ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Efektivní násobení polynomů
Student:	Michal Číla
Vedoucí:	Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

1. Nastudujte algoritmy pro efektivní násobení polynomů, konkrétně algoritmy Toom-Cook, Karatsuba a FFT.
2. Implementujte tyto algoritmy v jazyce C/C++ a zlepšete jejich časovou a paměťovou složitost s využitím tranformace zdrojového kódu za účelem efektivního využití skryté paměti a následné paralelizace zdrojového kódu s využitím OpenMP.
3. Otestujte algoritmy na zdrojová data získaná z veřejně dostupných zdrojů nebo na vlastní vygenerovaná data a proveďte měření.
4. Prozkoumejte numerickou stabilitu implementovaných algoritmů a diskutujte použitelnost algoritmů z tohoto hlediska.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

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

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 11. prosince 2015

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

Bakalářská práce

Efektivní násobení polynomů

Michal Číla

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

11. května 2016

Poděkování

Rád bych poděkoval vedoucímu práce, panu Ing. Ivanu Šimečkovi, Ph.D., za jeho ochotu a cenné rady, které mi poskytl při psaní této práce. Dále bych rád poděkoval své rodině za podporu nejen při studiu.

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 11. května 2016 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Číla, Michal. *Efektivní násobení polynomů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá efektivními algoritmy pro násobení polynomů, konkrétně algoritmy Karatsuba, Toom-Cook a Rychlá Fourierova transformace. Mimo jiné se práce zaměřuje na paralelizaci těchto algoritmů v jazyce C++ za použití knihovny OpenMP. V práci je dále provedeno měření skutečné efektivity a numerické stability implementovaných algoritmů.

Klíčová slova Polynom, násobení, Karatsuba, Toom-Cook, FFT, paralelizace, numerická stabilita.

Abstract

This thesis deals with efficient algorithms used for polynomial multiplication, particular with algorithms Karatsuba, Toom-Cook and Fast Fourier transform. Among other things this thesis focuses on parallelization of these algorithms in programming language C++ using OpenMP library. There is also testing of real effectiveness and numerical stability of implemented algorithms.

Keywords Polynomial, multiplication, Karatsuba, Toom-Cook, FFT, parallelization, numerical stability.

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Úvod

Polynomy jsou dnes využívány v mnoha odvětvích, používají se například v ekonomice pro analýzu nákladů, ve fyzice pro popis trajektorií projektil nebo v podnikání pro modelování obchodu, tedy například, jak bude ovlivňovat prodej zboží při zvyšování jeho ceny. Přímo násobení polynomů se využívá například v kryptografii pro výpočet sdíleného klíče v bezdrátových senzorových sítích.

Cílem této práce je rozbor efektivních algoritmů pro násobení polynomů. Konkrétně se jedná o algoritmy Karatsuba, Toom-Cook a Rychlá Fourierova transformace.

Algoritmy pro násobení polynomů jsem implementoval v programovacím jazyce C++ a následně paralelizoval za použití knihovny OpenMP.

Tato práce se dále zabývá testováním časové efektivity implementovaných algoritmů a testováním využití skryté paměti na dvou výpočetních strojích. Testování je realizováno srovnáním času výpočtu jedním i více vlákny.

Nakonec jsem v práci prozkoumal numerickou stabilitu implementovaných algoritmů a diskutoval použitelnost algoritmů z tohoto hlediska.

Kapitola 1

Základní pojmy a algoritmy

1.1 Polynomy

V této kapitole je čerpáno z [1].

Polynom *Polynom* je komplexní funkce komplexní proměnné, tedy $p : \mathbf{C} \to \mathbf{C}$, která má pro všechna $x \in \mathbf{C}$ funkční hodnotu $p(x)$ danou vzorcem

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i,$$

kde $a_0, \ldots, a_n$ jsou nějaká reálná či komplexní čísla, která nazýváme *koeficienty polynomu*.

Stupeň polynomu *Stupeň polynomu* s koeficienty $a_0, \ldots, a_n$ je největší index $k \in \mathbf{N}$ takový, že $a_k \neq 0$. Pokud jsou všechny koeficienty nulové (nulový polynom), prohlásíme, že stupeň je roven -1.

Součet dvou polynomů Nechť p je polynom stupně m a q je polynom stupně n. *Součet dvou polynomů* p a q je funkce s dána předpisem $s(x) = p(x) + q(x)$ pro všechna $x \in \mathbf{C}$. Součet polynomů p a q značíme $p + q$. Polymom $p + q$ je stupně nejvýše $max(m,n)$.

Násobek polynomu skalárem Nechť p je polynom stupně m. *Násobek polynomu p skalárem α* je funkce t dána předpisem $t(x) = \alpha \cdot p(x)$ pro všechna $x \in \mathbf{C}$. Násobek polynomu p skalárem α značíme αp. Polynom αp je polynom stupně m pro $\alpha \neq 0$, pro $\alpha = 0$ je to polynom stupně -1.

Součin polynomů Nechť p je polynom stupně m a q je polynom stupně n. *Součin polynomů* p a q je funkce u daná předpisem $u(x) = p(x)q(x)$ pro všechna $x \in \mathbf{C}$. Součin polynomů p a q značíme pq. Pro nenulové polynomy

p a q je pq polynom stupně $m + n$. Je-li p nebo q nulový, pak pq je polynom stupně -1.

1.2 Asymptotická složitost algoritmů

Tato kapitola čerpá z [2].

Asymptotická horní mez: $\mathcal{O}$-notace Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je *nejvýše řádu* $g(n)$, psáno $f(n) = \mathcal{O}(g(n))$, jestliže

$$(\exists c \in \mathbf{R}^+)(\exists n_0 \in \mathbf{N}^+)(\forall n \geq n_0)\Big(f(n) \leq c \cdot g(n)\Big).$$

$\mathcal{O}$-notaci používáme pro odhady složitosti algoritmů v nejhorších případech.

Asymptotická dolní mez: Ω-notace Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je *nejméně řádu* $g(n)$, psáno $f(n) = \Omega(g(n))$, jestliže

$$(\exists c \in \mathbf{R}^+)(\exists n_0 \in \mathbf{N}^+)(\forall n \geq n_0)\Big(c \cdot g(n) \leq f(n)\Big).$$

Ω-notaci používáme pro odhady složitosti algoritmů v nejlepších případech.

Asymptotická těsná mez: Θ-notace Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je *téhož řádu jako* $g(n)$, psáno $f(n) = \Theta(g(n))$, jestliže

$$(\exists c_1,c_2 \in \mathbf{R}^+)(\exists n_0 \in \mathbf{N}^+)(\forall n \geq n_0)\Big(c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\Big).$$

Θ-notaci používáme pro co nejpřesnější odhady složitosti algoritmů.

1.3 Mistrovská metoda

V této kapitole je čerpáno z [3].

Nechť $a \geq 1$ a $b > 1$ jsou konstanty, $f(n)$ funkce jedné proměnné. Uvažujme rekurentní rovnici

$$t(n) = at\Big(\frac{n}{b}\Big) + f(n).$$

Pak $t(n)$ má jedno z následujících asymptotických řešení:

1. Pokud $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$ pro nějakou konstantu $\varepsilon > 0$, pak $t(n) = \Theta(n^{\log_b a})$.
2. Pokud $f(n) = \Theta(n^{\log_b a})$, pak $t(n) = \Theta(n^{\log_b a} \log n)$.

3. Pokud $f(n) = \Omega(n^{\log_b a+\varepsilon})$ pro nějakou konstantu $\varepsilon > 0$ a pokud $af(\frac{n}{b}) \leq cf(n)$ pro nějakou konstantu $c < 1$ a všechna $n \geq n_0$, pak $t(n) = \Theta(f(n))$.

1.4 Algoritmy pro násobení polynomů

1.4.1 Triviální algoritmus

Triviální algoritmus je základním algoritmem pro násobení dvou polynomů. Mějme polynomy A a B. Bez újmy na obecnosti předpokládejme, že oba polynomy jsou stupně n. Potom polynom $C = A \cdot B$ získáme triviálním algoritmem pomocí následujícího vzorce:

$$C(x) = \sum_{i=0}^{n} \sum_{j=0}^{n} a_i b_j x^{i+j}.$$

Složitost triviálního algoritmu Podle pseudokódu 1 určíme složitost triviálního algoritmu. `For` cyklus na řádcích 2 až 4 pouze nastaví všechny koeficienty polynomu C na nulu. Stupeň polynomu C je součtem stupňů polynomů A a B. Předpokládáme-li stejný stupeň n polynomů A a B, je složitost tohoto `for` cyklu $\Theta(2n) = \Theta(n)$. Dostáváme se ke dvěma `for` cyklům na řádcích 5 až 9. Zde je zřejmá složitost těchto cyklů $\Theta(n^2)$. Tím dostáváme celkovou složitost triviálního algoritmu pro násobení polynomů $\Theta(n + n^2) = \Theta(n^2)$.

Algoritmus 1 Pseudokód triviálního algoritmu pro násobení polynomů

Vstup: Polynomy A, B
Výstup: Polynom C takový, že $C = A \cdot B$

```
1:  C . degree = A . degree + B . degree
2:  for i ← 0 to C . degree do
3:      C[i] ← 0
4:  end for
5:  for i ← 0 to A . degree do
6:      for j ← 0 to B . degree do
7:          C[i + j] ← C[i + j] + A [i] · B[j]
8:      end for
9:  end for
10: return C
```

1.4.2 Algoritmus Karatsuba

Algoritmus Karatsuba byl představen Anatolym Karatsubou v roce 1962 [4]. Je založen na metodě *rozděl a panuj*, tedy na rozdělení problému na podproblémy menší velikosti. Karatsubův algoritmus dokáže násobit dva polynomy

	Alg. Karatsuba		Triviální alg.	
n	# násobení	# sčítání	# násobení	# sčítání
2^1	3	4	4	1
2^2	9	16	16	4
2^3	27	52	64	49
2^4	81	160	256	225
2^5	243	484	1024	961

Tabulka 1.1: Srovnání počtu násobení a sčítání triviálního a Karatsubova algoritmu

stejného stupně, a tento stupeň musí být ve tvaru $2^x - 1$, kde $x \in \mathbf{N}$. Počet koeficientů tedy musí být 2^x. Pokud na vstupu nejsou polynomy v tomto tvaru, na místo chybějících koeficientů se doplní nuly.

Vzhledem k tomu, že pro počítač je násobení náročnější operací oproti sčítání, je smyslem algoritmu Karatsuba snížit počet operací násobení oproti triviálnímu algoritmu, a tím snížit časovou náročnost běhu výpočtu. Ukažme si na příkladu, jak Karatsuba násobí dva polynomy stupně 1. Mějme polynomy A a B ve tvaru:

$$A(x) = a_0 + a_1 x, \qquad B(x) = b_0 + b_1 x$$

a proměnné D_0, D_1, D_{01} obsahující součiny koeficientů ve tvaru:

$$D_0 = a_0 b_0, \qquad D_1 = a_1 b_1, \qquad D_{01} = (a_0 + a_1)(b_0 + b_1),$$

potom součin polynomů A a B spočteme následujícím způsobem:

$$C(x) = A(x) \cdot B(x) = D_1 x^2 + (D_{01} - D_0 - D_1)x + D_0.$$

K tomu, abychom získali polynom C, bylo zapotřebí čtyř operací sčítání, ale co je důležité, pouze tří operací násobení. Když použijeme triviální algoritmus pro polynomy stejného stupně, potřebujeme k výpočtu čtyři operace násobení a jednu operaci sčítání. Algoritmus Karatsuba tedy na násobení polynomů stupně 1 ušetří oproti algoritmu triviálnímu jednu operaci násobení za cenu tří operací sčítání navíc. V tabulce 1.1 vidíme srovnání počtu operací násobení a sčítání Karatsubova a triviálního algoritmu pro malá n.

Tento algoritmus lze použít rekurzivně, jak je vidět na řádcích 8 až 10 v pseudokódu 2 na stránce 7. Polynomy A a B jsou rozděleny na spodní a horní část takto:

$$A(x) = A_l(x) + A_u(x)x^{\frac{n}{2}}, \qquad B(x) = B_l(x) + B_u(x)x^{\frac{n}{2}}.$$

S těmito částmi pak zacházíme, jako kdyby to byly koeficienty. Algoritmus se pak stane rekurzivním, pokud budeme stejný postup aplikovat na tyto poloviční polynomy. V posledním kroku už pouze vynásobíme dva koeficienty.

Jelikož algoritmus volá vždy sama sebe na polynom s poloviční počtem koeficientů, algoritmus skončí přesně po $\log_2 n$ krocích.

Algoritmus 2 Pseudokód rekurzivního algoritmu Karatsuba

Vstup: Polynomy A, B a počet koeficientů těchto polynomů n.
Výstup: Polynom C takový, že $C = A \cdot B$.

```
1:  procedure KARATSUBA(A, B, n)
2:      if n = 1 then
3:          C = A · B
4:          return C
5:      end if
6:      A_l ← Low(A), A_u ← Up(A)
7:      B_l ← Low(B), B_u ← Up(B)
8:      D_0 ← Karatsuba(A_l, B_l, n/2)
9:      D_1 ← Karatsuba(A_u, B_u, n/2)
10:     D_01 ← Karatsuba(A_l + A_u, B_l + B_u, n/2)
11:     C ← D_0 + (D_01 − D_0 − D_1)x^{n/2} + D_1 x^n
12:     return C
13: end procedure
```

Složitost algoritmu Karatsuba Pro výpočet složitosti algoritmu Karatsuba využijeme Mistrovskou metodu. Označme n jako původní počet koeficientů polynomů A a B. Algoritmus volá sama sebe na polynomy s poloviční počtem koeficientů a toto volání proběhne třikrát v těle algoritmu, máme tedy funkci $t(n)$ v následujícím tvaru:

$$t(n) = 3t\Big(\frac{n}{2}\Big) + f(n).$$

Nyní je potřeba vypočítat složitost funkce $f(n)$. Rozebereme si jednotlivě řádky pseudokódu 2. Řádky 2 až 5 jsou triviální. Složitost těchto operací je $\Theta(1)$. O řádcích 6 a 7 platí totéž. Používají se koeficienty původních polynomů A a B. Nejdůležitější je řádek 11. Zde je potřeba přepočítat koeficienty polynomu D_{01}. Zajímá nás tedy počet koeficientů tohoto polynomu. Polynom D_{01} (a polynomy D_0 a D_1) je součinem polynomů, které mají poloviční počet koeficientů, tedy stupeň těchto polynomů je $\frac{n}{2} - 1$. Jak víme, násobením polynomů vznikne polynom, jehož stupeň je roven součtu stupňů oněch polynomů. Stupeň polynomu D_{01} je tedy roven $2 \cdot (\frac{n}{2} - 1) = n - 2$. Počet koeficientů tohoto polynomu je $n - 1$. Následně se sestaví polynom C z polynomů D_0, D_{01} a D_1. Všechny tyto polynomy mají $n - 1$ koeficientů. Asymptotická složitost řádku 11 je tedy $\Theta(n)$. Toto je zároveň asymptotická složitost celého těla algoritmu. Konečný tvar funkce $t(n)$ je:

$$t(n) = 3t\Big(\frac{n}{2}\Big) + \Theta(n).$$

Protože $\log_2 3 \doteq 1{,}585$, dostáváme 1. případ řešení Mistrovské metody, jelikož $f(n) = \mathcal{O}(n^{1,585-\varepsilon}) \Leftrightarrow \varepsilon \doteq 0{,}585$. Celková asymptotická složitost algoritmu Karatsuba je $\Theta(n^{\log_2 3}) \doteq \Theta(n^{1,585})$.

1.4.3 Algoritmus Toom-Cook

Algoritmus Karatsuba je základem pro algoritmus známý pod názvem Toom-Cook. Je pojmenovaný po svém objeviteli Andrei Toomovi a Stephenu Cookovi, který dovedl myšlenku algoritmu do zdárného konce. Algoritmus vypočítává součin polynomů pomocí metody *rozděl a panuj*. Algoritmus má mnoho podob, správně se mu říká Toom-Cook-r, kde r značí, na kolik částí se rozdělí původní polynomy. Algoritmus Karatsuba je právě Toom-Cook-2. Čím vyšší r, tím nižší asymptotická složitost. Obecně má Toom-Cook-r podle [5] asymptotickou složitost $\mathcal{O}(n^{\frac{\log(2r-1)}{\log r}})$. Čím vyšší r, tím složitější je však algoritmus naimplementovat. V této práci se dále budeme zabývat algoritmem Toom-Cook-3, tento algoritmus je také nejznámější, budeme jej nadále označovat jen jako Toom-Cook algoritmus.

Stejně jako algoritmus Karatsuba, i algoritmus Toom-Cook násobí dva polynomy stejného stupně. Rozdíl je v počtu koeficientů. Ten musí být dělitelný třemi. Pokud tomu tak není, doplní se chybějící koeficienty nulami. Je zřejmé, že v jednom běhu algoritmu se bude doplňovat jeden, maximálně dva koeficienty. Jak bylo zmíněno výše, algoritmus Toom-Cook rozdělí původní polynomy A a B s n koeficienty na 6 polynomů s $\frac{n}{3}$ koeficienty, tedy na polynomy A_0, A_1, A_2, B_0, B_1 a B_2, jak je vidět na řádcích 6 a 7 v pseudokódu 3 na stránce 10. Polynomy A a B máme ve tvaru:

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{3}} + A_2(x)x^{\frac{2n}{3}},$$
$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{3}} + B_2(x)x^{\frac{2n}{3}}.$$

Nechť C je polynom takový, že $C = A \cdot B$ a je ve tvaru:

$$C(x) = C_0(x) + C_1(x)x^{\frac{n}{3}} + C_2(x)x^{\frac{2n}{3}} + C_3(x)x^{n} + C_4(x)x^{\frac{4n}{3}}.$$

Potřebujeme zjistit koeficienty polynomů C_0, C_1, C_2, C_3 a C_4. Triviální myšlenkou je provést 9 násobení:

$$\begin{aligned}
C_0 &= A_0 \cdot B_0,\\
C_1 &= A_1 \cdot B_0 + A_0 \cdot B_1,\\
C_2 &= A_2 \cdot B_0 + A_1 \cdot B_1 + A_0 \cdot B_2,\\
C_3 &= A_2 \cdot B_1 + A_1 \cdot B_2,\\
C_4 &= A_2 \cdot B_2,
\end{aligned}$$

místo toho však podle [6] je výpočet těchto polynomů v algoritmu Toom-Cook řešený vyhodnocením polynomu C v 5 bodech, a to konkrétně 0, 1, -1, 2 a ∞. Tato vyhodnocení polynomu jsou opět polynomy, označme je W_0, W_1, W_{-1}, W_2 a W_∞. Tyto polynomy jsou získány 5 operacemi násobení:

$$\begin{aligned} W_0 &= A_0 \cdot B_0, \\ W_1 &= (A_0 + A_1 + A_2) \cdot (B_0 + B_1 + B_2), \\ W_{-1} &= (A_0 - A_1 + A_2) \cdot (B_0 - B_1 + B_2), \\ W_2 &= (A_0 + 2A_1 + 4A_2) \cdot (B_0 + 2B_1 + 4B_2), \\ W_\infty &= A_2 \cdot B_2. \end{aligned}$$

Získané polynomy jsou ve skutečnosti lineární kombinací hledaných polynomů C_0, C_1, C_2, C_3 a C_4, jedná se o soustavu 5 rovnic o 5 neznámých:

$$\begin{aligned} W_0 &= C_0, \\ W_1 &= C_0 + C_1 + C_2 + C_3 + C_4, \\ W_{-1} &= C_0 - C_1 + C_2 - C_3 + C_4, \\ W_2 &= C_0 + 2C_1 + 4C_2 + 8C_3 + 16C_4, \\ W_\infty &= C_4. \end{aligned}$$

Tato soustava je vyřešena několika elementárními operacemi sčítání, odečítání polynomů a násobení, dělení polynomů malým číslem. Označme si pomocné polynomy T_1 a T_2. Polynomy C_0, C_1, C_2, C_3 a C_4 získáme následujícími rovnicemi:

$$\begin{aligned} T_1 &= \frac{3W_0 + 2W_{-1} + W_2}{6} + 2W_\infty, \\ T_2 &= \frac{W_1 + W_{-1}}{2}, \end{aligned}$$

$$\begin{aligned} C_0 &= W_0, \\ C_1 &= W_1 - T_1, \\ C_2 &= T_2 - W_0 - W_\infty, \\ C_3 &= T_1 - T_2, \\ C_4 &= W_\infty. \end{aligned}$$

Celý postup je možné aplikovat rekurzivně, jak vidíme na stránce 10 v pseudokódu 3 na řádcích 8 až 12. Jedná se právě o získání polynomů W_0, W_1, W_{-1}, W_2 a W_∞. Algoritmus zastaví rekurzivní volání v době, kdy je počet koeficientů vstupních polynomů roven jedné. Zde dojde k vynásobení dvou čísel, a to jediných koeficientů vstupních polynomů.

Složitost algoritmu Toom-Cook Rozeberme si dopodrobna asymptotickou složitost dále zkoumaného algoritmu Toom-Cook-3 = Toom-Cook. Mějme polynom A s n koeficienty. Stejně, jako tomu bylo u výpočtu složitosti algoritmu Karatsuba, využijeme k výpočtu složitosti algoritmu Toom-Cook Mistrovskou metodu. V pseudokódu 3 je složitost operací na řádcích 2 až 7 $\Theta(1)$. Řádky 8 až 12 znamenají rekurzivní volání algoritmu na polynomy s $\frac{n}{3}$ koeficienty. Funkce $t(n)$ má tedy tvar:

$$t(n) = 5t(\frac{n}{3}) + f(n).$$

Nyní zbývá dopočítat složitost operací na řádcích 13 až 17. Polynomy W_0, W_1, W_{-1}, W_2 a W_∞ mají $\frac{2n}{3} - 1$ koeficientů, neboť se jedná o součiny polynomů majících $\frac{n}{3}$ koeficientů. Na řádcích 13 a 17 dochází ke sčítání a odčítání polynomů s $\frac{2n}{3} - 1$ koeficienty. Složitost těchto operací je tedy $\Theta(n)$. Výsledný tvar funkce $t(n)$ je:

$$t(n) = 5t(\frac{n}{3}) + \Theta(n).$$

Dostáváme první případ řešení Mistrovské metody, neboť $\log_3 5 \doteq 1{,}465$ a $f(n) = \mathcal{O}(n^{1{,}465-\varepsilon}) \Leftrightarrow \varepsilon \doteq 0{,}465$. Celková asymptotická složitost algoritmu Toom-Cook je tedy $\Theta(n^{\log_3 5}) \doteq \Theta(n^{1{,}465})$, čímž jsme potvrdili tvrzení v [5] o asymptotické složitosti algoritmu Toom-Cook, protože $n^{\log_3 5} = n^{\frac{\log 5}{\log 3}}$.

Algoritmus 3 Pseudokód rekurzivního algoritmu Toom-Cook

Vstup: Polynomy A, B, a počet koeficientů těchto polynomů n.
Výstup: Polynom C takový, že $C = A \cdot B$.

1: **procedure** ToomCook(A, B, n)
2: **if** $n = 1$ **then**
3: $C = A \cdot B$
4: **return** C
5: **end if**
6: $A_0 \leftarrow \text{FirstThird}(A)$, $A_1 \leftarrow \text{SecondThird}(A)$, $A_2 \leftarrow \text{ThirdThird}(A)$
7: $B_0 \leftarrow \text{FirstThird}(B)$, $B_1 \leftarrow \text{SecondThird}(B)$, $B_2 \leftarrow \text{ThirdThird}(B)$
8: $W_0 \leftarrow \text{ToomCook}(A_0, B_0, \frac{n}{3})$
9: $W_1 \leftarrow \text{ToomCook}(A_0 + A_1 + A_2, B_0 + B_1 + B_2, \frac{n}{3})$
10: $W_{-1} \leftarrow \text{ToomCook}(A_0 - A_1 + A_2, B_0 - B_1 + B_2, \frac{n}{3})$
11: $W_2 \leftarrow \text{ToomCook}(A_0 + 2A_1 + 4A_2, B_0 + 2B_1 + 4B_2, \frac{n}{3})$
12: $W_\infty \leftarrow \text{ToomCook}(A_2, B_2, \frac{n}{3})$
13: $T_1 \leftarrow \frac{3W_0+2W_{-1}+W_2}{6} - 2W_\infty$
14: $T_2 \leftarrow \frac{W_1+W_{-1}}{2}$
15: $C_0 \leftarrow W_0, C_1 \leftarrow W_1 - T_1, C_2 \leftarrow T_2 - W_0 - W_\infty, C_3 \leftarrow T_1 - T_2, C_4 \leftarrow W_\infty$
16: $C \leftarrow C_0 + C_1 x^{\frac{n}{3}} + C_2 x^{\frac{2n}{3}} + C_3 x^n + C_4 x^{\frac{4n}{3}}$
17: **return** C
18: **end procedure**

1.4.4 Rychlá Fourierova transformace

Rychlá Fourierova tranformace (FFT) je další z algoritmů využívajících strategii *rozděl a panuj*. Má mnoho využití, mimo jiné je určena právě pro násobení polynomů.

Pro FFT je nutné mít na vstupu dva polynomy stejného stupně, tento stupeň musí být roven $2^x - 1$, $x \in \mathbf{N}$, polynomy tedy musí mít 2^x koeficientů. Pokud tomu tak není, doplní se polynomu patřičný počet nulových koeficientů. Základní myšlenkou celého algoritmu je fakt, že polynom lze vyhodnotit nejen pomocí koeficientů, ale také pomocí funkčních hodnot v několika pevně daných bodech. Takový vektor funkčních hodnot si můžeme představit jako graf polynomu.

Mějme polynom A s n koeficienty a_0, $a_1, \ldots,$ a_{n-1}. Na správné vyhodnocení polynomu v n bodech FFT využívá komplexních čísel, konkrétně primitivní n-té odmocniny z jedné. To je takové číslo $\omega_n \in \mathbf{C}$, pro které platí $\omega_n = e^{i\frac{2\pi}{n}}$. FFT vyhodnocuje polynom v bodech ω_n^0, $\omega_n^1, \ldots,$ ω_n^{n-1}. Jinými slovy, pro funkční hodnotu $\hat{a}_k$ platí $\hat{a}_k = A(\omega_n^k)$, $k \in \langle 0, n-1 \rangle$. [7]

O výpočet funkčních hodnot vstupních polynomů se stará rekurzivní část FFT. Vstupní polynom se rozdělí na dva polynomy A_e a A_o. Polynom A_e obsahuje pouze sudé koeficienty původního polynomu, tedy a_0, $a_2, \ldots,$ a_{n-2} a polynom A_o pouze liché koeficienty původního polynomu a_1, $a_3, \ldots,$ a_{n-1}. Jak vidíme na stránce 12 v pseudokódu 4 na řádcích 7 a 8, na tyto polynomy je pak dále rekurzivně volán stejný algoritmus. Získáme tedy polynomy F_e a F_o. Z těch se pak pomocí `for` cyklu na řádcích 9 až 13 poskládá výsledný polynom F, obsahující ohodnocení polynomu A v n bodech.

Poté se v nerekurzivní části, konkrétně na stránce 12 v pseudokódu 5 na řádku 6, vypočítají funkční hodnoty výsledného polynomu. Jedná se jednoduše o vynásobení získaných funkčních hodnot obou vstupních polynomů. Tyto funkční hodnoty je pak nutné převést zpět do reprezentace polynomu pomocí koeficientů. K tomu je opět využita rekurzivní část FFT, ovšem s tím rozdílem, že místo primitivní n-té odmocniny z jedné ω_n je použito číslo komplexně sdružené k ω_n, tedy $\overline{\omega}_n$. Získané koeficienty se pak ještě vydělí číslem n a získáváme výsledné koeficienty vzniklého polynomu.

Složitost Rychlé Fourierovy transformace K výpočtu složitosti FFT nám opět pomůže Mistrovská metoda. Nejprve si rozebereme složitost rekurzivní části tohoto algoritmu pomocí pseudokódu 4. Řádky 2 až 6 mají složitost $\Theta(1)$, řádky 7 až 8 nám dávají tvar $t(n)$:

$$t(n) = 2t\Big(\frac{n}{2}\Big) + f(n),$$

složitost funkce $f(n)$ určuje `for` cyklus na řádcích 9 až 11. Ten má složitost $\Theta(\frac{n}{2}) = \Theta(n)$, dostáváme tedy:

$$t(n) = 2t\Big(\frac{n}{2}\Big) + \Theta(n).$$

To nám dává druhý případ řešení Mistrovské metody, neboť $\log_2 2 = 1$. Funkce $f(n) = \Theta(n^{\log_2 2}) = \Theta(n) \Rightarrow t(n) = \Theta(n \log n)$, tedy celková složitost rekurzivní části Rychlé Fourierovy transformace je $\Theta(n \log n)$.

Co se složitosti celého algoritmu Rychlé Fourierovy tranformace týče, v pseudokódu 5 vidíme tři volání funkce `FFTRec` na polynomy s n koeficienty a na řádcích 5 až 7 `for` cyklus se složitostí $\Theta(n)$, celkem tedy dostáváme složitost $\Theta(3n \log n + n) = \Theta(n \log n)$.

Algoritmus 4 Pseudokód rekurzivní části Rychlé Fourierovy transformace

Vstup: Polynom A, počet koeficientů n polynomu A, primitivní n-tá odmocnina z jedničky ω_n.
Výstup: Pole F, obsahující vyhodnocení polynomu A v n bodech.

```
1:  procedure FFTRec(A, n, ω_n)
2:      if n = 1 then
3:          F[0] = A[0]
4:          return F
5:      end if
6:      A_e ← even(A), A_o ← odd(A)
7:      F_e ← FFTRec(A_e, n/2, ω_n^2)
8:      F_o ← FFTRec(A_o, n/2, ω_n^2)
9:      for i ← 0 to n/2 − 1 do
10:         F[i] ← F_e[i] + ω_n^i · F_o[i]
11:         F[i + n/2] ← F_e[i] − ω_n^i · F_o[i]
12:     end for
13:     return F
14: end procedure
```

Algoritmus 5 Pseudokód celého algoritmu Rychlé Fourierovy transformace

Vstup: Polynomy A, B, primitivní n-tá odmocnina z jedničky ω_n.
Výstup: Polynom C takový, že $C = A \cdot B$.

```
1:  procedure FFT(A, B, ω_n)
2:      n ← A . degree + 1
3:      F_A ← FFTRec(A, n, ω_n)
4:      F_B ← FFTRec(B, n, ω_n)
5:      for i ← 0 to n − 1 do
6:          F_C[i] ← F_A[i] · F_B[i]
7:      end for
8:      C ← 1/n · FFTRec(F_C, n, ω̄_n)
9:      return C
10: end procedure
```

Kapitola 2

Použité prostředky

V této kapitole jsou popsány všechny prostředky, které jsem v práci využil. Najdeme zde například datové struktury použité při implementaci algoritmů. V kapitole je ve stručnosti popsána knihovna OpenMP, která je důležitým stavebním prvkem mé implementace. Na závěr této kapitoly jsou představeny prostředky určené k testování implementovaných algoritmů, jejichž realizace je popsána v následující kapitole.

2.1 Formát vstupních dat

Implementované algoritmy jsem testoval na náhodně vygenerované polynomy, které nejsou kromě hlavní paměti nikde jinde uloženy. Program je však možné spouštět i na polynomy uložené v textových souborech. Tyto soubory udávají na prvním řádku stupeň polynomu n a na dalších maximálně $n + 1$ řádcích (nulové koeficienty nejsou v souborech obsaženy) je zadán stupeň polynomu k a koeficient a_k. Pro znázornění zde uvádím příklad formátu souboru obsahujícího polynom $p(x) = 10 + 20x + 30x^3 + 40x^4 + 50x^5$:

```
5
0 10
1 20
3 30
4 40
5 50
```

2.2 Použité datové struktury

Polynomy ve svém programu ukládám do členské proměnné třídy `CPolynomial`. Tato členská proměnná je pole typu `double` a obsahuje koeficienty polynomu. Dále je jako členská proměnná uložen `integer` obsahující stupeň daného polynomu.

Konstruktory obsahuje třída dva. Jeden je pro načtení polynomu ze souboru do paměti a druhý je pro vytvoření polynomu z existujícího pole obsahujícího koeficienty polynomu. Destruktor pouze uvolní z paměti alokované pole pro uložení koeficientů. Mezi metody této třídy pak patří:

- `ExpandDegree`: Metoda rozšiřuje polynom na polynom daného stupně. Chybějící koeficienty jsou doplněny nulami.
- `Sum2Pols`: Metoda sečte dvě pole obsahující koeficienty polynomů. Je využívána v algoritmu Karatsuba.
- `Sum3Pols`: Metoda sečte tři pole obsahující koeficienty polynomů. Je využívána v algoritmu Toom-Cook.
- `GetCorrectDegree`: Metoda zjistí větší z počtu koeficientů dvou polynomů a vrátí tomuto číslu nejbližší mocninu dvojky.
- `GetComplex`: Metoda vrátí pole typu `complex<double>` obsahující koeficienty polynomu jako komplexní čísla. Imaginární část těchto koeficientů je nastavena na `0i`. Metodu využívá Rychlá Fourierova tranformace.
- `Trivial`: Triviální algoritmus určený pro násobení polynomů. Je využíván pro otestování funkčnosti ostatních algoritmů a pro srovnání doby běhu s ostatními algoritmy.
- `Karatsuba`: Přípravná část algoritmu Karatsuba.
- `KaratsubaRec`: Skutečný výpočet součinu dvou polynomů pomocí algoritmu Karatsuba.
- `FFT`: Nerekurzivní část algoritmu FFT.
- `FFTRec`: Rekurzivní část algoritmu FFT.
- `Toom3`: Přípravná část algoritmu Toom-Cook.
- `Toom3Rec`: Skutečný výpočet součinu dvou polynomů pomocí algoritmu Toom-Cook.

2.3 Knihovna OpenMP

V kapitole je čerpáno z [8].

Algoritmy pro násobení polynomů jsem po optimalizaci paralelizoval pomocí knihovny OpenMP. Jedná se o API pro programování vícevláknových aplikací. Funguje na principu označení bloku kódu programu, po kterém programátor požaduje, aby byl prováděn paralelně. OpenMP se postará o celý životní cyklus vlákna. Pro povolení vícevláknového běhu programu je nutné kompilátoru

`g++` přidat přepínač `-fopenmp` a ve zdrojovém kódu přidat knihovnu *omp.h*. OpenMP umožňuje snadnou paralelizaci sekvenčních aplikací pomocí předem definovaných direktiv.

Vlastnosti proměnných Daná vlastnost proměnné způsobí chování ostatních vláken vůči této proměnné. Pokud použijeme tyto vlastnosti na ukazatel, je důležité si uvědomit, že daná vlastnost platí pouze pro tento ukazatel, nikoliv na data, na která ukazuje. Mezi vlastnosti proměnných patří:

- `shared`: Způsobí, že daná proměnná bude sdílena všemi vlákny, v celém programu bude tedy existovat jediná instance této proměnné.
- `private`: Způsobí, že každé vlákno bude mít nezávislou instanci této proměnné. Proměnná vznikne jako neinicializovaná, po provedení paralelního bloku je obsah této proměnné dále nedefinován.
- `firstprivate`: Způsobí to samé, co vlastnost `private` s tím rozdílem, že tato proměnná je vytvořena s původní hodnotou, kterou obsahovala stejnojmenná proměnná před vstupem do paralelního bloku. Po provedení paralelního bloku je však obsah této proměnné dále nedefinován.
- `lastprivate`: Způsobí to samé, co vlastnost `firstprivate` s tím rozdílem, že hodnota této proměnné z poslední iterace paralelního cyklu bude překopírována do stejnojmenné proměnné hlavního vlákna procesu.

Funkční paralelismus Kromě datového paralelismu, na který je knihovna OpenMP primárně zaměřena, umožňuje dále tato knihovna i paralelismus funkční. Především nás zajímá mechanismus TASK, který spouští direktiva `#pragma omp task`. Tento mechanismus umožňuje paralelizaci například rekurzivních funkcí. Funguje tak, že direktivna `#pragma omp task` vygeneruje úlohu ke zpracování, uloží ji do tzv. *TASK pool* a nějaké vlákno začně úlohu vykonávat. Je možné ještě využít direktivu `#pragma omp taskwait`, která způsobí čekání na dokončení všech synovských úloh, pouze však přímých potomků. Příklad užití mechanismu TASK můžeme vidět ve zdrojovém kódu 2.1.

```
int fib ( int n ) {
  int i, j;
  if ( n < 2 ) return n;
  #pragma omp task shared ( i ) firstprivate ( n )
    i = fib ( n - 1 );
  #pragma omp task shared ( j ) firstprivate ( n )
    j = fib ( n - 2 );
  #pragma omp taskwait
  return i + j;
}
```

Zdrojový kód 2.1: Použití mechanismu TASK

TASK pool má nevýhodu v tom, že se jedná o sdílenou strukturu. Vložení do něj a výběr z něj jsou kritické sekce, a to způsobuje režii práce s TASK. Ve zdrojovém kódu 2.1 vidíme neefektivní využití mechanismu TASK pro výpočet Fibonacciho čísla. Každá větev je totiž prováděná pomocí jiného TASKu, dochází tedy k mnoha čtením a vkládání z/do *TASK pool*. Pro malá n je možné kód vykonávat sekvenčně, neboli nevytvářet další vlákna. K tomu může posloužit klauzule `if`. Není-li podmínka klauzule `if` splněna, současný TASK je pozastaven a začne se provádět nový TASK. Rodičovský TASK se pak obnoví po dokončení tohoto TASKu. Nepracuje se tedy zde s *TASK pool*. Pokud je podmínka splněna, nový TASK je vložen do *TASK pool* a vlákna si jej z něj mohou vyzvednout. Ve zdrojovém kódu 2.2 vidíme využití klauzule `if` na výpočet Fibonacciho čísla.

```
int fib ( n ) {
  int i, j;
  if ( n < 2 ) return n;
  #pragma omp task shared ( i ) if ( n > 20 )
    i = fib ( n - 1 );
  #pragma omp task shared ( j ) if ( n > 20 )
    j = fib ( n - 2 );
  #pragma omp taskwait
  return i + j;
}
```

Zdrojový kód 2.2: Efektivnější použití mechanismu TASK

Nevýhoda zdrojového kódu 2.2 spočívá v tom, že není pro výpočet využito i vlákno volající TASK. Ve zdrojovém kódu 2.3 vidíme optimalizaci, díky které se sníží počet čtení a vkládání z/do *TASK pool*.

```
int fib ( n ) {
  int i, j;
  if ( n < 2 ) return n;
  #pragma omp task shared ( i ) if ( n > 20 )
    i = fib ( n - 1 );
  j = fib ( n - 2 );
  #pragma omp taskwait
  return i + j;
}
```

Zdrojový kód 2.3: Efektivní použití mechanismu TASK

Nutno podotknout, že pro správné využití mechanismu TASK je nutné volání rekurzivní funkce, která tento mechanismus využívá, mít uvnitř paralelního bloku. V opačném případě by tato rekurzivní funkce byla prováděna pouze jedním vláknem. Ve zdrojovém kódu 2.4 můžeme vidět zavolání rekurzivní funkce, která vypočítá n-té Fibonacciho číslo.

```
int main () {
  int n = 100, result;
  #pragma omp parallel num_threads ( 4 )
    #pragma omp single
       result = fib ( n );
  printf ( "%d-te Fibonnaciho cislo je %d\n", n, result );
  return 0;
}
```

Zdrojový kód 2.4: Správné volání rekurizvní funkce využívající mechanismus TASK

2.4 Měřící prostředky

Pro testování implementovaných algoritmů jsem využíval dvou serverů. Prvním je svazek STAR (*star.fit.cvut.cz*), jehož konfigurace je následující:

- CPU: 2 ks Intel Xeon E5-2620 v2, 15 MB L3 cache, 6 jader, celkem 12 vláken, AVX sada instrukcí,
- RAM: 32 GB,
- OS: Gentoo 4.1.6,
- verze kompilátoru `g++`: 4.9.3.

K dispozici jsem měl i server HP ProLiant DL180 G6 s následující konfigurací:

- CPU: 2 ks Intel Xeon E5620, 12 MB L3 cache, 4 jádra, celkem 8 vláken, SSE4.2 sada instrukcí,
- RAM: 48 GB,
- OS: Ubuntu Server 14.04.4 LTS,
- verze kompilátoru `g++`: 4.8.4.

2.5 Nastavení kompilátoru

Kompilátor `g++` má mnoho přepínačů. Své implementace jsem při testování a měření kompiloval s následujícími přepínači:

- `-Ofast`: Zapne podporu automatických kompilátorových optimalizací zdrojového kódu.
- `-mavx` nebo `-msse4.2`: Nastaví kompilátor pro sadu instrukcí, kterou podporuje daný procesor.

Kapitola 3

Realizace

Kapitola popisuje realizaci algoritmů násobení polynomů. Pro každý algoritmus je popsána implementace jeho sekvenční verze, poté případné optimalizované a paralelizované verze.

3.1 Triviální algoritmus

Sekvenční část Triviální algoritmus jsem původně implementoval pouze ze dvou důvodů. Jedním bylo otestovat, zda ostatní algoritmy, na které je tato práce primárně zaměřena, dosahují stejných výsledků jako algoritmus triviální. Druhým důvodem bylo zkoumání numerické stability ostatních algoritmů, které je dále popsáno v kapitole 4. Jak ale v této kapitole zjistíme, triviální algoritmus je využit i v algoritmu Karatsuba a Toom-Cook.

Algoritmus jsem implementoval podle pseudokódu 1, který najdeme na stránce 5. Polynom C byl stejně jako polynomy A a B objekt třídy `CPolynomial`, pro který jsem alokoval potřebnou paměť před zahájením samotného výpočtu. Polynom C má naalokovanou paměť velikosti o jedna větší, než je součet stupňů polynomů A a B. Před zahájením výpočtu je nutné všechny koeficienty polynomu C nastavit na 0. Poté dojde k výpočtu, který je implementován podle řádků 5 až 9 v pseudokódu 1.

Paralelizace Paralelizaci triviálního algoritmu jsem vynechal. Jak zjistíme z měření obsaženého v kapitole 4, triviální algoritmus dosahoval tak špatných časových výsledků, že ani teoretické dvanáctinásobné zrychlení, které by měl umožnit výpočet provedený 12 vlákny, by triviálnímu algoritmu nepomohlo se vyrovnat s algoritmy ostatními. Jak bylo zmíněno výše, triviální algoritmus je sice obsažen v algoritmech Karatsuba a Toom-Cook, avšak paralelizace triviálního algoritmu by ani zde ničemu nepomohla, neboť algoritmy Karatsuba a Toom-Cook jsou paralelizovány pomocí mechanismu *TASK*, jak se dočteme dále v této kapitole.

d_{0_0}	d_{0_1}	$\cdots$	$d_{0_{n-2}}$	d_{01_0}	d_{01_1}	$\cdots$	$d_{01_{n-2}}$	d_{1_0}	d_{1_1}	$\cdots$	$d_{1_{n-2}}$

Tabulka 3.1: Struktura pole obsahujícího koeficienty polynomů D_0, D_{01} a D_1

Teoreticky by se triviální algoritmus dal paralelizovat pomocí direktivy `#pragma omp parallel for`, která umožňuje paralelizaci `for` cyklů. Nicméně by zde bylo mnoho sdílených přístupů do stejného paměťového místa více vlákny najednou. Bylo by tedy třeba triviální algoritmus upravit tak, aby každé vlákno dostalo na starost určitou část výpočtu, aby bylo sdílených přístupů do paměti co nejméně. To by se dalo udělat tak, že by se vstupní polynomy rozdělily na části, ty se mezi sebou vynásobily jednotlivými vlákny a nakonec by se z výsledků poskládal výsledný polynom.

3.2 Algoritmus Karatsuba

Sekvenční část Metodu `Karatsuba` jsem rozdělil na nerekurzivní a rekurzivní část.

Nerekurzivní část je přípravná, u vstupních polynomů se zde ověřuje, zda jejich stupeň je ve tvaru $2^x - 1$, případně se polynomy patřičně rozšiřují. Poté se zavolá rekurzivní část `KaratsubaRec`, která má jako parametry vstupní polynomy A, B a výsledný polynom $C = A \cdot B$, kde paměť potřebná pro uložení koeficientů výsledného polynomu je alokována v nerekurzivní části.

V rekurzivní části je postupováno dle pseudokódu 2 na straně 7. Pro polynomy A_l, A_u, B_l a B_u není alokována žádná paměť navíc, při celém výpočtu jsou využívána pole, ve kterých jsou uloženy koeficienty původních polynomů A a B, jen dochází k posunu ukazatelů (pro A_l a B_l není ukazatel posunut vůbec, pro A_u a B_u dochází k posunu ukazatelů o $\frac{n}{2}$, kde n je počet koeficientů polynomů A a B). Pouze pro polynomy $A_l + A_u$ a $B_l + B_u$ je alokována paměť navíc, neboť nelze přepisovat původní polynomy A a B a na místo jejich koeficientů zapisovat koeficienty polynomů $A_l + A_u$ a $B_l + B_u$.

Polynomy D_0, D_{01} a D_1 jsou ukládány do výsledného pole určeného pro koeficienty výsledného polynomu C, jak je vidět v tabulce 3.1.

Následně se přepočítají koeficienty polynomu D_{01} pomocí následujícího vzorce:

$$d_{01_i} = d_{01_i} - d_{0_i} - d_{1_i}, \qquad \forall i \in \langle 0,n-2 \rangle$$

d_{0_0}	d_{0_1}	d_{0_2}	d_{0_3}	d_{0_4}	d_{0_5}	d_{0_6}								
				+	+	+								
				d_{01_0}	d_{01_1}	d_{01_0}	d_{01_3}	d_{01_4}	d_{01_5}	d_{01_6}				
								+	+	+				
								d_{1_0}	d_{1_1}	d_{1_2}	d_{1_3}	d_{1_4}	d_{1_5}	d_{1_6}
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}

Tabulka 3.2: Postup konstrukce pole obsahujícího koeficienty polynomu C

Z polynomů D_0, D_{01} a D_1 je následně sestaven výsledný polynom C dle řádku 11 v pseudokódu 2. Postup je také znázorněn v tabulce 3.2, kde vidíme sestavení výsledného polynomu C, který vznikne vynásobením dvou polynomů A a B stupně 7. Polynom C má stupeň 14, polynomy D_0, D_{01} a D_1 jsou stupně 6, neboť jsou součinem polynomů stupně 4 (polynomy A_l, B_l, A_u, B_u, $A_l + A_u$ a $B_l + B_u$).

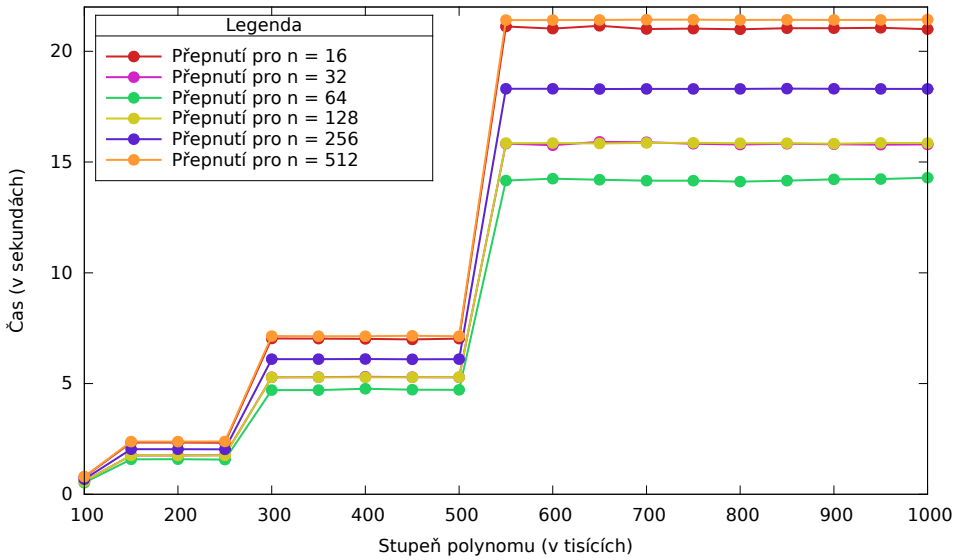
Optimalizace Algoritmus Karatsuba je kvůli režii spojené s rekurzivním voláním nevhodný pro polynomy malého stupně. Rozhodl jsem se tedy ve vhodné fázi výpočtu přepnout na triviální algoritmus, který se postará o zbytek výpočtu. Otázkou bylo, pro jaké n přepnutí provést. Měření sem provedl na polynomech stupně 100 tisíc až 1 milion. Na grafu 3.1 můžeme vidět testování hranice přepnutí algoritmu. Skoky v časech na tomto grafu jsou způsobeny nutným rozšířením stupně vstupních polynomů na tvar $2^x - 1$. Na serveru STAR i HP jsem dosáhl stejného výsledku, tedy nejlepší hranice přepnutí na triviální algoritmus je pro $n = 64$.

Paralelizace K paralelizaci jsem využil direktivu `task` knihovny OpenMP. Jeden `task` jsem použil pro výpočet polynomu D_0 a další `task` pro výpočet polynomu D_{01}. Pro výpočet polynomu D_1 jsem již další `task` nevytvářel. Tím jsem docílil toho, že k výpočtu polynomu D_1 bude využito i volající vlákno. Poté jsem vložil direktivu `taskwait`, aby při sestavování výsledného polynomu C bylo zajištěno, že výpočty polynomů D_0, D_{01} a D_1 jsou již hotové.

Paralelizace si vyžádala alokaci paměti pro pole obsahující koeficienty polynomů D_0, D_{01} a D_1, neboť použitím samotného pole C, jak je znázorněno v tabulce 3.2, by docházelo k paralelnímu přístupu na stejná paměťová místa, což by znehodnotilo výsledek výpočtu. Kapitola 4 však ukáže, že toto zvýšení paměťové náročnosti nezabrání výraznému snížení náročnosti časové.

3.3 Algoritmus Toom-Cook

Sekvenční část Jako tomu je u implementace algoritmu Karatsuba, i implementaci algoritmu Toom-Cook jsem rozdělil na nerekurzivní přípravnou část a rekurzivní část `Toom3Rec` obsahující samotný výpočet. V nerekurzivní části



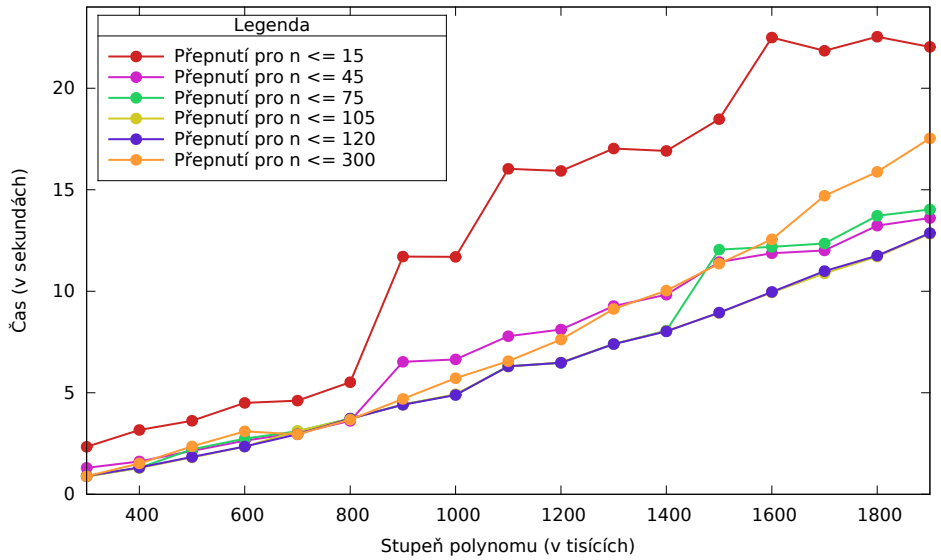
Obrázek 3.1: Hranice přepnutí algoritmu Karatsuba na triviální algoritmus

se ověřuje, zda je počet koeficientů původních polynomů dělitelný třemi a zda obsahují oba vstupní polynomy stejný počet polynomů. V opačném případě se polynomy patřičně rozšíří (doplní nulami). Poté si naalokuji paměť pro výsledný polynom a paměť pro pomocné pole w obsahující koeficienty polynomů W_0 W_1, W_{-1}, W_2 a W_∞ z pseudokódu 3, který najdeme na stránce 10.

Poté se zavolá rekurzivní část. Polynomy A_0, A_1, A_2, B_0, B_1 a B_2 jsou získány jednoduchým posunem ukazatelů o $\frac{n}{3}$. V rekurzi dochází k alokacím dvou polí, $sum1$ a $sum2$. Tato pole slouží ke sčítání polynomů, které jsou jako parametry rekurzivních volání, jak můžeme vidět na řádcích 9 až 11 v pseudokódu 3. Před provedením těchto volání dochází ke kontrole, zda třetina původního počtu koeficientů $third = \frac{n}{3}$ je stále dělitelná třemi. Pokud tomu tak není, navýší se proměnná $third$ o jedničku, či o dvojku, aby dělitelná třemi byla. Tato proměnná pak vstupuje jako parametr pěti rekurzivních volání na řádcích 8 až 12, kde se stává číslem n.

Jelikož při tvorbě třetin původních polynomů nedochází ke zbytečným alokacím, je nutné při ošetřování, zda je aktuální třetina $third$ dělitelná třemi, zajistit, aby polynom A_2, případně B_2, obsahoval na pozici posledního jednoho, či dvou koeficientů nulu. K tomu slouží pomocná proměnná $zero$. Při sčítání, které zajišťuje metoda `Sum3Pols`, vstupuje jako parametr, a místo koeficientů, které polynomy A_2 a B_2 skutečně obsahují, doplňuje na správnou pozici nulu.

Po provedení pěti rekurzivních volání využívám již naalokovaná pole $sum1$ a $sum2$ jako pomocné polynomy T_1 a T_2, které provádí výpočet na řádcích 13 a 14 v pseudokódu 3.



Obrázek 3.2: Hranice přepnutí algoritmu Toom-Cook na triviální algoritmus

Optimalizace I algoritmus Toom-Cook je vhodné přepnout na triviální algoritmus pro malá n. Graf 3.2 ukazuje dobu běhu algoritmu Toom-Cook, který je při různých hodnotách n přepnut na triviální algoritmus. Zde není tak jednoduché určit přesnou hranici přepnutí, neboť algoritmus během svého běhu dynamicky zjišťuje, zda je dané n dělitelné třemi. Oproti algoritmu Karatsuba tedy algoritmus Toom-Cook nedojde k přesně dané hranici n. Graf ukazuje nejlepší výsledek pro $n \leq 120$. Tato hodnota vyšla při měření na obou serverech. Pro znázornění jsem přidal hranici přepnutí pro $n \leq 300$, aby bylo vidět, že další zvýšení hranice už není vhodné.

Paralelizace K paralelizaci algoritmu Toom-Cook jsem rovněž využil direktivu `task` knihovny OpenMP. Direktivu jsem použil celkem čtyřikrát, a to pro výpočet polynomů W_0, W_1, W_{-1} a W_2. Pro polynom W_∞ již není vytvořen další `task`, aby bylo pro jeho výpočet použito i volající vlákno. Poté jsem použil direktivu `taskwait`, aby mohl výpočet polynomu C pokračovat s jistotou, že polynomy W_0, W_1, W_{-1}, W_2 a W_∞ jsou již získány.

Paralelizace si vyžádala další alokace paměti, konkrétně pro polynomy W_0, W_1, W_{-1}, W_2 a W_∞ a pro pomocná pole, uchovávající koeficienty polynomů, které vstupují jako parametry dalších rekurzivních volání, dle řádků 9 až 12 pseudokódu 3 na stránce 10. Ani toto přidání paměti navíc nezabránilo výraznému snížení časové složitosti algoritmu Toom-Cook, jak je vidět v kapitole 4.

3.4 Rychlá Fourierova transformace

Sekvenční část Implementace Rychlé Fourierovy transformace je podle pseudokódů 4 a 5, které najdeme na stránce 12, rozdělena na rekurzivní a nerekurizvní část. V nerekurzivní části se zjišťuje, zda počet koeficientů vstupních polynomů n je ve tvaru $n = 2^x$, $x \in \mathbf{N}$, případně dojde k doplnění chybějících koeficientů nulami. Následně je potřeba získat koeficienty vstupních polynomů jako komplexní čísla. K tomu slouží knihovna `complex`. Alokuji si nová pole typu `complex<double>` a do nich vkládám koeficienty původních polynomů.

Ještě zbývá získat primitivní n-tou odmocninu z jedné ω_n. Knihovna `complex` obsahuje funkci `polar`, která nám po zadání správných parametrů primitivní n-tou odmocninu z jedné vrátí.

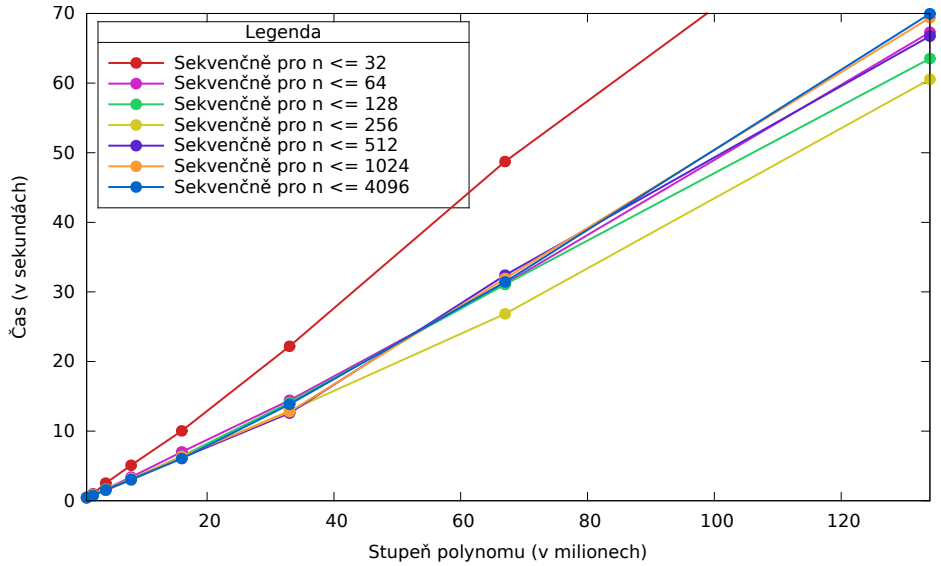
Poté dojde k zavolání rekurzivní části FFT na vstupní polynomy A a B, jejichž koeficienty už jsou uloženy jako komplexní čísla. Volání uloží do polí F_a a F_b typu `complex<double>` ohodnocení polynomů A a B v n bodech. Rekurzivní část FFT pracuje podle pseudokódu 4 s tím, že k získání polynomů A_e a A_o není alokována žádná paměť navíc, používá se původní paměť, ve které jsou uloženy koeficienty polynomu A. K tomuto účelu mám jako parametr rekurzivního volání funkce `FFTRec` navíc proměnnou určující velikost skoku, abychom docílili získání správných, tedy lichých či sudých, koeficientů.

Po získání polí F_a a F_b dojde k jejich vynásobení, vznikne nám pole F_c a na toto pole je pak podle řádku 8 pseudokódu 5 zavolána opět rekurzivní část FFT. Číslo komplexně sdružené $\overline{\omega}_n$ k číslu ω_n získáme pomocí funkce `conj` obsažené v knihovně `complex`. Výsledek tohoto rekurzivního volání je obsažen v předem naalokovaném poli C.

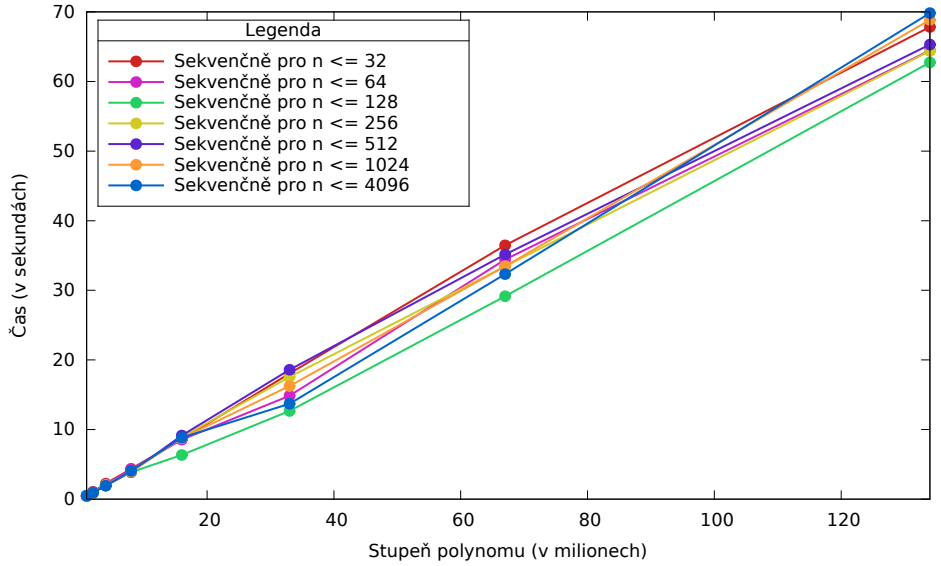
Následně dojde k vydělení reálných složek tohoto pole číslem n a uložení těchto čísel do výsledného polynomu.

Paralelizace K paralelizaci Rychlé Fourierovy transformace jsem opět využil direktivy `task`, kterou jsem aplikoval v rekurzivní části `FFTRec`. V nerekurzivní části `FFT` bylo potřeba volání částí rekurzivních vložit dovnitř paralelního bloku. Takto jsem zparalelizoval řádky 3, 4 a 8 pseudokódu 5.

Oproti algoritmům Karatsuba a Toom-Cook, které se pro malá n přepnou na algoritmus triviální, bylo u Rychlé Fourierovy tranformace nutné direktivu `task` omezit podmínkou, aby se pro malá n zbytečně nevytvářely další TASKy. Otestoval jsem na serverech STAR a HP, pro jaká n je nejlepší pokračovat sekvenčně, tedy jedním vláknem. Testování jsem provedl na polynomech s počtem koeficientů 2^{20} až 2^{27}. Grafy 3.3 a 3.4 ukazují výsledky tohoto měření. Na serveru STAR bylo nejlepší pokračovat v sekvenčním výpočtu pro $n \leq 256$ a na serveru HP pro $n \leq 128$. Důvodů, proč jsou výsledky odlišné, může být několik. Jedná se například o různé procesory, s rozdílnou sadou instrukcí. V kapitole 4 jsem tedy měření přizpůsobil těmto hodnotám na obou architekturách.



Obrázek 3.3: Omezení direktivy `task` pro malá n u FFT na serveru STAR



Obrázek 3.4: Omezení direktivy `task` pro malá n u FFT na serveru HP

Kapitola 4

Testování a diskuse

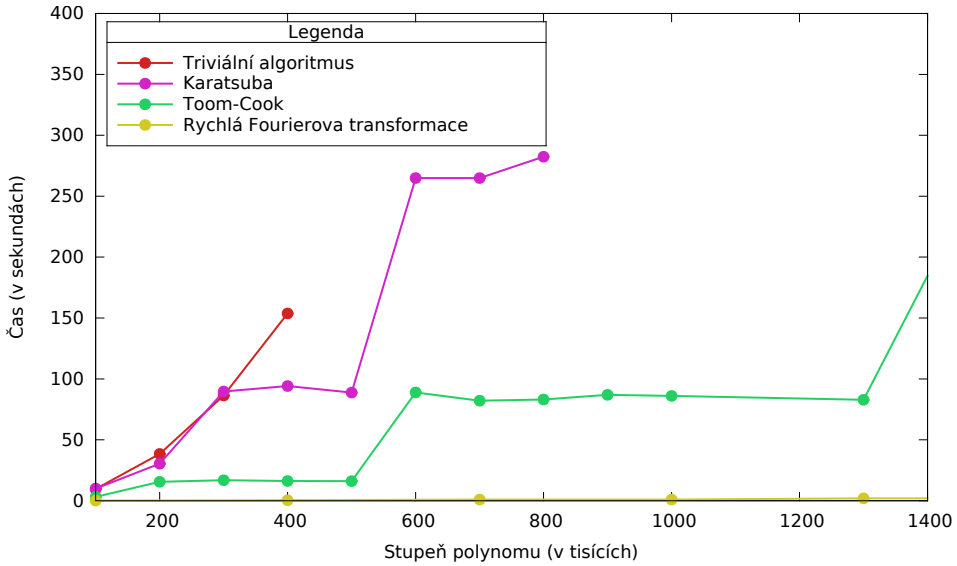
4.1 Testování implementovaných algoritmů

4.1.1 Sekvenční verze

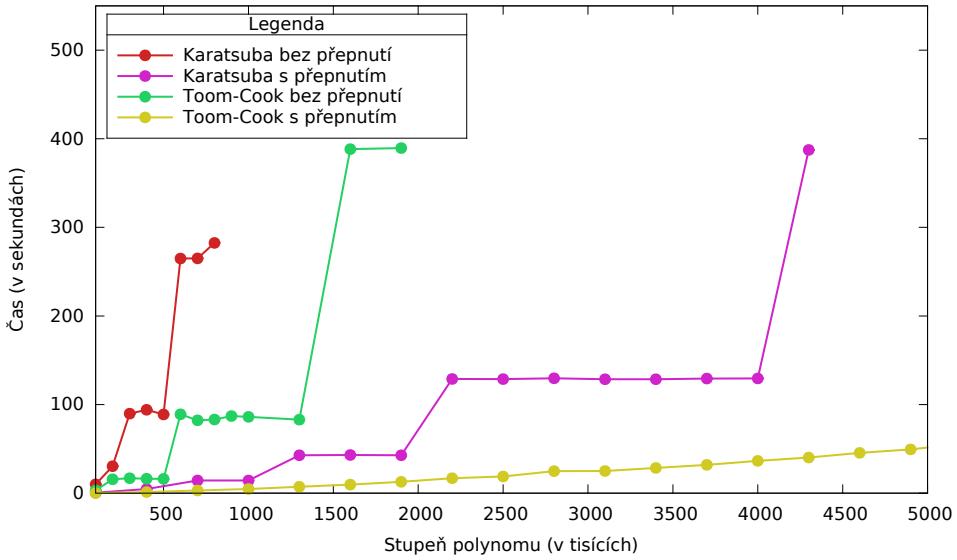
Sekvenční verzi implementovaných algoritmů jsem testoval na polynomech o počtu koeficientů 100 tisíc až 1 400 000. Na grafu 4.1 můžeme vidět výsledky tohoto testování. Triviální algoritmus a algoritmus Karatsuba nebylo třeba měřit pro vyšší stupně, neboť dosahovaly již tak vysokých časů. Nejlepších výsledků zjevně dosahuje Rychlá Fourierova transformace, která součin polynomů stupně 1 milion vypočte za zhruba 2 vteřiny. Algoritmus Toom-Cook polynomy stejného stupně spočte za přiblžně 80 vteřin. Nutno podotknout, že výsledky algoritmů Toom-Cook a Karatsuba jsou zde bez přepínání na algoritmus triviální, proto tyto algoritmy nedosahují výsledků, které bychom od nich očekávali.

4.1.2 Optimalizovaná verze

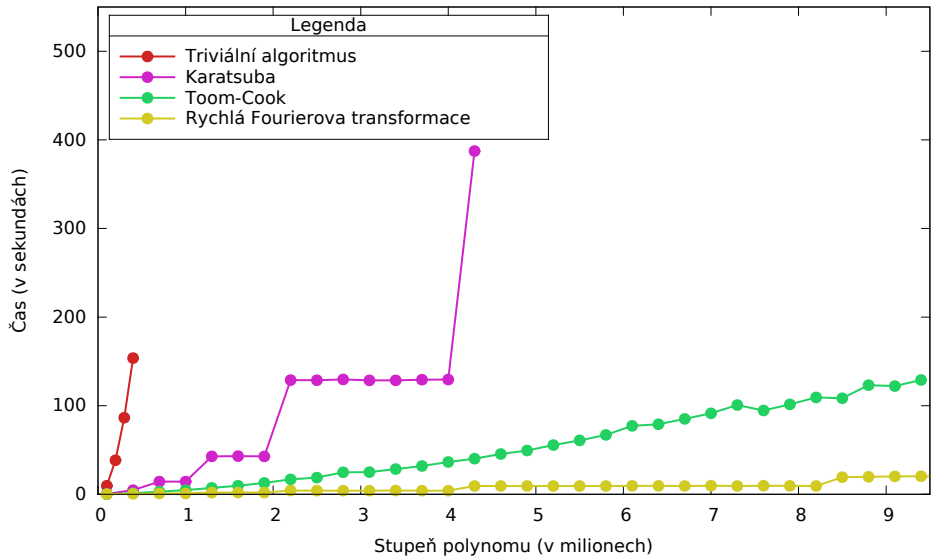
Optimalizace spočívala v přepnutí algoritmů Karatsuba a Toom-Cook na sekvenční verzi. Grafy 3.1 a 3.2 nám ukázaly nevhodnější n, pro která přepnout algoritmy na triviální algoritmus, který provede zbytek výpočtu. Na grafu 4.2 vidíme rozdíl v časové složitosti sekvenční a optimalizované verze algoritmů Karatsuba a Toom-Cook. Rozdíl je znatelný - algoritmus Karatsuba bez optimalizace spočetl součin polynomů stupně 700 tisíc za 264 vteřin, s přepnutím na triviální algoritmus stejný výpočet provedl za 14 vteřin. Algoritmus Toom-Cook provedl bez optimalizace součin polynomů stupně 1 milion za 86 vteřin, s přepnutím na triviální algoritmus tak provedl za necelých 5 vteřin. Rychlou Fourierovu transformaci jsem dále neoptimalizoval, protože výpočet sekvenční verze byl už tak velice rychlý a také FFT nelze přepnout na triviální algoritmus. Na grafu 4.3 vidíme výsledky všech algoritmů po optimalizaci algoritmů Karatsuba a Toom-Cook.



Obrázek 4.1: Sekvenční verze algoritmů



Obrázek 4.2: Rozdíl sekvenční a optimalizované verze algoritmů

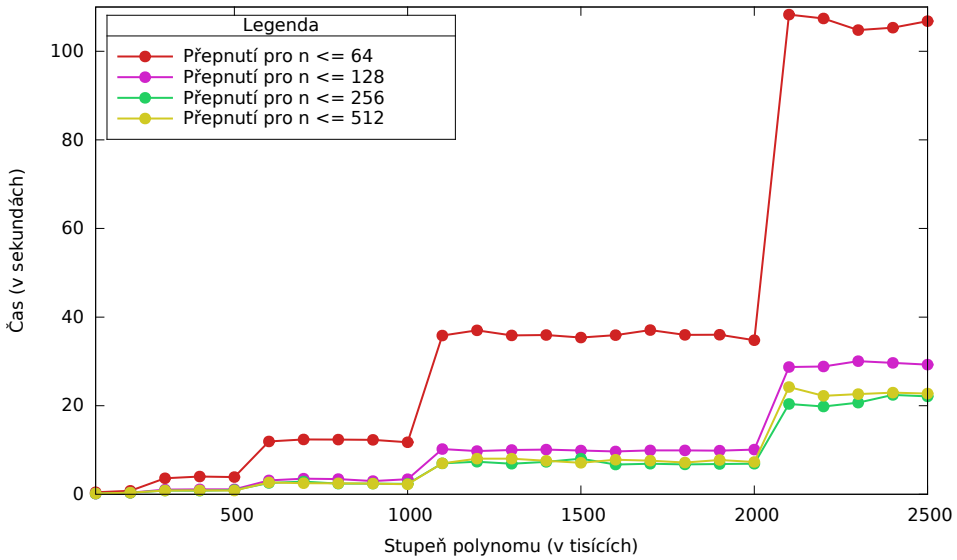


Obrázek 4.3: Optimalizovaná verze algoritmů

4.1.3 Paralelizovaná verze

Při měření paralelizované verze jsem se u algoritmů Karatsuba a Toom-Cook setkal s tím, že nejlepších výsledků jsem dosahoval při běhu se 4 vlákny, naopak spuštění programu s 8 (a na serveru STAR s 12) vlákny vykazovalo znatelně horší výsledky. Problémem bylo nevhodné přepnutí algoritmů na algoritmus triviální, neboť pro vícevláknový výpočet se vytvářela vlákna i pro malá n a režie spojená s vytvářením vláken výrazně znehodnotila časovou efektivitu algoritmů.

Provedl jsem tedy nová měření přepnutí algoritmů Karatsuba a Toom-Cook na algoritmus triviální, tentokrát s tím rozdílem, že výsledky ze serveru STAR, které znázorňují grafy 4.4 (algoritmus Karatsuba) a 4.6 (algoritmus Toom-Cook), jsou získány spuštěním algoritmů 12 vlákny, a výsledky ze serveru HP, znázorněny na grafech 4.5 (algoritmus Karatsuba) a 4.7 (algoritmus Toom-Cook), jsou získány spuštěním algoritmů 8 vlákny. Sekvenční optimalizovanou verzi algoritmu Karatsuba bylo vhodné přepnout na algoritmus triviální pro $n \leq 64$, paralelní verze si vyžádala přepnutí až pro $n \leq 256$ na serveru STAR a pro $n \leq 128$ na serveru HP. Algoritmus Toom-Cook bylo vhodné v jeho sekvenční optimalizované verzi přepnout na triviální algoritmus pro $n \leq 120$, jeho paralelní verzi je vhodné přepnout pro $n \leq 480$ na serveru STAR a pro $n \leq 240$ na serveru HP. Nutno podotknout, že v případě algoritmu Toom-Cook nemusí jít o přesné hodnoty, jde o nejlepší výsledky z množiny hodnot, kterou jsem si předem definoval. Oproti algoritmu Karatsuba totiž nelze definovat přesnou mez pro přepnutí na algoritmus triviální,



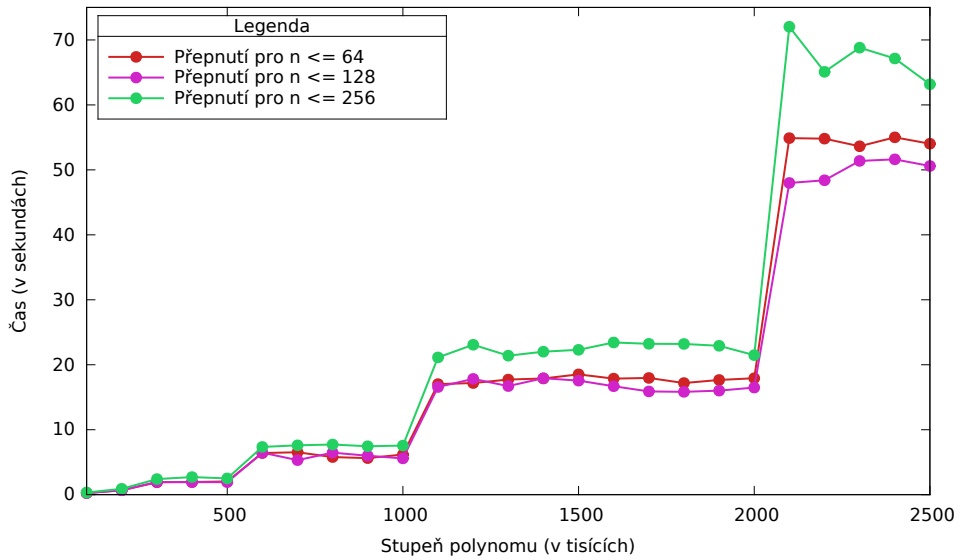
Obrázek 4.4: Přepnutí algoritmu Karatsuba na algoritmus triviální při vícevláknovém výpočtu na serveru STAR

neboť dochází k dynamickému rozdělování čísla n na třetiny.

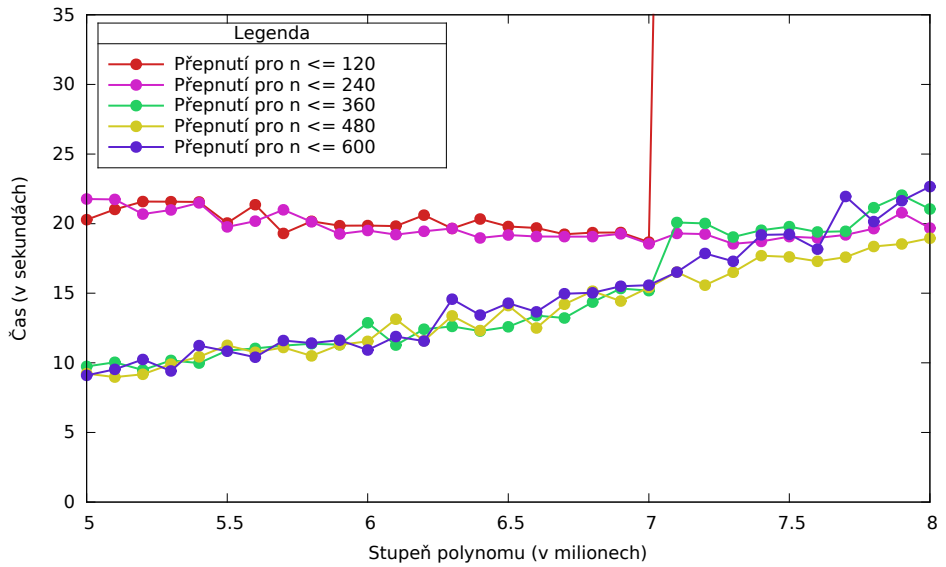
Se správnými hodnotami přepnutí algoritmu Karatsuba na algoritmus triviální jsem provedl měření časové efektivity algoritmu pro 1, 2, 4, 8 a na serveru STAR i pro 12 vláken. Výsledky měření můžeme vidět na grafech 4.8 a 4.9. Měření jsem provedl na polynomech stupně 100 tisíc až 2 a půl milionu. Algoritmus Karatsuba násobil polynomy stupně 2 a půl milionu zhruba 158 vteřin jedním vláknem, 12 vlákny mu stejný výpočet trval okolo 20 vteřin. Zrychlení je tedy výrazné.

Skutečné zrychlení algoritmu Karatsuba pro více vláken jsem také testoval, výsledky tohoto měření najdeme na grafech 4.10 a 4.11. Nutno podotknout, že hodnoty výsledků, které jsou na grafech zrychlení uvedené pro 1 vlákno, jsou hodnoty optimalizované verze algoritmu, nikoliv hodnoty paralelizované verze s nastaveným počtem vláken na jedno. Je tomu tak z důvodu znázornění skutečného zrychlení algoritmu více vlákny oproti skutečné sekvenční optimalizované verzi. Knihovna OpenMP má nemalou režii spojenou s vytvářením vláken, tudíž výsledky paralelizované verze s jedním vláknem jsou znatelně horší než výsledky sekvenční optimalizované verze. Algoritmus Karatsuba tak dosahuje skutečného šestinásobného zrychlení paralelní verze oproti verzi optimalizované na serveru STAR. Na serveru HP je zrychlení paralelní verze zhruba čtyřnásobné.

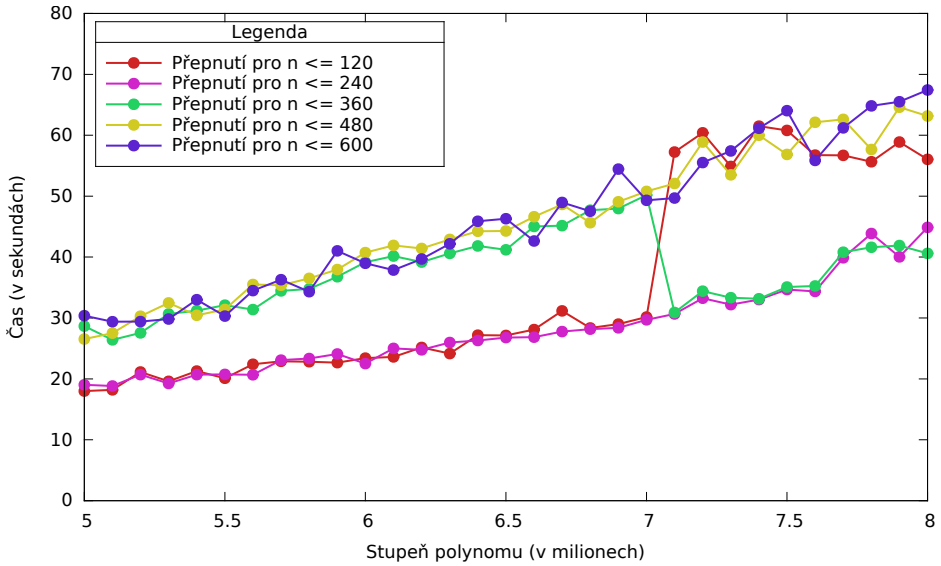
Algoritmus Toom-Cook jsem po nastavení správné meze pro přepnutí na triviální algoritmus také testoval při jeho spuštění 1, 2, 4, 8 a na serveru STAR i 12 vlákny. Výsledky měření jsou vidět na grafech 4.12 a 4.13. Měření



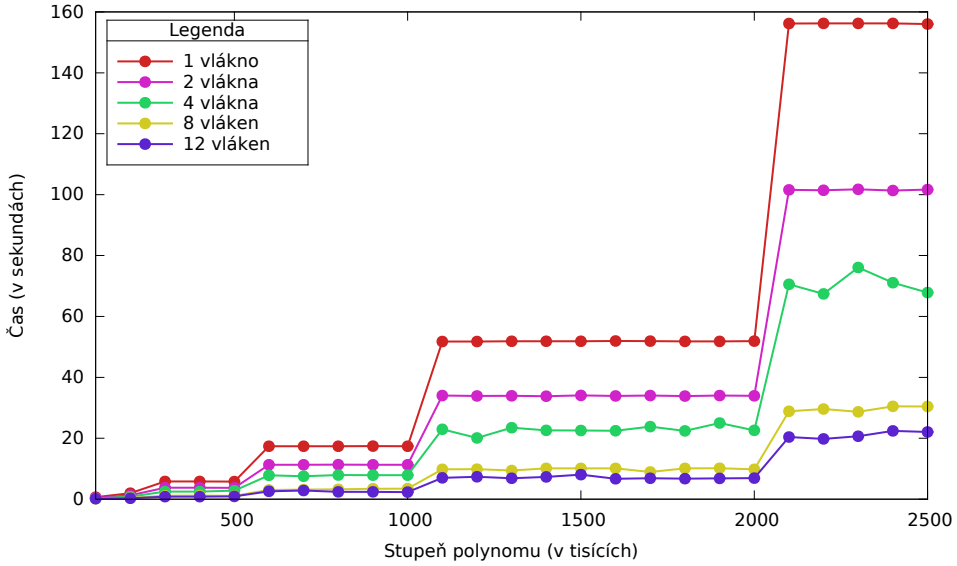
Obrázek 4.5: Přepnutí algoritmu Karatsuba na algoritmus triviální při vícevláknovém výpočtu na serveru HP



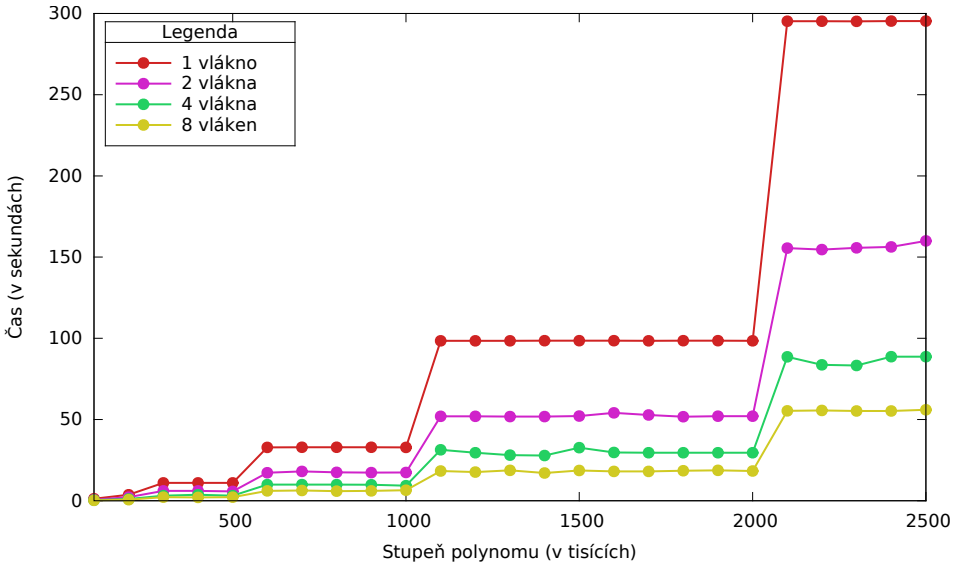
Obrázek 4.6: Přepnutí algoritmu Toom-Cook na algoritmus triviální při vícevláknovém výpočtu na serveru STAR



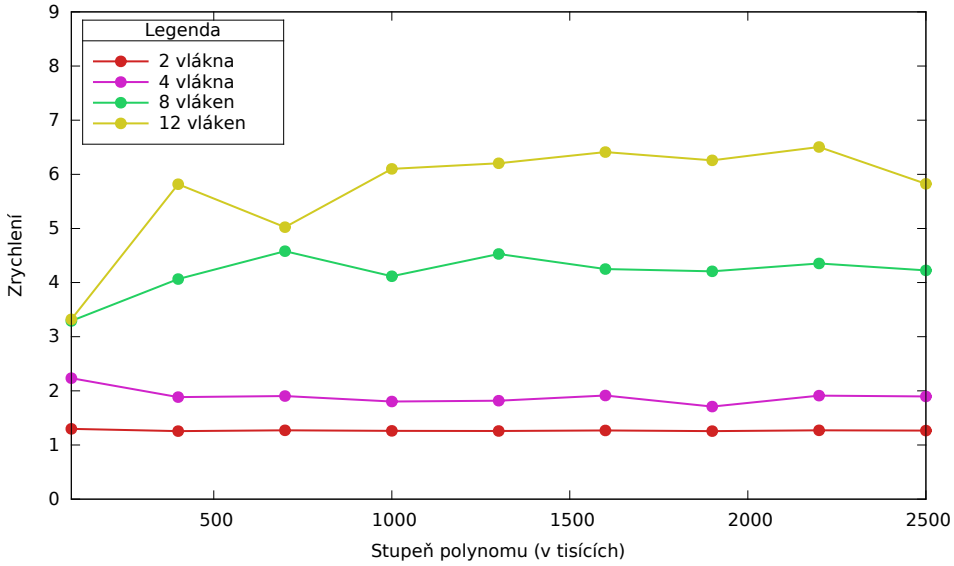
Obrázek 4.7: Přepnutí algoritmu Toom-Cook na algoritmus triviální při vícevláknovém výpočtu na serveru HP



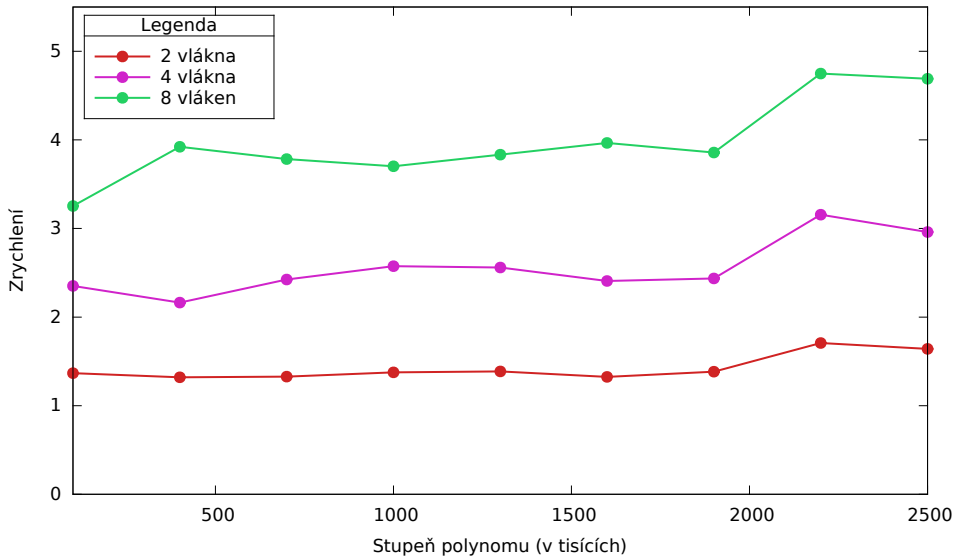
Obrázek 4.8: Algoritmus Karatsuba více vlákny na serveru STAR



Obrázek 4.9: Algoritmus Karatsuba více vlákny na serveru HP



Obrázek 4.10: Zrychlení algoritmu Karatsuba na serveru STAR

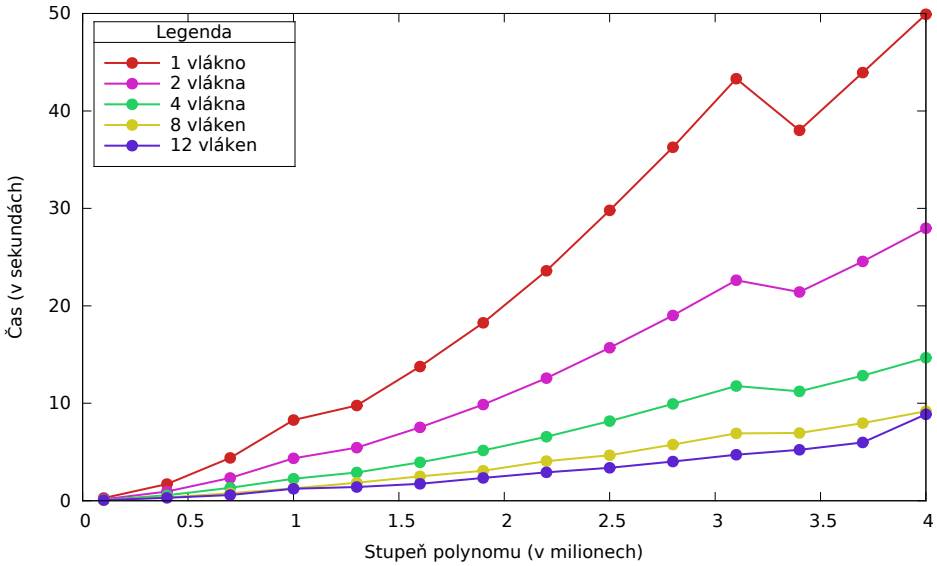


Obrázek 4.11: Zrychlení algoritmu Karatsuba na serveru HP

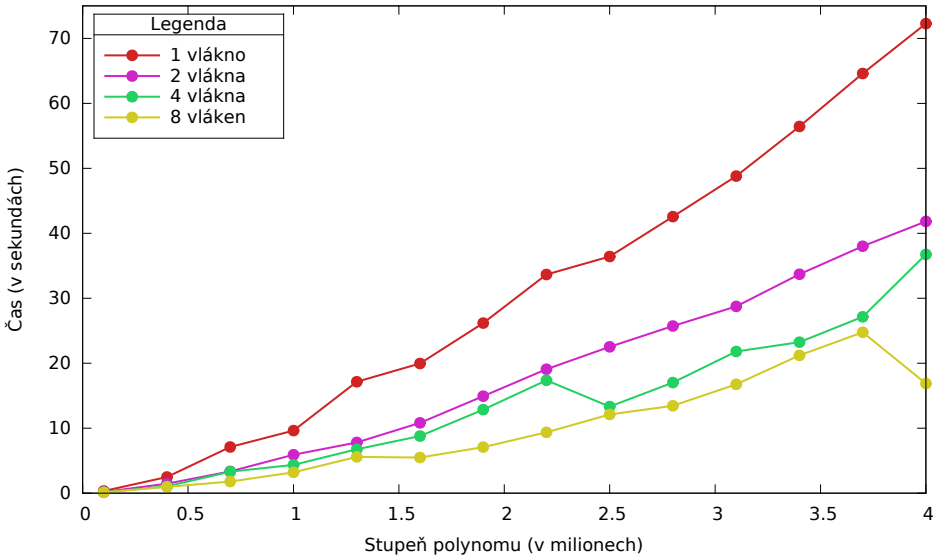
jsem provedl na polynomech stupně 100 tisíc až 4 miliony. Grafy obsahují na první pohled poněkud zarážející skoky v časech pro vyšší stupně polynomů. Je tomu tak proto, že ověřování, zda je počet koeficientů vstupního polynomů dělitelný třemi, probíhá dynamicky, tedy při každém průběhu algoritmu. Může tak snadno nastat, že pro vyšší stupeň polynomu dojde k menšímu počtu takovýchto dělení oproti polynomu s nižším stupněm polynomu. Grafy potvrzují efektivitu vícevláknového výpočtu. Zatímco jednovláknový výpočet součinu polynomů stupně 4 miliony trval na serveru STAR necelých 50 vteřin, výpočet 12 vlákny trval necelých 10 vteřin. Na serveru HP stejný výpočet jedním vláknem zabral zhruba 70 vteřin, výpočet 8 vlákny trval necelých 20 vteřin.

Zrychlení paralelizované verze algoritmu Toom-Cook oproti verzi optimalizované znázorňují grafy 4.14 a 4.15. Na serveru STAR je zrychlení při běhu programu 12 vlákny zhruba pětinásobné, na serveru HP dosahuje 8 vláknový výpočet zhruba třínásobného zrychlení. Důvodů, proč se nejedná o dvanáctinásobné zrychlení na serveru STAR a osminásobné zrychlení na serveru HP, je několik. Tím hlavním je režie spojená s vytvářením a udržováním vláken knihovnou OpenMP. Stejně jako grafy zrychlení algoritmu Karatsuba, i grafy zrychlení algoritmu Toom-Cook znázorňují zrychlení oproti optimalizované sekvenční verzi. Dalším důvodem menšího zrychlení můžou být nutné alokace paměti navíc oproti verzi sekvenční, se kterými je jistá časová režie navíc také spojena. Zrychlení je to nicméně pořád velmi dobré, algoritmus se mi podařilo pomocí knihovny OpenMP výrazně zrychlit oproti jeho sekvenční verzi.

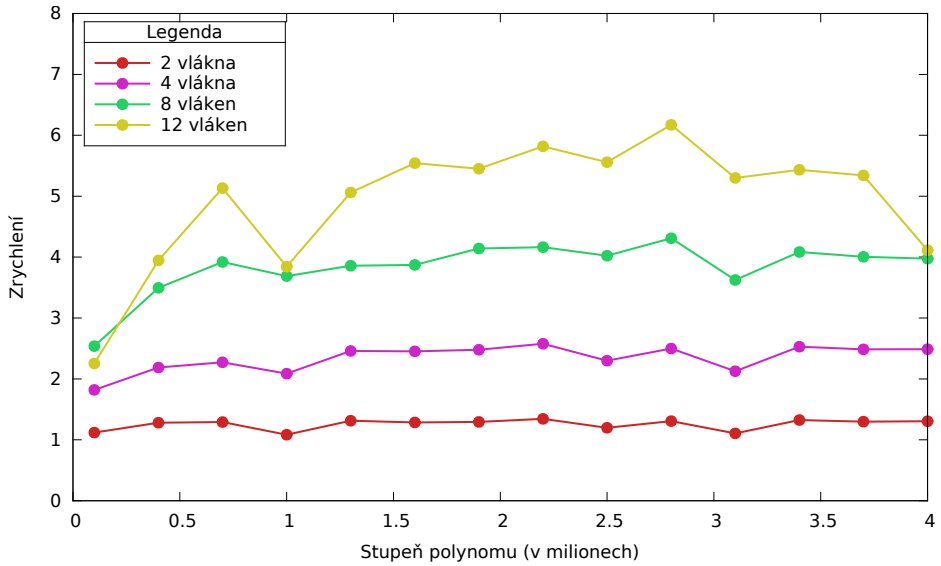
Posledním testovaným algoritmem je Rychlá Fourierova transformace. Výsledky měření doby výpočtu algoritmu FFT spuštěným 1, 2, 4, 8 a na serveru



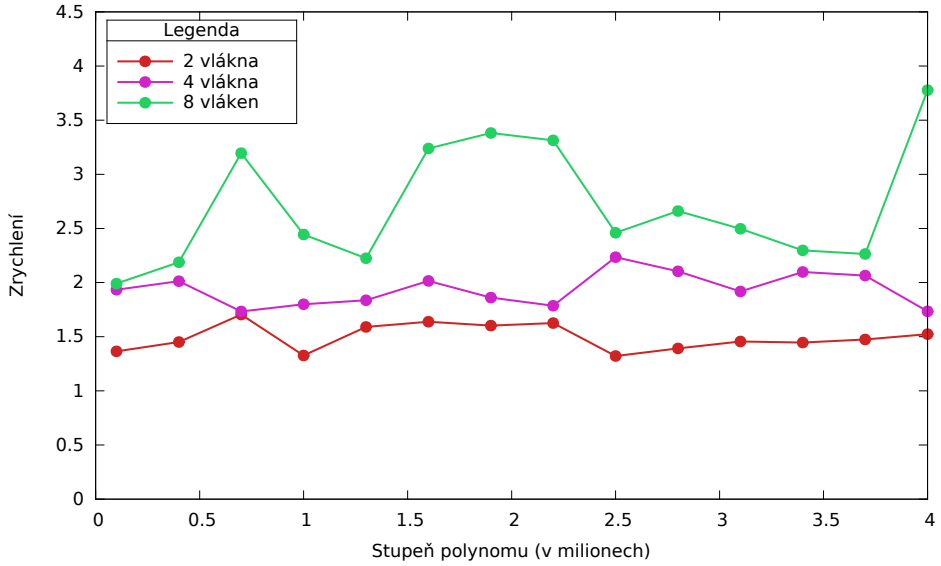
Obrázek 4.12: Algoritmus Toom-Cook více vlákny na serveru STAR



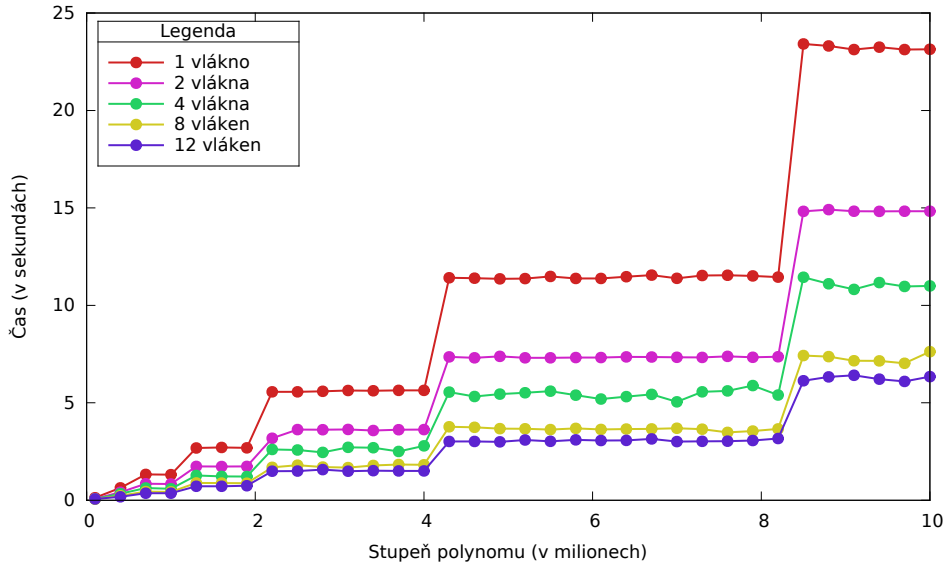
Obrázek 4.13: Algoritmus Toom-Cook více vlákny na serveru HP



Obrázek 4.14: Zrychlení algoritmu Toom-Cook na serveru STAR



Obrázek 4.15: Zrychlení algoritmu Toom-Cook na serveru HP

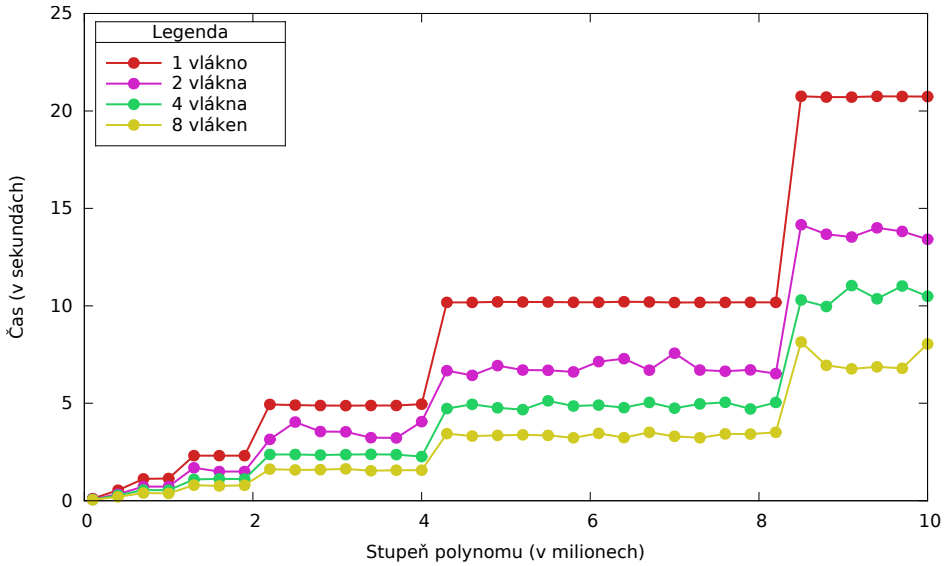


Obrázek 4.16: Rychlá Fourierova tranformace více vlákny na serveru STAR

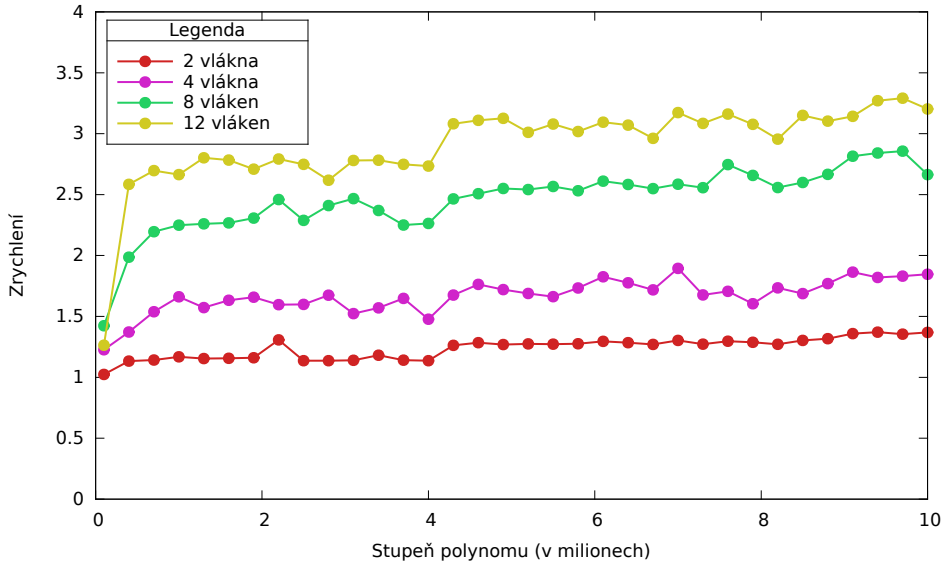
STAR i 12 vlákny jsou znázorněny na grafech 4.16 a 4.17. Grafy opět obsahují nepřiměřené skoky v časech. Rychlá Fourierova transformace (tak jako algoritmus Karatsuba) požaduje, aby počet koeficientů vstupních polynomů byl mocnina dvojky. Pokud tomu tak není, jsou chybějící koeficienty doplněny nulami. Měření probíhalo na polynomech stupně 100 tisíc až 10 milionů. Na serveru STAR jednovláknový výpočet součinu polynomů stupně 10 milionů trval necelých 24 vteřin, dvanáctivláknová verze zabrala pouhých 5 vteřin. Server HP vypočítal stejný součin jedním vláknem za 20 vteřin, osmivláknový výpočet trval zhruba 8 vteřin. Opět můžeme vidět výrazný nárůst časové efektivity použitím vícevláknového programování.

Zrychlení paralelizované Rychlé Fourierovy transformace, které je znázorněné na grafech 4.18 a 4.19, je na serveru STAR i HP zhruba třínásobné. Zrychlení není tak výrazné, jako tomu bylo u algoritmů Karatsuba a Toom-Cook, protože samotná sekvenční verze algoritmu FFT už je časově velmi efektivní, přesto vícevláknový výpočet časovou efektivitu ještě navýšil.

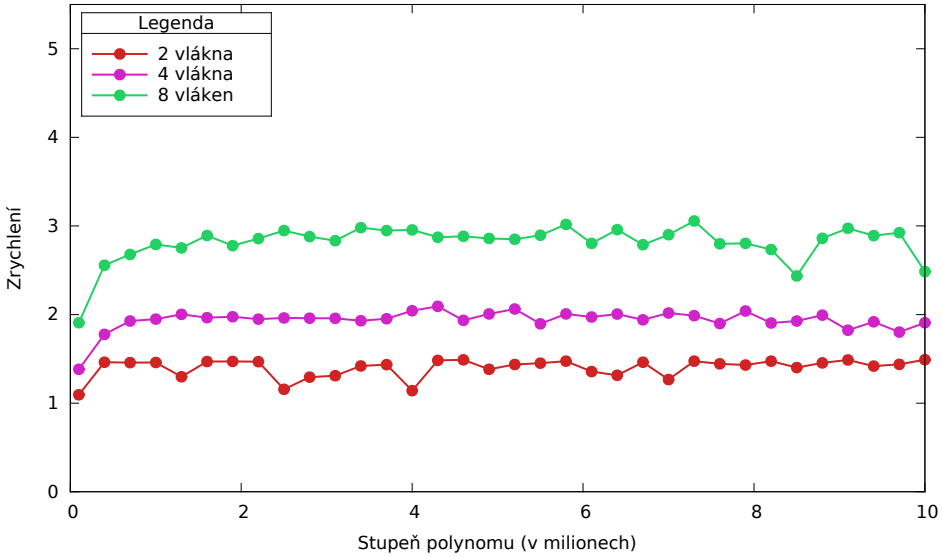
Na závěr této podkapitoly jsem přiložil grafy 4.20 a 4.21, znázorňující srovnání paralelizované verze algoritmů Karatsuba, Toom-Cook a Rychlé Fourierovy transformace. Měření jsem provedl na polynomech stupně 100 tisíc až 10 milionů. Dle očekávání nejlepších výsledků dosáhla Rychlá Fourierova transformace, která na serveru STAR vypočítá součin dvou polynomů stupně 10 milionů za zhruba 5 vteřin, na serveru HP za 8 vteřin. Algoritmus Toom-Cook spočítá stejný součin na serveru STAR za 26 vteřin, na serveru HP za 63 vteřin. Nakonec algoritmus Karatsuba vypočítá tento součin za 183 vteřin na serveru STAR, na serveru HP za necelých 500 vteřin.



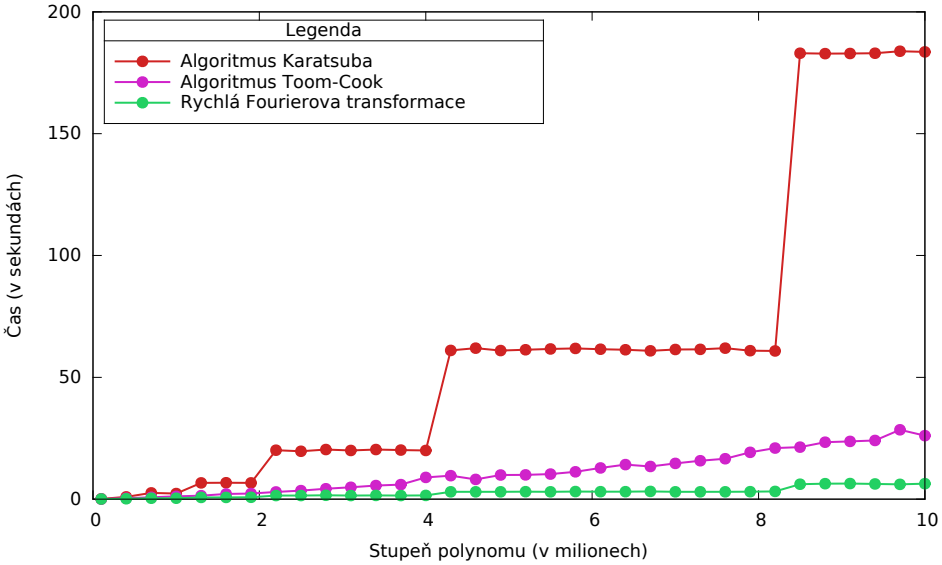
Obrázek 4.17: Rychlá Fourierova tranformace více vlákny na serveru HP



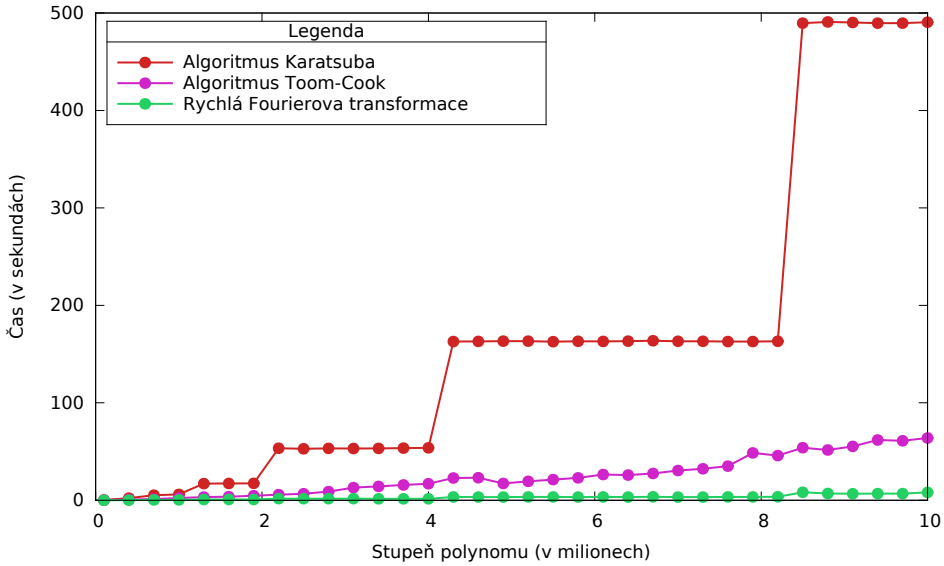
Obrázek 4.18: Zrychlení Rychlé Fourierovy tranformace na serveru STAR



Obrázek 4.19: Zrychlení Rychlé Fourierovy tranformace na serveru HP



Obrázek 4.20: Srovnání paralelizované verze algoritmů na serveru STAR



Obrázek 4.21: Srovnání paralelizované verze algoritmů na serveru HP

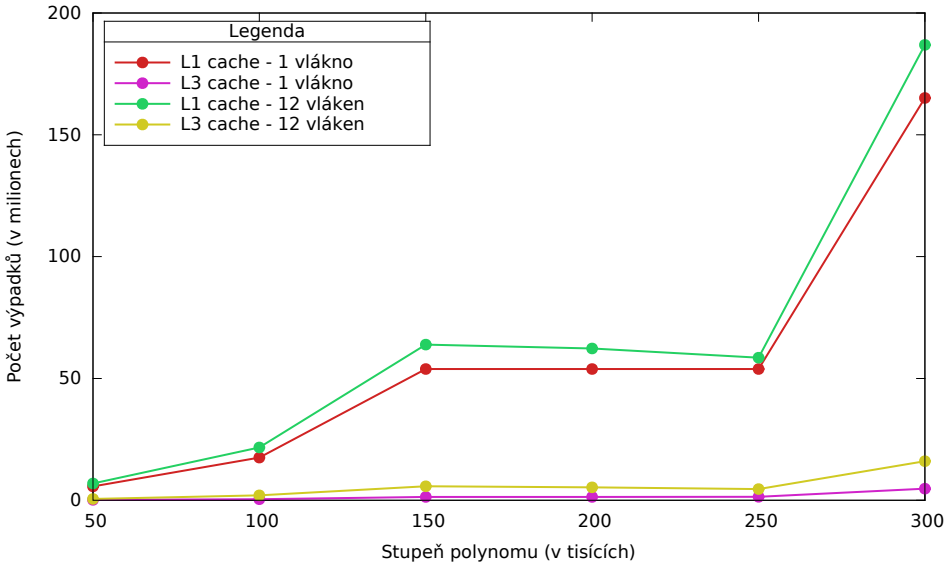
4.2 Využití skryté paměti

Pro získání informací o využití skryté (cache) paměti jsem použil nástroj Cachegrind. Program zobrazí informace o počtech přístupů do L1 a LL (*last level*) cache paměti. V našem případě je LL cache pamětí L3 cache paměť. Kromě počtu přístupů do L1 a L3 cache paměti Cachegrind zobrazuje počet výpadků dané cache paměti a *miss rate*, což je poměr počtu výpadků cache paměti ku počtu všech přístupů do dané cache paměti. A právě tyto informace jsem z měření svých algoritmů získával a zanesl do grafů. Každý algoritmus jsem měřil zvlášť pro 1 vlákno a pro 12 vláken, abych znázornil rozdíl vícevláknového výpočtu v přístupu do cache paměti oproti výpočtu sekvenčnímu. Měření jsem prováděl na polynomech stupně 50 tisíc až 300 tisíc. Hranici 300 tisíc jsem nastavil z důvodu časové náročnosti programu Cachegrind, kterému spuštění programů a vyhodnocení výsledků trvá mnohonásobně delší dobu, než by trval samotný výpočet programu.

4.2.1 Algoritmus Karatsuba

Počet výpadků cache paměti algoritmu Karatsuba je znázorněn na grafu 4.22. Z grafu je vidět mírný nárůst počtu výpadků jak v L1, tak v L3 cache paměti paralelizované verze oproti verzi sekvenční.

Nejlépe však znázorní efektivitu využití skryté paměti *miss rate*. Graf 4.23 znázorňuje miss rate L1 a L3 cache pamětí. *Miss rate* L1 cache paměti dosahuje zhruba 1,3 %. Paralelizace algoritmu zvýší *miss rate* na 1,5 %, což je



Obrázek 4.22: Počet výpadků skryté paměti algoritmu Karatsuba

stále dobrý výsledek. *Miss rate* L3 cache paměti sekvenční verze algoritmu Karatsuba je 0 %. Paralelizovaná verze navýší *miss rate* na 0,1 %.

4.2.2 Algoritmus Toom-Cook

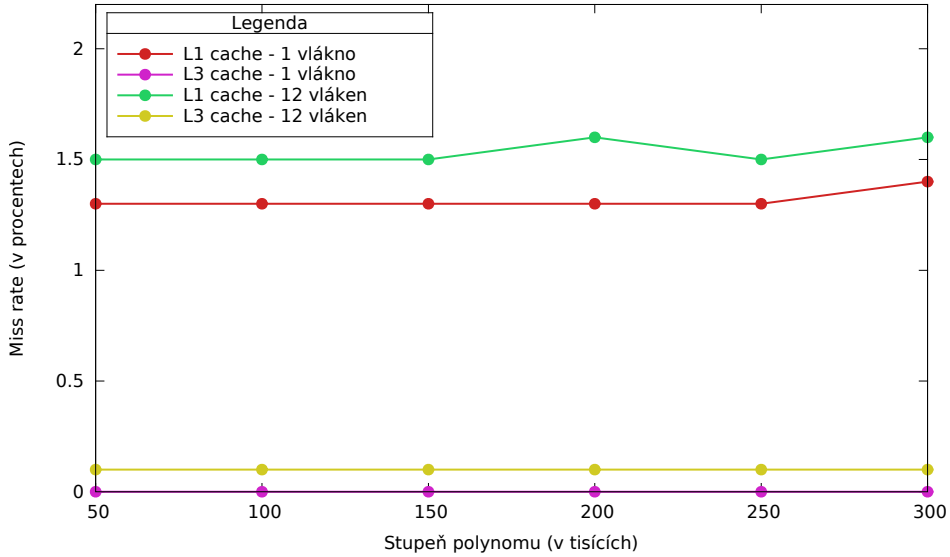
Graf 4.24 ukazuje počty výpadků L1 a L3 cache paměti pro testovaná data. Opět můžeme vidět nepatrný nárůst počtu výpadků paralelizované verze oproti verzi sekvenční. Nárůst je však minimální, což potvrzuje, že paralelizace (nijak výrazně) nesníží efektivitu využití skryté paměti verze sekvenční.

Miss rate L1 a L3 cache paměti znázorňuje graf 4.25. Paralelizace algoritmu navýší *miss rate* L1 cache paměti z původních zhruba 1,4 % na 1,5 %, *miss rate* L3 paměti zůstává po paralelizaci na původních zhruba 0,5 %.

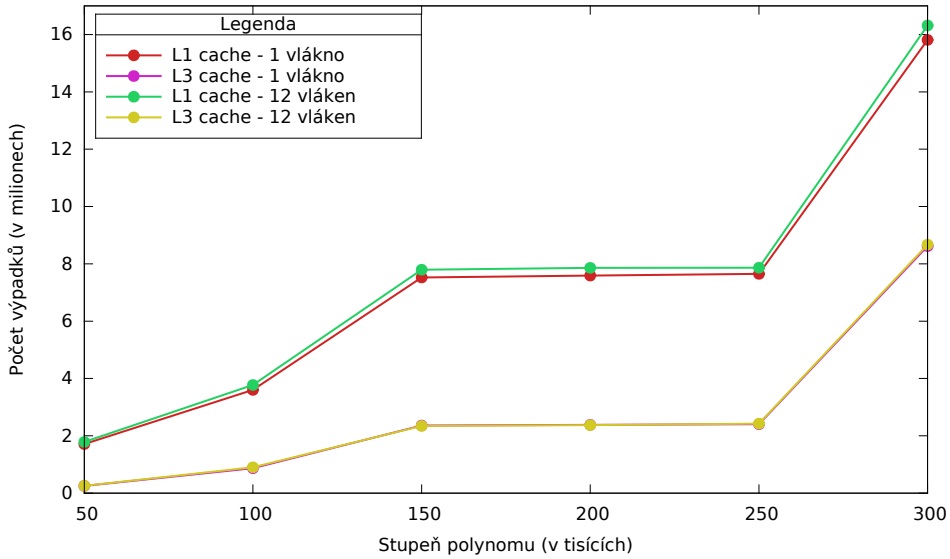
4.2.3 Rychlá Fourierova transformace

Na grafu 4.26 můžeme vidět počet výpadků skryté paměti Rychlé Fourierovy transformace. Ani zde paralelizace nezpůsobila výrazný nárůst počtu výpadků, navýšila pouze počet výpadků L1 cache paměti, počet výpadků L3 cache paměti zůstal přibližně stejný.

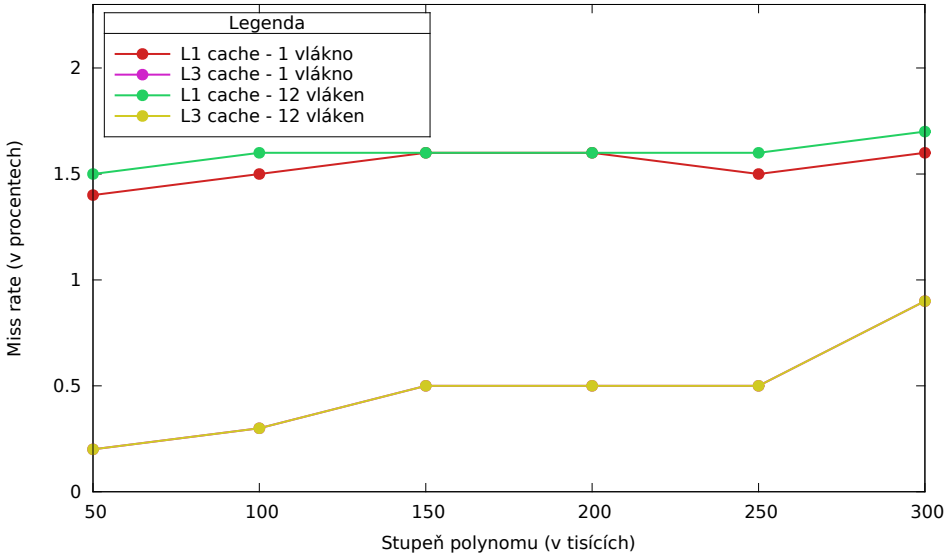
Jak můžeme vidět na grafu 4.27, *miss rate* L1 cache paměti paměti po paralelizaci algoritmu FFT naroste z původních 1,4 % na 1,6 %. Co se *miss rate* L3 cache paměti týče, i po paralelizaci algoritmu se stále drží na původních zhruba 0,5 %.



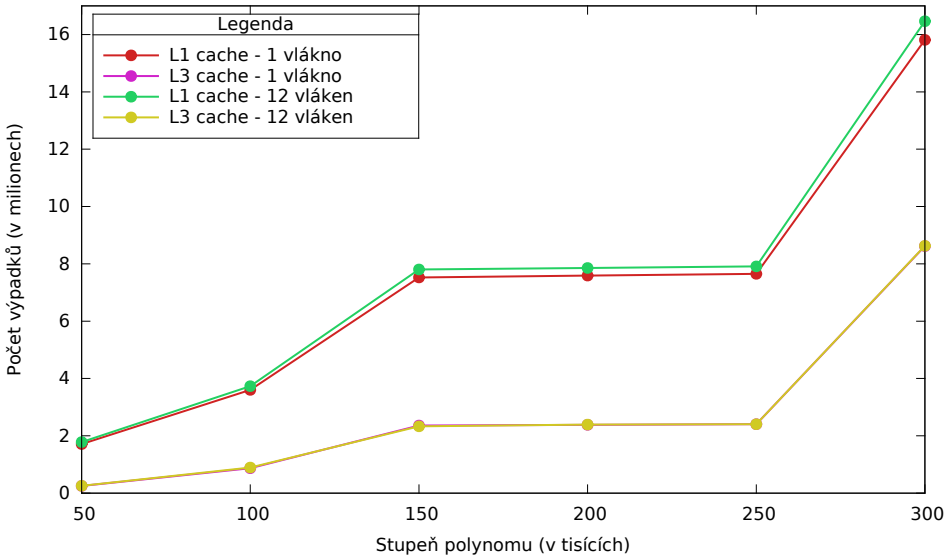
Obrázek 4.23: *Miss rate* skryté paměti algoritmu Karatsuba



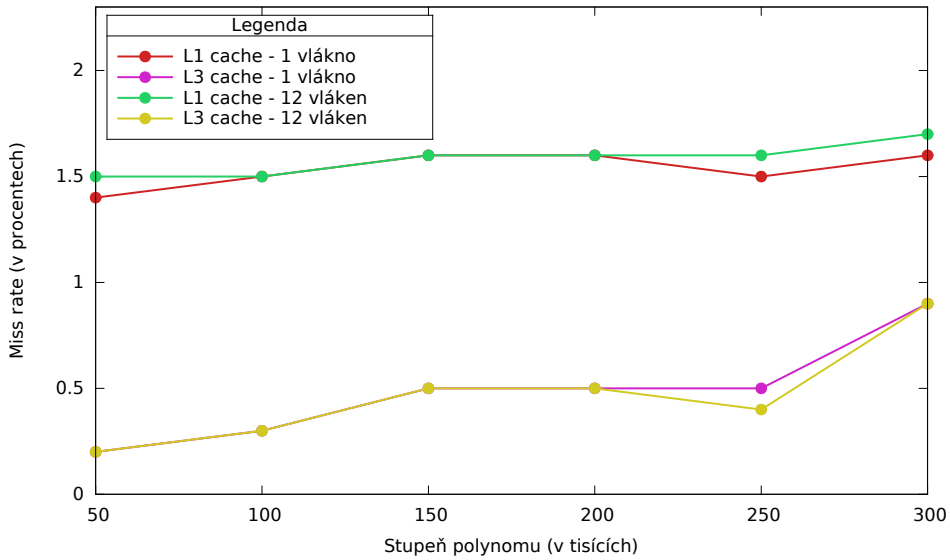
Obrázek 4.24: Počet výpadků skryté paměti algoritmu Toom-Cook



Obrázek 4.25: *Miss rate* skryté paměti algoritmu Toom-Cook



Obrázek 4.26: Počet výpadků skryté paměti algoritmu FFT



Obrázek 4.27: *Miss rate* skryté paměti algoritmu FFT

4.2.4 Shrnutí

Efektivita využití skryté paměti se ani po paralelizaci všech implementovaných algoritmů nijak výrazně nesnížila. Počty výpadků skryté paměti v milionech se můžou zdát jako vysoké číslo, *miss rate* nám však ukazuje, o jak malé číslo se ve skutečnosti jedná. *Miss rate* L1 cache paměti u mnou implementovnaých algoritmů nepřekročil hodnotu 1,6 % a *miss rate* L3 cache paměti nepřekročil hodnotu 1 %, což je velice dobrý výsledek, vzhledem k tomu, že podle [9, slajd 5] je typická hodnota *miss rate* pro L1 cache mezi 3 až 10 procenty, pro L1 cache to bývá okolo 1 % v závislosti na velikosti L3 cache. Cache paměť je tedy správně využívána. Je tomu tak i z důvodu správného uložení koeficientů polynomu v paměti. Nemalou roli v tak dobrých výsledcích hraje samozřejmě kapacita skryté paměti. L3 cache paměť na serveru STAR má 15 MB, na serveru HP je to 12 MB.

4.3 Numerická stabilita algoritmů

Při testování numerické stability algoritmů pro násobení polynomů jsem se zaměřil na tři hlediska. Zajímalo mne, jak numerickou stabilitu implementovaných algoritmů ovlivní stupeň vstupních polynomů, pokud budou všechny koeficienty vstupních polynomů nastaveny na konstantní hodnotu. Druhým hlediskem byl rozsah vstupních hodnot (hodnot koeficientů vstupních polynomů). V poslední řadě mě zajímalo, jak ovlivní chyba okolí koeficientů výstupního polynomu, pokud je jeden koeficient vstupních polynomů nastaven

na podstatně vyšší hodnotu oproti ostatním koeficientům.

Výsledky algoritmů Karatsuba, Toom-Cook a FFT jsem odečetl od výsledků triviálního algoritmu a získal jsem tak chybu ve výsledcích oproti výsledkům správným (rozdíly jsem si uchovával v absolutní hodnotě). Pro tyto hodnoty jsem dále spočetl průměrnou a maximální chybu a tyto hodnoty zanesl do grafu nebo do tabulky.

4.3.1 Stabilita v závislosti na stupni vstupních polynomů

Z důvodu vysoké časové náročnosti triviálního algoritmu jsem nemohl otestovat numerickou stabilitu na velké stupně vstupních polynomů. Otestoval jsem ji tedy na polynomech A a B stupně 100 tisíc až 1 milion. Všechny koeficienty vstupních polynomů byly nastaveny na pevně danou hodnotu (číslo 1234,567890123456789). Po vynásobení takových polynomů dostaneme polynom C, který má nejvyšší hodnotu koeficientu přibližně 150 miliard.

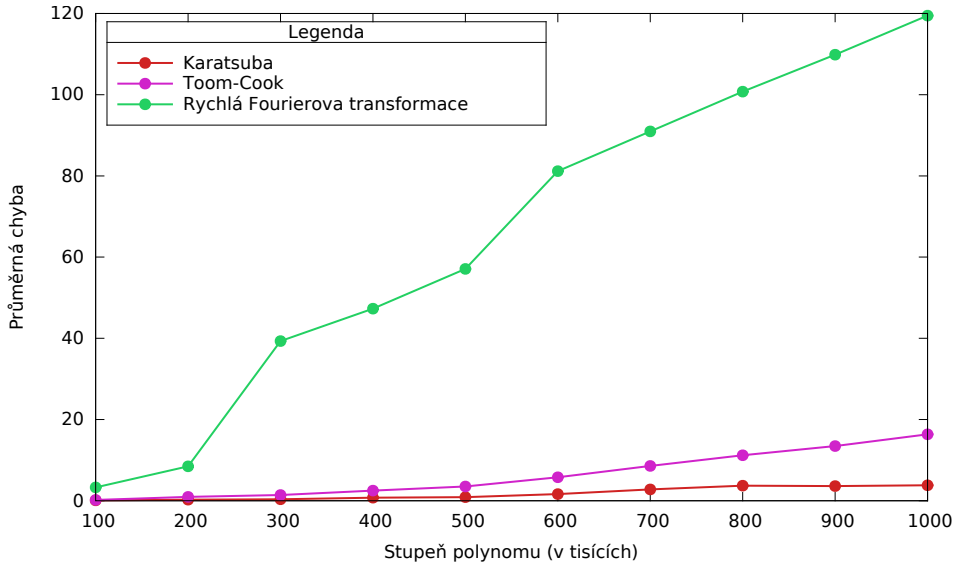
Znázornění průměrné chyby ve výsledku algoritmů Karatsuba, Toom-Cook a FFT najdeme na grafu 4.28. Algoritmus Karatsuba dosahoval chyby v hodnotách koeficientů okolo 3 až 5 oproti hodnotám získaným triviálním algoritmem. Algoritmus Toom-Cook dosahoval chyby až okolo 20. Naproti tomu Rychlá Fourierova transformace už dosáhla při násobení polynomů stupně 1 milion průměrné chyby přibližně 120.

Maximální chybu znázorňuje graf 4.29. Algoritmus Karatsuba dosahuje maximální chyby okolo 30, Rychlá Fourierova transformace chyby okolo 300. Algoritmus Toom-Cook zde nečekaně dosáhl větší maximální chyby než Rychlá Fourierova tranformace, a to dokonce až 800 v případě násobení polynomů stupně 1 milion. Nicméně průměrnou chybu má algoritmus Toom-Cook výrazně nižší oproti FFT, jedná se tedy spíše o ojedinělou odchylku.

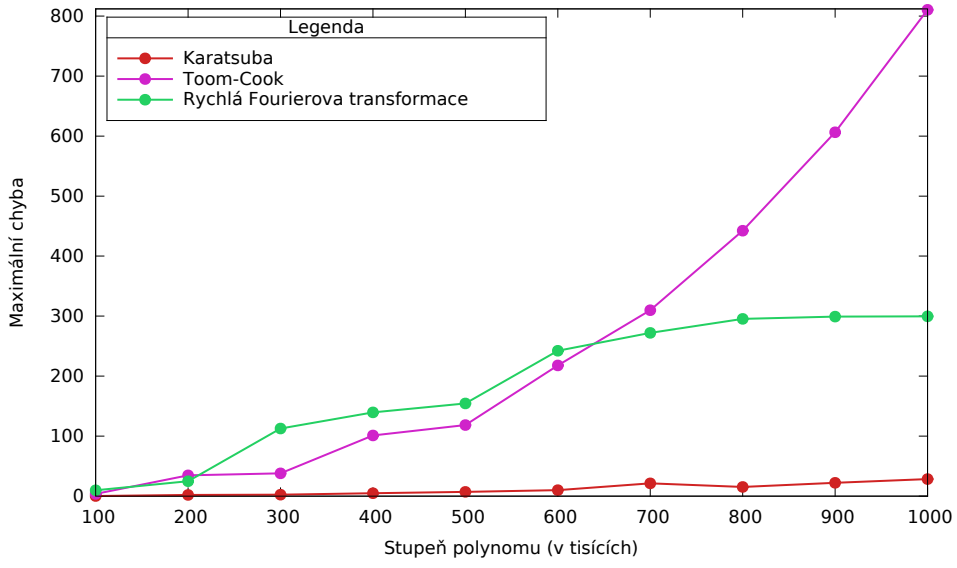
4.3.2 Stabilita v závislosti na rozsahu vstupních hodnot

Pro testování numerické stability algoritmů v závislosti na rozsahu vstupních hodnot jsem vygeneroval vstupní polynomy s náhodnými koeficienty, avšak s pevně danou nejvyšší hodnotou, jaké mohou koeficienty nabývat. Konkrétně jsem stabilitu testoval na hodnotách 10^1 až 10^9. Změřil jsem průměrnou a maximální chybu algoritmů ve výsledcích oproti výsledkům získaným triviálním algoritmem. Získané chyby jsou zaneseny do tabulky 4.1. Tyto výsledky nejsou znázorněny v grafu z důvodu velkého nárůstu chyby při navyšující se nejvyšší možné hodnotě koeficientu. Graf by v tomto případě nebyl moc vypovídající.

Z tabulky můžeme vidět, jak rychle narůstá chyba při narůstající nejvyšší možné hodnotě vstupních koeficientů. U všech algoritmů s každou další nejvyšší hodnotou narůstá chyba přibližně stonásobně, což je očekávaný výsledek vzhledem k tomu, že rozsah hodnot vstupních koeficientů narůstá desetinásobně. Nejvyšší chyby opět dosahuje Rychlá Fourierova transformace. Toom-Cook je na tom s průměrnou chybou o 3 řády lépe, maximální chybu má o řád



Obrázek 4.28: Průměrná chyba ve výsledků v závislosti na stupni polynomů



Obrázek 4.29: Maximální chyba ve výsledků v závislosti na stupni polynomů

	Průměrná chyba			Maximální chyba		
Nejvyšší hodnota koeficientu	Karatsuba	Toom-Cook	FFT	Karatsuba	Toom-Cook	FFT
10^1	$5 \cdot 10^{-8}$	10^{-6}	$3 \cdot 10^{-4}$	$2 \cdot 10^{-5}$	$9 \cdot 10^{-5}$	$7 \cdot 10^{-4}$
10^2	$4 \cdot 10^{-6}$	$9 \cdot 10^{-6}$	$2 \cdot 10^{-2}$	10^{-3}	$7 \cdot 10^{-3}$	$6 \cdot 10^{-2}$
10^3	$4 \cdot 10^{-4}$	$9 \cdot 10^{-4}$	2	0,2	0,66	6,22
10^4	$4 \cdot 10^{-2}$	0,9	234	13	75	616
10^5	4	86	23491	1212	5052	62118
10^6	389	8600	$2 \cdot 10^6$	10^5	$6 \cdot 10^5$	$6 \cdot 10^6$
10^7	38300	$8 \cdot 10^5$	$2 \cdot 10^8$	10^7	$6 \cdot 10^7$	$6 \cdot 10^8$
10^8	$4 \cdot 10^6$	$8 \cdot 10^7$	$2 \cdot 10^{10}$	10^9	$8 \cdot 10^9$	$6 \cdot 10^{10}$
10^9	$3 \cdot 10^8$	$8 \cdot 10^9$	$2 \cdot 10^{12}$	10^{11}	$5 \cdot 10^{11}$	$6 \cdot 10^{12}$

Tabulka 4.1: Chyba v závislosti na rozsahu vstupních hodnot

nižší. Algoritmus Karatsuba je v případě průměrné chyby o řád lepší oproti algoritmu Toom-Cook, maximální chybu mají přibližně stejnou.

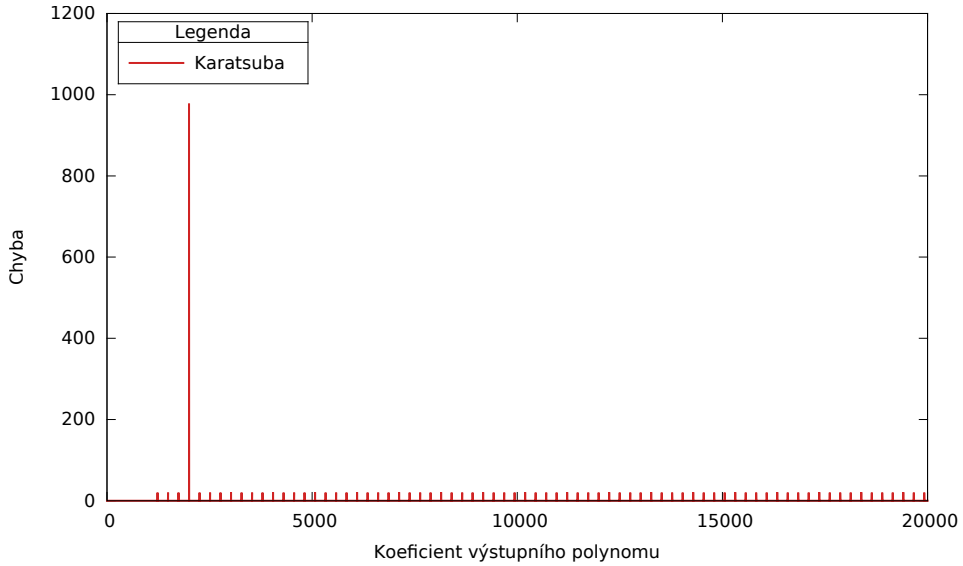
4.3.3 Stabilita v závislosti na konkrétní vysoké hodnotě

Numerickou stabilitu algoritmů v závislosti na konkrétní vysoké hodnotě jsem otestoval na součin polynomů A a B stupně 100 tisíc, přičemž kromě koeficientů a_{1000} a b_{1000} jsem všechny ostatní koeficienty nastavil na hodnotu 1. Koeficienty a_{1000} a b_{1000} jsem nastavil na hodnotu 10^8. Tak vysokou hodnotu jsem nastavil z důvodu dobrého znázornění chyby a okolí (ostatní koeficienty), které chyba ovlivňuje. Výsledky tohoto měření jsou vidět na grafu 4.30 pro algoritmus Karatsuba, 4.31 pro algoritmus Toom-Cook a nakonec na grafu 4.32 pro Rychlou Fourierovu transformaci. Aby byly grafy přehlednější, znázorňují chybu pouze do 20000. koeficientu. Dle očekávání je největší chyba výsledného polynomu $C = A \cdot B$ na všech grafech u koeficientu c_{2000}, neboť je výsledkem součinu $a_{1000} \cdot b_{1000}$, tedy koeficientů uchovávajících hodnotu 10^8.

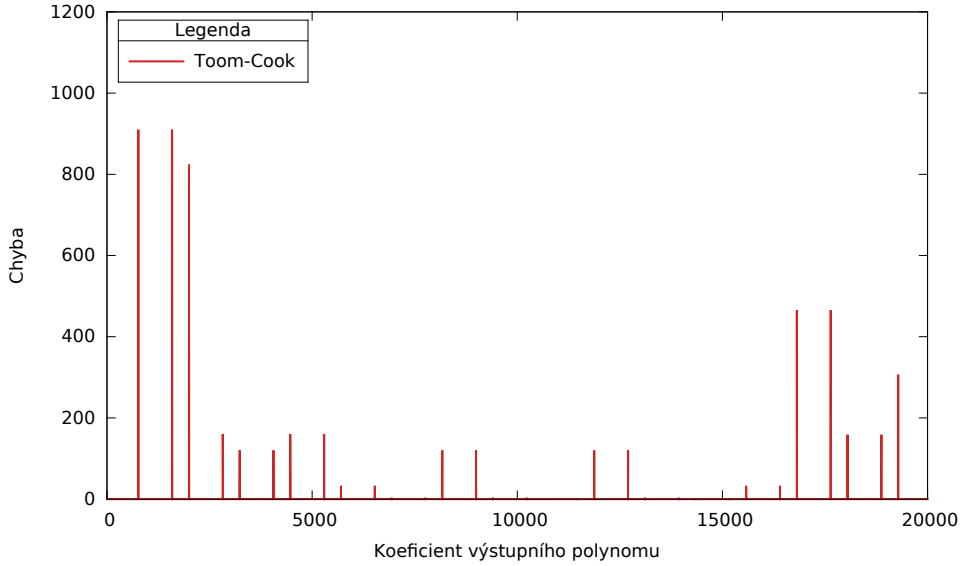
U algoritmu Karatsuba nedojde k výraznému ovlivnění okolí chybou ve výpočtu. Do koeficientu c_{999} nedojde k žádné chybě, poté až do posledního koeficientu dostáváme chybu na některých pozicích přibližně 20. Chyba koeficientu c_{2000} je přibližně 980.

Algoritmus Toom-Cook už má chybu na některých místech větší (místy až okolo 600), avšak nedosahuje tak častých chyb jako algoritmus Karatsuba. Chyba koeficientu c_{2000} je přibližně 820.

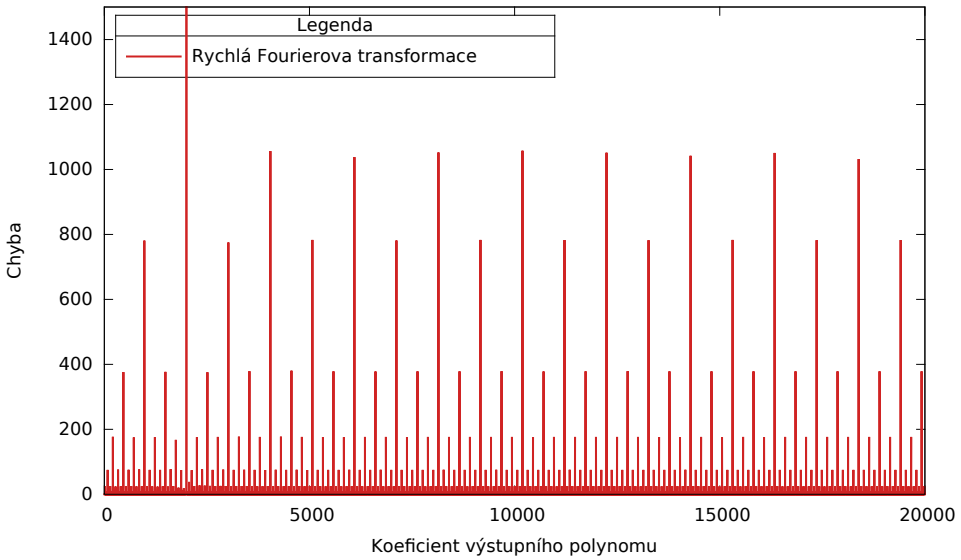
Nejhůře opět dopadla Rychlá Fourierova transformace. Chyba koeficientu c_{2000} je přes 600 tisíc. Dále se na ostatních koeficientech střídají chyby okolo 100 až 1000. Oproti algoritmům Karatsuba a Toom-Cook bezchybného výsledku Rychlá Fourierova transformace nedosáhne u žádného výsledného koeficientu.



Obrázek 4.30: Znázornění okolí ovlivněného chybou - algoritmus Karatsuba



Obrázek 4.31: Znázornění okolí ovlivněného chybou - algoritmus Toom-Cook



Obrázek 4.32: Znázornění okolí ovlivněného chybou - algoritmus FFT

4.3.4 Použitelnost algoritmů z hlediska numerické stability

Předchozí měření odhalila slabinu efektivních algoritmů pro násobení polynomů. Tím je numerická stabilita. Pro zaručený přesný výsledek součinu dvou polynomů je nutné násobit tyto polynomy triviálním algoritmem.

Algoritmus Karatsuba je schopný vynásobit dva polynomy přesně za podmínek nízkých hodnot vstupních koeficientů, avšak s rostoucím stupněm vstupních polynomů se chyby můžou dostavit i u nízkých hodnot koeficientů (v řádu tisíců). Ze všech tří zkoumaných algoritmů je však algoritmus nejpřesnější, zároveň ale také nejpomalejší.

Algoritmus Toom-Cook je méně přesný oproti algoritmu Karatsuba, obzvláště je náchylný na vysoké maximální chyby. Průměrné chyby si však drží blízko průměrným chybám algoritmu Karatsuba. S rostoucím stupněm vstupních polynomů však průměrná chyba roste výrazně rychleji, než je tomu tak u algoritmu Karatsuba.

Nakonec se Rychlá Fourierova transformace ukázala jako nejchybovější algoritmus. Za cenu velice rychlého výpočtu dostaneme ne úplně přesný výsledek. Zejména s rostoucím stupněm vstupních polynomů výrazně narůstá chybovost výpočtu.

4.4 Existující řešení

Algoritmy Karatsuba a FFT se zabýval v bakalářské práci [10] kolega Léhár. Kolega se zabýval taktéž optimalizací a paralelizací těchto algoritmů. Své implementace testoval také na serveru STAR, nicméně na starší architektuře. Není tedy relevantní srovnávat časovou efektivitu jeho implementace s mou implementací. Kolega se dále zabýval existujícími knihovnami pro jazyk C++, konkrétně knihovnami Armadillo a Blitz++. Tyto knihovny jsem také chtěl využít pro srovnání s mnou implementovanými algoritmy. V práci kolegy Lehára však bylo zjištěno, že tyto knihovny využívají k násobení polynomů triviálního algoritmu, časově by se tedy algoritmům Karatsuba, Toom-Cook ani FFT nemohly rovnat.

Algoritmům Karatsuba a Rychlé Fourierově transformaci se také věnoval kolega Brožek ve své bakalářské práci [11], ve které se zabýval implementací knihovny pro násobení polynomů. Kolega však reprezentoval koeficienty polynomů pouze jako celá čísla. Časové porovnání kolegovy implementace s mojí by tedy nebyla na místě, protože násobení celých čísel je pro procesor méně náročné než násobení čísel v plovoucí řadové čárce.

Aritmetické operace na polynomech v jazyce C++ umožňuje knihovna NTL [12]. Knihovna rozhodne na základě stupně vstupních polynomů a rozsahu hodnot vstupních koeficientů, který algoritmus využije pro násobení dvou polynomů. NTL používá pro násobení polynomů algoritmus triviální, Karatsuba, Schönhage-Strassenův a FFT [13, stránka 13]. Knihovna ovšem také požaduje pro reprezentaci koeficientů celá čísla, ani zde tedy není časové porovnání s mou implementací na místě.

Závěr

Cílem práce bylo prozkoumání algoritmů pro násobení polynomů, konkrétně algoritmů Karatsuba, Toom-Cook a Rychlé Fourierovy transformace.

V první kapitole jsou algoritmy představeny společně se základními pojmy nezbytnými k pochopení této problematiky. Dále je zjištěna asymptotická složitost a jsou uvedeny pseudokódy těchto algoritmů.

Druhá kapitola se věnuje prostředkům použitým v této práci. Kromě použitých datových struktur je představena knihovna OpenMP, důležitá složka použitá k paralelizaci algoritmů pro násobení polynomů. Dále jsou představeny měřící prostředky a nastavení kompilátoru.

V třetí kapitole je popsána implementace zkoumaných algoritmů. Je zde popsána sekvenční verze těchto algoritmů, případně jejich optimalizace a paralelizace.

V poslední kapitole je provedeno měření skutečné časové složitosti algoritmů na základě náhodně vygenerovaných vstupních dat. V této kapitole je ověřeno, že algoritmy dosahují předpokládané časové složitosti na základě asymptotické složitosti zjištěné v první kapitole. Dále je v této kapitole provedeno měření využití skryté paměti, kde se ukázalo, že skrytá paměť je správně využívána. Na závěr je prozkoumána numerická stabilita algoritmů pro násobení polynomů a následná diskuse nad použitelností algoritmů z tohoto hlediska.

Literatura

[1] OLŠÁK, Petr: *Úvod do algebry, zejména lineární*. FEL ČVUT, Praha, 2007, ISBN 978-80-01-03775-1.

[2] KOLÁŘ, Josef: BI-ZDM - Přednáška č. 1 - Odhady růstu funkcí. Říjen 2015, [cit. 2016-03-04].

[3] KOLÁŘ, Josef: BI-ZDM - Přednáška č. 9 - Složitost rekurzivních algoritmů, řešení rekurentních rovnic, Master Theorem. Říjen 2015, [cit. 2016-03-15].

[4] WEIMERSKIRCH André, CHRISTOF Paar: Generalizations of the Karatsuba Algorithm for Efficient Implementations [online]. 2006, [cit. 2016-03-18]. Dostupné z: `https://eprint.iacr.org/2006/224.pdf`

[5] BRENT, Richard P., ZIMMERMANN Paul: Modern Computer Arithmetic [online]. 2006, [cit. 2016-03-20]. Dostupné z: `http://www.loria.fr/~zimmerma/mca/mca-0.1.pdf`

[6] Toom 3-Way Multiplication [online]. 2015, [cit. 2016-03-24]. Dostupné z: `https://gmplib.org/manual/Toom-3_002dWay-Multiplication.html`

[7] JIA, Y.-B.: Polynomial Multiplication and Fast Fourier Transform [online]. 2015, [cit. 2016-03-20]. Dostupné z: `http://web.cs.iastate.edu/~cs577/handouts/polymultiply.pdf`

[8] ŠIMEČEK, Ivan: *Moderní počítačové architektury a optimalizace implementace algoritmů*. FIT ČVUT, Praha, 2015, ISBN 978-80-01-05658-5.

[9] Complete Powerpoint Lecture Notes for Computer Systems: A Programmer's Perspective (CS:APP) [online]. 2014, [cit. 2016-05-08]. Dostupné z: `http://courses.cs.vt.edu/cs2506/Fall2014/Notes/L16.CachePoliciesAndPerformance.pdf`

[10] LEHÁR, Adam: *Rychlé násobení smíšených polynomů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

[11] BROŽEK, Miloslav: *Knihovna pro násobení polynomů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

[12] A Tour of NTL: Examples: Polynomials [online]. 2016, [cit. 2016-05-10]. Dostupné z: `http://www.shoup.net/ntl/doc/tour-ex3.html`

[13] KOZLÍK, Andrew: *Rychlé násobení polynomů*. Bakalářská práce, Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, 2008.

Příloha A

Seznam použitých zkratek

FFT Fast Fourier Transformation (Rychlá Fourierova transformace)

API Application Programming Interface (Aplikační programovací rozhraní)

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `graphs` ..... grafy použité v práci
- `text` ..... text práce
  - `BP_Cila_Michal_2016.pdf` ..... text práce ve formátu PDF
  - `BP_Cila_Michal_2016_zadani.pdf` .... zadání práce ve formátu PDF