ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Algoritmy pro násobení polynomů
Student: Jan Rahm
Vedoucí: doc. Ing. Ivan Šimeček, Ph.D.
Studijní program: Informatika
Studijní obor: Teoretická informatika
Katedra: Katedra teoretické informatiky
Platnost zadání: Do konce letního semestru 2017/18

Pokyny pro vypracování

1) Nastudujte algoritmy triviální, Karatsuba [1] a Schönhage-Strassen [2] pro násobení polynomů.
2) Implementujte tyto algoritmy: triviální a Schönhage-Strassen v jazyce C/C++.
3) Implementované algoritmy optimalizujte pomocí transformací kódu [3], efektivní využití skrytých pamětí, vektorizace a vícevláknové paralelizace (pomocí technologie OpenMP).
4) Otestujte výkonnost implementovaných algoritmů na fakultním serveru Star a porovnejte s existujícími implementacemi algoritmů pro násobení polynomů (např. [4]).

Seznam odborné literatury

[1] WEIMERSKIRCH, A., PAAR, C.:Generalizations of the Karatsuba Algorithm for Efficient Implementations. IACR Cryptology ePrint Archive, 2006, p. 224.

[2] SCHÖNHAGE, A., STRASSEN V. : Schnelle Multiplikation großer Zahlen. Computing 7. 1971, p. 281-292.

[3] https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/kompilator.pdf

[4] Léhar A.: Rychlé násobení smíšených polynomu, BP, ČVUT FIT,2014

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 7. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Algoritmy pro násobení polynomů

Jan Rahm

Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

14. května 2017

Poděkování

Děkuji panu doc. Ing. Ivanu Šimečkovi, Ph.D. za ochotu, konzultace a cenné rady, které mi poskytl.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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 14. května 2017 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Jan Rahm. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Rahm, Jan. *Algoritmy pro násobení polynomů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Bakalářská práce se zabývá vybranými algoritmy pro násobení polynomů, zejména Karatsubova a Schönhageho–Strassenova algoritmu. Cílem práce je implementace algoritmů v jazyku C++, optimalizace pomocí transformace zdrojového kódu a paralizace pomocí knihovny OpenMP. Algoritmy jsou otestovány a změřeny na výpočetním serveru Star.

Klíčová slova Karatsuba, Schönhage–Strassen, násobení polynomů, optimalizace algoritmů, OpenMP

Abstract

The bachelor thesis deals with algorithms for multiplication of polynomials, especially Karatsuba and Schönhage-Strassen algorithm. The aim of the thesis is implementation algorithms in C++, optimization by source code transformation and parallelisation by OpenMP library. Algorithms are tested and measured on the Star computing server.

Keywords Karatsuba, Schönhage-Strassen, multiplication of polynomials, optimization of algorithms, OpenMP

Obsah

Seznam obrázků

Úvod

Polynomy mají širokosáhlé uplatnění v oblastech vědy, zejména matematiky. Využívají se například pro popis fyzikálních jevů, předpověď ekonomických modelů, a nebo v kryptografii pro šifrování klíčů.

S rostoucí velikostí polynomů roste i čas potřebný k jejich manipulaci, zejména k násobení. Proto je nutné se poohlédnout po algoritmech, které jsou rychlejší než klasické zacházení.

Cílem této bakalářské práce je implementace vybraných algoritmů, jmenovitě triviální, Karatsuby a Schönhage–Strassena, pro násobení polynomů, jejich optimalizace a paralelizace pomocí knihovny OpenMP. Jednotlivé implementace budou změřeny a porovnány mezi sebou a též porovnány s dalšími algoritmy pro násobení polynomů.

Kapitola 1

Základní pojmy a definice

Kapitola popisuje pojmy a algoritmy použité v bakalářské práci. Dále se zabývá knihovnami pro násobení polynomů. V kapitole je čerpáno z [11].

1.1 Základní pojmy

1.1.1 Polynom

Polynom p je funkce, pro kterou platí

$$p(x) = \sum_{i=0}^{n} \alpha_i x^i = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + ... + \alpha_n x^n,$$

kde $n \in N_0$ a $\alpha_0, \alpha_1, ..., \alpha_n \in R$. Čísla $\alpha_0, \alpha_1, ..., \alpha_n$ pak nazýváme koeficienty polynomu p.

1.1.2 Stupeň polynomu

Stupeň polynomu je nejvyšší index $i \in N_0$ takový, že koeficient $\alpha_i \neq 0$. Pokud jsou všechny koeficienty nulové, pak stupeň polynomu je roven -1.

1.1.3 Součet polynomů

Součet polynomů je funkce, pro kterou platí

$$s(x) = p(x) + q(x) = \sum_{i=0}^{n} \alpha_i x^i + \sum_{i=0}^{m} \beta_i x^i = \sum_{i=0}^{max(n,m)} (\alpha_i + \beta_i) x^i$$

1.1.4 Součin polynomů

Součin polynomů je funkce, pro kterou platí

$$s(x) = p(x) \cdot q(x) = \sum_{i=0}^{n} \alpha_i x^i \cdot \sum_{i=0}^{m} \beta_j x^i = \sum_{i=0}^{n} \sum_{j=0}^{m} \alpha_i \beta_j x^{i+j}$$

1.1.5 Diskrétní Fourierova transformace

Diskrétní Fourierova transformace je matematické zpracování řady čísel, které nachází uplatnění ve zpracování digitálního signálu, přesněji DFT umožňuje stanovit frekvenční spektrum digitálního signálu. Také může být použita k provedení konvoluce.

1.1.6 Rychlá Fourierova transformace

Rychlá Fourierova transformace je efektivní algoritmus (asymptotická složitost $O(n \log_2 n)$) pro vypočítání DFT a inverzní DFT (iDFT). FFT je velmi důležitá v mnoha matematických oblastech, například v digitálním zpracování signálu, v řešení parciálních diferenciálních rovnic a v rychlém násobení velkých celých čísel nebo polynomů.

1.1.7 Kořen jednoty

Pro číslo a platí

$$a^k \equiv 1 \pmod{m},$$

kde $a > 1$, je k-tý kořen jednoty.

1.1.8 Number-theoretic transform

Number-theoretic transform je druh FFT, který pracuje v modulární aritmetice s celými čísly. Rovnice FFT:

$$y_j \equiv \sum_{i=0}^{n-1} x_i r^{ij} \pmod{m},$$

kde $j \in n-1$, y a x jsou pole čísel velikosti n, r je n-tý kořen jednoty modulo m.

Pro inverzní NTT pak platí:

$$x_i \equiv \frac{1}{n} \sum_{i=0}^{n-1} y_j r^{ij} \pmod{m}$$

1.1.9 Vyvažovací vektor

Vyvažovací vektor je vektor A o velikosti n obsahující hodnoty a^j, $0 \leq j < n$, a je $2n$-tý *kořen jednoty.*
Pro A^{-1}, vektor obsahuje hodnoty a^{-j}, $0 \leq j < n$.

1.1.10 Konvoluce

Násobíme dvě čísla, 123 a 456, výsledek bez konečného *přenosu* (carrying) vypadá takto:

		1	2	3
	×	4	5	6
		6	12	18
	5	10	15	
4	8	12		
4	13	28	27	18

Sekvence čísel 4, 13, 28, 27, 18 se pak nazývá **acyklická konvoluce** čísel 123 a 456. Po provedení *přenosu* získáme výsledek 56088.

Vstupní sekvence má n elementů (zde 3), to znamená, že *acyklická konvoluce* má $2n - 1$ elementů. Pokud odečteme levé $(n - 1)$ elementy od pravých,

28	27	18
-	4	13
28	23	5

tak dostaneme **negacyklickou konvoluci**. *Přenosem* získáme hodnotu 3035, která odpovídá $3035 \equiv 56088 \pmod{1001}$. Resp. odpovídá hodnotě výsledku vstupní sekvence modulo $B^n + 1$ $(10^3 + 1)$.

Naopak pokud přičteme, tak dostaneme **cyklickou konvoluci**, která odpovídá hodnotě $3141 \equiv 56088 \pmod{999}$. Resp. modulo $B^n - 1$ $(10^3 - 1)$.

28	27	18
+	4	13
28	31	31

1.1.11 Konvoluční teorém

Konvoluční teorém je metoda, poskytující efektivní způsob výpočtu *cyklické konvoluce* dvou vstupních sekvencí.

$$CyclicConvolution(X, Y) = iDFT(DFT(X) \cdot DFT(Y))$$

Dva vektory X a Y, na každý použijeme DFT, následně jednotlivé položky vektorů vynásobíme v párech mezi sebou, $x_1 \cdot y_1, x_2 \cdot y_2, ..., x_n \cdot y_n$. Nakonec se provede iDFT.

Pro získání *negacyklické konvoluce* stačí mírně upravit předchozí rovnici,

$$NegacyclicConvolution(X, Y) = A^{-1} \cdot iDFT(DFT(A \cdot X) \cdot DFT(A \cdot Y))$$

kde A je vyvažovací vektor.

1.2 Algoritmy pro násobení polynomů

1.2.1 Triviální algoritmus

Triviální algoritmus spočívá v jednoduchém násobení každého členu z jednoho polynomu s každým členem z druhého polynomu.

Pseudokód:

```
for i to n do
  for j to m do
    res[i+j] = res[i+j] + pol1[i] * pol2[j];
  end
end
```

Asymptotická složitost: $O(n \cdot m)$, kde n, m jsou velikosti polynomů.

1.2.2 Karatsubův algoritmus

Karatsubův algoritmus [2] vymyslel roku 1960 matematik Anatoly Alexeevitch Karatsuba pro násobení velkých čísel, ale je aplikovatelný i pro násobení polynomů. Byl to též první algoritmus se složitostí menší než $O(n^2)$, přesněji $O(n^{\log_2 3})$.

Jedná se o rekurzivní algoritmus patřící do kategorie *"rozděl a panuj"*, to znamená, že řešený problém rozdělí na jednodušší podproblémy a postupným řešením těchto podproblémů se dojde k řešení celku.

Myšlenka algoritmu spočívá v menším počtu násobení, kde operace součtu polynomu je pro počítač jednodušší a rychlejší než operace násobení.

Mějme dva polynomy:

$$p(x) = \alpha_1 x^{n+1} + \alpha_2 x^n$$
$$q(x) = \beta_1 x^{n+1} + \beta_2 x^n$$

Platí:

$$p(x) * q(x) = z_2 x^{2n+2} + z_1 x^{2n+1} + z_0 x^{2n}$$

Kde:

$$z_2 = \alpha_1 \beta_1$$
$$z_1 = \alpha_1 \beta_2 + \alpha_2 \beta_1$$
$$z_0 = \alpha_2 \beta_2$$

Nyní upravíme:

$$z_1 = \alpha_1 \beta_2 + \alpha_2 \beta_1$$
$$z_1 = (\alpha_1 + \alpha_2)(\beta_1 + \beta_2) - \alpha_1 \beta_1 - \alpha_2 \beta_2$$
$$z_1 = (\alpha_2 + \alpha_1)(\beta_2 + \beta_1) - z_2 - z_0$$

Tedy místo 4 součinů dostáváme jen 3 a počet součtů se nám změnil z 1 na 4.

Pseudokód:

```
karatsuba(pol1, pol2, size)
  if ( size = 1 )
     return pol1*pol2

  m = size/2

  high1, low1 = split_at(pol1, m)
  high2, low2 = split_at(pol2, m)

  z0 = karatsuba(low1, low2, m)
  z1 = karatsuba((low1+high1), (low2+high2), m)
  z2 = karatsuba(high1, high2, m)

  return (z2*x^(2*m))+((z1-z2-z0)*x^(m))+(z0)
```

1.2.3 Schönhage–Strassenův algoritmus

Schönhage–Strassenův algoritmus [1] byl vytvořen Arnoldem Schönhagem a Volkerem Strassenem v roce 1971 pro násobení velkých celých čísel. Algoritmus pro výpočet využívá rekurzivně Rychlé Fourierovy transformace a dosahuje asymptotické složitosti $O(n \cdot \log(n) \cdot \log(\log(n)))$. Jednalo se o asymptoticky nejrychlejší algoritmus až do roku 2007, kdy byl oznámen nový algoritmus, tzv. Fürerův [3] algoritmus, ale ten je rychlejší jen pro astronomicky vysoká čísla, není využíván v praxi a také není předmětem zkoumání této bakalářské práce.

Stručný SSA pro acyklickou konvoluci:

1. Máme dva polynomy, *pol1* a *pol2*. Určíme si počet elementů $N = 2^k$ a modulo m, hodnota N je závislá, viz strana 21. Též vytvoříme dvě pomocná pole (*pom1*, *pom2*), každé velikosti N, a každé z nich bude vždy obsahovat *hodnotu mod m*.

2. Pokud je polynom kratší než $N/2$, vrať vynásobený polynom (například pomocí triviálního algoritmu).

3. Protože aplikujeme *acyklickou konvoluci* rozdělíme polynom na $N/2$ částí a vložíme do pomocného pole, zbytek pole naplníme nulou. *Pol1* do *pom1* a *pol2* do *pom2*.

4. Aplikujeme DFT na obě pole. V tomto případě NTT.

5. Následně provedeme násobení párů z pomocných polí, výsledky uložíme v dalším pomocném poli *res*. Zde můžeme použít rekurzivně SSA nebo jakýkoliv jiný algoritmus pro násobení polynomů.

6. Aplikujeme iDFT (inverzní NTT) na *res*.

7. Vlivem iDFT došlo k přesunu výsledků, a proto je nutné hodnoty v poli přeskládat.

8. Nakonec musíme posčítat jednotlivé hodnoty rozdělených polynomů do jednoho výsledku, zde *result*. Polynomy v *res* jsou posunuty o $N/2/2$.

Pseudokód SSA pro acyklickou konvoluci:

```
ss(pol1, pol2, size)
  if ( size < N/2 )
     return pol1*pol2

  split&padding( pol1, pom1 );
  split&padding( pol2, pom2 );

  DFT( pom1 );
  DFT( pom2 );

  multiply( pom1, pom2, res );

  iDFT( res );

  swapResult( res );

  sumResult( res, result );

  return result;
```

Ukázka výpočtu SSA
Máme dva polynomy tvaru

$$p(x) = 4 + 3x + 2x^2 + 1x^3$$

$$q(x) = 8 + 7x + 6x^2 + 5x^3$$

ty načteme a uložíme do polí příslušných velikostí, zde o velikosti 4.

Určíme si počet elementů N = 8 a modulo $m = 2^{16} + 1$.

Pro první polynom vytvoříme pomocné pole o velikosti N, tedy 8, rozdělíme polynom na N/2 částí, tedy 4, a vložíme do polí. Zbytek pole vyplníme 0. To samé uděláme i pro druhý polynom. Takto vypadá obsah polí:

4	3	2	1	0	0	0	0

8	7	6	5	0	0	0	0

Nyní použijeme DFT na každé pole zvlášť, výsledek vypadá takto:

10	4660	514	11796	2	61909	65027	52725

26	22136	514	27224	2	46489	65027	35257

Provedeme násobení párů v polí. Vyjde:

260	64059	2048	3004	4	30146	63489	33857

zde upozorňuji, že veškerá aritmetika probíhá v modulo m.

Aplikujeme iDFT:

32	0	5	16	34	60	61	52

A nakonec přeskládáme do požadovaného výsledku:

32	52	61	60	34	16	5	0

Protože jsme vstupní polynomy rozdělili na polynomy délky 1, tak není nutné provádět finální součet.

SSA a konvoluční teorém
Jak je vidět z popisu algoritmu, SSA využívá pro výpočet konvoluční teorém, přesněji acyklickou konvoluci. Pro ní potřebujeme pomocné pole dvojnásobné velikosti, proto v algoritmu dělíme polynom na $N/2$ částí, ale můžeme použít i jinou konvoluci, stačí pouze vhodně upravit pomocné pole.

- Acyklická konvoluce - Pomocné pole můžeme naplnit jen z poloviny, protože zbylé místo se využije pro výsledek konvoluce, viz strana 5. Tato konvoluce nám vrátí celý výsledek násobení dvou polynomů nebo čísel, za předpokladu, že je splněna podmínka, že se výsledek vejde do modula m. V ukázce výpočtu algoritmu to znamená, že $p(x) \cdot q(x) < 2^{16} + 1$.
- Negacyklická konvoluce - Pomocné pole můžeme naplnit celé, ale musíme na něj použít vyvažovací vektor před použitím DFT a po aplikaci iDFT použít na výsledek inverzní vyvažovací vektor. Konvoluce nám vrátí výsledek mod $(B^n + 1)$. Pokud se pohybujeme v bitové soustavě, tak vrácená hodnota se rovná výsledku mod $(2^k + 1)$. Toho se nechá využít při rekurzivním použití SSA, kdy vyšší a nižší úrovně rekurze dobře zapadají do sebe, protože obě jsou mod $(2^k + 1)$. Zde je důležitá poznámka, že při negacyklické konvoluci nižší úrovně rekurze mohou mít menší hodnotu modula a k dosažení plného výsledku je nutné na nejvyšší úrovni rekurze použít acyklickou konvoluci.
- Cyklická konvoluce - Pole naplníme celé, ale nepoužijeme vyvažovací vektor. Získáme tak výsledek mod $(B^n - 1)$. Tato konvoluce není v implementací SSA moc používána kvůli hodnotě modula.

1.3 Existující implementace

1.3.1 NTL

NTL (A Library for Doing Number Theory) [9] je knihovna jazyka C++, která poskytuje datové struktury a algoritmy pro práci s vektory, maticemi a polynomy nad celými čísly a konečnými tělesy.

Konkrétně pro násobení polynomů knihovna využívá triviální, Karatsubův, Schönhage–Strassenův algoritmus a funkci *HomMul*, která využívá Čínskou větu o zbytcích a rychlou Fourierovu transformaci.

Pro zlepšení výkonu je možné tuto knihovnu nakonfigurovat ve spojení s knihovnou GMP (GNU Multiple-Precision Library). GMP je knihovna zabývající se aritmetikou pro dlouhá celá čísla. Je ručně optimalizovaná pro širokou škálu procesorových architektur, to jí umožňuje až několikanásobné zrychlení.

1.3.2 CLN

CLN (Class Library for Numbers) [8] je knihovna pro efektivní výpočty s libovolnými druhy čísel v libovolné přesnosti. Knihovna je napsaná v jazyku C++ a podporuje i operace s polynomy.

Pro násobení CLN využívá triviální, Karatsubův a blíže nespecifikovaný optimální algoritmus pro velké hodnoty.

Pro zrychlení je jádro CLN napsáno v nízkoúrovňovém jazyce assembler pro některé procesory a je možné CLN nakonfigurovat s knihovnou GMP, ze které pak využívá efektivní nízkoúrovňové metody.

1.3.3 FLINT

FLINT (Fast Library for Number Theory) [10] je knihovna zabývající se teorii čísel, napsaná v jazyku C.

FLINT je závislý na knihovně GMP, a samostatná implementace je pak hlavně optimalizovaná pro procesory typu x86 a x86-64.

Knihovna podporuje celá, reálná a komplexní čísla, modulární aritmetiku, konečná tělesa, polynomy a další matematické oblasti.

Pro násobení polynomů využívá klasický, Karatsubův, Schönhage-Strassenův algoritmus a Kroneckerovu substituci.

Kapitola 2

Realizace

Kapitola obsahuje popis prostředků použitých pro implementaci a popis obecných, optimalizovaných a paralelizovaných implementací algoritmů.

2.1 Možné optimalizace

2.1.1 Vektorizace

Vektorizace umožňuje provést operaci pro více dat v jedné instrukci. Toho je vhodné využít například ve *for* cyklu, kde dochází k opakování stejných operací [6].

V práci je využita automatická vektorizace cyklu pomocí kompilátoru, kde musí být splněny následující podmínky:

- V cyklu se nesmí nacházet datové závislosti.
- Vektorizovaný cyklus musí být ten nejvíce uvnitř.
- Cyklus nesmí obsahovat podmínky a volání funkcí.
- Položky v paměti musí být vhodně umístěny, tak aby na sebe navazovaly.

2.1.2 Paralelizace

Paralelizace umožňuje běh programu rozdělit na více částí, které pak běží současně, a tím dosáhnout zrychlení.

Existuje několik nástrojů jak docílit paralelizace, jedním z nich je POSIX Threads. Je to standartní rozhraní určené k vytváření a manipulaci s vlákny. Výhodou POSIX je, že vše je plně ovládáno programátorem, to je ale také jeho největší nevýhodou, protože použití je složité i pro nejjednodušší funkce a je nutné ručně ošetřit všechny možné problémy.

Dalším z nástrojů, určených pro paralelizaci, je OpenMP, které je také využito v této práci. OpenMP je API pro programování vícevláknových aplikací. Programátor pomocí direktiv označí vybrané bloky kódu, které se mají paralelizovat, následně se OpenMP postará o celou paralelizaci.

Pro povolení vícevláknového běhu programu je nutné kompilátoru přidat přepínač `-fopenmp`.

Použité direktivy:

- `num_threads(x)` - Vytvoří x vláken pro paralelní zpracování následujícího kódu.
- `for` - Zparalelizuje následující *for* cyklus.
- `schedule(static)` - Jednotlivým vláknům přiřadí konstantní počet úloh.
- `task` - Vgeneruje úlohu ke zpracování a uloží ji do tzv. TASK pool.

2.2 Triviální algoritmus

2.2.1 Implementace

Implementace triviálního algoritmu je opravdu triviální. Dva jednoduché *for* cykly implementované podle pseudokódu na stránce 6.

Tento algoritmus je implementován kvůli otestování funkčnosti ostatních algoritmů v této práci. Pro svoji funkčnost předpokládá dvě vstupní pole polynomů a jedno výstupní s inicializovanými hodnotami prvků pole na nula.

2.2.2 Optimalizace

Pro optimalizaci triviálního algoritmu je využita automatická optimalizace kompilátoru a vektorizace, která je možná díky implementaci polynomů jako jednoduchá pole.

2.2.3 Paralelizace

U triviálního algoritmu existují tři možnosti paralelizace [4], popsáno podle pseudokódu na straně 6:

- Paralelizace vnějšího cyklu - Vlákna čtou souběžně všechny prvky pole *pol2* a zároveň zapisují do pole *res*. Proto jsou nutné atomické operace, protože se zapisuje do sdílené části paměti.
- Paralelizace vnitřního cyklu - Všechna vlákna čtou postupně *pol1* a v jeden okamžik vždy čtou stejný prvek pole. Nedochází tak k zápisu do sdílené části paměti, ale je nutná synchronizace na konci každého vnitřního cyklu a není možné cyklus vektorizovat. Takto je paralelizována implementace triviálního algoritmu.
- Paralelizace podle oblastí zápisu - Vlákna si rozdělí výstupní pole*res* na několik disjunktních částí, každé vlákno pak bude pouze zapisovat do své části paměti. Ale je nutné manuálně plánovat a vyvažovat zatížení jednotlivých vláken.

2.3 Karatsubův algoritmus

2.3.1 Implementace

Implementace Karatsubova algoritmu vyžaduje stejné vstupní parametry jako triviální algoritmus. Tedy dvě pole polynomů, které chceme vynásobit, jejich velikost a správně inicializované pole pro výsledek, to znamená, že pole je dostatečné velikosti a inicializované na nulové hodnoty.

Implementace je velice podobná pseudokódu na stránce 7. Rekurze začíná podmínkou na ověření velikosti, v této naivní implementaci pokud je polynom délky jedna, tak se použije triviální algoritmus, zde je vhodné poznamenat, že velikost délky může být jiná, a tak dojde k přepnutí na triviální algoritmus dříve, touto vlastností se více zabývá sekce optimalizace Karatsubova algoritmu.

Následuje rozdělení polynomů pomocí ukazatelů na dvě části a dojde k alokaci dvou pomocných polí. Na obě nižší a i vyšší části polynomů se rekurzivně zavolá Karatsubův algoritmus a výsledky se uloží do pomocných polí. Dojde tedy k výpočtu *z0* a *z2* jak je uvedeno v pseudokódu. *Z1* se ukládá do pole pro výsledek.

Dojde k zavolání funkce minusPol(), která odečte hodnoty *z0* a *z2* od *z1*. Následuje přičtení *z0* a *z2* k finálnímu výsledku (v pseudokódu se jedná o poslední řádek return).

Pomocná pole se následně použijí pro výpočet součtu *low1 + high1* a *low2 + high2* a dojde k poslednímu rekurzivnímu volání Karatsuby (výpočet *z1*).

2.3.2 Optimalizace

Zapnutím kompilátorové vektorizace došlo k vektorizaci funkcí minusPol() a sumPol() použitých v algoritmu. Funkce slouží pro zjednodušení manipulace s polynomy, v tomto případě pro součet a rozdíl.

Velice zajímavou myšlenkou je přepnutí Karatsubova algoritmu na triviální pro určitou velikost polynomu. Tím by odpadla část režie s rekurzivní alokací algoritmu na zásobníku, a tak by se dosáhlo celkového zrychlení.

2.3.3 Paralelizace

V Karatsubově algoritmu byla paralelizována rekurzivní volání *karatsuba*, viz strana 7, pomocí direktivy *task* z knihovny OpenMP.

Dále bylo vyzkoušeno paralelizování jednotlivých *for* cyklů pro součet a rozdíl polynomů.

Veškerá paralelizace byla zkoušena na nejrychlejší implementaci Karatsuby, tedy na optimalizované verzi pomocí přepnutí na triviální algoritmus při velikosti polynomu 2^5.

2.4 Schönhage–Strassenův algoritmus

2.4.1 Implementace

SSA je implementován pomocí *acyklické konvoluce* podle pseudokódu na stránce číslo 8. Implementace využívá předpokladu délky polynomu rovné 2^k, to má za následek o trochu jednodušší implementaci a umožňuje jednoduché použití Karatsubova algoritmu pro přepnutí SSA rekurze. Také očekává stejné vstupní parametry jako triviální a Karatsubův algoritmus.

V implementaci je zvoleno modulo $m = 2^{16} + 1$ a $N = 16$. To znamená, že maximální hodnota vypočítaného polynomu může být $\pm 2^{15}$. Omezení na tuto hodnotu je z důvodu velikosti datového typu int (2^{32}) kvůli kroku násobení párů v algoritmu. Ale protože se první bit používá pro znaménko, tak z důvodů bezpečnosti jsou v implementaci SSA použity datové typy int64_t.

Hodnota $N = 16$ byla zvolena na základě měření, viz strana 21.

V algoritmu se využívají dvě datové struktury, *InstanceOfSSPolynom* a *instaceSS*. *InstanceOfSSPolynom* je dvourozměrné pole ukládající si rozdělený polynom a *instaceSS* je struktura obsahující uložená data instance SSA, podrobně N, m a *kořen jednoty*.

2.4.2 Optimalizace

Stejně jako pro ostatní algoritmy, tak i pro Schönhage–Strassenův algoritmus je využita kompilátorová optimalizace a vektorizace.

Také se nabízí možnost přepnutí SSA na triviální, nebo Karatsubův algoritmus, případně kombinaci obou dvou.

2.4.3 Paralelizace

Paralelizace SSA byla provedena na nejrychlejší optimalizované implementaci algoritmu, to znamená na *SSA/Kara32*, viz strana 26.

Bylo vyzkoušeno paralelizování kroku párových násobení a for cyklů pro manipulaci s polynomy.

Kapitola 3

Testování a diskuze

Tato kapitola obsahuje měření a testování algoritmů, jejich porovnání a diskuzi ohledně jejich implementací.

3.1 Použité prostředky

3.1.1 Reprezentace hodnot

Algoritmy jsou testovány na polynomech uložených v souborech. Hodnoty jsou uloženy od nejmenšího po největší stupeň koeficientu polynomu.

Příklad dat v souboru pro polynom stupně 7:

```
8
1       1   1   1   1   0   0   1
XXX
1       0   1   1   1   1   1   1
```

První řádek udává velikost polynomů, pak následuje řádek s hodnotami koeficientů prvního polynomu. Třetí řádek je oddělovač druhého polynomu.

Polynomy musí být velikosti 2^k z důvodu Karatsubova algoritmu, kdy by jinak vstupní polynom musel být vždy zvětšen na nejbližší vyšší mocninu 2. Takto porovnávané algoritmy jsou vždy měřeny na stejně velkých polynomech a nedochází tak k znevýhodňování Karatsubova algoritmu.

Tento předpoklad je nadále využit i při implementaci SSA, díky kterému se zjednodušila jeho implementace. Také je pak velice jednoduché použít Karatsubův algoritmus v SSA.

3.1.2 Nastavení kompilátoru

Použitý kompilátor je g++ verze 5.4.0 s přepínači `-Wall -pedantic -std=c++11`.
Dále pro optimalizaci a paralelizaci bylo použito:

- `-O3`
- `-fopt-info-vec`
- `-ffast-math`
- `-ftree-vectorize`

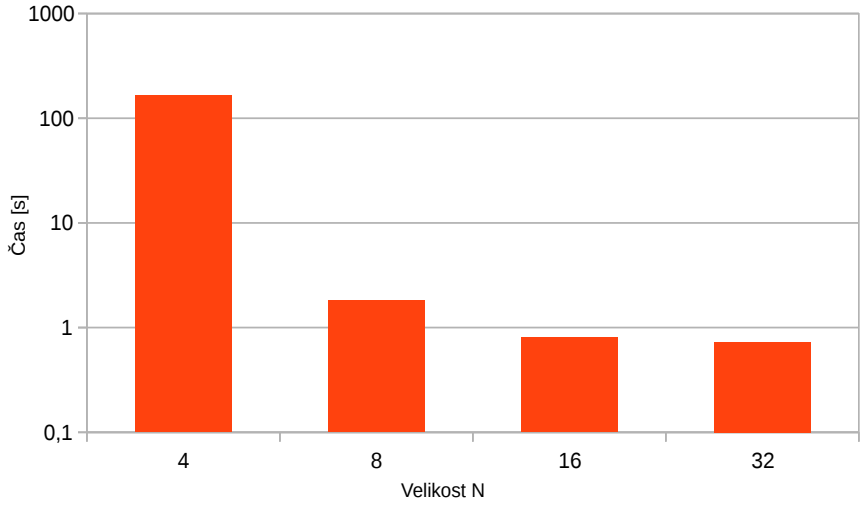
3.1.3 Testovací server

Měření a testování probíhalo na fakultním výpočetním serveru STAR (star.fit.cvut.cz) [5].
Parametry:

- CPU: 2x 6core Xeon 2620 v2 @ 2.1Ghz, celkem 12 vláken
- RAM: 32 GB

3.2 Vliv implementace

U Schönhage–Strassenův algoritmus bylo potřeba zvolit velikost N, jak je vidět na grafu níže, nejlépe se jeví hodnota 32.



Obrázek 3.1: Volba N u SSA

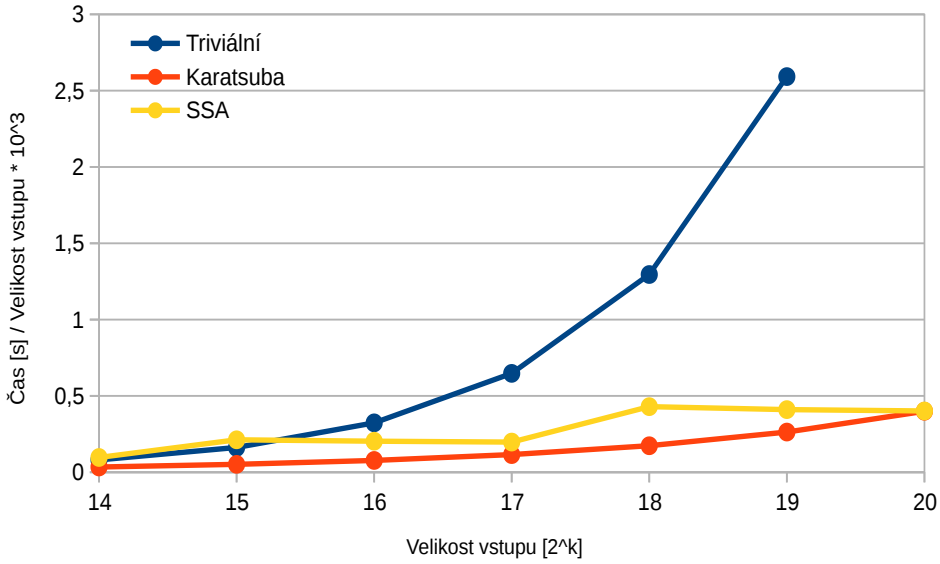
Byla ale zvolena hodnota 16, protože podíl zlepšení mezi velikostí 32 a 16 je minimální a s rostoucí hodnotou N dochází k dřívějšímu použití triviálního algoritmu u párového násobení, proto může být rychlejší. Měření proběhlo na polynomu velikosti 2^{13}

Dále hodnota N je závislá na velikosti modula m a *kořenu jednoty* a. Musí platit:

$$a^N \equiv 1 \pmod{m} \quad \wedge \quad a^{N/2} \not\equiv 1 \pmod{m}$$

Proto není možné v modulu $2^{16}+1$ zvolit hodnotu N větší jak 32, specificky 64, 128, atd.

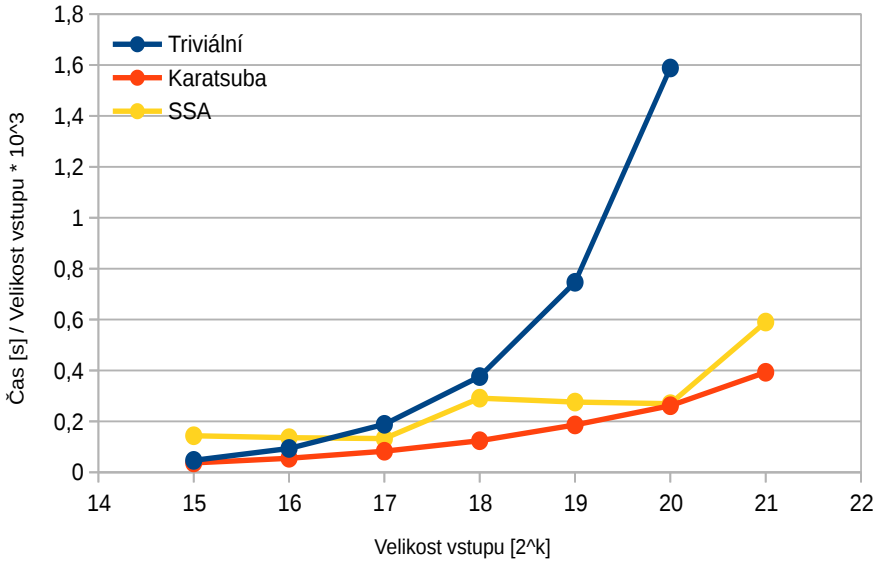
Hodnoty změřených naivních implementací, graf 3.2, svým výkonem rozhodně neoslní, nejlépe si vedl Karatsubův algoritmus, ale je též patrné, že na delších polynomech je dohnán SSA.



Obrázek 3.2: Porovnání naivních implementací

3.3 Vliv optimalizace

Na grafu 3.3 je patrné, že vlivem zapnutých kompilátorových optimalizací, došlo k výraznému zrychlení implementací.



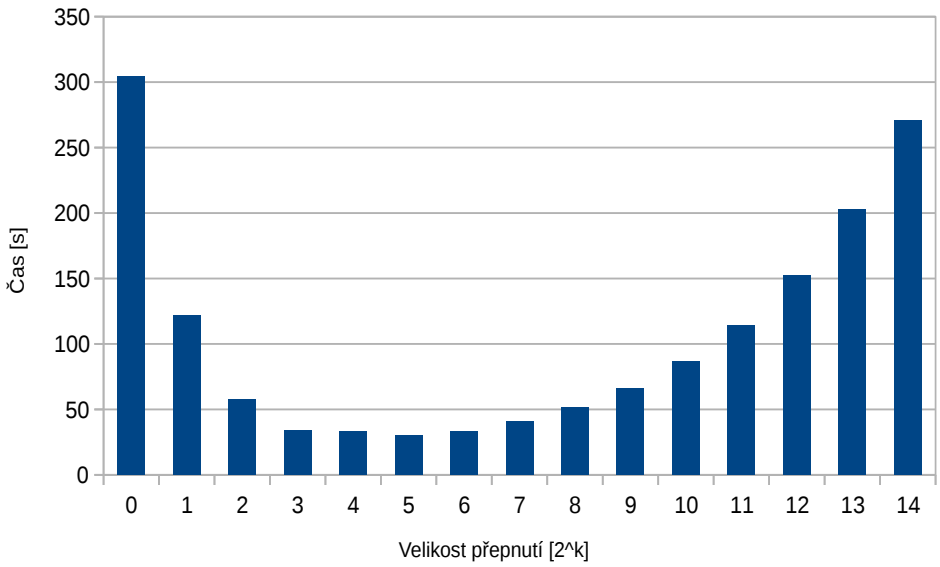
Obrázek 3.3: Porovnání algoritmů s kompilátorovými optimalizacemi

Následující grafy zobrazují přepnutí implementací na jednodušší algoritmy. Přepnutí bylo vždy měřeno na polynomu velikosti 2^{20} a na optimalizovaných implementací algoritmů pomocí kompilátoru.

Nejvhodnější hranice se může lišit pro polynomy různého stupně, zejména pro nízké nebo vysoké stupně. Pro většinu polynomů bude takto určená hranice vyhovující volbou.

3.3.1 Karatsubův algoritmus

Jako první je zde uveden graf přepnutí Karatsuby na triviální algoritmus.



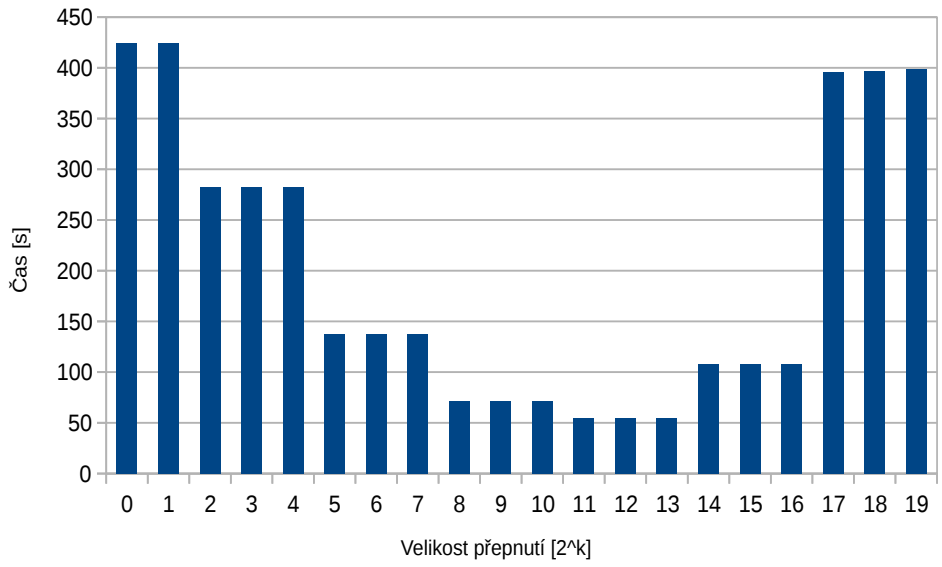
Obrázek 3.4: Přepnutí Karatsuby na triviální algortimus

Z grafu je patrné, že nejlepší hodnota pro přepnutí je 2^5, tedy je vhodné přepnout Karatsubův algoritmus na triviální při násobení polynomů délky 32.

3.3.2 Schönhage–Strassenův algoritmus

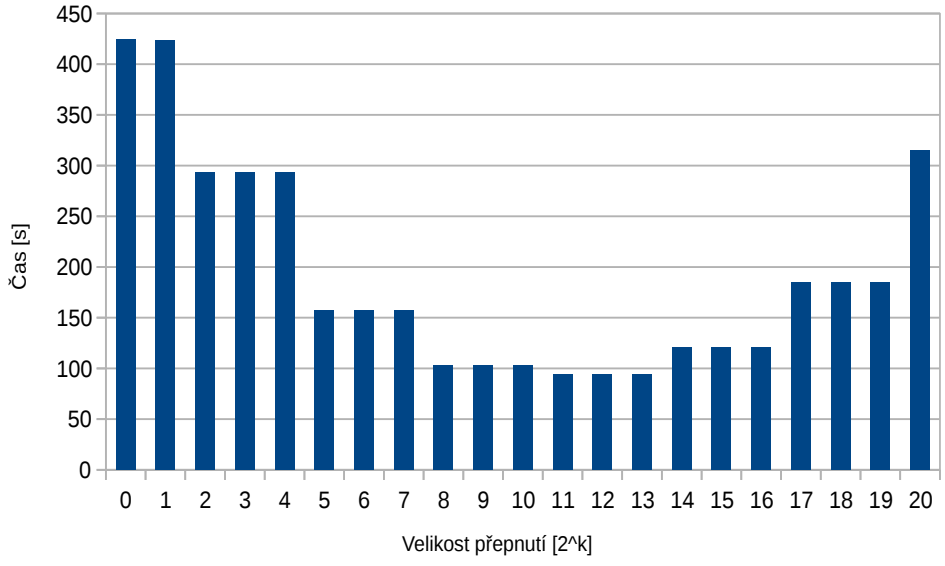
Přepnutí SSA na triviální algoritmus je vhodné při velikosti 2^{11} až 2^{13} členů polynomů, jak je viditelné na grafu 3.5.

Také z grafu 3.5 vyplývá, že rychlost výpočtu násobku se mění s velikostí N, graf je zarovnán po trojicích, v tomto případě je $N = 16$. Protože v implementaci SSA je zvolena *acyklická konvoluce*, pole pro uložení polynomu využívá jen polovinu své kapacity, tedy 2^3. Proto se rychlost mění vždy o 3 dílky mocniny čísla 2.



Obrázek 3.5: Přepnutí SSA na triviální algoritmus

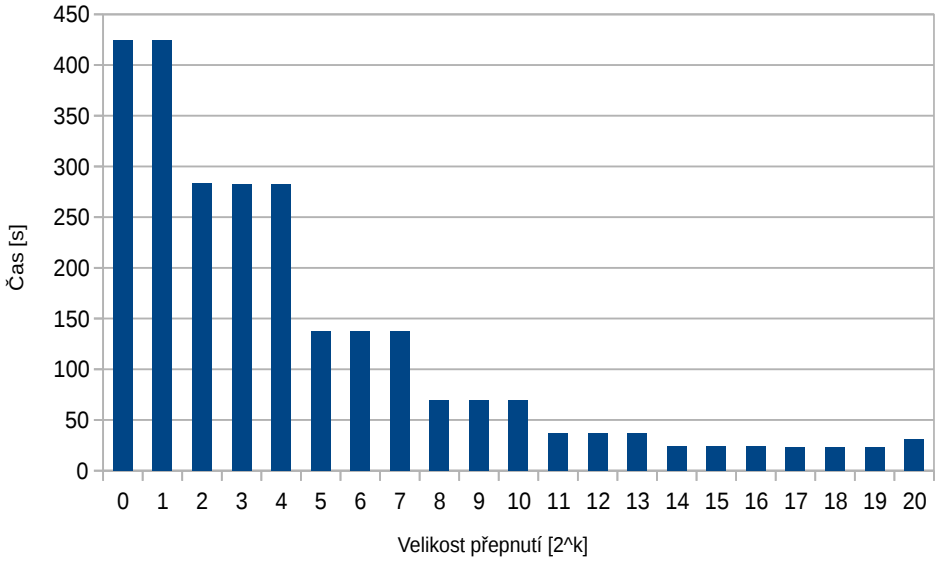
Graf 3.6 zobrazuje přepnutí Schönhage–Strassenova algoritmu na Karatsubův algoritmus, který nevyužívá optimalizaci přepnutí Karatsuby na triviální algoritmus.



Obrázek 3.6: Přepnutí SSA na Karatsubův algoritmus

Z grafu 3.6 opět vyplývá, že vhodná hodnota pro přepnutí je 2^{11} až 2^{13} členy polynomu.

Poslední graf 3.7 přepnutí ukazuje přepnutí SSA na optimalizovaný Karatsubův algoritmus pomocí přepnutí na triviální algoritmus při hodnotě délky polynomu 2^5.



Obrázek 3.7: Přepnutí SSA na Kara32

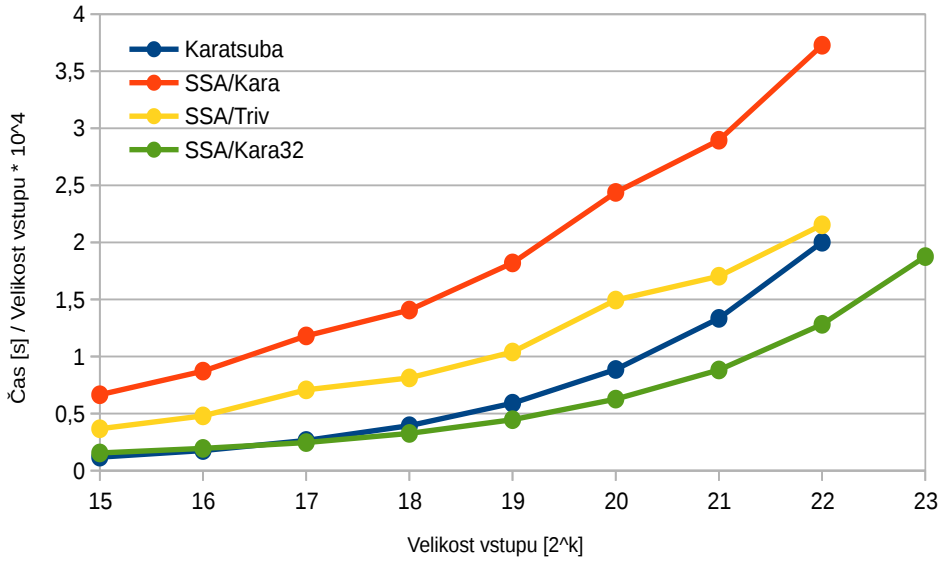
Graf 3.7 nám ukazuje, že je vhodné přepnout SSA při 2^{19} členů polynomu, ale protože měření bylo prováděno na polynomech délky 2^{20}, tak je to trochu zvláštní výsledek.

Když se podíváme na poslední údaj v grafu 3.7, na hodnotu 2^{20}, tak výsledný čas měření odpovídá času samostatného Karatsuby s přepnutím na hodnotě 2^5. To je v pořádku, protože pouze došlo k použití Karatsuby, viz první podmínka v pseudokódu na straně 8.

To znamená, že je vhodné volat SSA jen do určité hloubky rekurze, v tomto případě rovnou použít, při kroku násobení párů v algoritmu, Karatsubův algoritmus s přepnutím na hodnotě 2^5.

Na následujícím grafu 3.8 jsou zachyceny časy implementací algoritmů s nejlepší hranicí přepnutí na jednodušší algoritmus pro různě velké vstupy. Na první pohled je ihned viditelné, že implementace *SSA/Kara32* je nejrychlejší. Pro *SSA/Kara* platí, že je pomalá vlivem režie rekurze Karatsuby, zatímco rychlost *SSA/Triv* se pro delší polynomy přibližuje ke Karatsubovi a určitě by ho i překonal.

Za zmínku stojí, že vlivem modula $2^k + 1$ je možné nahradit většinu násobení bitovými posuny a tím dosáhnout dalšího zrychlení.



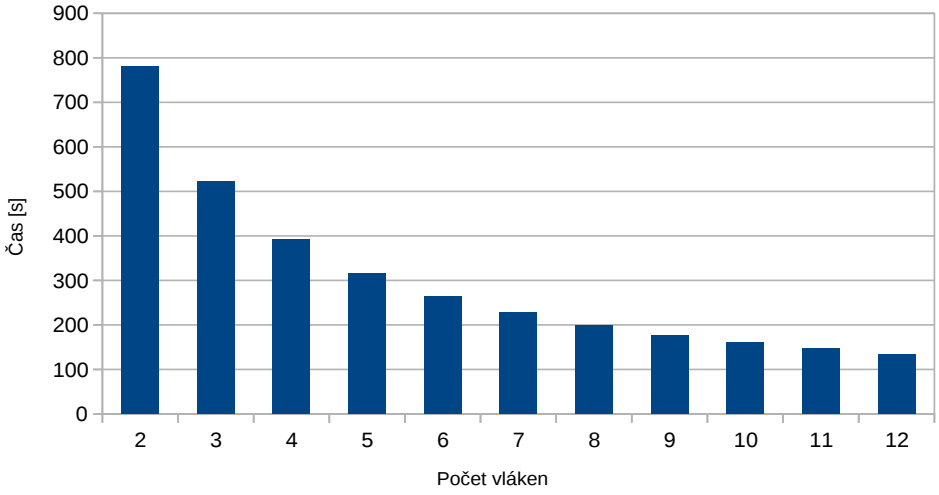
Obrázek 3.8: Porovnání optimalizovaných algoritmů

Legenda ke grafu 3.8:

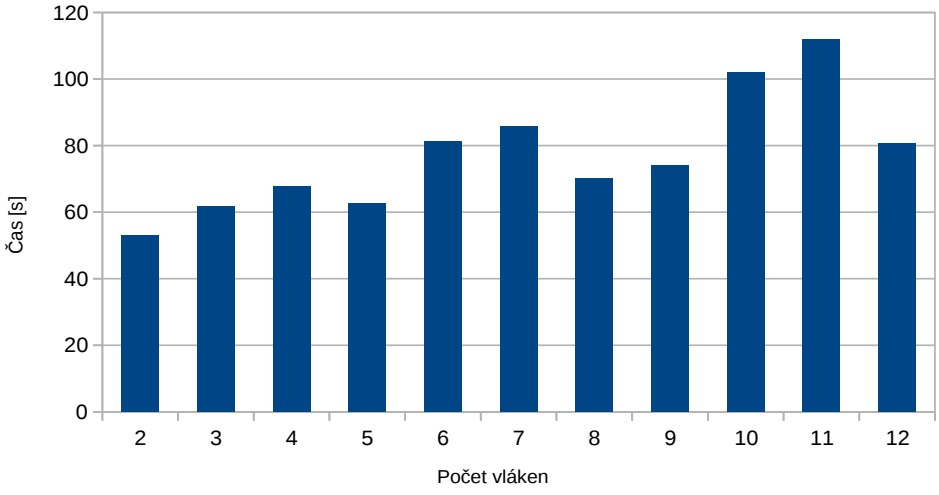
- Karatsuba - Karatsubův algoritmus s přepnutím na triviální algoritmus na hodnotě 2^5 délky polynomů.
- SSA/Kara - SSA s přepnutím na Karatsubův algoritmus na hodnotě 2^{12}.
- SSA/Triv - SSA s přepnutím na triviální algoritmus na hodnotě 2^{12}.
- SSA/Kara32 - SSA, pro násobení párů používá Karatsubův algoritmus s přepnutím na hodnotě 2^5.

3.4 Vliv paralelizace

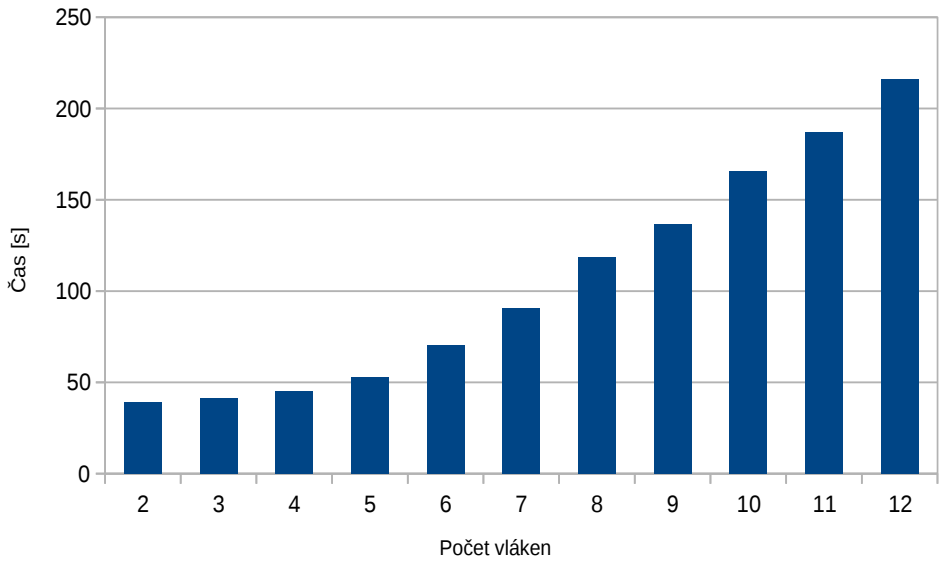
Následující grafy zobrazují paralelní implementace algoritmů pro různý počet vláken. Tyto implementace byly vždy měřeny na polynomech velikosti 2^{20} a na nejrychlejších optimalizovaných verzí implementací.



Obrázek 3.9: Paralelizace triviálního algoritmu



Obrázek 3.10: Paralelizace *for* cyklů u Karatsuby



Obrázek 3.11: Paralelizace rekurze u Karatsuby

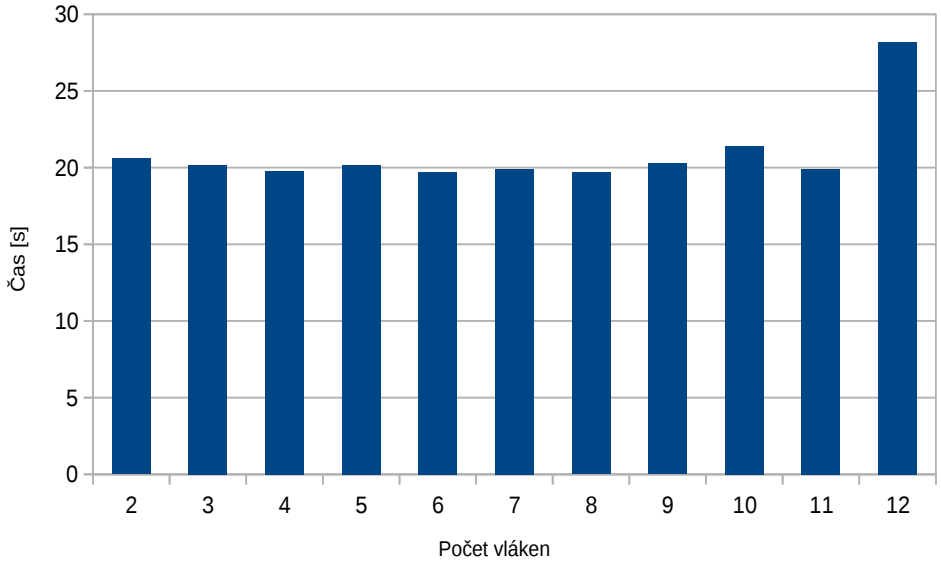
Na prvním grafu 3.9 je vidět, že s rostoucím počtem vláken dochází ke zrychlení triviálního algoritmu. Při počtu 12 vláken byla implementace na polynomu délce 2^{20} zrychlena až 12.3-krát. To znamená, že paralelizovaný triviální algoritmus je velice výkonný oproti optimalizované verzi.

Zatímco další dva grafy, 3.10 a 3.11, žádné zrychlení s rostoucím počtem vláken neukazují.

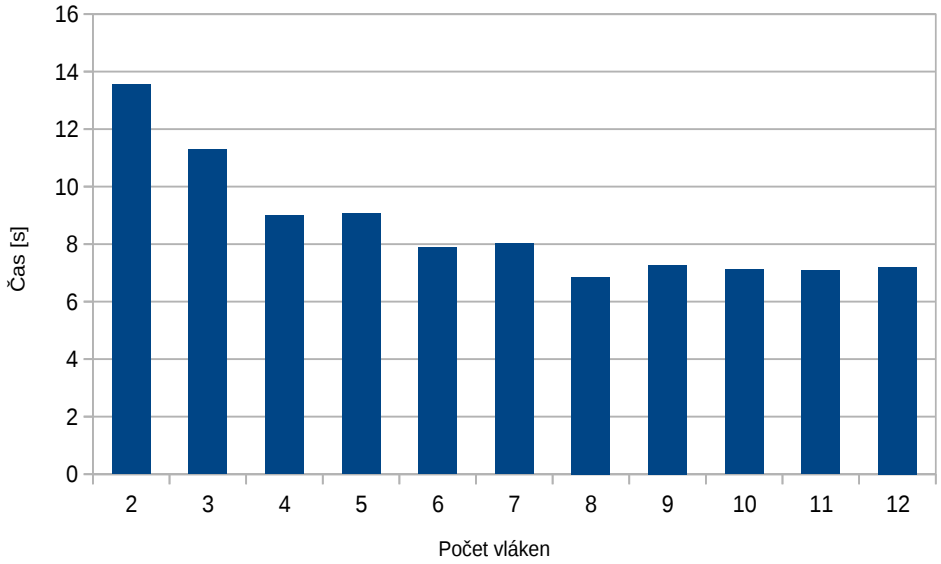
U grafu 3.10 je to nejspíše vlivem režie direktivy *for*, kdy dochází k jejímu častému volání, a tak je Karatsubův algoritmus zpomalován.

Graf 3.11 zobrazuje implementaci využívající direktivy *task* pro rekurzivní volání Karatsuby, ale jak je na grafu vidět, tak k žádnému zrychlení nedošlo. Nejspíše za to může přílišné rozdrobení rekurze na mnoho *tasků* a takto získaná režie navíc způsobuje zpomalení implementace. Toto rozdrobení by ale mělo částečně řešit přepnutí Karatsuby na triviální algoritmus, tedy pokud by došlo k přepnutí na vyšší velikosti polynomu, tak by se mohlo dosáhnout lepších výsledků.

Vzhledem k zhoršení rychlostí obou paralelizací Karatsubova algoritmu, nemá smysl měřit implementaci spojující oba přístupy.

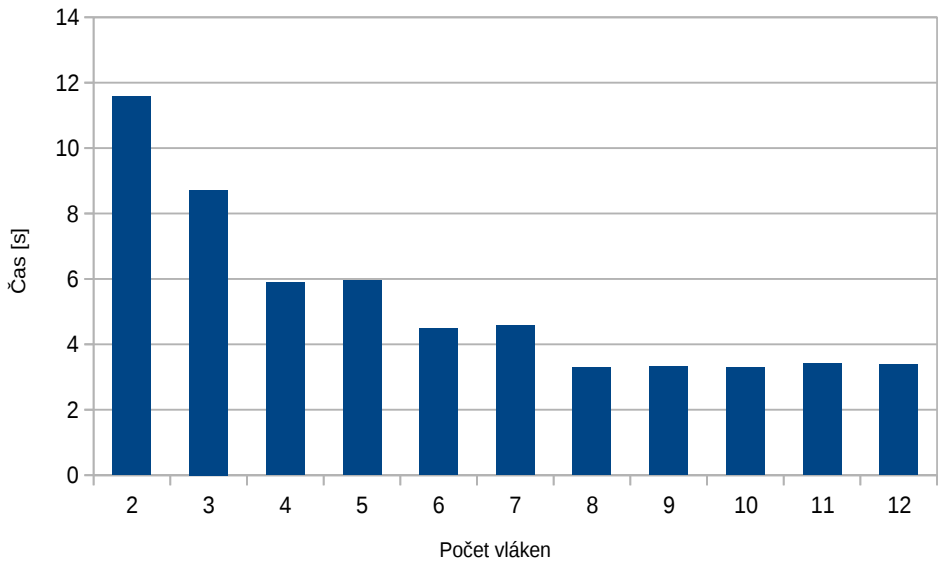


Obrázek 3.12: Paralelizace *for* cyklů u SSA



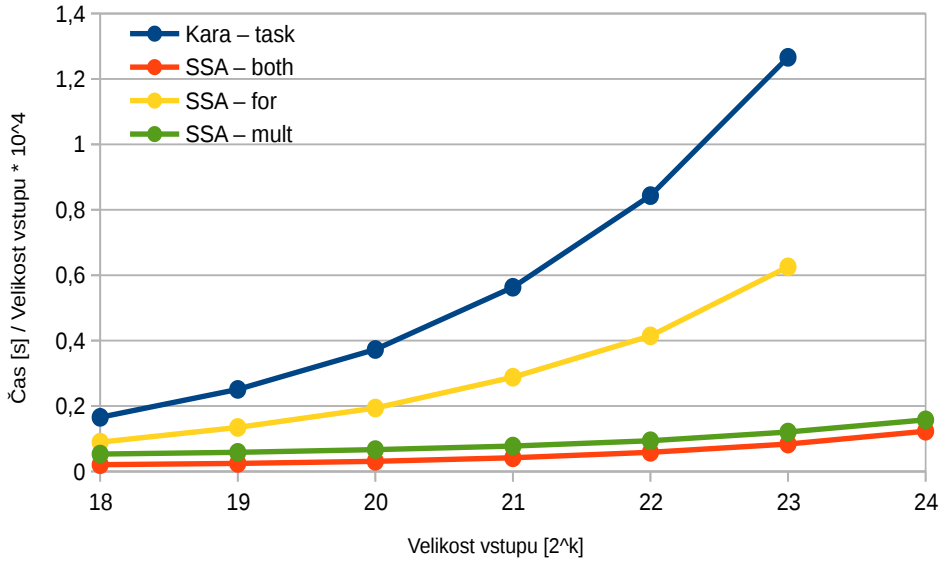
Obrázek 3.13: Paralelizace násobení u SSA

Z grafů 3.12 a 3.13 je patrné, že optimální počet vláken je 8. Tato hodnota není náhodná, protože závisí na velikosti N. Zajímavé by bylo změřit implementaci pro 16 vláken, ale vlivem maximálního počtu 12 vláken na serveru Star by k žádnému dalšímu zrychlení nedošlo.



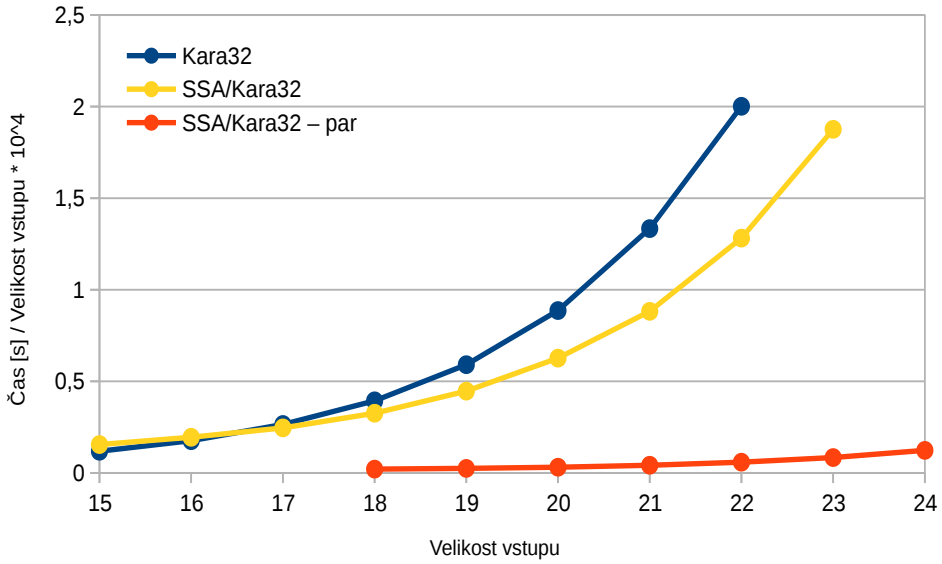
Obrázek 3.14: Paralelizace násobení a *for* cyklů u SSA

Kombinaci paralelizace *for* a párového násobení v SSA je vidět v grafu 3.14. Pro tuto implementaci je optimální počet vláken 10, kdy 8 vláken zaměstná výpočet násobení a zbytek je využit pro *for* cykly.



Obrázek 3.15: Porovnání paralelizovaných algoritmů

Na grafu 3.15 je patrné, že velký vliv na paralelizace SSA má paralelizace párového násobení. Pokud se přidá i paralelizace *for* cyklů, dojde k dalšímu zrychlení, i když ne k tak významnému.



Obrázek 3.16: Porovnání paralelizovaných a optimalizovaných algoritmů

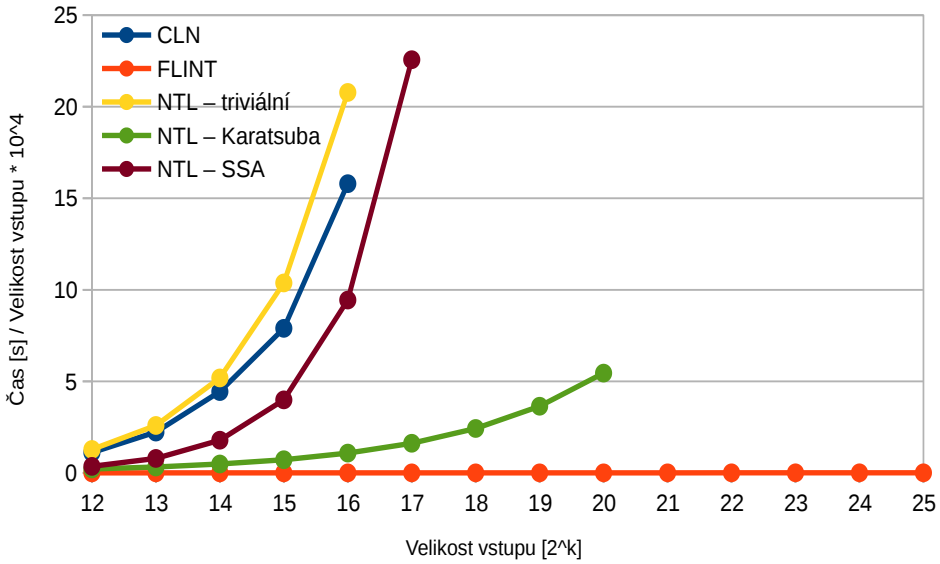
Legenda ke grafům 3.15 a 3.16:

- Kara – task - Paralelizovaný Karatsubův algoritmus pomocí direktivy *task*.
- SSA – both - SSA s paralelizací *for* cyklů a párového násobení.
- SSA – for - SSA, paralelizace *for* cyklů.
- SSA – mult - SSA, paralelizace párového násobení.
- Kara32 - Karatsubův algoritmus s přepnutím na triviální algoritmus při délce polynomu 2^5.
- SSA/Kara32 - SSA s přepnutím na Kara32.
- SSA – par - SSA s paralelizací *for* cyklů a párového násobení.

Triviální algoritmus není již v grafech zobrazen vzhledem k jeho velké časové náročnosti.

3.5 Existující implementace

Pro porovnání s dalšími existujícími implementacemi algoritmů pro násobení polynomů byly použity knihovny NTL verze 10.3.0, CLN verze 1.3.4, FLINT verze 2.5.2 a GMP verze 6.1.2. Každá knihovna využívá pro násobení různé algoritmy podle délky polynomu, kromě NTL, kdy toto rozpoznání nefungovalo na serveru Star, proto jsou implementace změřeny samostatně.



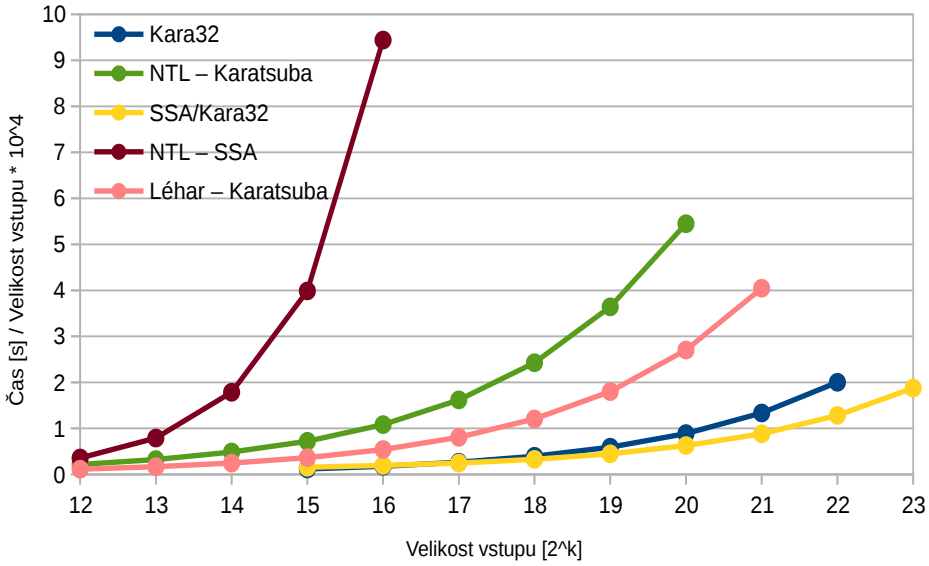
Obrázek 3.17: Porovnání existujících implementací

Na grafu 3.17 je porovnání cizích implementací mezi sebou. Na první pohled je patrné, že FLINT je nejrychlejší a dosahuje vysoké rychlosti i pro velice vysoké délky polynomů.

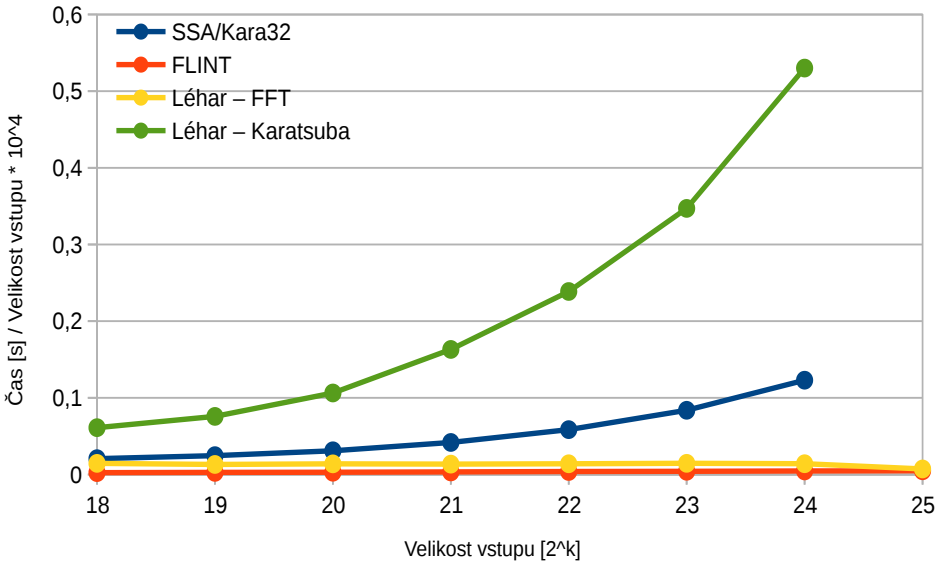
Z grafu pro CLN vyplývá, že pro násobení polynomů se používá pouze triviální algoritmus a tedy k přepnutí na lepší algoritmus podle délky polynomu nedošlo.

Následující graf 3.18 zobrazuje porovnání nejlepších sekvenčních optimalizovaných implementací z této bakalářské práce s implementacemi z knihoven a bakalářské práce Adama Léhara [12]. Na něm se ukazuje, že knihovní násobení polynomů má stále prostor pro zlepšení.

Poslední graf 3.19 ukazuje porovnání paralelních implementací, kde jednoznačně vede FLINT s těsně následující implementací FFT od Léhara.



Obrázek 3.18: Porovnání sekvenčních implementací



Obrázek 3.19: Porovnání paralelních implementací

Závěr

Cílem této bakalářské práce byla analýza a implementace algoritmů pro násobení polynomů, konkrétně triviální, Karatsubův a Schönhage–Strassenova, jejich optimalizace a paralelizace pomocí knihovny OpenMP. Tento cíl byl rozhodně splněn.

V první kapitole jsou popsány a definovány pojmy použité v bakalářské práci, následuje popis algoritmů a dalších existujících implementací.

Druhá kapitola se zabývá implementací algoritmů, jejich optimalizací a paralelizací.

Ve třetí a poslední kapitole se nachází popis testovacího prostředí a použitých prostředků. Obsahuje také měření a testování implementací na výpočetním serveru. Algoritmy byly následně porovnány mezi sebou a také s dalšími existujícími implementacemi.

Literatura

[1] SCHÖNHAGE, A., STRASSEN V. Schnelle Multiplikation großer Zahlen. *Computing 7.* 1971, str. 281–292.

[2] WEIMERSKIRCH, A., PAAR, C. *Generalizations of the Karatsuba Algorithm for Efficient Implementations.* IACR Cryptology ePrint Archive, 2006: str. 224.

[3] FÜRER, M. Faster Integer Multiplication, *Proceedings of the 39th annual ACM Symposium on Theory of Computing.*, 2007, str. 55-67

[4] ŠIMEČEK, I. *Paralelní programování v OpenMP.* [online]. 2016, [cit. 2017-04-1]. Dostupné z: `https://edux.fit.cvut.cz/courses/MI-PDP.16/_media/lectures/mi-pdplecture05-openmpprogramming.pdf`

[5] ŠIMEČEK, I.: *Výpočetní prostředky.* [online]. 2015, [cit. 2017-04-1]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-EIA/tutorials/vyp_pr`

[6] ŠIMEČEK, I. *Použití vektorizace v C/C++ (GCC a ICC).* [online]. 2016, [cit. 2017-04-1]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/vektorizace.pdf`

[7] ŠIMEČEK, I. *Technologie OpenMP.* [online]. 2016, [cit. 2017-04-1]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/omp.pdf`

[8] HAIBLE, B.; KRECKEL, R. B.: *CLN, a Class Library for Numbers.* [on-line]. 2014, [cit. 2017-05-1]. Dostupné z: `https://www.ginac.de/CLN/cln.html#Speed-efficiency`

[9] SHOUP, V.: *A Tour of NTL: NTL Implementation and Portability.* [online]. 2016, [cit. 2017-05-1]. Dostupné z: `http://www.shoup.net/ntl/doc/tour-impl.html`

[10] HART, W.: *FLINT: Fast Library for Number Theory.* [on-line]. 2016, [cit. 2017-05-1]. Dostupné z: `http://www.shoup.net/ntl/doc/tour-impl.html`

[11] KORTEKAAS, T.: *Multiplying large numbers and the Schönhage-Strassen Algorithm.* [on-line]. 2015, [cit. 2017-04-1]. Dostupné z: `https://tonjanee.home.xs4all.nl/SSAdescription.pdf`

[12] Léhar, A. *Rychlé násobení smíšených polynomů.*, Praha, 2014, Bakalářská práce, ČVUT, Fakulta informačních technologií

Příloha **A**

Seznam použitých zkratek

SSA Schönhage–Strassenův algoritmus

FFT Fast Fourier transform (Rychlá Fourierova transformace)

NTT Number-theoretic transform

DFT Discrete Fourier transform (Diskrétní Fourierova transformace)

iDFT Inverse Discrete Fourier transform (Inverzní Diskrétní Fourierova transformace)

GMP GNU Multiple-Precision Library

FLINT Fast Library for Number Theory

NTL A Library for Doing Number Theory

CLN Class Library for Number

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `input_data` ..... vstupní data
    - `thesis`
      - `naive` ..... naivní implementace
      - `optimalized` ..... optimalizované implementace
      - `parallel` ..... paralelní implementace
    - `other`
      - `NTL` ..... NTL implementace
      - `CLN` ..... CLN implementace
      - `FLINT` ..... FLINT implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `images` ..... obrázky použité v práci
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF