určena pro *master* část paralelního výpočtu,

void _process_results(mpz_t number, mpz_t result) pro zprocesování výsledků od ostatních procesů provádějící výpočet,

void _slave_part(mpz_t number) určena pro výpočetní procesy, které nezpracovávají výsledky od ostatních procesů,

void _factorize_parallel(int argc, char argv, mpz_t number)** implementace inicializace komunikačního prostředí, rozdělení práce mezi řídící proces a početní procesy, následné neblokující zaslání zprávy pro ostatní procesy, že mohou ukončit veškeré výpočty a volání finální funkce pro ukončení komunikace.

6.2 Implementace Pollardovy–ρ metody

Implementaci tohoto algoritmu reprezentuje třída *PollardRho*. Tato třída má navíc jednu členskou proměnnou. Ta uchovává funkci, která generuje další prvek posloupnosti. Ve třídě je navíc přidána struktura **Value**, která uchovává hodnotu struktury **mpz_t**. Tuto strukturu využíváme v množině (kontejneru **set**) uchovávající duplikátní hodnoty pro paralelní výpočet.

6.2.1 Sekvenční výpočet

Sekvenční část vychází přesně z algoritmu 2. Implementace abstraktní metody **factorize** spočívá v tomto:

1. Nejdříve si vygenerujeme jisté číslo x. K tomu slouží metoda **_generate**, která přijímá argument, do které uloží výsledek a hranice intervalu, ze kterého je číslo generováno,

2. následně je volána metoda **_factorize**, přijímající počáteční hodnotu, složené číslo, které faktorizujeme, parametr, do kterého ukládáme výsledek a indikátor, zda se jedná o paralelní či sekvenční výpočet.

6.2.2 Paralelní výpočet

Abstraktní metody jsou implementovány takto:

_master_part neobsahuje v těle metody žádný kód. Pro tento algoritmus není potřeba implementace této metody.

_slave_part vybere interval, ze kterého generujeme hodnoty. Dále máme paralelní region, kde si v cyklu každé vlákno vygeneruje unikátní hodnotu a spustí výpočet v metodě **_factorize** s indikátorem paraleního výpočtu. Tento cyklus běží do doby, než nějaké vlákno nalezne korektní hodnotu nebo dokud není proces ukončen hlavním procesem.

_process_results funguje velmi podobně jako předešlá metoda. Také nejdříve je určen interval a následně si každé vlákno vygeneruje unikátní hodnotu pro výpočet. V cyklu zde navíc je kontrolováno, zda nějaký proces našel hledaný faktor.

factorize_parallel metoda vyčistí pole duplikátů a následně zavolá nadřazenou rodičovskou metodu, která inicializuje komunikační prostředí.

6.3 Implementace Pollardovy $p-1$ metody

Implementaci reprezentuje třída *Pollard*. Velmi se podobá implementaci Pollardovy ρ–metody. Proto jí zde nebudeme příliš podrobně rozebírat. Tato třída obsahuje navíc implementaci struktury **Value**, která je zde ze stejného důvodu, jako tomu bylo u předešlého algoritmu.

6.3.1 Sekvenční výpočet

Implementace metody **factorize** probíhá opět tak, že se nejdříve vygeneruje hodnota a, která je v daném rozmezí. Následně se spustí metoda **_factorize**, která provádí výpočet definovaný v algoritmu 3.

6.3.2 Paralelní výpočet

Paralelní výpočet se velmi podobá paralelnímu výpočtu z předešlého algoritmu. Implementace abstraktních metod je tedy úplně stejná, ve smyslu rozdělení práce vláknům.

6.4 Lenstrův algoritmus

Implementaci reprezentuje třída *Lenstra*. Tato třída obsahuje ve své soukromé **private** části definici dvou struktur:

Point je struktura reprezentující bod na eliptické křivce, tato struktura má 3 členské proměnné, které jsou souřadnicemi daného bodu,

EllipticCurve je struktura reprezentující eliptickou křivku splňující rovnici $y^2 = x^3 + ax + b$. Tato struktura má 3 členské proměnné, která představuje parametr a, b a modulus, ve kterém výpočet probíhá.

Obě struktury mají definované konstruktory, které incializují své proměnné a destruktory pro jejich dealokaci. Struktury mají navíc definovaný kopírující konstruktor a operátor přiřazení, aby byla pohodlnější práce s těmito strukturami. Navíc struktura **EllipticCurve** má přetížený operátor pro porovnání. Je to z toho důvodu, že duplikátní křivky si nedržíme v poli nýbrž v C++ kontejneru **set**, která toto přetížení vyžaduje. Dalšími soukromými členskými proměnnými jsou pomocné body. Proměnná **INFINITY_POINT** reprezentuje bod v nekonečnu a proměnná **FAILURE** reprezentující chybu při výpočtu násobku bodu.

Jak již bylo v předešlé kapitole řečeno, že jsme získali faktor čísla zjistíme tak, že nalezená inverze je právě tím faktorem. To si budeme indikovat tak, že první souřadnici bodu nastavíme na zápornou hodnotu. V jiném případě nemůžeme mít zápornou hodnotu v první souřadnici.

Pro tuto třídu je nutné si definovat násobení bodu číslem. K tomu využijeme algoritmu **Double-and-add** uvedený v [2]. Tento algoritmus implementuje soukromá metoda **_mul_points**. Pseudokód lehce upraven pro naše potřeby je uveden v algoritmu 10. Pro součet bodů slouží metoda **_add_points**, která používá definovaný součet.

Algorithm 10 Pseudokód Double-and-add algoritmu

```
1:  function DoubleAndAdd(point p, multiplier i)
2:      res = INFINITY_POINT, q = p
3:      mult = i
4:      while mult != 0 do
5:          if multiplier je lichý then
6:              ret = p + ret
7:              if ret = FAILURE then return FAILURE
8:              else if ret.x < 0 then return ret
9:              end if
10:         end if
11:         p = p + p
12:         if p = FAILURE then return FAILURE
13:         else if p.x < 0 then return p
14:         end if
15:         mult = ⌊mult/2⌋
16:     end whilereturn ret
17: end function
```

Zbývá nám nyní jen určit, jak budeme generovat eliptickou křivku a bod. To můžeme provést tak, že si libovolně zvolíme bod $P = (p_1, p_2) \in R^2$ a $a \in R$. Pro připomenutí pracujeme nad okruhem $R := \mathbb{Z}_n$. Zbývající hodnotu eliptické křivky b vypočítáme jako $b \equiv p_2^2 - p_1^3 - ap_1 \pmod{n}$.

6.4.1 Sekvenční výpočet

Implementace metody **factorize** provádí následující kroky v nekonečném cyklu:

1. Vygeneruj křivku *ecc* a bod p
2. Zavolej metodu **_factorize**, do které vstupují jako parametry vygenerovaná křivka, bod, proměnná s výsledkem a indikátor paralelního výpočtu (zde ji nastavíme na nepravdu **false**).
3. Pokud je výsledek faktorem, potom ukonči cyklus, v opačném případě vygeneruj novou křivku.

Tato metoda opakuje stále dokola tyto 3 kroky, dokud nenalezne hledaný faktor. Nyní si rozebereme metodu **_factorize**:

1. Nastav násobící koeficient (z algoritmu 3.3 hodnota i) na číslo 2.
2. Spusť cyklus, který běži do doby, než je bod p roven bodu v nekonečnu, nebo bodu indikujícímu chybu při výpočtu
3. V tomto cyklu vynásob bod s hodnotou i.
4. Pokud nový bod má první hodnotu nastavenou na zápornou hodnotu, ulož do proměnné s výsledkem největšího společného dělitele faktorizovaného čísla a třetí souřadnicí bodu.

6.4.2 Paralelní výpočet

_master_part není implementována. Metoda má prázdné tělo, pro tento algoritmus není potřeba.

_slave_part vytvoří paralelní blok, kde každé vlákno si vytvoří proměnnou pro uchování výsledku faktorizace. Následuje cyklus, ve kterém postupně každé vlákno přijme od hlavního procesu unikátní eliptickou křivku a bod. Následně se spustí výpočet metodou **_factorize**, kde nastavíme indikátor paralelního výpočtu na pravdu (**true**).

_process_results si nejdříve vytvoří kontejner pro ukládání použitých eliptických křivek. Následuje paralelní region, kde si každé vlákno vytvoří lokální proměnné pro eliptickou křivku, bod a výsledek. Hlavní vlákno je použito jako generátor křivek a bodu. To je tedy jen v metodě **_parallel_generator**. Dalším krokem pro pracovní vlákna je cyklus, kde si každé vlákno vygeneruje vlastní unikátní eliptickou křivku a bod a následuje výpočet. Následuje opět kontrola příchozí zprávy reprezentující potenciální faktor čísla.

factorize_parallel metoda pouze volá rodičovskou metodu, která inicializuje komunikační prostředí.

6.5 Kvadratické síto

Implementace se nachází ve třídě *QuadraticSieve*. Spolu s touto třídou je implementována třída *Matrix*, která reprezentuje matici, kterou využijeme k nalezení řešení homogenní rovnice. Tato třída je obalem nad strukturou **nmod_mat_t**, aby bylo jednodušší s touto stukturou manipulovat a řešení zůstalo přehlednější.

Ve třídě *QuadraticSieve* jsou implementovány další dvě struktury. Jedna struktura *Factor* nám uchovává 1 hodnotu, kterou je členská proměnná zvaná (**prime**), jež uchovává prvočíslo, které se nachází v naší faktorové bázi. Dále tato struktura implementuje mimo konstruktoru a destruktoru také přiřazovací operátor a kopírující konstruktor z důvodu jednodušší manipulace s touto hodnotou a jejím možném použití v dynamickém poli *vector*.

Druhá struktura se nazývá *NumberHolder*. Ta nám uchovává naše původní hodnoty x_i, x_i^2, které jsou definované v popisu algoritmu. Uchovává také hodnoty odstraněných exponentů.

Pro tento algoritmus je důležité, abychom na začátku měli zkontrolováno, zda se nejedná o perfektní čtverec faktorizovaného čísla a zda faktorizované číslo není prvočíslem.

6.5.1 Sekvenční výpočet

Sekvenční výpočet začíná v metodě **factorize**, která otestuje předané číslo, zda se jedná o prvočíslo. Následně je zkontrolováno, jestli se nejedná o perfektní čtverec. Navazuje volání metody **_factorize**.

V metodě **_factorize** si nejdříve inicializujeme naše hranice pro faktorovou bázi pomocí privátní metody **_get_bound**. Ty máme již určené v jedné z předešlých kapitol, kde jsme si popsali hodnoty těchto hranic. Aby bylo možné zkusit faktorizovat i malé čísla, je zde určená minimální hranice, která má hodnotu 100.

V dalším kroku sestavujeme naši faktorovou bázi. To nám zajišťuje metoda *_factor_base*.

Dále nastavujeme náš interval, ze kterého vybíráme naše možné hodnoty pro určení relací. Zde máme interval nastaven takto:

$$[\lfloor\sqrt{n}\rfloor - M; \lfloor\sqrt{n}\rfloor + M]$$

Aktuální hodnotu si držíme ve struktuře *NumberHolder*. Zde si uchováváme náš čtverec tohoto čísla, od kterého je odečtena hodnota $\lfloor\sqrt{n}\rfloor$. Dále k tomu si držíme naši původní hodnotu x. Od hodnoty uchovávající čtverec odebíráme nejvyšší exponenty prvočísla. Pokud máme získaný exponent lichý, uložíme ho v této struktuře do pole na patřičné místo (každý prvek pole reprezentuje paritu exponentu daného prvočísla z faktorové báze). Do připraveného pole s relacemi si uložíme použitou hodnotu x a k tomu její čtverec zredukovaný

o odmocninu z faktorizovaného čísla n. To se nám bude hodit pro sestavování levé a pravé strany požadované kongruence.

Dalším krokem máme nalezení řešení homogenní soustavy rovnic. K tomu využijeme již zmiňovanou třídu *Matrix*, která nám obaluje práci s nalezením tohoto řešení. Jelikož potřebujeme vždy transponovat řešení této rovnice, neboť $\mathbb{A}\vec{x} \equiv 0 \pmod 2 \Leftrightarrow (\vec{x}^T\mathbb{A}^T)^T \equiv 0 \pmod 2$, tak metoda třídy Matrix **null_solution** nám rovnou toto řešení transponuje. Do této metody vstupuje již připravená struktura popisující matici **nmod_mat_t**, která má požadované rozměry. Posíláme zde matici, jelikož každá řádka této matice reprezentuje jedno možné řešení. Hodnost této matice je potom počet nalezených řešení.

Na závěr z nalezených řešení sestavíme požadovanou kongruenci. Postupně procházíme řešení a vybíráme taková $Q(x_i)$ pro x^2 a x_i pro y^2, respektive vybíráme z pole uložených relací takové hodnoty ze struktur *NumberHolder*, které odpovídají danému řešení. Na závěr otestujeme zda se jedná o perfektní čtverce a nalezneme jejich kořeny, jejichž rozdíl a součet použijeme pro nalezení faktoru.

6.5.2 Paralelní výpočet

Popíšeme si jednotlivé metody pro paralelní výpočet:

_gather_factor_base určí hranice intervalu hodnot každého procesu pro získání faktorové báze. Každý proces jej pak následně rozešle ostatním procesům a dále je i od ostatních získá.

_master_part nejdříve volá metodu **_gather_factor_base**. Následně inicializuje výslednou matici, která je využita pro soustavu homogenních rovnic. Dalším krokem je volání metody *_sieve_parallel*, která představuje prosévací část. Po prosévací části se rozešle všem procesorům zpráva, že výpočet je dokonán a můžou přestat s hledáním relací. Hlavní proces potom pokračuje dále sekvenčně a nalezne požadované řešení.

_slave_part se velmi podobá předešlé metodě. Opět získá svojí část faktorové báze, kterou rozešle všem ostatním. Inicializace matice je zde pouze jen na rozměry 1×1, aby mohla být provolána metoda *_sieve_parallel*. Po návratu z této metody je proces ukončen.

_process_results zde představuje pouhé uložení výsledku do parametru *result*.

factorize_parallel provolává rodičovskou metodu pro inicializaci komunikačního prostředí.

_sieve_parallel určí, jakou část daný procesor má vykonávat. Pokud se jedná o pracovní proces, tak nalezené relace pouze zasílá hlavnímu procesu. Hlavní proces naopak své nalezené relace uloží do pole s relacemi a s nimi i přijaté relace od ostatních procesů.

_master_get_exp je metoda pro hlavní proces, která přijme vektor exponentů a hodnotu x od pracovního procesu.

_slave_send_exp naopak zasílá hlavnímu procesu vektor exponentů společně s hodnotou x.

6.6 Obecné číselné síto

Poslední algoritmus se nachází ve třídě **GNFS**. Společně s touto třídou se zde nachází implementace třídy **PolynomGenerator**. Jak již název napovídá, zprostředkovává nám práci pro generování polynomu. V jejím konstruktoru je inicializace náhodného generátoru čísel. V jejím destruktoru pak uvolnění paměti alokované tímto generátorem. Jsou zde pouze dvě metody:

generate_polynomial je veřejná metoda, která na základě určeného kořenu m a stupně polynomu nám vygeneruje požadovaný polynom, který splňuje požadované podmínky, které jsou popsány například v algoritmu 6.

_set_poly_coeffs je soukromá metoda, která nejdříve nastaví koeficienty polynomu tak, aby vyhovovaly podmínkám. Následně dochází k redukci koeficientů, kterou jsme si již dříve popsali v popisu tohoto algoritmu.

Třída GNFS uchovává další soukromou strukturu *Pair*, která uchovává páry dvou hodnot, které následně využijeme pro faktorové báze. Sice knihovny C++ nám nabízí již definovanou třídu *pair*, avšak my potřebujeme obalit struktury reprezentující dané hodnoty, protože nemají ve výchozím stavu popis přiřazení pomocí operátoru =. Tato struktura implementuje stejně jako v předchozích algoritmech konstruktor, destruktor, kopírující konstruktor a definici přetížení operátoru =.

Struktura si uchovává také soukromé členské proměnné:

_rational_bound uchovává hranici pro racionální faktorovou bázi,

_algebraic_bound uchovává hranici pro algebraickou faktorovou bázi,

_quad_base uchovává počet prvků v kvadratické bázi,

_qcb_min uchovává minimální hodnotu prvočísla pro kvadratickou bázi,

_qcb_max uchovává maximální hodnotu prvočísla pro kvadratickou bázi,

_rfb uchovává pole páru pro racionální faktorovou bázi,

_afb uchovává pole páru pro algebraickou faktorovou bázi,

_qcb uchovává pole páru pro kvadratickou bázi.

6.6.1 Sekvenční výpočet

Sekvenční výpočet je definován v metodě **factorize**. Nejdříve nastaví vhodné parametry pro vygenerování polynomu a pro hranice faktorových bází v metodách **_get_poly_degree** a **_set_bounds**. Tyto hodnoty jsme převzali z doporučení uvedeném v [11], jelikož se jedná o jisté aproximace. Hodnotu kořene m volíme náhodně a splňuje tyto podmínky:

$$\lfloor n^{\frac{1}{st(f)+1}} \rfloor \leq m \leq \lceil n^{\frac{1}{st(f)}} \rceil$$

Dalším krokem je sestavení bází v metodě **_set_bases**. To opět opisuje popis, který máme uveden v algoritmu 6. Navazuje prosévací část popsaná v algoritmu 7. Zde si objevené relace ukládáme do pole struktur *Pair*, kterou je zároveň návratovou hodnotou.

Následně se dostáváme do stejné části jako je tomu u předešlého algoritmu. Pomocí metody **_set_matrix** sestavíme z daných relací matici, jak je popsáno v algoritmu 8.

Finální krok je v metodě **_get_factor**. Zde opět nalezneme řešení homogenní soustavy rovnic. Dále procházíme postupně všechna nalezená řešení a sestavujeme z nich hledané perfektní čtverce. Pro hledání kořene hodnoty y^2 používáme metodu **_algebraic_sqrt**, která nám vrací přesnou hodnotu tohoto kořene. Zde nemůžeme použít klasickou metodu pro hledání kořene, jelikož hledáme kvadratické reziduum nad obecným tělesem.

Pro nalezení kvadratického rezidua nad algebraickým tělesem použijeme algoritmus *Tonneli-Shanks*. Tento algoritmus je možné zobecnit a použít ho nad jakýmkoliv tělesem, jak je uvedeno v [23].

6.6.2 Paralelní výpočet

Implementace jednotlivých metod pro paralelní výpočet je:

factorize_parallel zkontroluje, zda faktorizované číslo je složené, vyčistí kontejner uložených duplikátů a faktorových bází a zavolá rodičovskou metodu inicializující komunikační prostředí,

_master_part rozesílá všem procesům unikátní polynom společně s jeho kořenem,

_slave_part přijme polynom s jeho kořenem. Dále se sekvenčně nastaví hranice pro relace a faktorové báze. Následně je volána metoda **_get_relations_parallel**, která nám vrátí hledané relace. Poslední částí je sekvenční krok sestavení matice, hledání řešení, a z řešení hledání rozkladu čísla. Nalezené řešení zašle hlavnímu procesu,

_get_relations_parallel vrací nalezené relace. Relace jsou zde hledány pomocí vláken. V paralelním bloku každé vlákno vezme vlastní hodnotu **b**, která je uvedena v algoritmu 7 a pomocí této hodnoty hledá relace,

_process_results funguje podobně jako metoda **_slave_part**. Hlavní proces si vygeneruje vlastní polynom. Dále opět nastaví hranice a faktorové báze, pokračuje hledáním relací a na závěr se pokusí nalézt hledaný rozklad čísla. Pokud se nepovede ho najít, přijme zprávu od pracovních procesů.

6.7 Dostupné implementace

Nyní máme popsanou naší implementaci vybraných algoritmů. Existují i jiné projekty, které implementují tyto algoritmy. Jedním z takových projektů je SageMath [16], který implementuje Pollardovu $p - 1$ a ρ metodu pro čísla do jisté velikosti, Lenstrův algoritmus pro určitou velikost čísla a kvadratické síto.

Lenstrův algoritmus implementuje projekt GMP-ECM [24], jehož vývojáři se zabývají zejména touto metodou. Tento projekt využívá také SageMath, jak se můžeme dočíst v dokumentaci.

Implementaci GNFS a kvadratického síta můžeme najít například v [25] anebo v [26]

Některé z těchto projektů však nezahrnují propojení OpenMPI a OpeMP pro paralelizaci algoritmů. Můžeme v těchto projektech spíše najít použití jedné z těchto knihoven.

Kapitola 7

Analýza výsledků

V této kapitole si jako první ukážeme, jak vypadá běžná analýza moderních faktorizačních algoritmů. Dále si ukážeme naměřené výsledky implementací spuštěných na školním klastru STAR. Pro měření využijeme několik hodnot, které mají jistou délku v cifrách, abychom mohli následně rozhodnout, který algoritmus je vhodný pro kterou sadu čísel a jaká je jejich přibližná spotřeba paměti.

Implementace budeme spouštět jak v paralelní verzi tak i v sekvenční verzi. Testování probíhá tak, že spustíme implementace nad testovacími daty 10krát a budeme vybírat minimální čas, který je potřeba pro výpočet.

7.1 Analýza moderních faktorizačních algoritmů

Dříve existovala *RSA Challenge* (viz [27]) od roku 1991, která dříve také nabízela i peněžní výhru, pokud se někomu povede faktorizovat čísla vygenerované společností RSA Laboratories.

Od roku 2007 je tato výzva zrušena a společnost nevytváří nové klíče RSA, které byly pro výzvu použity. Společnost však stále nechává vygenerované klíče dostupné, ale již bez ceny.

Faktorizace nad těmito klíči se stále provádí. Příkladem může být zpráva z dubna roku 2020, kdy bylo faktorizováno 250 ciferné číslo pomocí algoritmu GNFS (více viz [29]).

Stroje, na kterých tyto faktorizace probíhají, jsou vždy popsané v publikacích, které popisují skutečnost, že byl prolomen další klíč. Například u posledního, tedy 250 ciferného čísla, běžel algoritmus na několika desítkách tísíc strojích po celém světě a výpočet trval několik měsíců.

7.2 Konfigurace klastru STAR

Nyní si popíšeme konfiguraci klastru STAR. V popisu vycházíme z [14]. Dostupných uzlů pro měření máme 3. Každý uzel reprezentuje jeden procesor:

- **Model: Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz**
- **#CPU jader: 20**
- **#sockets, #cores per socket, #threads per socket: (2, 10, 1)**
- **CPU Cache L1 - L2 - L3: 32KB - 256KB - 25600KB**

Dostupná paměť pro uzel je **64GB RAM**. V měření jsme omezeni 10 minutami na úlohu. Je tedy nutné volit vhodné délky hodnot, aby implementace zvládly v tomto čase číslo faktorizovat.

7.3 Parametry kompilace

Program jsme zkompilovali pomocí **mpic++**, který je zde použit z důvodu použití knihovny OpenMPI, která tento kompilátor vyžaduje. Byly použity následující přepínače:

- **-fopenmp** je nutný pro použití knihovny **OpenMP**,
- **-std=c++17** pro využití standardu C++17,
- **-O3** pro optimalizace 3. úrovně,
- **-lflint** pro určení cesty ke knihovně FLINT pro linker,
- **-lgmp** pro určení cesty ke knihovně GMP pro linker,

7.4 Použité testovací hodnoty

Testovací hodnoty použijeme jako složená čísla typu $n = pq$, kde p, q jsou prvočísla, jejichž délka v cifrách je stejná, nebudeme tedy testovat složená čísla, kde jeden faktor je v řádech nižší než druhý faktor. Prvočísla byla volena náhodně. Zvolili jsme těchto 14 hodnot:

1. Hodnota $100003 \cdot 10007 =$

   1000730021, má 10 cifer, počet bitů je 32

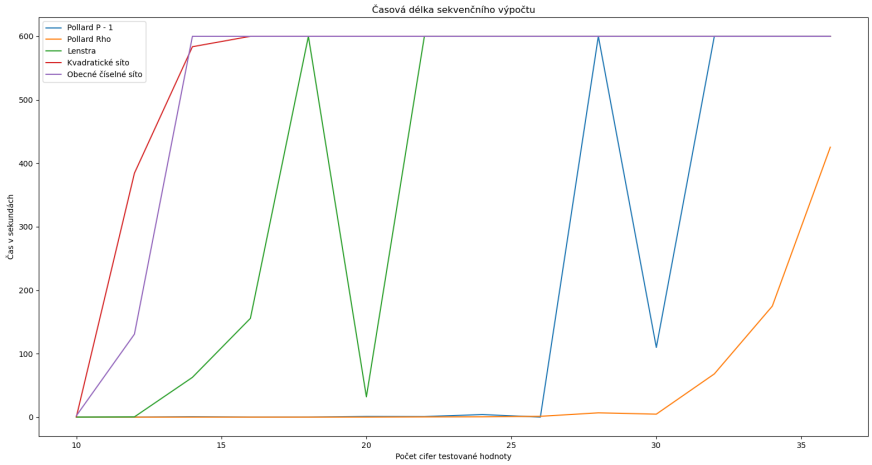
2. $169399 \cdot 732097 =$

   124016499703, má 12 cifer, počet bitů je 39,

3. $9015689 \cdot 2179753 =$
   19651975144817, má 14 cifer, počet bitů je 47,
4. $66877373 \cdot 37411007 =$
   2501949869444611, má 16 cifer, počet bitů je 54,
5. $616711813 \cdot 193781603 =$
   119507403712176239, má 18 cifer, počet bitů je 59,
6. $7156658203 \cdot 3334377187 =$
   23862997847239614961, má 20 cifer, počet bitů je 67,
7. $69042605417 \cdot 73174579949 =$
   5052163649973526983733, má 22 cifer, počet bitů je 75,
8. $498796229131 \cdot 797594702249 =$
   397837229856663925015619, má 24 cifer, počet bitů je 81,
9. $6557409984317 \cdot 4779898644569 =$
   313437550959200558472243 73, má 26 cifer, počet bitů je 87,
10. $62176039100899 \cdot 46283009871953 =$
    2877694231505844147237185747, má 28 cifer, počet bitů je 94,
11. $817931922805289 \cdot 501274865319727 =$
    410008714444926584643751636103, má 30 cifer, počet bitů je 101,
12. $7878127367160683 \cdot 6144986336391269 =$
    484109850275525191682490812767 27, má 32 cifer, počet bitů je 108,
13. $78227414794747619 \cdot 13127269084928071 =$
    1026912323828935219557287213512949, má 34 cifer, počet bitů je 112,
14. $819251619519795947 \cdot 244278502983555437 =$
    200125559183149097928582026802413839, má 36 cifer, počet bitů je 120.

7.5 Výsledky měření sekvenčního výpočtu

Na obrázku 7.1 můžeme vidět časový průběh algoritmů. Nejlepším algoritmem pro hodnoty, které mají maximálně 36 cifer, se zdá být Pollardova ρ–metoda. Můžeme si povšimnout, že i když Lenstrův algoritmus, případně číselná síta, mají asymptoticky lepší časovou složitost, tak nejsou vhodné pro takovou délku čísel.



Obrázek 7.1: Graf časů při sekvenčním výpočtu

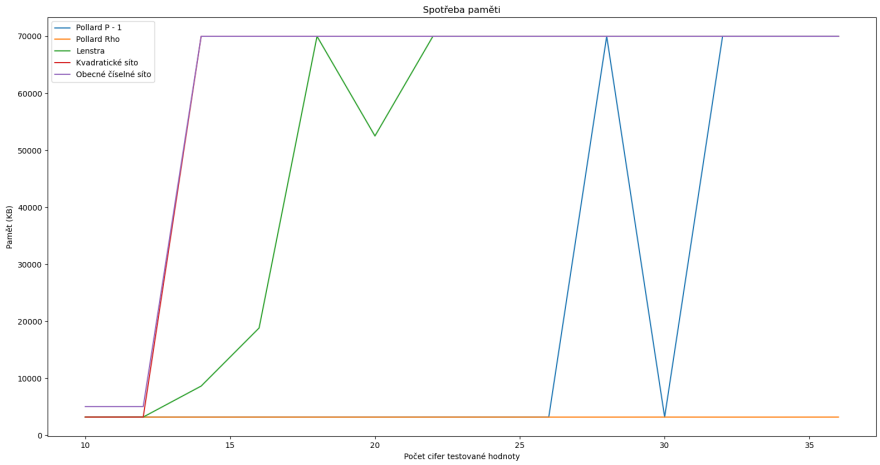
Naopak nejhůře si vedl algoritmus GNFS, který by měl mít ze všech uvedených algoritmů nejlepší časovou složitost. Můžeme si však všimnout toho, že Pollardova ρ–metoda od 34 ciferného čísla začíná využitý čas růst. To je zapříčiněno tím, že již máme větší interval, ze kterého vybíráme hodnoty, nad kterým je následně spuštěn výpočet. Máme tedy menší šanci, že vybereme vhodnou hodnotu jako první člen posloupnosti.

Pollardova $p-1$ metoda se zdá být vhodná pro čísla do 26 cifer. Pokud bychom přidali k této metodě hranici, která by určovala kolik maximálně iterací při výpočtu může být vykonáno a následně vybrána jiná hodnota pro výpočet, tak by mohly být výsledky jiné. Stejně tak pro Lenstrovův algoritmus by mohly být výsledky jiné, pokud bychom určili stejnou hranici a následný výběr jiné eliptické křivky a bodu.

Využití číselných sít pro testované hodnoty není vhodné. Problémem je, že číselná síta stráví nejvíce času v prosévací části, tedy v hledání relací. Velikost intervalu pro číselná síta je vždy závislá na složeném čísle.

Na obrázku 7.2 můžeme vidět graf spotřeby paměti pro jednotlivé algoritmy. Hodnota 200 000 KB reprezentuje neznámou spotřebu paměti, neboť výpočet byl předčasně ukončen z důvodu časové limitace ze strany výpočetního svazku. Můžeme si všimnout, že Pollardova ρ–metoda a $p-1$ metoda opravdu využívá pouze konstantní hodnotu přibližně 4000 KB. Naopak Lenstrův algoritmus využívá více paměti. Zde je to z důvodu uchovávání elptických křivek a bodů a také je potřeba ukládat proměnné pro pomocné hodnoty.

Rozeberme si nyní číselná síta. V tabulce 7.1 vidíme velikost faktorové báze, počet nalezených relací a počet řešení pro jistý počet cifer. Počet relací ani velikost faktorové báze není nijak zvlášť velká. Pro faktorizaci tedy



Obrázek 7.2: Graf potřebné paměti pro sekvenční výpočet

Počet cifer	Velikost faktorové báze	Počet relací	Počet řešení
10	13	23	12
12	13	23	10
14	14	24	13

Tabulka 7.1: Vygenerované parametry pro kvadratické síto

Počet cifer	Polynom	Velikost matice	Počet řešení
10	$x^3 + 476x^2 + 492x + 293$	103×103	41
12	$9x^3 + 2045x^2 + 2017x + 2259$	112×112	34

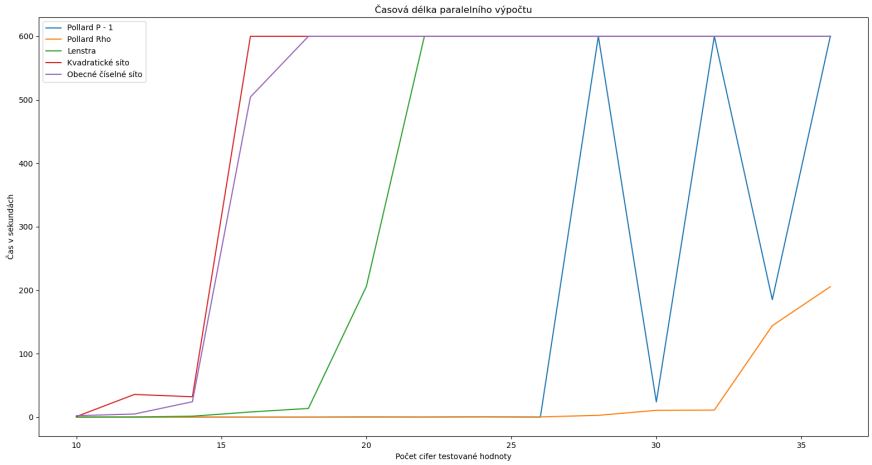
Tabulka 7.2: Vygenerované parametry pro GNFS

nemusíme hledat velký počet těchto relací.

Tabulka 7.2 nám zobrazuje vygenerované parametry pro obecné číselné síto, velikost matice a počet nalezených řešení. Zde již oproti kvadratickému sítu můžeme vidět, že matice pro tento algoritmus je větší a počet řešení je také větší. Máme tedy větší šanci, že v těchto nalezených řešeních spíše najdeme hledané perfektní čtverce.

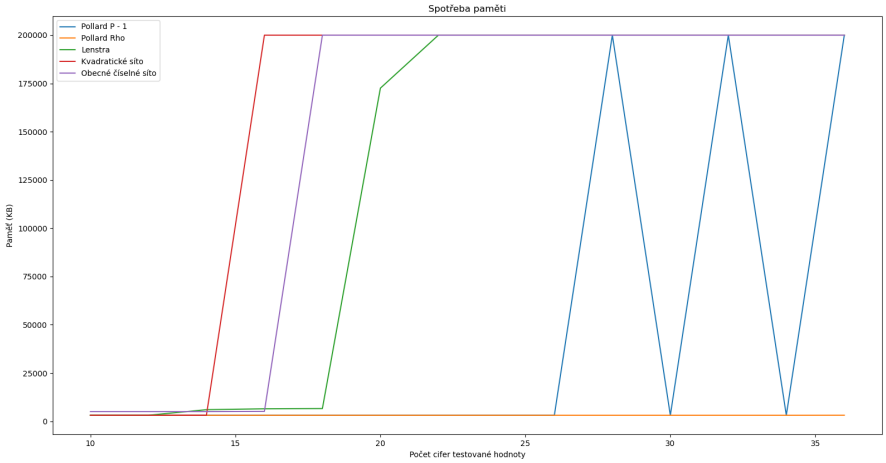
7.6 Výsledky měření paralelního výpočtu

Na obrázku 7.3 můžeme vidět, že nejlépe si opět vedla Pollardova ρ–metoda. Křivka této metody roste o něco pomaleji než je tomu v případě sekvenčního výpočtu. Ostatní algoritmy mají při paralelizaci také příznivé výsledky. Vidíme,



Obrázek 7.3: Graf časů při paralelním výpočtu

že časy číselných sít se zmenšily. Kvadratické číselné síto opět provedlo výpočet pro hodnoty do 14 cifer, avšak s lepším časovým výsledkem. Paralelizace naopak více prospěla obecnému číselnému sítu, protože se povedlo faktorizovat i 16 ciferné číslo.



Obrázek 7.4: Graf potřebné paměti pro paralelní výpočet

Na obrázku 7.4 je graf spotřeby paměti pro paralelní výpočet. Spotřeba je zde měřena jako průměr pro 1 proces. Hodnota 200 000KB opět reprezentuje

nedokončený výpočet. U Pollardovy $p-1$ a ρ–metody si můžeme všimnout opět konstantní spotřeby paměti.

U Lenstrova faktorizačního algoritmu si můžeme všimnout, že s rostoucím časem nám i roste spotřebovaná paměť. To je zde opět z důvodu udržovaných unikátních eliptických křivek.

Spotřeba paměti je pro číselná síta relativně nízká oproti ostatním algoritmům i přes to, že je nutné si držet hodnoty pro matici, pro relace atd. V některých případech je tato spotřeba nižší než například u Lenstrova algoritmu, který si drží jen eliptickou křivku a bod.

Počet cifer	Velikost faktorové báze	Počet relací	Počet řešení
10	13	23	12
12	13	23	10
14	14	24	11

Tabulka 7.3: Vygenerované parametry pro kvadratické síto (paralelní výpočet)

Pokud srovnáme tabulku 7.3 s tabulkou 7.1, můžeme si všimnout, že jediný rozdíl je v poslední řádce v počtu řešení. Je to z důvodu nalezení jiných relací než v případě sekvenčního řešení (relace byly vybírány z více částí intervalu zároveň).

Počet cifer	Polynomy	Velikost matice	Počet řešení
10	$53x^3 + 205x^2 + 89x + 186$	99×99	34
	$5x^3 + 14x^2 + 122x + 469$	94×94	32
	$x^3 + 109x^2 + 615x + 896$	100×100	41
12	$12x^3 + 844x^2 + 1139x + 1558$	122×122	46
	$382x^3 + 330x^2 + 54x + 289$	109×109	40
	$29x^3 + 1154x^2 + 41x + 1293$	116×116	44

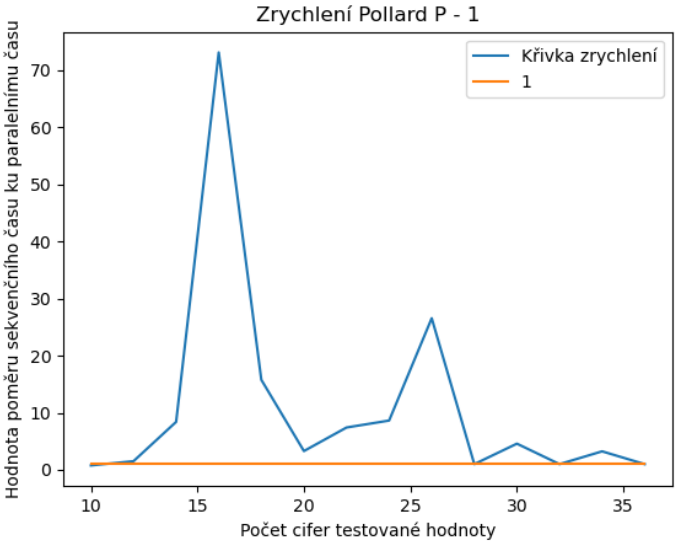
Tabulka 7.4: Vygenerované parametry pro obecné číselné síto (paralelní výpočet)

V tabulce 7.4 můžeme nalézt vygenerované hodnoty společně s velikostí matice a počtem řešení pro obecné číselné síto. Každá trojice v řádce reprezentuje hodnoty pro jednotlivé uzly, na kterých probíhal test.

Nyní si ukážeme grafy s poměry časů, které budeme počítat takto:

$$\frac{SEQ_{time}}{PAR_{time}},$$

kde SEQ_{time} je naměřený čas sekvenčního běhu programu a PAR_{time} je naměřený čas paralelního běhu. Na každém grafu je vyobrazena pomocná hranice, která ukazuje, zda se pro daný počet cifer testované hodnoty zrychlil výpočet nebo neopak zpomalil. Hodnoty pod hranicí značí zpomalení a hodnoty nad hranicí zrychlení.



Obrázek 7.5: Poměr časů pro Pollardovu $p-1$ metodu

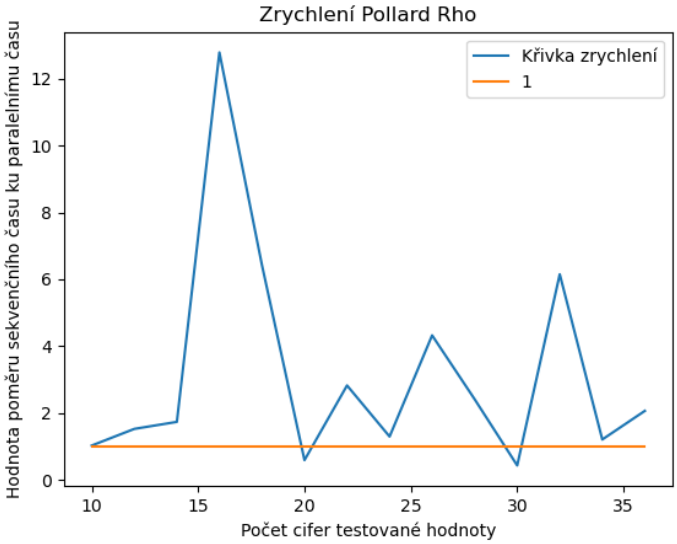
Obrázek 7.5 reprezentuje graf poměru časů pro Pollardovu $p-1$ metodu. Zrychlení je zde nejvyšší pro 16 ciferné číslo. K žádnému výraznému poklesu nedošlo. Pro 10 ciferné číslo si můžeme všimnout malého zpomalení. To může být zapříčiněno například tím, že nějaký čas strávil paralelní výpočet na generování unikátních hodnot a rozeslání hodnot a jejich následné případné přijetí a ukončení všech procesů.

Na obrázku 7.6 můžeme na grafu poměru časů pro Pollardovu ρ–metodu vidět stejné nejvýraznější zrychlení jako u předešlého grafu pro 16 ciferné číslo. Propadů tu však můžeme zpozorovat mnohem více. Ty jsou zde opět z důvodu, kdy časy nalezení prvočinitelů jsou si velmi podobné se sekvenčnímu řešení, ale musí se čekat na ukončení práce vláken, zaslání a přijmutí zpráv a ukončení procesů.

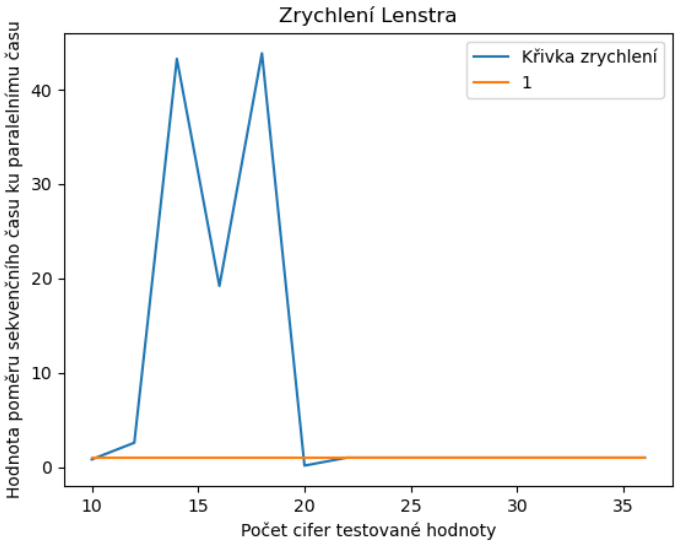
Obrázek 7.7 představuje poměry časů pro Lenstrův algoritmus. Zde již máme více výrazných zrychlení zejména pro hodnoty od 14 ciferné testované hodnoty do 18 ciferné. Po těchto hodnotách následuje mírný propad. Zde je tento propad z důvodu nepříliš vhodného generování eliptických křivek a bodů pro paralelní výpočet.

Kvadratické síto, podle obrázku 7.8, se také vyrázně zrychlilo pro 12 a 14 ciferné číslo. Následně však byl program ukončen a graf tedy pokračuje ve stejné linii jako pomocná hranice. Pro 10 ciferné číslo je nepatřičný propad opět z důvodu zasílání zpráv.

Poslední obrázek 7.9 ukazuje graf poměru časů pro GNFS. Opět zde lze vidět výrazné zrychlení pro 12 a 14 ciferné číslo. Pro 16 ciferné číslo není již zrychlení tak dobré.

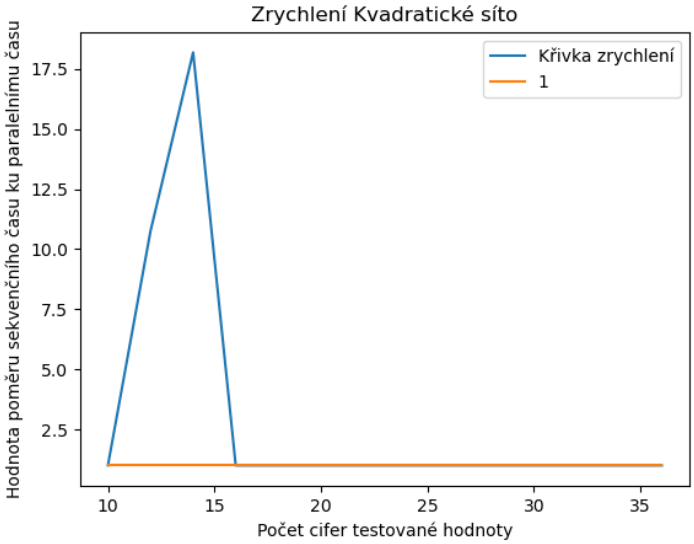


Obrázek 7.6: Poměr časů pro Pollardovu ρ–metodu

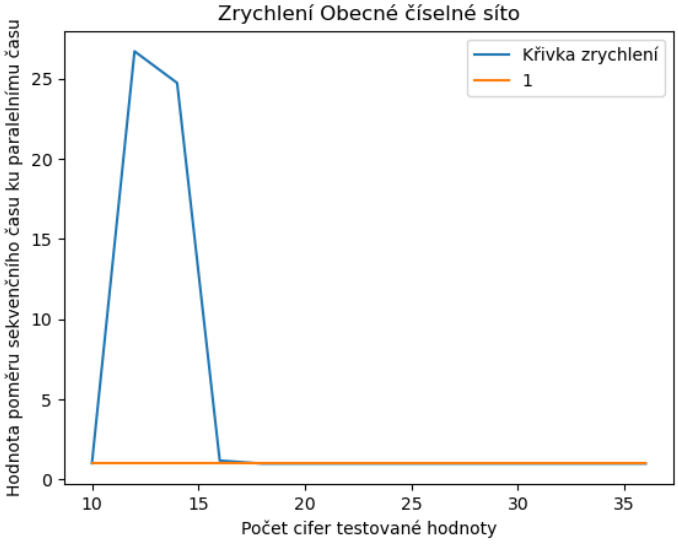


Obrázek 7.7: Poměr časů pro Lenstrův algoritmus

Na závěr si popíšeme tabulky, které reprezentují teoretický a naměřený nárůst pro jednotlivé algoritmy pro sekvenční výpočet.



Obrázek 7.8: Poměr časů pro kvadratické síto



Obrázek 7.9: Poměr časů pro GNFS

V tabulce 7.5 můžeme vidět jaký je teoretický a naměřený nárust pro Pollardovu $p-1$ metodu. Řádky bez hodnoty jsou zde z důvodu předčasného ukončení výpočtu. Můžeme si z hodnot všimnout, že neodpovídají zcela předpokládanému nárůstu.

#Cifer	#Bitů	Předpoklad	Naměřený
12	40	10^1	$2.483 \cdot 10^1$
14	47	10^2	$9.696 \cdot 10^2$
16	54	10^3	$5.254 \cdot 10^1$
18	60	10^4	$1.072 \cdot 10^2$
20	67	10^5	$1.583 \cdot 10^3$
22	74	10^6	$1.344 \cdot 10^3$
24	80	10^7	$6.318 \cdot 10^3$
26	87	10^8	$5.061 \cdot 10^1$
28	94	10^9	–
30	100	10^{10}	$1.721 \cdot 10^5$
32	107	10^{11}	–
34	113	10^{12}	–
36	120	10^{13}	–

Tabulka 7.5: Nárust pro Pollardovu $p - 1$ metodu

#Cifer	#Bitů	Předpoklad	Naměřený
12	40	$3.161 \cdot 10^0$	$2.627 \cdot 10^0$
14	47	10^1	$5.663 \cdot 10^0$
16	54	$3.164 \cdot 10^1$	$3.151 \cdot 10^1$
18	60	10^2	$4.345 \cdot 10^1$
20	67	$3.164 \cdot 10^2$	$1.845 \cdot 10^2$
22	74	10^3	$3.239 \cdot 10^3$
24	80	$3.164 \cdot 10^3$	$5.610 \cdot 10^3$
26	87	10^4	$1.437 \cdot 10^4$
28	94	$3.164 \cdot 10^4$	$7.727 \cdot 10^4$
30	100	10^5	$5.376 \cdot 10^4$
32	107	$3.164 \cdot 10^5$	$7.784 \cdot 10^5$
34	113	10^6	$1.999 \cdot 10^6$
36	120	$3.164 \cdot 10^6$	$4.863 \cdot 10^6$

Tabulka 7.6: Nárůst pro Pollardovu ρ–metodu

Z tabulky 7.6 můžeme vyčíst hodnoty nárůstu pro Pollardovu ρ–metodu. Opět si zde můžeme všimnout, že naměřený nárůst je nižší, než předpokládaný nárůst. U 22 ciferné hodnoty si můžeme všimnout, že zde je již naměřený nárůst vyšší než bylo očekáváno. U 30 ciferné hodnoty je naopak předpokládaný nárůst vyšší než naměřený. Náš algoritmus se však drží přibližné hodnoty teoretického nárůstu. Zde jsou menší odchylky z důvodu, že složitost, kterou počítáme je pouze asymptotická a nikoliv úplně přesná.

Tabulka 7.7 reprezentuje nárůst pro Lenstrův algoritmus. Zde si můžeme všimnout, že rozdíl mezi naměřeným nárůstem a teoretickým nárůstem je

#Cifer	#Bitů	Předpoklad	Naměřený
12	40	$4.592 \cdot 10^{0}$	$3.588 \cdot 10^{2}$
14	47	$1.901 \cdot 10^{1}$	$5.974 \cdot 10^{4}$
16	54	$7.246 \cdot 10^{1}$	$1.484 \cdot 10^{5}$
18	60	$2.577 \cdot 10^{2}$	–
20	67	$8.649 \cdot 10^{2}$	$3.050 \cdot 10^{4}$
22	74	$2.759 \cdot 10^{3}$	–
24	80	$8.427 \cdot 10^{3}$	–
26	87	$2.475 \cdot 10^{4}$	–
28	94	$7.021 \cdot 10^{4}$	–
30	100	$1.930 \cdot 10^{5}$	–
32	107	$5.158 \cdot 10^{5}$	–
34	113	$1.343 \cdot 10^{6}$	–
36	120	$3.415 \cdot 10^{6}$	–

Tabulka 7.7: Nárůst pro Lenstrův algoritmus

větší oproti předešlým algoritmům. Tyto rozdíly jsou zde opět z toho důvodu, že asymptotická složitost není úplně přesná. Dalším důvodem může být také fakt, že výběr eliptických křivek a bodu nemusel být příliš vhodný.

#Cifer	#Bitů	Předpoklad	Naměřený
12	40	$3.137 \cdot 10^{0}$	$4.152 \cdot 10^{2}$
14	47	$9.107 \cdot 10^{0}$	$6.305 \cdot 10^{2}$
16	54	$2.484 \cdot 10^{1}$	–
18	60	$6.433 \cdot 10^{1}$	–
20	67	$1.595 \cdot 10^{2}$	–
22	74	$3.808 \cdot 10^{2}$	–
24	80	$8.796 \cdot 10^{2}$	–
26	87	$1.973 \cdot 10^{3}$	–
28	94	$4.313 \cdot 10^{3}$	–
30	100	$9.210 \cdot 10^{3}$	–
32	107	$1.925 \cdot 10^{4}$	–
34	113	$3.945 \cdot 10^{4}$	–
36	120	$7.945 \cdot 10^{4}$	–

Tabulka 7.8: Nárůst pro kvadratické síto

V tabulce 7.8 máme nárůsty pro kvadratické síto. Zde máme pro porovnání pouze dvě hodnoty. I z těchto dvou hodnot vidíme, že rozdíl je opět o něco větší než u prvních zmíněných algoritmů.

Poslední tabulka 7.9 obsahuje vypočítané nárůsty pro obecné číselné síto. Zde máme pouze jednu hodnotu. Z ní můžeme vyčíst, že sice není rozdíl příliš

#Cifer	#Bitů	Předpoklad	Naměřený
12	40	$3.362 \cdot 10^{0}$	$5.576 \cdot 10^{1}$
14	47	$1.010 \cdot 10^{1}$	–
16	54	$2.782 \cdot 10^{1}$	–
18	60	$7.132 \cdot 10^{1}$	–
20	67	$1.723 \cdot 10^{2}$	–
22	74	$3.962 \cdot 10^{2}$	–
24	80	$8.726 \cdot 10^{2}$	–
26	87	$1.850 \cdot 10^{3}$	–
28	94	$3.798 \cdot 10^{3}$	–
30	100	$7.567 \cdot 10^{3}$	–
32	107	$1.468 \cdot 10^{4}$	–
34	113	$2.782 \cdot 10^{4}$	–
36	120	$5.159 \cdot 10^{4}$	–

Tabulka 7.9: Nárůst pro obecné číselné síto

velký, avšak mezi dalšími testovanými hodnotami by neměl být nárůst tak velký. Zde hraje velkou roli fakt, že pro tento algoritmus je časová složitost ukázaná pomocí heuristiky, podle které odpovídá časová složitost přibližně funkci uvedené v kapitole 3.6.

7.7 Resumé

Z měření se nám zdá být nejefektivnějším algoritmem Pollardova ρ–metoda pro testované hodnoty. Samozřejmě je ale nutné si uvědomit, že jsme při měření byli omezeni časem a nemohli jsme tedy vyzkoušet hodnoty, které mají například 50 cifer, abychom oveřili, jestli nepřijde jakýsi zlom v křivkách, kdy například Lenstrův algoritmus by vyšel jako výhodnější pro vyšší hodnoty. Bylo by vhodné také implementovat různé optimalizace jak pro Lenstrův algoritmus, tak pro číselná síta.

Z tabulky 3.6 však můžeme vidět, že náš výsledek je očekávaný. Pro hodnoty do 40 cifer má nejmenší počet kroků právě Pollardova ρ–metoda. Dále pak pro čísla do 90 cifer by mělo mít nejmenší počet kroků kvadratické síto a od 90 cifer výš by mělo být nejrychlejší potom obecné číselné síto.

V [28] jsou uvedené doposud největší rozklady nejen pomocí našich vybraných faktorizačních algoritmů. Pro Pollardovu $p-1$ metodu měl nalezený prvočinitel 58 cifer. Pro Lenstrův algoritmus je to 67 cifer, pro kvadratické síto je to pak 135 cifer a pro GNFS 250 cifer.

Pokud se rozhodujeme, kdy který algoritmus použít, pak v [30] je to uvedeno takto:

- Pollardova $p-1$ metoda je vhodná pro B-hladká čísla, kde B je relativně

nízká hranice. Hranice hodnoty B bývá do 10^6. Tedy hodnoty do 10^{12}.

- Pollardova ρ–metoda je vhodná pro hodnoty do 2^{68}.
- Lenstrův algoritmus pak pro hodnoty do 10^{50}.
- Kvadratické síto je uváděno jako vhodné pro hodnoty od 10^{50} do 10^{100}.
- GNFS je poté nejvhodnější pro hodnoty vyšší než 10^{100}.

Závěr

Cílem této práce bylo seznámit se s faktorizačními algoritmy, implementovat je a sekvenční řešení těchto algoritmů také paralelizovat. Dalším cílem bylo také zanalyzovat jejich časovou a paměťovou složitost.

V první kapitole jsme si popsali základní matematické struktury, které jsou nutným základem pro vybrané faktorizační algoritmy.

V druhé kapitole jsme si ukázali využití problému faktorizace v kryptografii. Popsali jsme si jak šifrovací algoritmus funguje a kde se využívá.

Ve třetí kapitole se věnujeme faktorizačním algoritmům, abychom pochopili fungování jednotlivých vybraných algoritmů. Popsali jsme si zde teoretickou časovou a paměťovou složitost.

Čtvrtá kapitola se zabývá návrhem paralelizace jednotlivých algoritmů. Popsali jsme si zde také jak metody operují nad distribuovanou pamětí a také nad sdílenou pamětí.

Pátá kapitola obsahuje popis vybraných technologií, které jsou nutné pro praktickou část této práce a tedy implementaci jednotlivých faktorizačních algoritmů.

V šesté kapitole jsme si popsali, jak jsme implementaci provedli. Popsali jsme si zde implementace jednotlivých metod, které poté tvoří celek pro různá měření.

Poslední kapitola se zabývá analýzou výsledků naměřených hodnot implementovaných faktorizačních algoritmů.

V praktické části této práce jsme úspěšně implementovali vybrané faktorizační algoritmy v základní verzi bez různých optimalizací. Povedlo se také implementovat jejich paralelizaci. Ukázali jsme si pro určitou sadu čísel, které mají jistou délku v počtu cifer, který algoritmus je nejvhodnější. Bohužel kvůli časovému omezení na výpočetním svazku jsme si neukázali, pro kterou sadu čísel je lepší právě využít jiné algoritmy. Bylo by vhodné faktorizační algoritmy dále optimalizovat. Zejména vhodná optimalizace by byla pro Lenstrův algoritmus, kvadratické síto a obecné číselné síto. Zanalyzovali jsme si také časové a paměťové nároky pro jednotlivé faktorizační metody a ukázali si, že i když

mají některé faktorizační algoritmy asymptoticky lepší časovou složitost, tak pro některé námi testované hodnoty jsou méně efektivní. U algoritmů jsme si také ukazali, že čím méně jsou algoritmy výpočetně náročné, tak je těžší je implementovat.

Příloha A

Použití aplikace

Pro kompilaci tohoto programu je nutné mít doinstalované knihovny, které jsou popsané v předešlé kapitole. Doporučený kompilátor pro tuto implementaci nosí název **mpic++**. Implementace využívá některých technik, které jsou od standardu C++11. Může být tedy potřeba tento standard specifikovat.

```
[kuba@archlinux implementation]$ ./factorizer -h
Usage:    ./factorizer [OPTION] ALGORITHM INTEGER
This program factorizes integer. Integer has to be greater than 1.
Options:
    -d                Start parallel
    -h, --help        Show this menu
Algorithms:
    -G                General Number Field Sieve
    -L                Lenstra's algorithm
    -P                Pollard p - 1 method
    -R                Pollard-rho method
    -Q                Quadratic Sieve
[kuba@archlinux implementation]$
```

Obrázek A.1: Návod k použití

Na obrázku A.1 můžeme vidět použití naší aplikace po zkompilování. Rozebereme si jednotlivé přepínače.

Přepínače **Options**:

- Přepínač **-d** nám indikuje, zda má být výpočet prováděn paralelně. Výchozí stav aplikace je sekvenční výpočet.
- Přepínač **-h/–help** slouží k pomocnému výpisu, který můžeme vidět na obrázku A.1.

Tyto přepínače jsou nepovinné, není tedy nutné je uvádět.

Dalším přepínači slouží k výběru algoritmu:

- Přepínač **-G** reprezentuje GNFS algoritmus,

- přepínač **-L** reprezentuje Lenstrův algoritmus,
- přepínač **-P** reprezentuje Pollardovu $p - 1$ metodu,
- přepínač **-R** reprezentuje Pollardovu ρ–metodu,
- přepínač **-Q** reprezentuje faktorizování pomocí kvadratického síta.

Posledním parametrem této aplikace je číslo, které chceme faktorizovat. Toto číslo by mělo být větší než číslo 1.

Pokud spouštět program tak, aby byl prováděn paralelní výpočet, je nutné k tomu využít spouštěcí aplikaci pro danou implementaci standardu MPI. Například takovou aplikací může být **mpirun** nebo **mpiexec**.

Příloha B

Seznam použitých zkratek

GCD – Greatest Common Divisor (největší společný násobek)

GMP – GNU Multiple Precision

FLINT – Fast Library for Number Theory

MPI – Message Passing Interface

RSA – Rivest, Shamir, Adleman (šifrova pojmenovaná podle jejich tvůrců)

SPMD – Single Program Multiple Data

Příloha C

Obsah přiložené SD karty

- `readme.txt` ..... stručný popis obsahu SD karty
- `implementation` ..... zdrojové kódy implementace
  - `gnfs` ..... zdrojové kódy implementace GNFS
  - `lenstra` ..... zdrojové kódy implementace Lenstrova algoritmu
  - `pollard_p` ..... zdrojové kódy implementace Pollardovy $p - 1$ metody
  - `pollard_rho` ..... zdrojové kódy implementace Pollardovy ρ–metody
  - `quadratic_sieve` ..... zdrojové kódy implementace kvadratického síta
  - `tests` ..... zdrojové kódy testů implementací
- `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `src` ..... zdrojové soubory LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF

Bibliografie

1. BAUMHOF, Andreas. Dostupné také z: https://www.quintessencelabs.com/blog/breaking-rsa-encryption-update-state-art/. [cit. 2020-05-10].
2. KALVODA, T.; STAROSTA, Š.; PETR, I. *Matematika pro kryptologii – poznámky k přednáškám*. Dostupné také z: https://courses.fit.cvut.cz/MI-MKY/lectures/index.html.
3. PELANTOVÁ, Edita; MASÁKOVÁ, Zuzana. *Teorie čísel*. České vysoké učení technické v Praze, 2017. ISBN 978-80-01-06030-8.
4. *Digital Signature Standard (DSS)*. National Institute of Standards a Technology, 2013. Dostupné z DOI: http://dx.doi.org/10.6028/nist.fips.186-4. Technická zpráva. [cit. 2020-04-02].
5. POLLARD, J. M. A monte carlo method for factorization. 1975, s. 331–334. Dostupné z DOI: https://doi.org/10.1007/BF01933667.
6. POLLARD, J. M. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society*. 1974, roč. 76, č. 3, s. 521–528. Dostupné z DOI: 10.1017/S0305004100049252.
7. CHAREST, Anna-Sophie. Pollard's p-1 and Lenstra's factoring algorithms. 2005. Dostupné také z: http://www.math.mcgill.ca/darmon/courses/05-06/usra/charest.pdf. [cit. 2020-02-03].
8. POMERANCE, Carl. The Quadratic Sieve Factoring Algorithm. 1985, s. 169–182.
9. BRIGGS, Matthew E. An Introduction to the General Number Field Sieve. 1998. Dostupné také z: https://intranet.math.vt.edu/people/brown/doc/briggs_gnfs_thesis.pdf. [2020-04-01].
10. GARRETT, Stephani Lee. *ON THE QUADRATIC SIEVE*. the Faculty of The Graduate School at The University of North Carolina at Greensboro, 2008. Dostupné také z: https://libres.uncg.edu/ir/uncg/f/umi-uncg-1581.pdf.

11. MURPHY, Brian Antony. *Polynomial Selection for the Number Field Sieve Integer Factorisation Algorithm.*

12. CASE, Michael. A Beginner's Guide To The General Number Field Sieve. Dostupné také z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.2389&rep=rep1&type=pdf`. [cit. 2020-04-25].

13. RIESEL, Hans. *Prime Numbers and Computer Methods for Factorization.* The Royal Institute of Technology, 1994. ISBN 978-0-8176-8297-2. Dostupné také z: `https://epdf.pub/prime-numbers-and-computer-methods-for-factorization-second-edition.html`. [cit. 2020-04-20].

14. *Popis klastru STAR.* Dostupné také z: `https://courses.fit.cvut.cz/MI-PDP/labs/klastr-star.html`. [cit. 2020-05-02].

15. GRANLUND, Torbjörn et al. Dostupné také z: `https://gmplib.org/`. [cit. 2020-04-20].

16. *Dokumentace SageMath.* Dostupné také z: `https://doc.sagemath.org/html/en/a_tour_of_sage/`. [cit. 2020-05-02].

17. HART, William; JOHANSSON, Fredrik; PANCRATZ, Sebastian. Dostupné také z: `http://www.flintlib.org/index.html`. [cit. 2020-04-20].

18. *Open MP.* Dostupné také z: `www.openmp.org`. [cit. 2020-05-13].

19. Dostupné také z: `https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads`. [cit. 2020-05-14].

20. LANGR, Daniel; ŠIMEČEK, Ivan; TVRDÍK, Pavel. *Úvod do OpenMP.* Dostupné také z: `https://courses.fit.cvut.cz/MI-PDP/media/lectures/MI-PDP-Prednaska02-OpenMP.pdf`. [cit. 2020-05-14].

21. Dostupné také z: `https://www.mpi-forum.org/docs/`. [cit. 2020-05-12].

22. *Open MPI.* Dostupné také z: `https://www.open-mpi.org/`. [cit. 2020-04-20].

23. ADJ, Gora; RODRÍGUEZ-HENRÍQUEZ, Francisco. Square root computation over even extension fields. Dostupné také z: `https://eprint.iacr.org/2012/685.pdf`. [cit. 2020-14-05].

24. ZIMMERMANN, Paul. Dostupné také z: `https://gforge.inria.fr/projects/ecm/`. [cit. 2020-05-16].

25. MONICO, Chris. Dostupné také z: `http://www.math.ttu.edu/~cmonico/software/ggnfs/`. [cit. 2020-05-16].

26. KECHLIBAR, Marian. Dostupné také z: `http://www.karlin.mff.cuni.cz/~krypto/Implementace_NFS.html`. [cit. 2020-05-16].

27. Dostupné také z: https://web.archive.org/web/20131110032059/http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm#. [cit. 2020-05-25].

28. Dostupné také z: https://dlmf.nist.gov/27.19. [cit. 2020-05-25].

29. Dostupné také z: https://www.schneier.com/blog/archives/2020/04/rsa-250_factore.html. [cit. 2020-05-25].

30. LENSTRA, Arjen K. Integer Factoring. 2000, s. 31–58.