ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Algoritmy pro násobení polynomů
Student:	Jakub Holub
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 triviální, Karatsuba, Toom-3, Toom-4 pro násobení polynomů.

2) Implementujte tyto algoritmy v jazyce C/C++ a analyzujte přesnost jejich výsledků.

3) Implementované algoritmy optimalizujte pomocí transformací kódu, efektivní využití skrytých pamětí, vektorizace a paralelizace (pomocí technologie OpenMP).

4) Otestujte výkonnost implementovaných algoritmů na fakultním serveru Star.

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 5. února 2016

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

Bakalářská práce

Algoritmy pro násobení polynomů

Jakub Holub

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

16. května 2016

Poděkování

Děkuji vedoucímu práce Ing. Ivanu Šimečkovi Ph.D. za cenné rady při psaní této práce.

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 16. května 2016 .....

Odkaz na tuto práci

Holub, Jakub. *Algoritmy pro násobení polynomů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato bakalářská práce se zaměřuje na vybrané algoritmy pro násobení polynomů. V teoretické části jsou tyto algoritmy popsány a je vysvětlen jejich princip. Praktická část se zaměřuje na implementaci a následnou optimalizaci těchto algoritmů. Pro implementované algoritmy je provedena analýza přesnosti výsledků. Optimalizované algoritmy jsou otestovány a změřeny na výpočetním serveru. Nakonec je provedeno porovnání s již existujícími implementacemi a knihovnami.

Klíčová slova polynom, násobení, optimalizace, paralelizace, OpenMP

Abstract

This bachelor thesis deals with chosen algorithms for multiplication of polynomials. These algorithms are described in the theoretical part of the thesis with explanation of their principle of operation. The practical part is focused on implementation of these algorithms followed by their optimization. For implemented algorithms an analysis of the accuracy of results is executed. Optimized algorithms are then tested and measured on a computer server. In the end a comparision with already existing implementations and libraries is performed.

Keywords polynomial, multiplication, optimisation, parallelization, OpenMP

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Pro násobení polynomů existuje mnoho algoritmů. Tyto algoritmy byly většinou původně zamýšleny pro násobení čísel, ale jsou jednoduše použitelné pro násobení polynomů.

Tato práce se zaměřuje na vybrané algoritmy, které řeší násobení polynomů. Nejdříve jsou tyto algoritmy popsány a je vysvětlen jejich princip. Následně je popsána jejich implementace v jazyce C++ a analyzována přesnost jejich výsledků. Algoritmy jsou poté optimalizovány pomocí transformací kódu, vektorizace a paralelizace pomocí technologie OpenMP. Cílem práce je tedy efektivní implementací těchto algoritmů, zejména rozdělením výpočtů algoritmů do více vláken nad sdílenou pamětí.

V poslední části jsou optimalizované algoritmy testovány na výpočetním serveru. U algoritmů je testován vliv počtu vláken na rychlost výpočtu. Jednotlivé algoritmy jsou nakonec porovnány mezi sebou a také s již existujícími implementacemi a knihovnami.

Kapitola 1

Základní pojmy a algoritmy

V této kapitole jsou popsány základní pojmy a algoritmy, kterými se tato práce zabývá. Po dohodě s vedoucím bakalářské práce byly vybrány algoritmy triviální, Karatsuba, Toom-3 a Toom-4. Pro následující popis algoritmů jsou předpokládaný polynomy p a q, jejíchž stupeň je n.

1.1 Základní pojmy

1.1.1 Polynom

Polynom (někdy nazýván jako mnohočlen) p o jedné proměnné x je funkce, která je tvaru

$$p(x) = \sum_{i=0}^{n} \alpha_i x^i$$

kde $n \in \mathbb{N}_0$ a zároveň pro každé i platí $\alpha_i \in \mathbb{R}$. Čísla $\alpha_0, \alpha_1, ..., \alpha_n$ nazýváme koeficienty polynomu [1].

1.1.2 Nulový polynom

Nulový polynom je takový polynom, kde pro každé $i \in [0, n], i \in \mathbb{N}_0$ platí $\alpha_i = 0$.

1.1.3 Stupeň polynomu

Stupeň polynomu p je nejvyšší i, pro které $\alpha_i \neq 0$. Stupeň nulového polynomu má hodnotu -1 [1].

$$deg(p) = max\{i \in \mathbb{N}_0 \mid \alpha_i \neq 0\}$$

1.1.4 Součin polynomů

Součinem polynomů p stupně n a polynomu q stupně m o stejné proměnné x vznikne nový polynom r stupně maximálně $n + m$, a to součinem každého členu polynomu p s každým členem polynomu q.

$$p(x) = \sum_{i=0}^{n} \alpha_i x^i, \qquad q(x) = \sum_{i=0}^{m} \beta_i x^j$$

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

1.2 Triviální algoritmus

Jedná se o nejpřímočařejší algoritmus pro součin polynomů. Postupně se násobí každý člen prvního polynomu s každým členem polynomu druhého. Asymptotická složitost tohoto algoritmu je tedy $\mathcal{O}(n^2)$. Algoritmus provede $(n+1)^2$ operací součinu a $(n+1)^2$ operací součtu.

Algoritmus 1 Triviální algoritmus

```
1: i ← 0
2: j ← 0
3: for i to deg(p) do
4:     for j to deg(q) do
5:         r[i + j] ← r[i + j] + p[i] ∗ q[i]
6:     end for
7: end for
```

1.3 Karatsubův algoritmus

1.3.1 Popis algoritmu

Karatsuba představil svůj algoritmus pro násobení polynomů v roce 1960 [2]. Byl původně zamýšlen pro násobení čísel, ale je také snadno aplikovatelný pro násobení polynomů. Jednalo se o první algoritmus, který byl asymptoticky rychlejší než triviální algoritmus. Byl také jako jeden z prvních typu rozděl a panuj, tedy je rekurzivně aplikován pro menší vstupy. Jeho asymptotická složitost je $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$. Je založený na myšlence, že operace součtu je asymptoticky rychlejší než operace součinu a snaží se tedy minimalizovat počet součinů. Místo 4 operací součinu provede pouze 3, ale zvýší počet operací součtu. Stupně násobených polynomů musí být shodné a musí být ve tvaru $2^k - 1$, kde $k \in \mathbb{N}_0$. Tedy počet koeficientů polynomu je ve tvaru 2^k. Tento tvar je požadován z toho důvodu, že algoritmus se spouští rekurzivně a rozděluje

polynomy na poloviny. Pokud se stupně vstupních polynomů liší, je nutné rozšířit tyto polynomy na nejbližší vyšší stupeň tvaru 2^k-1, přičemž nové členy polynomu budou mít nulové koeficienty. Pro polynomy stupně 0 se použije triviální algoritmus.

1.3.2 Princip algoritmu

Předpokládejme následující dva polynomy:

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

Součin těchto polynomů lze rozepsat následujícím způsobem:

$$r(x) = p(x) \cdot q(x) = a * x^{2n} + b * x^{2n+1} + c * x^{2n+2}$$

Přičemž platí

$$\begin{aligned} a &= \alpha_1\beta_1 \\ b &= \alpha_1\beta_2 + \alpha_2\beta_1 \\ c &= \alpha_2\beta_2 \end{aligned}$$

Takto rozepsaný součin polynomů vyžaduje 4 operace součinu. Karatsuba přišel s nápadem, že je možné ušetřit jeden součin na úkor vyššího počtu součtů. Tedy že část b lze zapsat následovně [3].

$$b = (\alpha_1 + \alpha_2)(\beta_1 + \beta_2) - a - c$$

To je ekvivalentní hodnota s původním b.

$$\begin{aligned} b &= (\alpha_1 + \alpha_2)(\beta_1 + \beta_2) - a - c \\ &= \alpha_1\beta_1 + \alpha_1\beta_2 + \alpha_2\beta_1 + \alpha_2\beta_2 - \alpha_1\beta_1 - \alpha_2\beta_2 \\ &= \alpha_1\beta_2 + \alpha_2\beta_1 \end{aligned}$$

V důsledku této úpravy již není nutné počítat z původního b součiny $\alpha_1\beta_2$ a $\alpha_2\beta_1$, ale pouze součin součtů $\alpha_1 + \alpha_2$ a $\beta_1 + \beta_2$. Každé volání algoritmu tedy vyžaduje 3 operace součinu a 4 operace součtu.

Tabulka 1.1: Porovnání počtu součinů triviálního a Karatsubova algoritmu

Stupeň n	**Triviální algoritmus**	**Karatsubův algoritmus**
0	1	1
1	4	3
3	16	9
7	64	27
15	256	81
31	1 024	243
127	16 384	2 187
1 023	1 048 576	59 049
4 095	16 777 216	531 441
16 383	268 435 456	4 782 969
65 535	4 294 967 296	43 046 721

Z tabulky 1.1 je možné zjistit, že součinů provede triviální algoritmus více pro libovolný stupeň $n > 0$.

Karatsubův algoritmus se spouští rekurzivně 3krát pro násobení polynomů poloviního stupně. Algoritmus se tedy dá zapsat pomocí rekurzivního vzorce $T(n) = 3 \cdot T(n/2) + f(n)$, kde $f(n)$ jsou operace mimo rekurzivní volání. Z tohoto vzorce lze ověřit, že asymptotická složitost je skutečně $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$. Algoritmus teoreticky provede

$$3^{log_2(n+1)}$$

součinů.

Algoritmus 2 Karatsubův algoritmus

```
1: function KARATSUBA(p, q)
2:     if deg(p) == 0 then
3:         product ← TRIVIAL(p, q)
4:     else
5:         a ← KARATSUBA(LOW(p), LOW(q))
6:         b ← KARATSUBA(LOW(p) + HIGH(p), LOW(q) + HIGH(q))
7:         c ← KARATSUBA(HIGH(p), HIGH(q))
8:         product ← a + (b − a − c) · x^(deg(p)/2) + c · x^deg(p)
9:     end if
10:    return product
11: end function
```

1.4 Toom-k algoritmy

Toom-k algoritmy, někdy také nazývané Toom-Cook k-way algoritmy, vymyslel Andrei Toom v roce 1963. Později je v roce 1966 Stephen Cook formálně

popsal a publikoval. Tyto algoritmy jsou rychlejší zobecněnou formou Karatsubova algoritmu. Stejně jako u Karatsubova algoritmu se nejdříve polynomy rozdělí na menší polynomy stejného stupně. Právě písmeno k v názvu algoritmu určuje, na kolik částí se polynom má rozdělit. Následně jsou rozdělené polynomy vyhodnoceny v $2k-1$ různých bodech. Odpovídající vyhodnocené polynomy v těchto bodech se poté vynásobí mezi sebou rekurzivním použitím stejného algoritmu. Pro polynomy stupně 0 se použije triviální algoritmus. Po vypočítání součinu se provede interpolace, kdy se zjistí koeficienty vynásobených polynomů. Z těchto polynomů se poté za pomoci posunů stupňů sestaví výsledný polynom, který je součinem původních polynomů [4].

Body, ve kterých se polynomy vyhodnotí, se volí tak, aby následně nebyla příliš komplikovaná interpolace pro získání koeficientů. Často se jedná o čísla blízko 0.

Algoritmus lze zapsat rekurzivním vzorcem

$$T(n) = (2k-1) \cdot T(n/k) + f(n),$$

kde $f(n)$ jsou operace mimo rekurzivní volání. Asyptotická složitost je potom $\Theta(n^{\log_k(2k-1)})$.

Tabulka 1.2: Různé varianty Toom-k algoritmu

k	**Složitost**	**Poznámka**
2	$\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$	Karatsubův algoritmus
3	$\Theta(n^{\log_3 5}) \approx \Theta(n^{1.465})$	Nejčastější varianta, přijatelná režie
4	$\Theta(n^{\log_4 7}) \approx \Theta(n^{1.404})$	Vyšší režie
...	...	...
16	$\Theta(n^{\log_{16} 31}) \approx \Theta(n^{1.239})$	Komplikovaná implementace, vysoká režie
k	$\Theta(n^{\log_k(2k-1)})$	-

S roustoucí konstantou k se snižuje asymptotická složitost, ale kód algoritmu se stává složitějším. Navíc se provede více konstantních operací, které asymptotická složitost nevyjadřuje. Nejčastěji používanou variantou je pro $k = 3$, kde je přijatelná režie a implementace není příliš složitá [4].

1.4.1 Toom-3 algoritmus

Jedná se o Toom-k algoritmus, pro který $k = 3$. Vstupní polynomy jsou nejdříve rozděleny na stejně velké třetiny:

$$p = p_0 + p_1 + p_2$$
$$q = q_0 + q_1 + q_2$$

Stupeň polynomů tedy musí být ve tvaru $3^l - 1$, kde $l \in \mathbb{N}_0$. Pokud mají polynomy rozdílné stupně, je potřeba rozšířit je na nejbližší vyšší společný stupeň takového tvaru, přičemž nové členy polynomu budou mít nulové koeficienty.

Poté se polynomy vyhodnotí v různých bodech a, b, c, d, e:

$$\begin{aligned}
p(a) &= a^0p_0 + a^1p_1 + a^2p_2 & q(a) &= a^0q_0 + a^1q_1 + a^2q_2 \\
p(b) &= b^0p_0 + b^1p_1 + b^2p_2 & q(b) &= b^0q_0 + b^1q_1 + b^2q_2 \\
p(c) &= c^0p_0 + c^1p_1 + c^2p_2 & q(c) &= c^0q_0 + c^1q_1 + c^2q_2 \\
p(d) &= d^0p_0 + d^1p_1 + d^2p_2 & q(d) &= d^0q_0 + d^1q_1 + d^2q_2 \\
p(e) &= e^0p_0 + e^1p_1 + e^2p_2 & q(e) &= e^0q_0 + e^1q_1 + e^2q_2
\end{aligned}$$

Takto vyhodnocené polynomy se mezi sebou vynásobí:

$$\begin{aligned}
r(a) &= p(a) \cdot q(a) \\
r(b) &= p(b) \cdot q(b) \\
r(c) &= p(c) \cdot q(c) \\
r(d) &= p(d) \cdot q(d) \\
r(e) &= p(e) \cdot q(e)
\end{aligned}$$

Pro výpočet těchto součinů se rekurzivně aplikuje Toom-3 algoritmus. Po vypočítání součinů je potřeba zpetně zjistit koeficienty r_0, r_1, r_2, r_3, r_4 těchto polynomů, tedy provede se interpolace:

$$\begin{pmatrix} r(a) \\ r(b) \\ r(c) \\ r(d) \\ r(e) \end{pmatrix} = \begin{pmatrix} a^0 & a^1 & a^2 & a^3 & a^4 \\ b^0 & b^1 & b^2 & b^3 & b^4 \\ c^0 & c^1 & c^2 & c^3 & c^4 \\ d^0 & d^1 & d^2 & d^3 & d^4 \\ e^0 & e^1 & e^2 & e^3 & e^4 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}$$

Po úpravě tedy

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} a^0 & a^1 & a^2 & a^3 & a^4 \\ b^0 & b^1 & b^2 & b^3 & b^4 \\ c^0 & c^1 & c^2 & c^3 & c^4 \\ d^0 & d^1 & d^2 & d^3 & d^4 \\ e^0 & e^1 & e^2 & e^3 & e^4 \end{pmatrix}^{-1} \begin{pmatrix} r(a) \\ r(b) \\ r(c) \\ r(d) \\ r(e) \end{pmatrix}$$

Z těchto polynomů se již za pomocí vhodných posunů a následnému sečtení sestaví výsledný polynom.

Algoritmus lze zapsat rekurzivním vzorcem

$$T(n) = 5 \cdot T(n/3) + f(n),$$

kde $f(n)$ jsou operace mimo rekurzivní volání. Odtud tedy asymptotická složitost $\Theta(n^{\log_3 5}) \approx \Theta(n^{1.465})$.

1.4.2 Toom-4 algoritmus

Jedná se o Toom-k algoritmus, pro který $k = 4$. Vstupní polynomy jsou nejdříve rozděleny na stejně velké čtvrtiny:

$$p = p_0 + p_1 + p_2 + p_3$$
$$q = q_0 + q_1 + q_2 + q_3$$

Stupeň polynomů tedy musí být ve tvaru $4^l - 1$, kde $l \in \mathbb{N}_0$. Pokud mají polynomy rozdílné stupně, je potřeba rozšířit je na nejbližší vyšší společný stupeň takového tvaru, přičemž nové členy polynomu budou mít nulové koeficienty.

Poté se polynomy vyhodnotí v různých bodech a, b, c, d, e, f, g:

$$\begin{aligned}
p(a) &= a^0p_0 + a^1p_1 + a^2p_2 + a^3p_3 & q(a) &= a^0q_0 + a^1q_1 + a^2q_2 + a^3q_3 \\
p(b) &= b^0p_0 + b^1p_1 + b^2p_2 + b^3p_3 & q(b) &= b^0q_0 + b^1q_1 + b^2q_2 + b^3q_3 \\
p(c) &= c^0p_0 + c^1p_1 + c^2p_2 + c^3p_3 & q(c) &= c^0q_0 + c^1q_1 + c^2q_2 + c^3q_3 \\
p(d) &= d^0p_0 + d^1p_1 + d^2p_2 + d^3p_3 & q(d) &= d^0q_0 + d^1q_1 + d^2q_2 + d^3q_3 \\
p(e) &= e^0p_0 + e^1p_1 + e^2p_2 + e^3p_3 & q(e) &= e^0q_0 + e^1q_1 + e^2q_2 + e^3q_3 \\
p(f) &= f^0p_0 + f^1p_1 + f^2p_2 + f^3p_3 & q(f) &= f^0q_0 + f^1q_1 + f^2q_2 + f^3q_3 \\
p(g) &= g^0p_0 + g^1p_1 + g^2p_2 + g^3p_3 & q(g) &= g^0q_0 + g^1q_1 + g^2q_2 + g^3q_3
\end{aligned}$$

Takto vyhodnocené polynomy se mezi sebou vynásobí:

$$
\begin{aligned}
r(a) &= p(a) \cdot q(a) \\
r(b) &= p(b) \cdot q(b) \\
r(c) &= p(c) \cdot q(c) \\
r(d) &= p(d) \cdot q(d) \\
r(e) &= p(e) \cdot q(e) \\
r(f) &= p(f) \cdot q(f) \\
r(g) &= p(g) \cdot q(g)
\end{aligned}
$$

Pro výpočet těchto součinů se rekurzivně aplikuje Toom-4 algoritmus. Po vypočítání součinů je potřeba zpetně zjistit koeficienty r_0, r_1, r_2, r_3, r_4, r_5, r_6 těchto polynomů, tedy provede se interpolace:

$$
\begin{pmatrix} r(a) \\ r(b) \\ r(c) \\ r(d) \\ r(e) \\ r(f) \\ r(g) \end{pmatrix} = \begin{pmatrix} a^0 & a^1 & a^2 & a^3 & a^4 \\ b^0 & b^1 & b^2 & b^3 & b^4 \\ c^0 & c^1 & c^2 & c^3 & c^4 \\ d^0 & d^1 & d^2 & d^3 & d^4 \\ e^0 & e^1 & e^2 & e^3 & e^4 \\ f^0 & f^1 & f^2 & f^3 & f^4 \\ g^0 & g^1 & g^2 & g^3 & g^4 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{pmatrix}
$$

Po úpravě tedy

$$
\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{pmatrix} = \begin{pmatrix} a^0 & a^1 & a^2 & a^3 & a^4 \\ b^0 & b^1 & b^2 & b^3 & b^4 \\ c^0 & c^1 & c^2 & c^3 & c^4 \\ d^0 & d^1 & d^2 & d^3 & d^4 \\ e^0 & e^1 & e^2 & e^3 & e^4 \\ f^0 & f^1 & f^2 & f^3 & f^4 \\ g^0 & g^1 & g^2 & g^3 & g^4 \end{pmatrix}^{-1} \begin{pmatrix} r(a) \\ r(b) \\ r(c) \\ r(d) \\ r(e) \\ r(f) \\ r(g) \end{pmatrix}
$$

Z těchto polynomů se již za pomocí vhodných posunů a následnému sečtení sestaví výsledný polynom.

Algoritmus lze zapsat rekurzivním vzorcem

$$T(n) = 7 \cdot T(n/4) + f(n),$$

kde $f(n)$ jsou operace mimo rekurzivní volání. Odtud tedy asymptotická složitost $\Theta(n^{\log_4 7}) \approx \Theta(n^{1.404})$.

Kapitola 2

Použité prostředky a optimalizační techniky

V této kapitole jsou popsány fundamentální prostředky a základní optimalizační techniky, které byly využity při implementaci zvolených algoritmů.

2.1 Jazyk C++

V práci je použit programovací jazyk C++. Jedná se o jazyk kompilovaný, zdrojový kód je tedy jednorázově přeložen do strojového kódu a vznikne samostatný program. Přestože je tento jazyk nezávislý na platformě, nabízí například přímý přístup do paměti nebo konstrukce, které lze převést přímo do strojového kódu. Lze tedy provádět optimalizace přímo pro konkrétní hardware. Proto je tento jazyk vhodný pro programy, u kterých je prioritou rychlost výpočtu, a z tohoto důvodu byl zvolen pro implementaci těchto algoritmů.

2.2 OpenMP

Knihovna OpenMP umožňuje snadnou paralelizaci C/C++ programů. Tato knihovna se skládá z direktiv pro překladač, které inicializují samotnou paralelizaci, tedy rozdělení práce programu na více vláken nad sdílenou pamětí. V důsledku díky tomu lze dosáhnout mnohanásobného zrychlení programů. Na rozdíl od dalších knihoven poskytuje vyšší úroveň abstrakce a je tedy nezávislý na cílové platformě.

2.2.1 Direktiva parallel

Jedná se o základní direktivu, která označuje začátek paralelního bloku. Vytvoří zadaný počet vláken, kód bloku se zduplikuje pro každé vlákno, a každé vlákno provede výpočty tohoto bloku. Na konci paralelního bloku vznikne bariéra, která čeká na dokončení výpočtů všech vláken [5].

2.2.2 Direktiva for

Direktiva *for* slouží pro paralelizaci `for` cyklu. Rozdělí cyklus na disjunktní bloky a každému vláknu přidělí jeden blok. Tato direktiva by měla být uvnitř direktivy *parallel*, v opačném případě by výpočty v cyklu provádělo pouze rodičovské vlákno. Tuto kombinaci direktiv lze zapsat i zkráceně jako *parallel for*. Pomocí této direktivy lze paralelizovat zároveň i vnořené cykly pomocí parametru *collapse*(x), kde x je počet `for` cyklů, které se mají paralelizovat [5].

Pomocí parametru *schedule* lze nastavit, jakým způsobem se má plánovat rozdělení iterací mezi vlákna. Umožňuje následující volby [6][5]:

- `Static` - každému vláknu se přiřadí část kontinuálních iterací cyklu. Tato část má pevnou velikost. Pokud tato velikost není uvedena, rozloží se iterace rovnoměrně mezi vlákna.
- `Dynamic` - každému vláknu se dynamicky přiřadí velikost kontinuálních iterací cyklu. Jakmile vlákno dokončí aktuální část iterací, obdrží k vykonání další část. Pokud velikost není stanovena, předpokládá se 1.
- `Guided` - podobné plánování jako `dynamic`, ale velikost začíná jako vysoké číslo (zpravidla počet iterací ku počtu vláken) a postupně se snižuje na základě počtu zbývajících iterací k vykonání. Velikost jako parametr určuje minimální počet iterací k provedení jedním vláknem.
- `Runtime` - plánování se rozhodne za běhu až v okamžiku provádění.
- `Auto` - plánování rozhodne kompilátor a operační systém.

2.2.3 Direktiva task

Direktiva *task* je vhodná pro paralelizaci volání funkcí, zejména rekurzivních. Rodičovské vlákno vytvoří úlohu, kterou vlákna v paralelním bloku začnou vykonávat. Tato direktiva tedy musí být uvnitř bloku *parallel*. Zpravidla u direktivy *task* není žádoucí, aby každé vlákno vykonávalo stejný kód paralelního bloku, a proto se používá direktiva *single*. Ta zajistí, že kód paralelního bloku provádí pouze jedno vlákno. Teprve až při zavolání direktivy *task* se použijí další vlákna [5].

Při použití direktivy *task* se úloha předá k vykonání, ale rodičovské vlákno pokračuje dále v kódu. Pokud je žádoucí čekat na výsledek, který vypočte *task*, je třeba použít bariéru *taskwait*, která čeká na dokončení všech přímých úloh. Až poté pokračuje rodičovské vlákno dále v kódu.

Vzhledem k velké režii spojené s touto direktivou lze použití *task* podmínit parametrem *if*, na základě kterého se buď použije *task* nebo v opačném případě kód vykoná přímo rodičovské vlákno. To je výhodné zejména u rekurzivních algoritmů, kde se postupně snižuje velikost množiny, se kterou kód pracuje.

2.2.4 Parametr shared

Tento parametr u direktiv značí proměnné, které jsou sdílené napříc všemi vlákny. Všechna vlákna tedy přistupují ke stejné proměnné [7].

2.2.5 Parametr private

Tento parametr u direktiv značí proměnné, které mají být pro každé vlákno unikátní. Každé vlákno si tedy vytvoří vlastní kopii této proměnné a s ní pracuje. Tato kopie ale nemá inicializovanou hodnotu. Pokud by vlákna měla mít tuto proměnnou inicializovanou na hodnotu, která byla nastavena před vstupem do paralelního bloku, bylo by možné použit parametr *firstprivate* [7].

2.2.6 Direktiva atomic

Příkaz, který následuje za touto direktivou, je prováděň atomicky, tedy vykonává ho vždy jen jedno vlákno současně. Tato direktiva je proto užitečná například při zápisu do stejné proměnné [5].

2.3 Rozbalení cyklu

Rozbalení cyklu (loop unrolling) je technika, při níž se zvýší inkrementace cyklu o určitý faktor, přičemž v těle cyklu se provede kód pro všechny hodnoty mezi aktuální a následující iterací.

Algoritmus 3 Rozbalení cyklu

```
1:  for i = 0; i < n; i = i + 1 do                ▷ před rozbalením cyklu
2:      CODE(i)
3:  end for
4:
5:
6:  factor ← 4
7:  for i = 0; i < n; i = i + factor do            ▷ po rozbalení cyklu
8:      CODE(i)
9:      CODE(i + 1)
10:     CODE(i + 2)
11:     CODE(i + 3)
12: end for
```

Tato technika se používá z důvodu, že moderní procesory mají hlubokou pipeline a podmíněné skoky je brzdí. Rozbalení cyklu sníží počet iterací cyklu a tím tedy i počet podmíněných skoků. Zároveň mohou být instrukce díky delšímu vnitřku cyklu lépe naplánované [8].

2.4 Vektorizace

Vektorizace umožňuje provést logické či matematické operace pro více dat najednou v jedné instrukci. To je výhodné zejména uvnitř cyklu, kde jsou v iteracích opakovaně prováděny stejné operace. Překladač k tomu využívá speciální `SIMD` (Single instruction, multiple data) instrukce z vektorových sad instrukcí. Vektorizaci je možné aktivovat několika způsoby [9]:

- Samostatný assemblerový soubor, který je připojen k výsledku.
- Assemblerový kód vložený přímo v kódu C++.
- Instrinsic instrukce, které se chovají jako vestavěné funkce, které odpovídají jednotlivým vektorovým instrukcím. Překladač se poté postará o vhodné přeložení.
- Automatická vektorizace, kterou zajistí kompilátor.

V této práci je využita automatická vektorizace. Aby kompilátor mohl zajistit automatickou vektorizaci, musí cyklus splňovat následující podmínky [9]:

- Nesmí obsahovat podmínky či volání funkcí.
- Data, s kterými se v cyklu pracuje, by mezi iteracemi měla v paměti navazovat.
- Mezi iteracemi by neměly být datové závislosti, které by mohla vektorizace narušit.
- Cyklus musí být nejvnitřnější.

Kapitola 3

Implementace a optimalizace algoritmů

V této kapitole je popsána implementace, sekvenční optimalizace a paralelní optimalizace jednotlivých algoritmů. Zároveň je zde analyzována přesnost algoritmů.

3.1 Společné prvky implementace algoritmů

Pro implementaci všech algoritmů je jako datový typ pro uložení polynomu využito jednoduché pole. Nabízí se pro polynomy vytvořit speciální datový typ, ve kterém by byl uložen jak polynom samotný, tak i jeho stupeň a u některých algoritmů případně i velikost alokovaného pole. To by ale zbytečně komplikovalo kód a přidalo i menší režii. Vzhledem k tomu, že v této práci je snaha o maximální rychlost implementovaných algoritmů, je upřednostněna na úkor čitelnosti kódu.

Ukazatele na pole, zejména v parametrech funkcí, jsou v kódu označeny klíčovým slovem `restrict`, které překladači napoví, že na toto pole ukazuje pouze tento jeden ukazatel. Díky tomu může překladač vygenerovat efektivnější kód [8].

3.2 Triviální

3.2.1 Implementace

Triviální algoritmus je nejpřímočařejší algoritmus pro násobení polynomů. Je složen ze dvou vnořených cyklů, ve kterých se násobí každý člen prvního polynomu s každým členem druhého polynomu.

3.2.2 Sekvenční optimalizace

Vzhledem k vlastnosti algoritmu, kdy jsou iterativně násobeny koeficienty obou polynomů, tedy kontinuálně v poli, je pro tento algoritmus využita automatická vektorizace, která značně urychlí výpočet. Manuální rozbalení cyklu zde tedy nemá velký význam využít.

3.2.3 Paralelní optimalizace

K samotné paralelizaci tohoto algoritmu lze přistoupit několika přístupy [10]:

1. První možnost je použití paralelizace na vnější cyklus. V takovém případě ale jednotlivé zapisované oblasti nejsou disjuktní, a proto vznikají kolize při zápisu. Je tedy nutné zápis provádět jako atomické operace, což umožňuje direktiva *atomic* z knihovny OpenMP. To ale degraduje samotné použití paralelizace, protože jednotlivá vlákna musejí čekat na zápis. V důsledku je zrychlení jen nepatrné.

2. Druhou možností je použití paralelizace na vnitřní cyklus. Zapisované oblasti v tomto případě jsou disjunktní, nevznikají tedy žádné kolize a vlákna se zbytečně nezdržují. Nevýhoda tohoto řešení je větší režie synchronizace vláken.

3. Třetí možností je rozdělit vlákna tak, aby každé počítalo navzájem disjunktní oblasti. Tato varianta však vyžaduje manuální plánování a vyvažování zátěže.

V této práci je implementovaná druhá varianta, tedy paralelizace vnitřního cyklu, jelikož při měření byla nejrychlejší. Pro paralelizaci je použita direktiva *for* z knihovny OpenMP, která je určena pro paralelizaci `for` cyklu. Plánování této direktivy je nastaveno jako *static* bez pevné velikosti, tedy každé vlákno dostane stejně velkou část iterací cyklu.

3.3 Karatsubův algoritmus

3.3.1 Implementace

Jednou z částí, které je v implementaci potřeba vyřešit, je rozdělování polynomů na poloviny. Bylo by možné vytvořit 4 nové polynomy poloviního stupně, do kterých se překopírují odpovídající poloviny původních polynomů. To by ale brzdilo rychlost výpočtu, jelikož by se v každém rekurzivním volání musela alokovat nová pole a následně do nich uložit obsah z původních polynomů. Díky ukazatelům v jazyku C++ však lze tuto režii obejít. Místo nových polí se vytvoří pouze 4 ukazetele, které budou ukazovat na odpovídající místa původních polí. Počet provedených operací pro rozdělení polynomů na poloviny je tedy konstantní.

V implementaci je potřeba v každém rekurzivním volání kromě koncových alokovat dvě pomocná pole o velikosti aktuálního stupně polynomů. Tato pomocná pole jsou nejprve využita pro uložení výsledků součinu dolních a horních polovin polynomu. Pole jsou nutná z toho důvodu, že jejich výsledek se využívá dvakrát a nelze tedy poslat jako parametr přímo ukazatel na výsledný polynom. Po uložení výsledků z těchto polí jsou následně ještě využita pro poslední třetí volání. Do každého pole se uloží součet dolní s horní polovinou jednoho polynomu. Pro toto poslední volání je již možné použit jako výsledný polynom ukazatel na odpovídající místo výsledného polynomu z aktuálního volání. Výsledek třetího volání se tedy již nemusí nijak zpracovávat. Celkově je zde tedy mírná režie s alokací a uvolňování paměti, avšak toto řešení má také výhodu, že u paralelní optimalizace nebude nutné použít atomické operace při zápisu do paměti.

Na rozdíl od pseudokódu (2) algoritmu, kde je sčítání a odčítání polynomů bráno jako jedna operace, jsou polynomy uloženy v poli. Počet operací součtu bude tedy ve skutečnosti násobně vyšší v závislosti na stupni polynomů n. Tyto součty je třeba provést v cyklu a budou tedy vedle samotného rekurzivního volání kritickou částí algoritmu, na které je vhodné se v sekvenční optimalizaci zaměřit.

3.3.2 Sekvenční optimalizace

V každém volání algoritmu je třeba alokovat 2 nová pole. To není příliš vhodné, jelikož režie alokace a následného uvolnění paměti zabírá určitý čas. Tuto režii lze alespoň mírně snížit tím, že se provede pouze jedna alokace pole o velikosti součtu velikostí původních dvou polí. Druhé pole bude potom pouze jako ukazatel mířit na patřičné místo prvního pole.

Všechny cykly jsou za pomoci překladače automaticky vektorizovány. Nemá zde tedy velký smysl použit manuální rozbalení cyklu, jelikož by se tím vektorizace do jisté míry degradovala.

Vzhledem k tomu, že algoritmus je rekurzivní, vzniká na zásobníku každým voláním nový aktivační záznam s parametry a návratovou adresou. Je s tím tedy spojená určitá režie a požadavek na místo na zásobníku. Nabízí se tedy možnost v určitém stupni přepnout algoritmus na triviální a urychlit tím celý výpočet. Samotný stupeň přepnutí má smysl zjistit až při paralelní optimalizaci, kde navíc přibude režie spojená právě s paralelizací.

3.3.3 Paralelní optimalizace

Pro rekurzivní algoritmus lze z knihovny OpenMP použit direktivu *task*, která slouží převážně právě k těmto účelům. Před každé rekurzivní volání se přidá tato direktiva a vlákna začnou vykonávat tuto úlohu. Vzhledem k tomu, že rodičovské vlákno pokračuje dále a nečeká na výsledek *task*, je třeba přidat direktivu *taskwait*, která čeká na dokončení všech přímých potomků. Aby

rodičovské vlákno zbytečně nečinně nečekalo na dokončení všech přimých potomků, lze ho využít pro výpočet posledního rekurzivního volání, a tím urychlit výpočet.

S direktivou *task* je spojená i určitá režie. Proto jí není vhodné využívat pro malé výpočty, tedy v tomto případě pro koncové rekurzivní volání, kdy je stupeň polynomů už velmi nízký.

Zjištění této hranice bylo provedeno měřením. Výsledky měření znázorňuje graf 4.2, z kterého lze zjistit, že nejvhodnější hranicí je stupeň 255.

3.4 Toom-k algoritmy

Toom-k algoritmy jsou zobecněnou formou Karatsubova algoritmu, takže jejich implementace a optimalizace je velice podobná. I tak jsou zde ale určité rozdíly.

Důležitou částí algoritmu je vhodný výběr různých bodů pro vyhodnocení polynomů. Tyto body mohou být zvoleny libovolně s jednou podmínkou, a to aby byla možná následná inverze matice bodů. Body se zpravidla volí blízko čísla 0, aby bylo co nejjednodušší jejich vyhodnocení a následná interpolace, díky které se zjistí koeficienty součinů takto vyhodnocených polynomů. Vyjímkou je často používaný bod s hodnotou ∞, který představuje výraz $\lim_{x\to\infty} \frac{p(x)}{x^{deg(p)}}$. To ve výsledku znamená, že hodnota $p(\infty)$ je koeficient u členu s nejvyšší mocninou. Výběr vhodných bodů je tedy také způsob optimalizace.

3.5 Toom-3 algoritmus

3.5.1 Implementace

V implementaci byly použity body $0, 1, -1, -2, \infty$. Tyto body jsou vybrány z důvodu, aby bylo časově jednoduché vyhodnocení polynomů a následná interpolace.

$$\begin{aligned}
p(0) &= 0^0p_0 + 0^1p_1 + 0^2p_2 && = p_0 \\
p(1) &= 1^0p_0 + 1^1p_1 + 1^2p_2 && = p_0 + p_1 + p_2 \\
p(-1) &= (-1)^0p_0 + (-1)^1p_1 + (-1)^2p_2 && = p_0 - p_1 + p_2 \\
p(-2) &= (-2)^0p_0 + (-2)^1p_1 + (-2)^2p_2 && = p_0 - 2p_1 + 4p_2 \\
p(\infty) &= p_2 && = p_2
\end{aligned}$$

Pro vyhodnocení tedy stačí 2 součiny a 6 součtů. Analogicky se vyhodnotí i druhý polynom. Tyto polynomy jsou následně vynásobeny rekurzivním voláním.

Koeficienty vynásobených polynomů v těchto bodech jsou vypočítány následovně:

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ (-2)^0 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix}$$

Pokud by se pro vypočítání koeficientů použilo přímé vynásobení matice bodů s vektorem vyhodnocených polynomů, bylo by nutné provést poměrně hodně operací součinu a dělení. Proto je snaha tento součin optimalizovat. Toho lze dosáhnout šikovnou sekvencí operací, která tento součin vypočte co nejefektivněji. Avšak nalézt takovou sekvenci je poměrně složitý proces a závisí i na zvolených bodech pro vyhodnocení polynomů. V této práci je použita sekvence, kterou představil Marco Bodrato [11].

$$\begin{aligned} r_0 &= r(0) \\ r_4 &= r(\infty) \\ r_3 &= \frac{r(-2) - r(1)}{3} \\ r_1 &= \frac{r(1) - r(-1)}{2} \\ r_2 &= r(-1) - r_0 \\ r_3 &= \frac{r_2 - r_3}{2} + 2r_4 \\ r_2 &= r_2 + r_1 - r_4 \\ r_1 &= r_1 - r_3 \end{aligned}$$

Tato sekvence vyžaduje pouze 3 operace dělení, 1 operaci násobení a 8 operací součtu. To je oproti přímému násobení značně menší počet operací.

3.5.2 Sekvenční optimalizace

Vyhodnocení, interpolace a následné sestavení polynomu probíhá v cyklech v závislosti na aktuálním stupni polynomů. Čas pro vykonání těchto cyklů se podařilo snížit díky rozbalení cyklů. Pro vyhodnocení byl cyklus rozbalen 3krát. Tento počet byl vybrán zejména proto, že stupeň polynomů je tvaru $3^l - 1$, tedy počet koeficientů je násobkem čísla 3. Díky tomu není nutné provádět operace pro zbytkové iterace. Po vynásobení vyhodnocených polynomů vzniknou polynomy dvojnásobného stupně, takže lze provést rozbalení cyklu 6krát beze zbytku. Toho bylo využito pro interpolaci a sestavení výsledného polynomu, kde bylo rozbalení aplikováno 6krát.

3.5.3 Paralelní optimalizace

Paralelní optimalizace je stejná jako u Karatsubova algoritmu. Jediný rozdíl je ve zvolené hranici pro přepnutí na triviální algoritmus.

Zjištění této hranice bylo provedeno měřením. Výsledky měření znázorňuje graf 4.3, z kterého lze zjistit, že nejvhodnější hranicí je stupeň 242.

3.6 Toom-4 algoritmus

Implementace a optimalizace Toom-4 algoritmu je takřka totožná s algoritmem Toom-3. Jsou zde tedy pouze zmíněny odlišnosti.

3.6.1 Implementace

V implementaci byly použity body $0, -2, 1, -1, 2, \frac{1}{2}, \infty$.

$$
\begin{aligned}
p(0) &= 0^0p_0 + 0^1p_1 + 0^2p_2 + 0^3p_3 && = p_0 \\
p(-2) &= (-2)^0p_0 + (-2)^1p_1 + (-2)^2p_2 + (-2)^3p_3 && = p_0 - 2p_1 + 4p_2 - 8p_3 \\
p(1) &= 1^0p_0 + 1^1p_1 + 1^2p_2 + 1^3p_3 && = p_0 + p_1 + p_2 + p_3 \\
p(-1) &= (-1)^0p_0 + (-1)^1p_1 + (-1)^2p_2 + (-1)^3p_3 && = p_0 - p_1 + p_2 - p_3 \\
p(2) &= 2^0p_0 + 2^1p_1 + 2^2p_2 + 2^3p_3 && = p_0 + 2p_1 + 4p_2 + 8p_3 \\
p\left(\frac{1}{2}\right) &= \left(\frac{1}{2}\right)^0 p_0 + \left(\frac{1}{2}\right)^1 p_1 + \left(\frac{1}{2}\right)^2 p_2 + \left(\frac{1}{2}\right)^3 p_3 && = 8p_0 + 4p_1 + 2p_2 + p_3 \\
p(\infty) &= p_3 && = p_3
\end{aligned}
$$

Pro vyhodnocení tedy stačí 9 součinů a 15 operací součtu. Analogicky se vyhodnotí i druhý polynom. Tyto polynomy jsou následně vynásobeny rekurzivním voláním.

Koeficienty vynásobených polynomů v těchto bodech jsou vypočítány následovně:

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 & 0^5 & 0^6 \\ (-2)^0 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 & (-2)^5 & (-2)^6 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 & 1^5 & 1^6 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 & (-1)^5 & (-1)^6 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^5 & 2^6 & \\ \left(\frac{1}{2}\right)^0 & \left(\frac{1}{2}\right)^1 & \left(\frac{1}{2}\right)^2 & \left(\frac{1}{2}\right)^3 & \left(\frac{1}{2}\right)^4 & \left(\frac{1}{2}\right)^5 & \left(\frac{1}{2}\right)^6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} r(0) \\ r(-2) \\ r(1) \\ r(-1) \\ r(2) \\ r\left(\frac{1}{2}\right) \\ r(\infty) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & \frac{1}{60} & -\frac{2}{3} & -\frac{2}{9} & \frac{1}{36} & \frac{2}{45} & -2 \\ -\frac{5}{4} & -\frac{1}{24} & \frac{2}{3} & \frac{2}{3} & -\frac{1}{24} & 0 & 4 \\ \frac{5}{2} & 0 & \frac{3}{2} & -\frac{7}{18} & -\frac{1}{18} & -\frac{1}{18} & \frac{5}{2} \\ \frac{1}{4} & \frac{1}{24} & -\frac{1}{6} & -\frac{1}{6} & \frac{1}{24} & 0 & -5 \\ -\frac{1}{2} & -\frac{1}{60} & -\frac{1}{3} & \frac{1}{9} & \frac{1}{36} & \frac{1}{90} & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r(0) \\ r(-2) \\ r(1) \\ r(-1) \\ r(2) \\ r\left(\frac{1}{2}\right) \\ r(\infty) \end{pmatrix}$$

Stejně jako u Toom-3 algoritmu, i u tohoto algoritmu je použita efektivní sekvence operací pro vypočítání koeficientů, kterou představil Marco Bodrato [11].

$$
\begin{aligned}
r_0 &= r(0) & r_6 &= r(\infty) \\
r_5 &= r\left(\frac{1}{2}\right) + r(2) & r_1 &= \frac{r(2) - r(-2)}{2} \\
r_4 &= r(2) - r_0 & r_4 &= \frac{r_4 - r_1}{4} - 16r_6 \\
r_3 &= \frac{r(1) - r(-1)}{2} & r_2 &= r(1) - r_3 \\
r_5 &= r_5 - 65r_2 & r_2 &= r_2 - r_6 - r_0 \\
r_5 &= \frac{r_5 + 45r_2}{2} & r_4 &= \frac{r_4 - r_2}{3} \\
r_2 &= r_2 - r_4 & r_1 &= r_5 - r_1 \\
r_5 &= \frac{r_5 - 8r_3}{9} & r_3 &= r_3 - r_5 \\
r_1 &= \frac{\frac{r_1}{15} + r_5}{2} & r_5 &= r_5 - r_1
\end{aligned}
$$

Tato sekvence (po řádcích) vyžaduje pouze 8 operací dělení, 4 operaci násobení a 18 operací součtu. To je oproti přímému násobení značně menší počet operací.

3.6.2 Sekvenční optimalizace

U Toom-4 algoritmu bylo provedeno rozbalení cyklu pro vyhodnocení 4krát a pro interpolaci a sestavení výsledného polynomu 8krát.

3.6.3 Paralelní optimalizace

Paralelní optimalizace je stejná jako u Toom-3 algoritmu, potažmo Karatsubova algoritmu. Jediný rozdíl je ve zvolené hranici pro přepnutí na triviální algoritmus.

Zjištění této hranice bylo provedeno měřením. Výsledky měření znázorňuje graf 4.4, z kterého lze zjistit, že nejvhodnější hranicí je stupeň 255.

3.7 Přesnost implementovaných algoritmů

V jazyce C++ se při použití mnoha matematických operací s datovým typem `double` ztrácí přesnost výsledků. To je způsobeno tím, že tento datový typ

neumí reprezentovat všechny hodnoty a s přibývajícími operacemi chyba narůstá. Vzhledem k tomu, že každý algoritmus používá odlišný způsob výpočtu součinu polynomů, bude mezi nimi určitý rozdíl v přesnosti výsledků.

Samotná přesnost výsledku také závisí na stupních vstupních polynomů. Čím větší je stupeň polynomů, tím více se provede matematických operací a ztratí se na přesnosti. Přesnost ještě mohou ovlivnit samotné hodnoty koeficientů vstupních polynomů, jelikož z nich vycházejí veškeré výpočty.

Nejpřesnější je algoritmus triviální, který počítá součin přímočarým způsobem. Zbývající algoritmy sice provádí méně součinů, ale naopak provádí více operací součtu. U Toom-3 a Toom-4 algoritmů se navíc vyskytuje násobení či dělení několika konstantami. Největší odchylka koeficientů od skutečných hodnot se ve výsledném polynomu vyskytuje u členů, jejichž exponent má hodnotu blízko poloviny stupně tohoto polynomu, jelikož s těmito koeficienty je provedeno nejvíce operací.

Měření přesnosti těchto algoritmů se věnuje podkapitola 4.5.

Kapitola 4

Testování a měření

Tato kapitola se zabývá měřením a testováním implementovaných algoritmů na výpočetním serveru. Vstupní polynomy pro testování byly vygenerovány vlastním náhodným generátorem polynomů.

4.1 Výpočetní server

Implementované algoritmy jsou testovány na fakultním výpočetním serveru `Star`. Tento server umožňuje využít pro výpočty až 12 vláken nad sdílenou pamětí. Na serveru přijímá jednotlivé požadavky k výpočtu fronta, která postupně zpracovává příchozí požadavky. Díky tomu je zajištěno, že jednotlivé procesy, které jsou určeny ke zpracování, nejsou ovlivněny jinými procesy. Na tomto serveru tedy stejný výpočet puštěný opakovaně bude časově trvat s malou odchylkou téměř stejně dlouho. Proto je tento server vhodný pro měření.

4.2 Nastavení kompilátoru

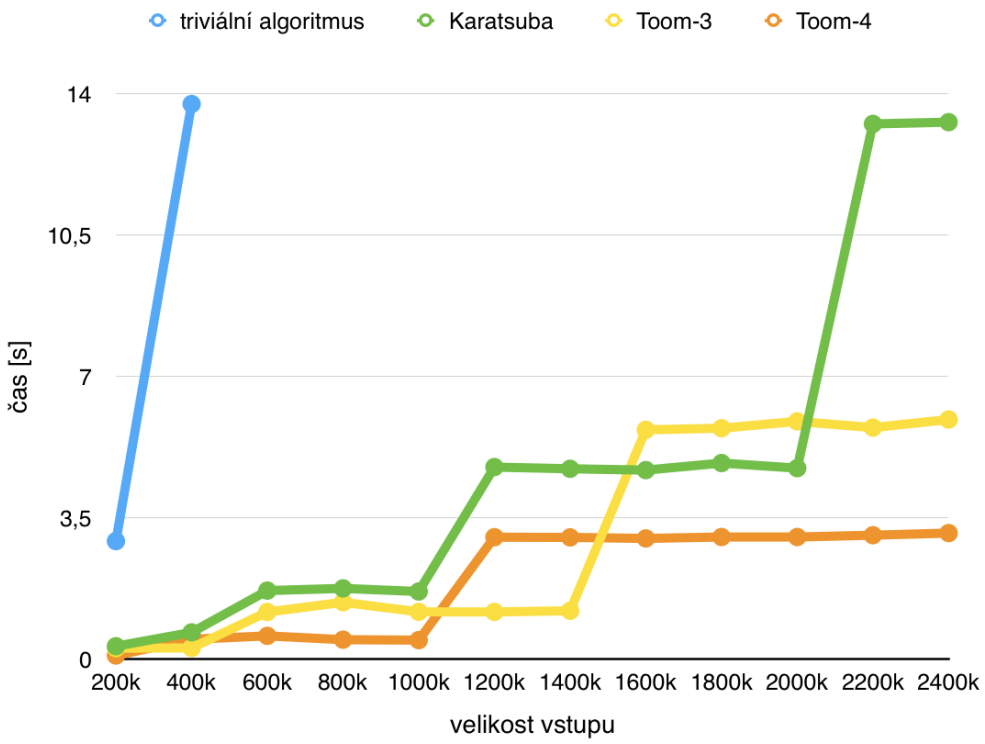
Důležitou součástí je i nastavení kompilátoru g++, tedy zvolené přepínače, kterými lze výrazně urychlit výpočet. Dále lze pomocí přepínače nastavit použitou sadu vektorových instrukcí.

Použité nastavení pro měření:

- `O3` - jednotný přepínač, který aktivuje mnoho dalších přepínačů, které optimalizují výsledný program
- `ffast-math` - zrychluje některé matematické operace
- `funroll-loops` - provede rozbalení cyklů, u kterých je možné zjistit počet iterací při kompilaci
- `ftree-vectorize` - aktivuje automatickou vektorizaci

- `mavx` - vektorizace využije AVX instrukce

4.3 Srovnání algoritmů



Obrázek 4.1: Srovnání implementovaných algoritmů. Velikost vstupu je uvedena v tisících. Algoritmy jsou spuštěny s 24 vlákny

Z grafu (4.1) lze zjistit, že asymptotické předpoklady algoritmů odpovídají skutečnému měření. Triviální algoritmus je příliš pomalý a jeho měření pro vyšší vstupy by bylo zbytečné.

Zajímavější jsou zbývající algoritmy. U těch lze zpozorovat, jak postupně skokovitě stoupá čas potřebný k výpočtu. To je důsledek toho, že stupně vstupních polynomů jsou zarovnány nulovými koeficienty na nejbližší vyšší společný stupeň $a^k - 1$, kde a je konstanta podle daného algoritmu. Dále z výsledků měření bylo možné zjistit, že u těchto algoritmů při skoku na další mocninu k se potřebný čas pro výpočet zvýšil tolikrát, kolikrát je v daném algoritmu rekurzivní volání funkce.

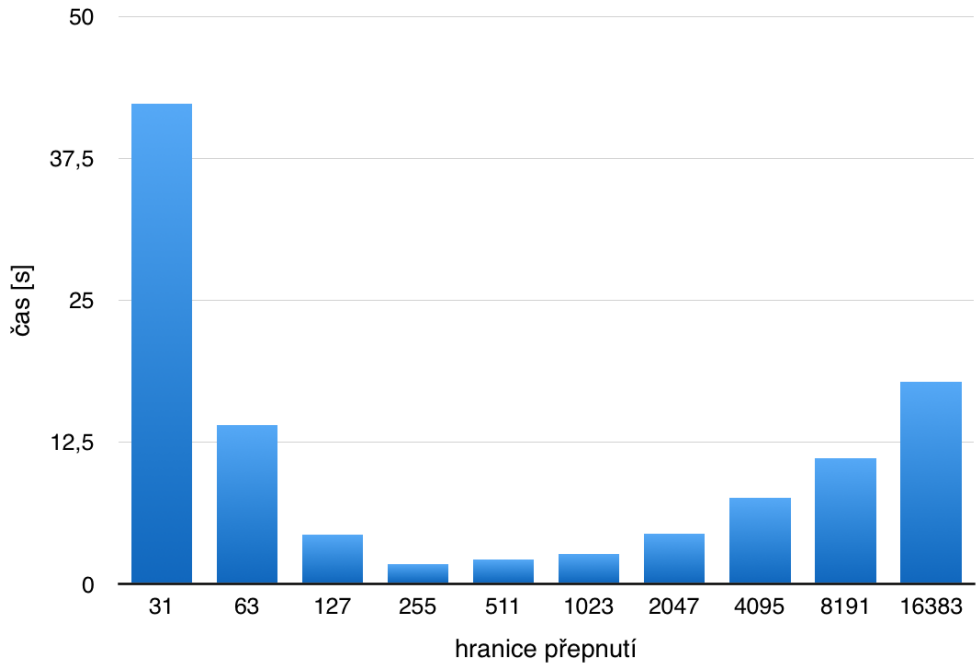
4.4 Zjištění hranice přepnutí

U algoritmů Karatsuba, Toom-3 a Toom-4 je v nejnižší hladině rekurze použit pro součin triviální algoritmus. Tato podkapitola se zabývá nalezením nejvhodnějších hranic stupňů polynomů, při které má dojít k přepnutí na triviální algoritmus. Při použití příliš nízké hranice by výpočet brzdila režie spojená s paralelizací. Na druhé straně při zvolení příliš vysoké hranice bude výpočet brzdit triviální algoritmus, který je pro polynomy vyššího stupně pomalejší. Zvolený způsob měření pro zjištění této hranice testuje všechny relevantní hloubky rekurze, najde tedy nejvhodnější hranici.

Hranice přepnutí těchto algoritmů je měřena na polynomech o stupni 1 milionu. Nejvhodnější hranice se může lišit pro polynomy různého stupně, zejména pro extrémně nízke či vysoké stupně. Pro většinu polynomů bude však takto hranice vyhovující volbou.

4.4.1 Karatsubův algoritmus

Vzhledem k podstatě Karatsubova algoritmu bude hranice přepnutí ve tvaru $2^l - 1$, kde $k \in \mathbb{N}_0$.

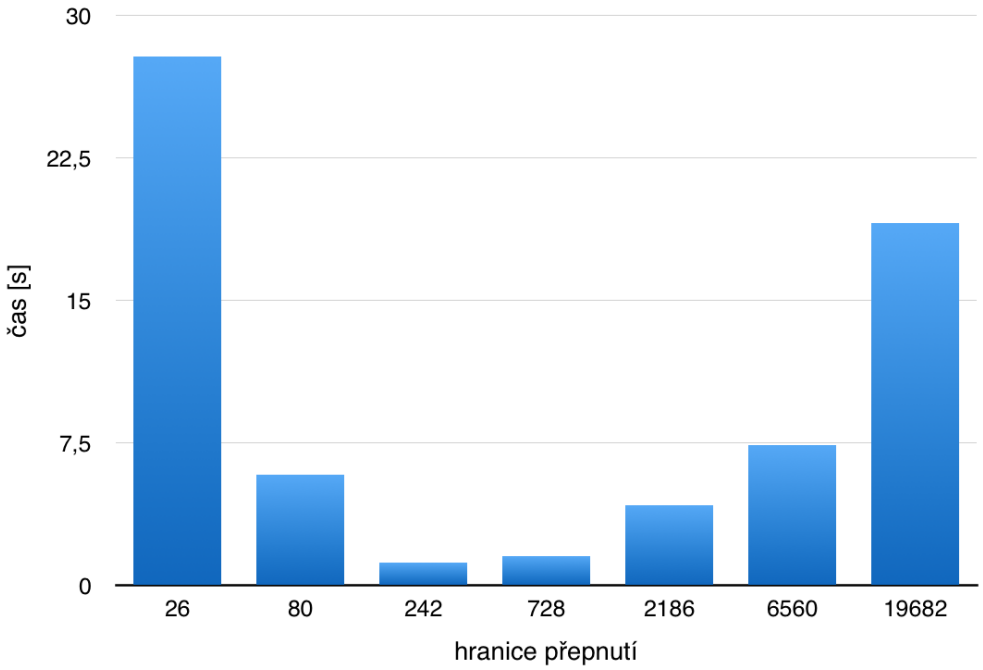


Obrázek 4.2: Časy pro různé hranice přepnutí Karatsubova algoritmu pro polynomy o stupni 1 milionu

Z grafu 4.2 lze zjistit, že hranice přepnutí vychází nejlépe při stupni 255.

4.4.2 Toom-3 algoritmus

Vzhledem k podstatě Toom-3 algoritmu bude hranice přepnutí ve tvaru $3^l - 1$, kde $l \in \mathbb{N}_0$.

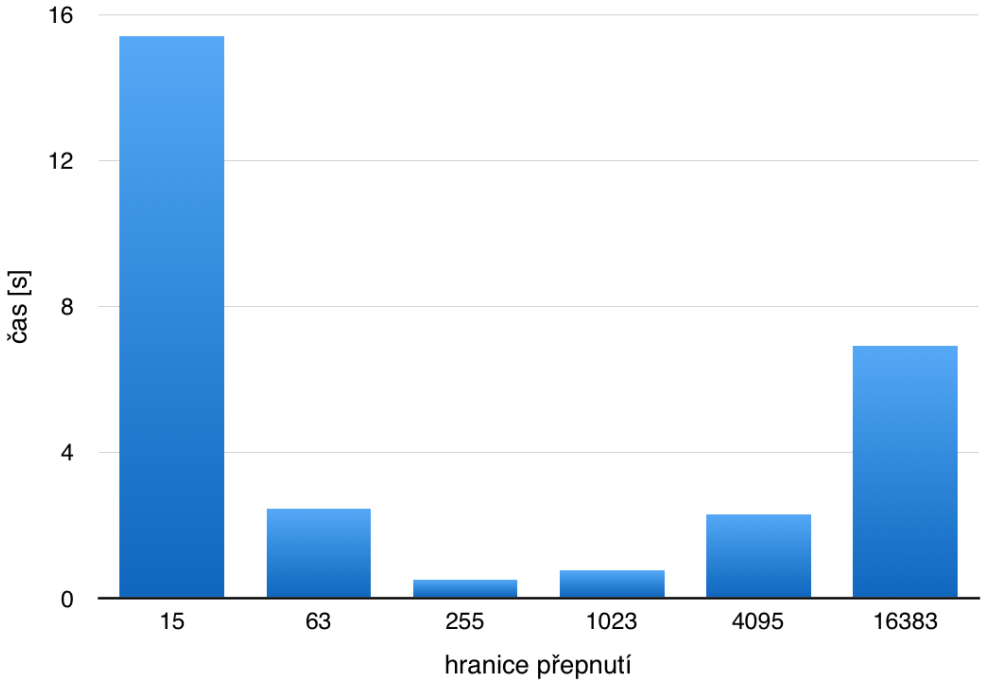


Obrázek 4.3: Časy pro různé hranice přepnutí Toom-3 algoritmu pro polynomy o stupni 1 milionu

Z grafu 4.3 lze zjistit, že hranice přepnutí vychází nejlépe při stupni 242.

4.4.3 Toom-4 algoritmus

Vzhledem k podstatě Toom-4 algoritmu bude hranice přepnutí ve tvaru $4^l - 1$, kde $l \in \mathbb{N}_0$.



Obrázek 4.4: Časy pro různé hranice přepnutí Toom-4 algoritmu pro polynomy o stupni 1 milionu

Z grafu 4.4 lze zjistit, že hranice přepnutí vychází nejlépe při stupni 255.

4.5 Přesnost algoritmů

Tato podkapitola se věnuje měření přesnosti výsledků implementovaných algoritmů. U algoritmů Karatsuba, Toom-3 a Toom-4 je zkoumána průměrná a maximální odchylka výsledků oproti triviálnímu algoritmu. Přesnost je měřena pro různě velké stupně vstupních polynomů a pro různé maximální hodnoty koeficientů vstupních polynomů.

4.5.1 Karatsubův algoritmus

Tabulka 4.1: Průměrné a maximální hodnoty odchylek Karatsubova algoritmu oproti triviálnímu algoritmu. Sloupce reprezentují maximální hodnotu koeficientů výsledných polynomů, řádky stupně polynomů.

Kar.	$10^3 \cdot 10^3 = 10^6$		$10^6 \cdot 10^6 = 10^{12}$		$10^9 \cdot 10^9 = 10^{18}$	
	avg	**max**	**avg**	**max**	**avg**	**max**
1 000	$84 * 10^{-9}$	$86 * 10^{-8}$	$81 * 10^{-3}$	$96 * 10^{-2}$	$91 * 10^{3}$	$13 * 10^{5}$
10 000	$41 * 10^{-7}$	$16 * 10^{-5}$	$40 * 10^{-1}$	$14 * 10^{1}$	$40 * 10^{5}$	$18 * 10^{7}$
100 000	$13 * 10^{-5}$	$12 * 10^{-3}$	$13 * 10^{1}$	$19 * 10^{3}$	$31 * 10^{7}$	$15 * 10^{9}$

4.5.2 Toom-3 algoritmus

Tabulka 4.2: Průměrné a maximální hodnoty odchylek Toom-3 algoritmu oproti triviálnímu algoritmu. Sloupce reprezentují maximální hodnotu koeficientů výsledných polynomů, řádky stupně polynomů.

Toom-3	$10^3 \cdot 10^3 = 10^6$		$10^6 \cdot 10^6 = 10^{12}$		$10^9 \cdot 10^9 = 10^{18}$	
	avg	**max**	**avg**	**max**	**avg**	**max**
1 000	$88 * 10^{-9}$	$57 * 10^{-8}$	$85 * 10^{-3}$	$63 * 10^{-2}$	$94 * 10^{3}$	$79 * 10^{4}$
10 000	$25 * 10^{-7}$	$21 * 10^{-6}$	$24 * 10^{-1}$	$23 * 10^{0}$	$24 * 10^{5}$	$22 * 10^{6}$
100 000	$74 * 10^{-6}$	$10 * 10^{-4}$	$73 * 10^{0}$	$86 * 10^{1}$	$73 * 10^{6}$	$95 * 10^{7}$

4.5.3 Toom-4 algoritmus

Tabulka 4.3: Průměrné a maximální hodnoty odchylek Toom-4 algoritmu oproti triviálnímu algoritmu. Sloupce reprezentují maximální hodnotu koeficientů výsledných polynomů, řádky stupně polynomů.

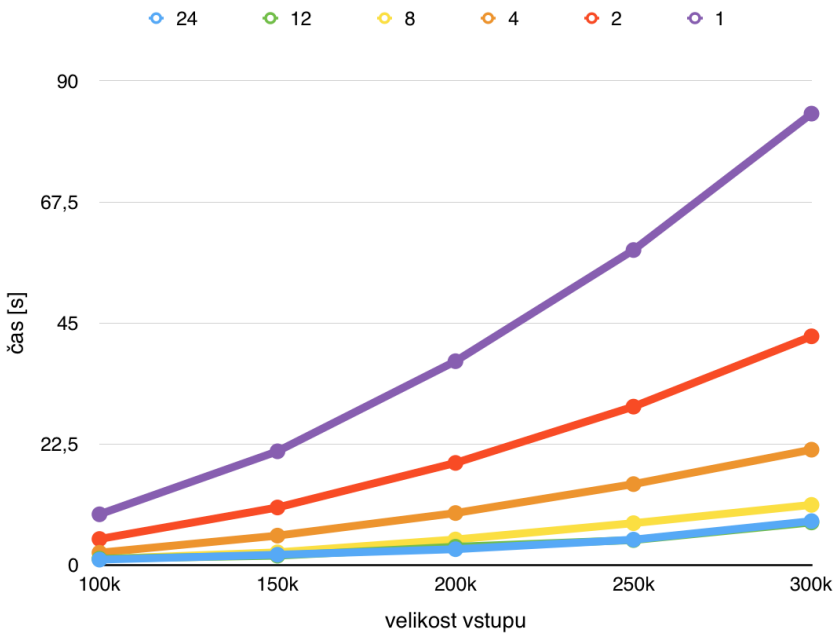
Toom-4	$10^3 \cdot 10^3 = 10^6$		$10^6 \cdot 10^6 = 10^{12}$		$10^9 \cdot 10^9 = 10^{18}$	
	avg	**max**	**avg**	**max**	**avg**	**max**
1 000	$17 * 10^{-8}$	$19 * 10^{-7}$	$19 * 10^{-2}$	$23 * 10^{-1}$	$19 * 10^{4}$	$26 * 10^{5}$
10 000	$37 * 10^{-6}$	$82 * 10^{-5}$	$37 * 10^{0}$	$82 * 10^{1}$	$35 * 10^{6}$	$76 * 10^{7}$
100 000	$50 * 10^{-4}$	$36 * 10^{-2}$	$52 * 10^{2}$	$31 * 10^{4}$	$53 * 10^{8}$	$38 * 10^{9}$

4.5.4 Zhodnocení přesnosti

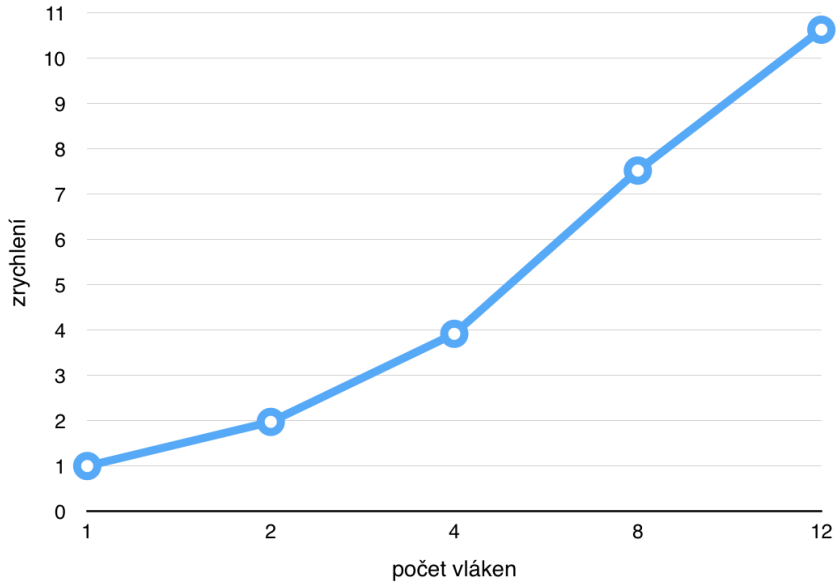
Z tabulek 4.1, 4.2 a 4.3 lze zjistit, že průměrná odchylka algoritmů Karatsuba a Toom-3 je řádově velice podobná. Oproti tomu rychlejší algoritmus Toom-4 má průměrnou odchylku zpravidla o řád horší.

Dále lze pozorovat, že pokud se zvýší řád maximálních hodnot koeficientů vstupních polynomů, tak se průměrná odchylka řádově posune, ale to je zapříčiněno způsobem, jakým datový typ `double` ukládá hodnoty. Není to tedy způsobeno tím, že by algoritmus byl nepřesnější. Naopak stupeň polynomů vliv na přesnost má, jelikož algoritmy při vyšším stupni provedou více operací.

4.6 Triviální algoritmus

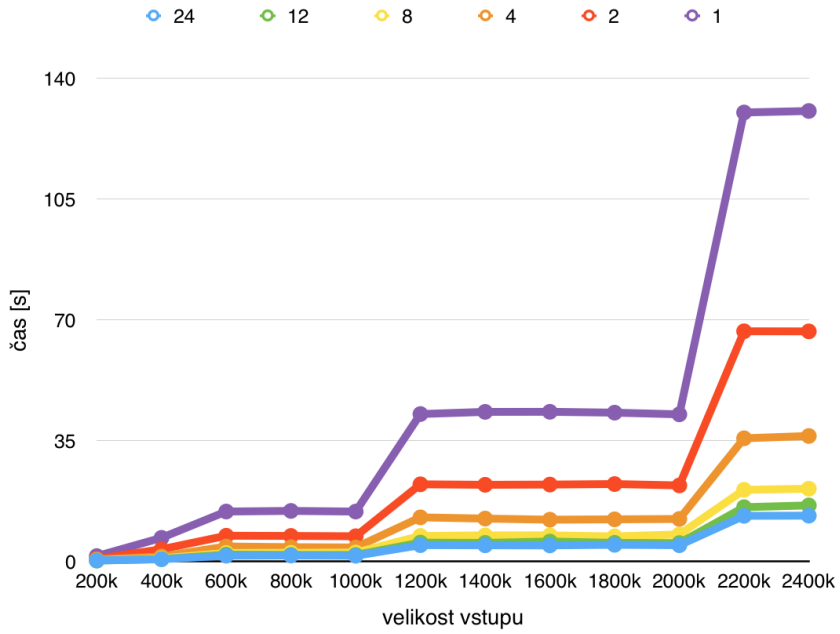


Obrázek 4.5: Měření triviálního algoritmu pro různý počet vláken. Velikost vstupu je uvedena v tisících

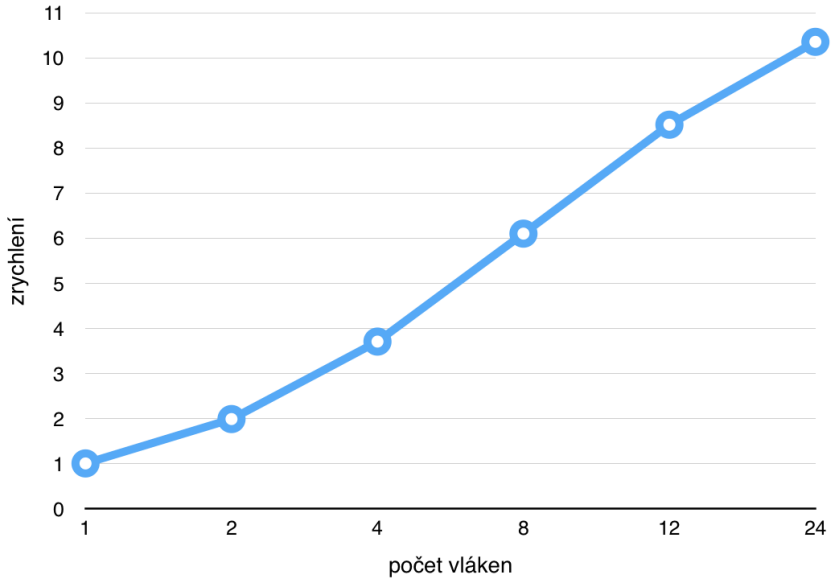


Obrázek 4.6: Zrychlení triviálního algoritmu

4.7 Karatsubův algoritmus

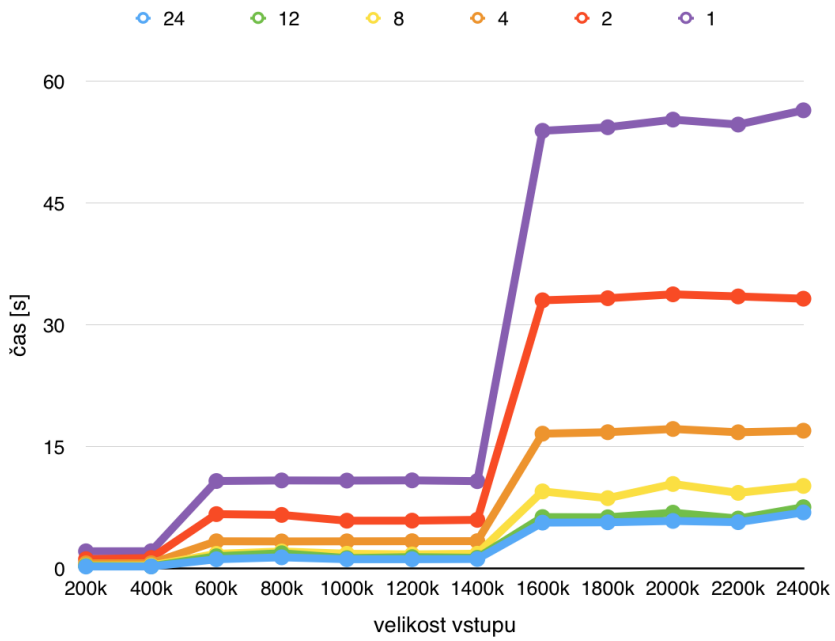


Obrázek 4.7: Měření Karatsubova algoritmu pro různý počet vláken. Velikost vstupu je uvedena v tisících

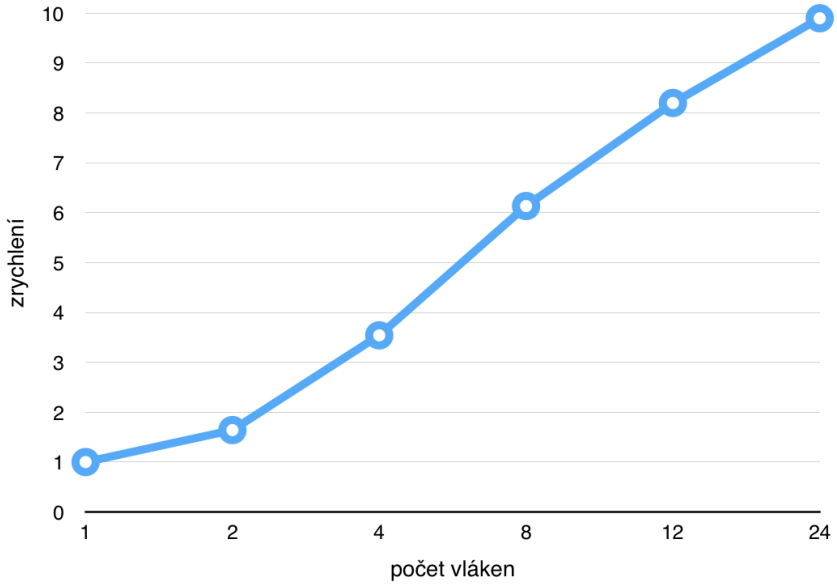


Obrázek 4.8: Zrychlení Karatsubova algoritmu

4.8 Toom-3 algoritmus

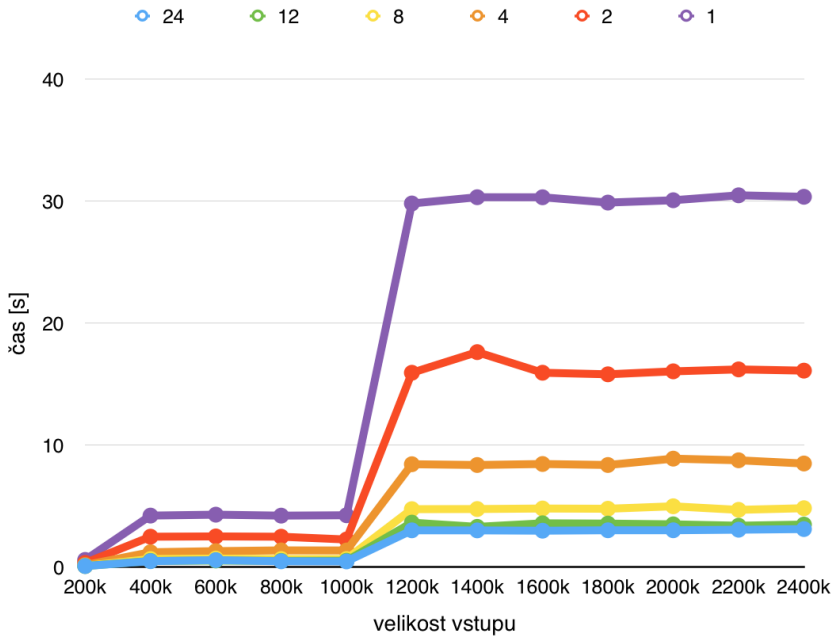


Obrázek 4.9: Měření Toom-3 algoritmu pro různý počet vláken. Velikost vstupu je uvedena v tisících

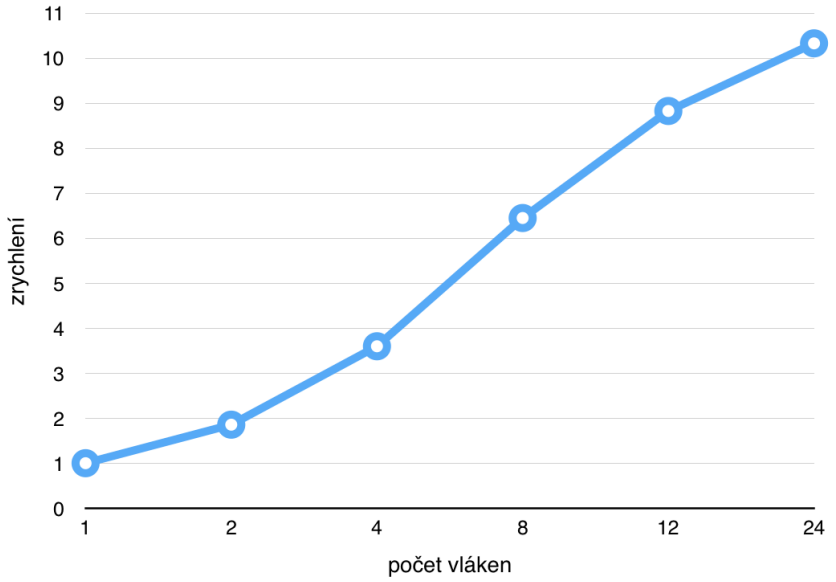


Obrázek 4.10: Zrychlení Toom-3 algoritmu

4.9 Toom-4 algoritmus



Obrázek 4.11: Měření Toom-4 algoritmu pro různý počet vláken. Velikost vstupu je uvedena v tisících



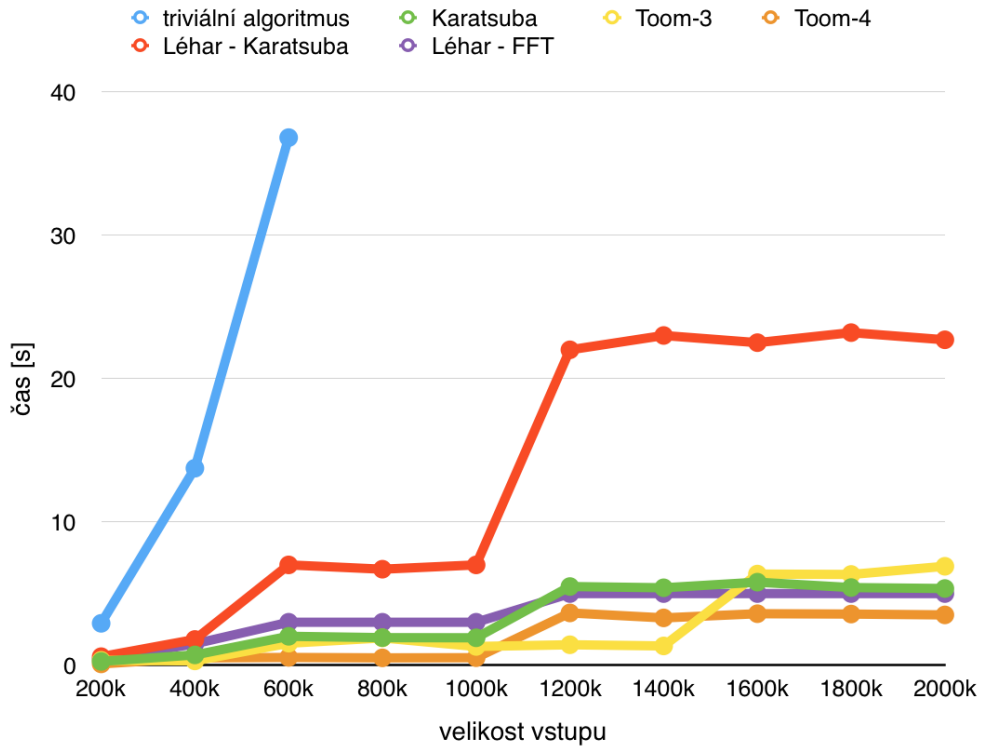
Obrázek 4.12: Zrychlení Toom-4 algoritmu

4.10 Porovnání s existujícími implementacemi

V této podkapitole jsou implementované algoritmy porovnány s existujícími implementacemi a knihovnami.

4.10.1 Bakalářská práce Adama Léhara

Adam Léhar ve své bakalářské práci v roce 2014 zpracoval pro součin polynomů Karatsubův algoritus a Rychlou Fourierovu transformaci. Tyto algoritmy optimalizoval a následně testoval na stejném výpočetním serveru `Star` [12].



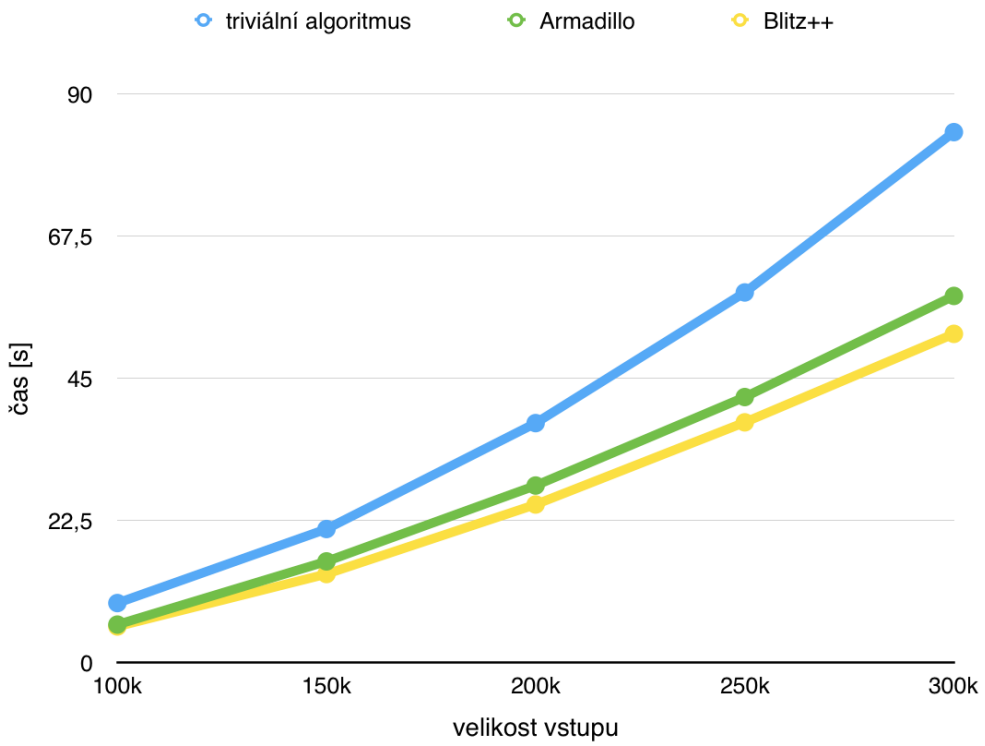
Obrázek 4.13: Porovnání implementovaných algoritmů s algoritmem Karatsuba a Rychlé Fourierovy transformace od Adama Léhara při využití 12 vláken

Z grafu 4.14 lze vidět, že Karatsubův algoritmus od Adama Léhara je časově náročnější než v implementace z této práce. To může být způsobeno špatným nastavením kompilátoru nebo méně efektivní implementací. Naopak Rychlá Fourierová transformace, která má asymptotickou složitost $\mathcal{O}(n \log n)$ a je tedy z těchto algoritmů asymptoticky nejrychlejší, dosahuje dobrých vý-

sledků a při ještě vyšších stupních polynomů by se mohl stát z těchto algoritmů nejrychlejším.

4.10.2 Další implementace

Pro další porovnání implementovaných algoritmů byly vybrány knihovny Armadillo a Blitz++. Tyto knihovny nepodporují paralelní výpočet, srovnání je tedy provedeno pouze pro sekvenční algoritmy.



Obrázek 4.14: Porovnání implementovaného triviálního algoritmu s knihovnami Armadillo a Blitz++

Z grafu 4.14 lze vidět, že tyto knihovny nabízejí rychlejší výpočet než triviální algoritmus, ale proti dalším algoritmům by neobstály.

Závěr

Cílem této bakalářské práce byla efektivní implementace vybraných algoritmů pro násobení polynomů. Tento cíl byl splněn.

Jednotlivé algoritmy byly nejdříve popsány, následně byla provedena jejich implementace a sekvenční optimalizace pomocí transformací kódu či vektorizace. Na základě implementace algoritmů byla provedena analýza přesnosti algoritmů. Vyráznému zrychlení výpočtů pomohla paralelizace pomocí technologie OpenMP. Výpočty algoritmů tedy byly rozděleny do více vláken nad sdílenou pamětí.

V poslední části bylo provedeno testování a měření na výpočetním serveru. Implementované algoritmy byly porovnány mezi sebou a u jednotlivých algoritmů byl změřen vliv počtu vláken na čas výpočtu a zrychlení. Nakonec bylo provedeno porovnání implementovaných algoritmů s existujícími implementacemi a knihovnami.

Literatura

[1] Štampach, F.; Klouda, K.: BI-LIN - Přednáška č.2 - Polynomy. Březen 2014, [cit. 2016-04-15].

[2] Karatsuba, A. A.: The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, 1995: s. 169–183, [cit. 2016-04-16].

[3] Weisstein, E. W.: Karatsuba Multiplication [online]. [cit. 2016-04-16]. Dostupné z: `http://mathworld.wolfram.com/KaratsubaMultiplication.html`

[4] macro: Toom-Cook multiplication [online]. [cit. 2016-04-18]. Dostupné z: `http://everything2.com/title/Toom-Cook+multiplication`

[5] Šimeček, I.: BI-EIA - Technologie OpenMP. 2015, [cit. 2016-04-22].

[6] Green, R. W.: OpenMP* Loop Scheduling [online]. 2012, [cit. 2016-04-23]. Dostupné z: `https://software.intel.com/en-us/articles/openmp-loop-scheduling`

[7] Yliluoma, J.: Guide into OpenMP: Easy multithreading programming for C++ [online]. 2007, [cit. 2016-04-22]. Dostupné z: `http://bisqwit.iki.fi/story/howto/openmp/`

[8] Šimeček, I.; Šoch, M.: BI-EIA - Kompilátorové optimalizace I: Metody transformací zdrojových kódů. 2015, [cit. 2016-04-24].

[9] Šimeček, I.: BI-EIA - Použití vektorizace v C/C++ (GCC). 2015, [cit. 2016-04-24].

[10] Šimeček, I.: BI-EIA - Úvod do paralelního počítání. 2015, [cit. 2016-04-25].

[11] Bodrato, M.: Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. 2007, [cit. 2016-04-28].

[12] Léhar, A.: *Rychlé násobení smíšených polynomů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014, [cit. 2016-05-06].

Příloha A

Seznam použitých zkratek

SIMD Single instruction, multiple data

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src` ..... zdrojové kódy implementace
- `text` ..... text práce
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `BP_Holub_Jakub_2016.pdf` ..... text práce ve formátu PDF