VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

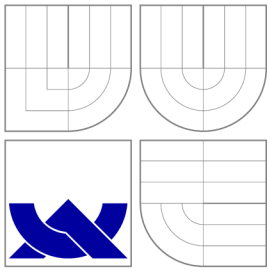
PARALELIZACE V JAZYCE RUST

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

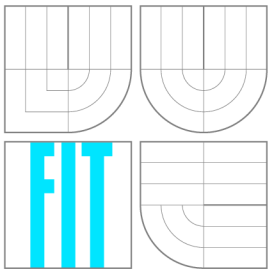
AUTOR PRÁCE
AUTHOR

ONDŘEJ ŠLAMPA

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PARALELIZACE V JAZYCE RUST

PARALLEL PROGRAMMING IN RUST LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ ŠLAMPA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JOZEF KOBRTEK
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zabývá paralelizací v jazyce Rust. Cílem této práce je zhodnotit výkon a použitelnost jazyka Rust pro tvorbu paralelních aplikací ve srovnání s již používanou alternativou - OpenMP. Toto porovnání bylo provedeno na výpočtu n-rozměrné konvoluce. V závěru se nachází zhodnocení výsledků a návrhy pro jejich další využití.

Abstract

Topic of this thesis is parallelization in Rust. Aim of this thesis is to compare performance and usability of Rust language with already used alternative - OpenMP. Computation of n-dimensional comvolution was used for benchmark. In conclusion there is evaluation of results and suggestions for their future use.

Klíčová slova

Rust, OpenMP, paralelizace, paralelní programování, konvoluce.

Keywords

Rust, OpenMP, paralelization, parallel programing, convolution.

Citace

Ondřej Šlampa: Paralelizace v jazyce Rust, bakalářská práce, Brno, FIT VUT v Brně, 2014

Paralelizace v jazyce Rust

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jozefa Kobrtka.

.....
Ondřej Šlampa
21. května 2014

Poděkování

Chtěl bych poděkovat panu Ing. Jozefu Kobrtkovi za přínosné a pravidelné konzultace, odborné rady a bezproblémovou komunikaci.

Obsah

Kapitola 1

Úvod

Během posledních 60 let se poměr mezi cenou výkonem procesorů zlepšil přibližně 100 miliardakrát. Většinu předchozích dvou desetiletí využili hardwarový architekti tento pokrok ke zvyšování rychlosti tranzistorů. Nevýhodou tohoto přístupu je zvyšování spotřeby energie a teploty. Trendem posledních let je tvorba procesorů obsahujících několik jader, kde různé jádra provádějí současně různé výpočty. Vícejádrové procesory sice vyřešily problémy s výkonem, ale přinášejí také komplikace. Tou hlavní je bezpečná paralelizace programů.

Cílem této práce je porovnámí paralelizace v jazyce Rust a API OpenMP. Pro srovnání jsou použity algoritmy pro výpočet n-rozměrné konvoluce. Oba způsoby paralelizace budou porovnány podle výkonu a podle programátorské přívětivosti.

Druhá kapitola této práce se zabývá popisem technologií pro tvorbu paraleních programů: jazykem Rust a API OpenMP. Třetí kapitola definuje matematickou operaci konvoluce a způsoby jejího výpočtu. Čtvrtá kapitola se věnuje implementacím metod ze třetí kapitoly pomocí technologií popsaných v druhé kapitole. Pátá kapitola popisuje, jakým způsobem byly změřeny výsledky používané v následující kapitole. Šestá kapitola porovnává implementace podle několika kritérií. V závěru se nachází zhodnocení výsledků a návrhy pro jejich další využití.

Kapitola 2

Přístupy k paralelizaci

Tato kapitola je věnována popisu technologií, použitých v této práci. V první podkapitole je popsána historie a vlastnosti programovacího jazyka Rust. Důraz je kladen hlavně na výhody, které tyto vlastnosti přinášejí, oproti jazyku C a způsoby meziprocesorové komunikace které Rust podporuje. Následující podkapitola se věnuje historii a použití API OpenMP při paralelním programování v jazyce C.

2.1 Jazyk Rust

Rust je staticky typovaný programovací jazyk se zaměřením na bezpečnost práce s pamětí, paralelní programování a výkon. Podporuje čistě funkcionální, procedurální i objektově orientovaný styl programování. Syntaxe a sémantika Rustu je inspirována syntaxí a sémantikou jazyka C++.[2] Autorem jazyka je Graydon Hoare, který začal na Rustu pracovat v roce 2006. O tři roky později se k vývoji přidal jeho zaměstnavatel — Mozilla a v roce 2012 byla vydána první verze překladače.[15]

2.1.1 Základní konstrukce jazyka Rust

Jazyk Rust obsahuje několik typů proměnných, které je možné rozdělit do tří skupin na primitivní, textové a složené datové typy. Proměnné jsou definovány pomocí klíčového slova `let` a jsou implicitně neměnitelné. Měnitelné proměnné jsou definovány pomocí dvojice klíčových slov `let mut`. Explicitní určení datového typu je volitelné, pokud není uveden, tak je odvozen z hodnoty, která je proměnné přiřazena.

Rust obsahuje deset primitivních datových typů, pět znaménkových a pět bezznaménkových, které představují celá čísla: `int`, `i8`, `i16`, `i32`, `i64`, `uint`, `u8`, `u16`, `u32`, `u64`. Čísla s desetinou čárkou představují datové typy `f32` a `f64`. Zbývající primitivní datové typy jsou `bool` a `()`, který nabývá pouze hodnoty `()` a používá se jako implicitní typ návratové hodnoty funkcí.

Textové datové typy jsou dva a to `char`, představující jeden znak textu, a `str`, řetězec znaků. Typ `char` je zapisován pomocí jednoduchých uvozovek a `str` pomocí složených uvozovek.

Složené datové typy jazyka Rust jsou struktura, vektor a entice. Pro neměnitelné složené datové typy platí, že všechny jejich části jsou také neměnitelné, a pro měnitelné složené datové typy platí, že všechny jejich části jsou také měnitelné. Není tak možné vytvořit strukturu, která by byla částečné měnitelná a částečně neměnitelná. Struktury Rustu jsou velmi podobné strukturám jazyka C. Struktury jsou definovány pomocí klíčového slova `struct`

```
fn main(){
    let a=10i;                  //datový typ int
    let b:int=10;               //datový typ int
    let c="Já jsem řetězec.";   //datový typ str
    let mut d=1.4;              //datový typ f32
    let vector=[1,2,3];         //vektor obsahující čísla typu int
    let tuple=(4, 5.3);         //entice
}
```

Ukázka kódu 2.1: Definice proměnných

```
    struct MyStruct{
        value:int,
        string:~str,
        real:f64
    }
```

Ukázka kódu 2.2: Definice struktury `MyStruct`

následovaného jménem struktury a seznamem prvků struktury. Seznam prvků struktury se zapisuje do složených závorek a je tvořen definicí prvků oddělených čárkou. Každý prvek struktury je definován pomocí svého jména a datového typu a ty jsou odděleny dvoutečkou. Definice struktury je zobrazena v ukázce kódu 2.2. Typ vektor je podobný typu pole z jazyka C, narozdíl od něj ale obsahuje kontrolu indexů. Při pokusu o přístup mimo vektor je vyvolána chyba programu a nedojde tak k porušení paměti[1]. Entice jsou bezejmené struktury složené z bezejmených prvků a jsou značeny pomocí závorek.[3] Ukázka kódu 2.1 zobrazuje definice proměnných různých datových typů.

Definice funkcí začínají klíčovým slovem `fn`, obsahují vstupní proměnné a jejich typy oddělené dvoutečkou. Pokud funkce vrací hodnotu je její datový typ uveden v hlavičce funkce za operátorem `->`. Funkce v ukázce kódu 2.3 provede součet dvou celých čísel a vrátí výsledek. Podobně jako v jazyce C vstupním bodem programu je funkce `main`.[3]

Rust, stejně jako jazyk C, obsahuje větvení `if` a cyklus `while`. Druhým cyklem je nekonečný cyklus `loop`. Funkci konstrukce `switch` z jazyka C plní v Rustu `match`. Novou konstrukcí je `spawn`, který slouží k tvorbě nových vláken. Při vytvoření nového vlákna je vlastnictví paměti (viz. 2.1.2) využívané v tomto vlákně přesunuto do tohoto vlákna. Když vlákno dojde na konec konstrukce `spawn` je ukončeno a odstraněno z paměti.[3]

2.1.2 Správa paměti

Rust nabýzí několik způsobů zprávy paměti: počítání referencí, garbage collector a chytré ukazatele, které jsou součástí syntaxe jazyka a jsou doporučeným způsobem správy paměti.[14]

```
fn sum(a:int, b:int)->int{
    let c=a+b;
    return c;
}
```

Ukázka kódu 2.3: Definice funkce

[1] Segmentation fault

```
fn main(){
    spawn(proc(){
        println("Vedlejší vlákno.");
    })
    println("Hlavní vlákno.");
}
```

Ukázka kódu 2.4: Tvorba vláken

```
fn is_empty(list:&[int])->bool{
    return list.len()==0;
}
fn main(){
    let list=~[1,2,3];
    if is_empty(list){
        println("Seznam je prázdný.");
    }
}
```

Ukázka kódu 2.5: Přetypování ukazatelů

Chytré ukazatele

Jazyk Rust neumožňuje vytvořit ukazatel, který nikam neukazuje, tzv. null poiter, z toho plyne, že každý ukazatel musí ukazovat na platný úsek paměti nebo být označen jako ukazatel s přesunutým vlastnictvím[2].

Vlastnící ukazatel[3], dále jen ukazatel, je ukazatel, který "vlastní"danou paměť. To znamená, že na tuto paměť ukazuje právě jeden ukazatel a když pozbude platnosti, tak je tato paměť uvolněna. Tento ukazatel může předat vlastnění paměti jinému ukazatelu, pokud to udělá, je označen, že jeho hodnota byla přesunuta[4]. Problémy, které jsou způsobeny přesouvání vlastnění paměti, je možné odhalit za překladu. Ve zdrojovém kódu je značen tyldou před hodnotou nebo datovým typem.

Reference je ukazatel, který ukazuje na paměť vlastněnou jiným ukazatelem a který se nestará o její uvolnění. Používá se hlavně při volání funkcí, kdy se ve volající funkci předává vlastnící ukazatel, ale ve volané funkci se přijímá reference. Díky tomuto přetypování je možné se vyhnout předání vlastnění při volání funkcí. Toto demonstruje ukázka kódu 2.5.[3]

2.1.3 Zasílání zpráv

Je nejčastěji používaný způsob synchronizace několika procesů. Je založený na tom, že procesy si mezi sebou posílají zprávy. Výhodou zasílání zpráv je to, že programátor nepracuje přímo se sdílenou pamětí a díky tomu se předchází problémům, které při této práci vznikají. Nevýhodou jsou vyšší časové a paměťové nároky při posílání větších zpráv a nadbytečné kopírování dat v rámci jednoho programu, protože každý proces má své vlastní kopie proměnných.

Pro zasílání zpráv se v Rustu používají kanály. Každý kanál se skládá ze struktury `Sender`, která slouží k odesílání zpráv, a `Receiver`, která přijímá zprávy. `Sender` může

[2] ownership
[3] owned pointer
[4] moved

```
fn main(){
    let (sender, receiver):(Sender<int>, Receiver<int>)=std::comm::channel();
    spawn(proc(){
        sender.send(12);
    })
    println!("Obdržel jsem {}.", receiver.recv());
}
```

Ukázka kódu 2.6: Kanál

```
fn main(){
    let shared="Sdílený řetězec";
    let arc=Arc::new(shared);
    spawn(proc(){
        println!("{}", arc.get());
    })
}
```

Ukázka kódu 2.7: Sdílená paměť

být nakopírován a používán v několika různých procesech. Kanál má určený datový typ a může přenášet pouze data tohoto typu. Nový kanál je vytvořen pomocí funkce `channel` v modulu `std::comm`. Ukázka kódu 2.6 demonstruje použití kanálu v programu, kde jedno vlákno pošle číslo druhému vláknu.[3]

2.1.4 Sdílená paměť

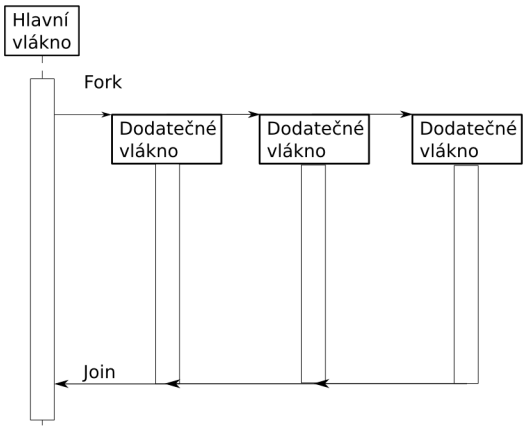
Rust podporuje i synchronizaci procesů na základě sdílené paměti. Pro sdílení dat se používá struktura `Arc` (Atomically Reference Counted wrapper), která je jen referencí na sdílená data. Při vytváření procesů se tak kopírují jen reference na sdílená data a sdílená data jsou tak v systému jen jednou. Tímto způsobem je možné sdílet jen neměnitelná data. [3]

2.2 OpenMP

OpenMP je API pro tvorbu paralelních programů v jazycích C, C++ a Fortran založených na sdílení paměti. Skládá se z direktiv pro překladač, knihovny funkcí a proměnných prostředí.[11] Počátky OpenMP sahají do 80. let 20. století ke standardu ANSI X3H5. Technické limitace a vzrůst popularity programovacích modelů založených na oddělené paměti způsonili ukončení dalšího vývoje. Vznik OpenMP byl motivován snahou vývojářů o vytvoření standardu pro tvorbu paralelních programů, který by zajišťoval přenositelnost kódu mezi podporovatelnými platformami. Standard ANSI X3H5 byl použit jako základ, na kterém byla vytvořena první specifikace API - OpenMP 1.0. Verze pro Fortran vyšla v říjnu 1997 a verze pro C/C++ o rok později. Nejnovější verze (v době psaní této práce) je OpenMP 4.0, která vyšla v červenci 2013.[7]

2.2.1 Programovací model

OpenMP podporuje programování na principu fork-join. Program začíná jako jedno hlavní vlákno. Na začátku paralelní části programu vytvoří hlavní vlákno dodatečná vlákna po-



Obrázek 2.1: Paralelní část programu v OpenMP.

mocí rozdvojení[5]. Všechna vlákna tvoří tým vláken, které budou vykonávat paralelní část programu. Na konci paralelní části programu vlákna čekají na ostatní vlákna. Když všechna vlákna dojdou na konec paralelní části programu, sloučí[6] se do jednoho hlavního vlákna. Toto může být provedeno několikrát během programu. Počet dodatečných vláken může být určen OpenMP nebo zadán programátorem.[8]

2.2.2 Základní konstrukce

V jazyce C se konstrukce OpenMP zapisují pomocí implementačních direktiv, které se značí pomocí `#pragma omp` a příslušných klauzulí. Výhodou tohoto přístupu je to, že pokud překladač direktivám nerozumí, tak je ignoruje. Díky tomu je možné přeložit programy využívající OpenMP i na překladačích, které OpenMP nepodporují. Paralelní část programu se značí direktivou `#pragma omp parallel` a je tvořena následujícím příkazem nebo sekvencí příkazů. Pro proměnné platí, že promněnné definované mimo paralelní část programu jsou sdílené všemi vlákny a proměnné definované v ní jsou soukromé pro každé vlákno.

Jednou ze základních konstrukcí OpenMP je paralelizace cyklu `for` pomocí direktivy `#pragma omp parallel for`. Díky této konstrukci může být prováděno několik smyček cyklu zároveň. Iterační proměnná je soukromá pro každé vlákno. Pokud má několik vláken přistupovat k jedné sdílené proměnné, měli by být tyto přístupy umístěny v kritické sekci. Ta se definuje direktivou `#pragma omp critical(name)`, kde `name` je jméno této sekce. OpenMP zajišťuje, že v danný okamžik nemůžou být vykonávány dvě kritické sekce se stejným jménem. Počet vláken používaných v paralelní sekci programu je možné nastavit pomocí direktivy `num_threads(number_of_threads)`, kde `number_of_threads` je požadovaný počet vláken.[7] Ukázka těchto direktiv je v ukázce kódu 2.8.

Další kontrukcí OpenMP je paralelizace sekcí programu pomocí direktiv `#pragma omp parallel sections` a `#pragma omp section`. Tyto direktivy rozdělují sekvenci příkazů na sekce, které je možné provádět odděleně v různých vláknech a v různém pořadí. Klauzuje `if(podminka)` přidaná k direktivě obsahující `parallel` způsobí, že paralelizace bude provedena pouze pokud platí podmínka.[7] Tyto direktivy jsou demonstrovány v ukázce kódu 2.9.

[5] fork
[6] join

```
int* array=malloc(10*sizeof(10));
#pragma omp parallel for num_threads(4)
for(int i=0; i<10; i++){
    #pragma omp critical(array_update)
    array[i]=i;
}
```

Ukázka kódu 2.8: Direktivy `for` a `critical`

```
#pragma omp parallel sections if(use_threads)
{
    printf("První sekce");
    #pragma omp section
    printf("Druhá sekce");
}
```

Ukázka kódu 2.9: Direktivy `sections`, `section` a `if`

Kapitola 3

Diskrétní konvoluce

Tato kapitola je věnována teoretickým základům, které jsou nutné pro porozumění této práce. V první podkapitole definuji pojem diskrétní konvoluce, následují dvě podkapitoly se věnují způsobům výpočtu konvoluce a poslední podkapitola popisuje jak převést konvoluci n-rozměrných signálů na konvoluci jedrozměrných signálů.

3.1 Definice

Diskrétní konvoluce je matematická operace nad dvěma diskrétními signály x a h definovaná vzorcem 3.1. Jejím výsledkem je diskrétní signál y.

$$y[n] = (x * h)[n] = \sum_{i=-\infty}^{\infty} x[n-i]h[i] \tag{3.1}$$

Problémem předchozí definice je to, že pracuje nad signály nekonečné délky. Vzorec 3.2, kde m je délka signálu h, vznikl upravením vzorce 3.1 pro signály o konečné délce. [12]

$$y[n] = (x * h)[n] = \sum_{i=0}^{m-1} x[n-i]h[i] \tag{3.2}$$

3.2 Metoda overlap-save

Metoda overlap-save je způsob výpočtu konvoluce signálu x a jádra h o délce m. Výsledný signál y je rozdělen na segmenty o délce l, které jsou vypočteny samostatně a následně konkatenovány. Každý segment začíná na $kl + m - 1$ prvku výstupního signálu, kde k je index segmentu. Ve vzorci 3.3 definuji signál x_k, který je ktý segment vstupního signálu x. Vzorec 3.4 je definicí signálu y_k, který je ktý segment výstupního signálu y.

$$x_k[n] = \begin{cases} x[kl+n] & 0 \leq n < l+m-1 \\ 0 & \text{ostatní} \end{cases} \tag{3.3}$$

$$y_k[n] = (x_k * h)[n] \tag{3.4}$$

Poté pro $kl+m-1 \leq n < kl+l+m-1$ (obsah ktého segmentu) a $m-1 \leq n-kl < l+m-1$ můžeme napsat rovnici 3.5.[13]

$$y[n] = \sum_{i=0}^{m-1} h[i] \cdot x[n-i] = \sum_{i=0}^{m-1} h[i] \cdot x_k[n-kl-i] = (x_k * h)[n-kl] = y_k[n-kl] \quad (3.5)$$

3.3 Konvoluční teorém

Konvoluční teorém je metoda výpočtu konvoluce a říká, že Fourierova transformace konvoluce se rovná skalárnímu součinu Fourierových transformací operandů konvoluce. Jinými slovy můžeme říct, že konvoluce v časové oblasti se rovná skalárnímu součinu ve frekvenční oblasti. Toto je zapsáno rovnicí 3.6.

$$\mathcal{F}\{x * h\} = \mathcal{F}\{x\} \cdot \mathcal{F}\{h\} \quad (3.6)$$

Aplikováním inverzní Fourierovy transformace na obě strany rovnice 3.6 dostaneme rovnici 3.7, kterou je možné použít na výpočet kovoluce.[9]

$$x * h = \mathcal{F}^{-1}\{\mathcal{F}\{x\} \cdot \mathcal{F}\{h\}\} \quad (3.7)$$

3.4 Převod dvourozměrné konvoluce na jednorozměrnou

Nechť $X_{M\times N}$ a $Y_{K\times L}$ jsou dvourozměrné signály, kde spodní index značí velikost signálu. Konvoluce těcht dvou signálů je signál $Z_{M+K-1\times N+L-1}$, získaný pomocí rovnice 3.8 pro $0 \leq i < M+K-1$ a $0 \leq j < N+L-1$.

$$Z(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m,n)Y(i-m, j-n) \quad (3.8)$$

Nechť $X'_{(M+K-1)\times(N+L-1)}$ a $Y'_{(M+K-1)\times(N+L-1)}$ jsou dvourozměrné signály, které vznikly rozšířením signálů $X_{M\times N}$ a $Y_{K\times L}$ podle rovnice 3.9.

$$A'_{(M+K-1)\times(N+L-1)} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-1} & a_{0,N} & 0 & \cdots & 0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-1} & a_{1,N} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{M-1,0} & a_{M-1,1} & \cdots & a_{M-1,N-1} & a_{M-1,N} & 0 & \cdots & 0 \\ a_{M,0} & a_{M,1} & \cdots & a_{M,N-1} & a_{M,N} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \quad (3.9)$$

Nechť X'' a Y'' jsou jednorozměrné signály, které vznikly ze signálů X' a Y' konkatenací řádků, tak aby první a poslední vzorky signálů X'' a Y'' odpovídal prvním a posledním vzorkům signálů X a Y. Nadbytečné vzorky ze signálů X' a Y' se v signálech X'' a Y'' neobjeví. Délka signálu X'' je $(M-1)(N+L-1)+N$ a délka Y'' je $(K-1)(N+L-1)+L$. Konvolucí signálů X'' a Y'' vznikne signál Z' o délce $(N+L-1)(M+K-1)$ viz rovnice 3.10. Signály Z' a Z mají stejný počet vzorků, díky tomu je můžeme mezi sebou převést.[10]

$$
\begin{aligned}
l_{Z'} = l_{X''} + l_{Y''} - 1 = \\
(M-1)(N+L-1) + N + (K-1)(N+L-1) + L - 1 = \\
(N+L-1)(M-1+K-1) + N + L - 1 = \\
(N+L-1)(M-1+K-1+1) = \\
(N+L-1)(M+K-1)
\end{aligned}
\tag{3.10}
$$

3.5 Převod n-rozměrné konvoluce na jednorozměrnou

Zobecněním konvoluce dvourozměrných signálů získám konvoluci obecně n-rozměrných signálů. Algoritmus převodu je založený na myšlence, že každý n-rozměrný signál můžu považovat za pole (n-1)-rozměrných signálů. Na tyto signály je rekurzivně použit algoritmus převodu. Výsledné jednorozměrné signály jsou konkatenovány a doplněny o nuly tak, aby se délka doplněného signálu rovnala délce odpovídající části výsledného signálu.[10]

Kapitola 4

Implementace výpočtu

Tato kapitola se zabývá implementaci výpočtu diskrétní konvoluce pomocí různých algoritmů v jazyce Rust a v jazyce C pomocí OpenMP. První podkapitola obsahuje návrh výsledných aplikací. Následující podkapitoly se zabývají implementací jednotlivých metod výpočtu. Druhá podkapitola se věnuje naivní metodě výpočtu konvoluce, následující dvě podkapitoly se zabývají algoritmy v jazyce Rust, pátá podkapitola popisuje implementaci naivní metody pomocí OpenMP a poslední je věnována využití Fourierovy transformace.

4.1 Návrh

Výsledkem implementace jsou dva programy. Jeden vytvořený pomocí jazyka C a druhý pomocí jazyka Rust. Obě aplikace jsou zpouštěny pomocí příkazové řádky a neobsahují grafické uživatelské rozhraní. Obvyklý běh obou programů je možné popsat pomocí následujícího výčtu.

1. Analýza argumentů příkazové řádky.
2. Načtení vstupního signálu a kernelu.
3. Převod vstupního signálu a kernelu na jednorozměrné signály.
4. Výpočet konvoluce podle zvolené metody.
5. Převod jednorozměrného výstupního signálu na vícerozměrný.
6. Uložení výstupního signálu.

První program je napsám pomocí jazyka C podle standardu C99. Ke čtení a ukládání obrázků používá knihovnu DevIL.[5] Paralelizace tohoto programu je provedena pomocí API OpenMP. Pro výpočet rychlé Fourierovy transformace je použita knihovna FFTW.[6] Výpočet konvoluce implementuje pomocí tří metod: jednovláknové naivní metody, vícevláknové naivní metody a konvolučního teorému. Druhý program je napsán pomocí jazyka Rust ve verzi 0.10. Ke čtení a ukládání obrázků rovněž používá knihovnu DevIL.[5] Konvoluci dokáže spočítat pomocí tří metod: jednovláknové naivní metody, vícevláknové naivní metody a metody overlap-save. Kompletní popis rozhraní obou programů je v příloze C.

4.2 Naivní metoda

Nejjednodušším způsobem jak vypočítat konvoluci dvou signálů je naivní metoda, protože vychází z definice konvoluce podle vzorce 3.2, který je použit pro výpočet každého prvku výsledného signálu.

Při použití jen jednoho vlákna jsou všechny prvky výsledného signálu vypočteny sériově v jednom cyklu.

4.2.1 Paralelní naivní metoda v jazyce Rust

Tato metoda výchází z předchozího způsobu výpočtu, ale pro výpočet prvků je použito několik pracovních vláken. Každý prvek výsledného signálu je vypočítán samostatně v jednom z těchto vláken. Pro každé dva prvky tak platí, že můžou být vypočteny v různých vláknech.

Výpočet v hlavním vlákně se skládá ze čtyř částí: vytvoření pracovních vláken, odeslání zadání práce, odeslání příkazů k ukončení a sběru výsledků práce. Při tvorbě pracovního vlákna je do něj vložena reference (viz. 2.1.4) na vstupní signál a jádro. Pracovní vlákno přijímá zadání práce od hlavního vlákna a vykonává je. Pro komunikaci mezi hlavním vláknem a pracovními vlákny se používají dva kanály. Jeden pro zadávání práce pracovním vláknům a druhý pro sbírání výsledků práce hlavním vláknem. Zadání práce je dvojce (A, I), kde A je akce, kterou má pracovní vlákno provést, a I je index prvku výsledného vlákna. Akce A může nabývat dvou hodnot `Compute` a `Terminate`. První značí, že vlákno má vypočítat Itý prvek výstupního signálu, druhá ukončí vlákno. Výsledek práce je dvojce (I, R), kde I je index prvku výsledného signálu a R hodnota tohoto prvku. Tento algoritmus je popsán pomocí sekvenčního diagramu v příloze D.2.

4.3 Metoda overlap-save v jazyce Rust

Výhodou této metody je omezení komunikace mezi vlákny, protože počet segmentů se rovná počtu pracovních vláken a do pracovních vláken tak není potřeba posílat příkazy a data. Ty jsou do pracovních vláken vložena při jejich vytvoření.

Hlavní vlákno vytvoří zadaný počet pracovních vláken a každému přitom předá index segmentu výstupního signálu, který má spočítat, odpovídající segment vstupního signálu a refenci na kernel. Poté přijme segmenty výstupního signálu od pracovních vláken. Pracovní vlákno vypočítá segment výstupního signálu, odešle ho hlavnímu vláknu a ukončí se. Pro komunikaci mezi hlavním vláknem a pracovními vlákny se používá jeden kanál, který je využit ke sbírání segmentů výstupního signálu. Každý segment výstupního signálu je poslán jako dvojce (I, S), kde I je index segmentu a S je obsah segmentu. Tento algoritmus je zobrazen na sekvenčního diagramu v příloze D.1.

4.4 Naivní metoda pomocí OpenMP

Tato metoda je paralelizací naivní metody pomocí OpenMP. Výhodou této metody je to, že se od neparalelní metody liší jen direktivami OpenMP.

Všechny prvky výsledného signálu jsou počítány v cyklu, který je paralelizován pomocí direktivy `#pragma omp parallel for if(parallel) num_threads(number_of_threads)`, kde `parallel` je proměnná typu `bool` a říká jestli má být cyklus paralelizován a kde

`number_of_threads` je počet vláken, které mají být využity pro výpočet. Přístup do výsledného signálu je ošetřen direktivou `#pragma omp critical`.

4.5 Výpočet pomocí konvolučního teorému

Implementace je provedena pomocí knihovny FFTW, která vypočítá rychlou Fourierovu a inverzní Fourierovu transformaci. Je to jedna z nejrychlejších knihoven pro výpočet Fourierovy transformace.[6] Kompletní postup výpočtu krok po kroku je uveden níže.

1. Převedu vstupní signál a kernel z datovéhu typu `double` na typ `fftw_complex`.
2. Transformuji vstupní signál a kernel pomocí Fourierovy transformace.
3. Provedu skalární násobení vstupního signálu a kernelu.
4. Výsledek násobení transformují pomocí inverzní Fourierovy transformace.
5. Převedu výstupní signál z datovéhu typu `fftw_complex` na typ `double`.

Kapitola 5

Měření výkonu

Tato kapitola popisuje jakým způsobem byl testovám výkon implementovaných metod. První podkapitola popisuje měřící skript a druhá výsledky, které byly získány pomocí tohoto skriptu.

5.1 Měřicí skript

Skript je napsaný v jazyce Python 3 a využívá program `time` pro měření času. Skládá z případů, kde případ je identifikován podle použitého jazyka, metody a počtu vláken. Případ je reprezentován třídou `Case`. Provedení měření se skládá ze tří částí: vytvoření seznamu případů, měření doby výpočtu jednotlivých případů a výpisu výsledků.

Vytvoření seznamu případů je možné ovlivnit pomocí argumentů příkazové řádky. Pokud nejsou zadány žádné jsou vytvořeny všechny možné případy pro 1 až 16 vláken. Pokud jsou zadány argumenty provedou pouze v nich definované případy. Argumenty `rust` a `c` říkají, přidá do seznamu všechny případy zadaného jazyka pro 1 až 16 vláken. Argument `<jazyk>:<metoda>` přidá do seznamu případy zadaného jazyka a metody pro 1 až 16 vláken. Argument `<jazyk>:<metoda>:n` přidá do seznamu případ se zadaným jazykem a metodou pro n vláken.

Měření doby výpočtu se provádí pro každý případ odděleně. Jako vstupní signál je použit soubor `lena.jpg` ze složky `inputs`. Kernely jsou načítány ze složky `kernels`. Pro každý případ je provedeno 100 měření na různých kernelech. Výsledné signály jsou uloženy do složky `outputs`. Ve výpisu výsledků jsou uvedeny aritmetické průměry všech měření.

5.2 Výsledky

Měření proběhlo na čtyřjádrovém procesoru Intel Core i5-3570K. Výsledky měření jsou rozděleny na dvě části tabulka 5.1 zobrazuje jednovláknové metody a tabulka 5.2 vícevláknové metody.

Metoda	Délka výpočtu (ms)
konvoluční teorém	25,4
jednovláknová naivní C	51,3
jednovláknová naivní Rust	71,5

Tabulka 5.1: Délky výpočtu jednovláknových metod

počet vláken	vícevláknová naivní v C (ms)	vícevláknová naivní v Rustu (ms)	overlap-save v Rustu (ms)
1	50,5	90,4	75,8
2	22,3	40,9	38,9
3	20,1	31,0	30,4
4	11,7	28,3	24,2
5	18,4	30,7	25,9
6	17,4	28,5	25,4
7	17,4	26,3	24,7
8	16,7	28,9	24,7
9	17,5	29,1	25,8
10	17,6	26,8	25,8
11	17,1	28,8	25,0
12	17,1	30,3	25,7
13	16,8	30,7	25,5
14	16,7	29,8	28,1
15	16,9	30,5	25,2
16	17,5	31,1	26,2

Tabulka 5.2: Délky výpočtu vícevláknových metod

Kapitola 6

Porovnání implementací

V této kapitole jsou porovnány implementace metod z kapitoly 4. První podkapitola obsahuje porovnání všech metod podle délky výpočtu. Druhá podkapitola se věnuje porovnání rychlosti implementací paralelních metod s náhledem k počtu vláken použitých při výpočtu. Třetí podkapitola je věnována porovnání metod podle počtu odeslaných zpráv. Poslední podkapitola porovnává implementace z pohledu programátora.

6.1 Podle délky výpočtu

Nejdůležitějším kritériem porovnávání implementací je délka výpočtu. Všechny implementace jsou porovnány vzhledem k referenční metodě využívající konvoluční teorém, protože k její implementaci byla využita profesionální knihovna FFTW. Tato metoda spočítala výsledný signál za 25,4 ms.

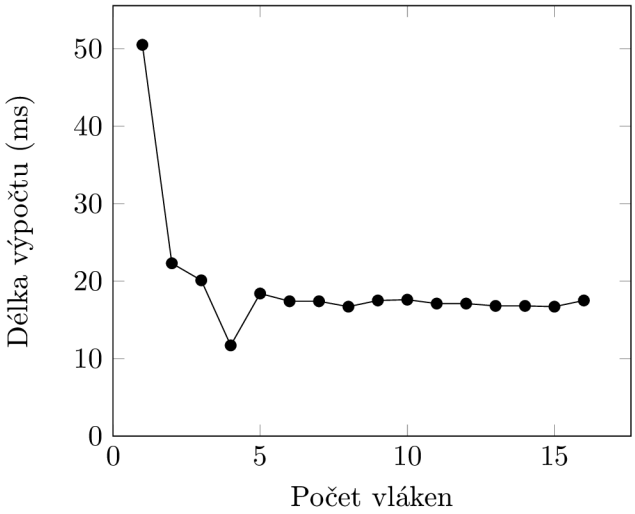
Nejrychlejší implementací byla vícevláknová metoda pomocí OpenMP, která provedla výpočet výsledného signálu za 11,7 ms, to je méně než polovična času, který potřebovala refenční metoda. Metoda overlap-save a vícevláknová naivní metoda v Rustu spočítaly signál za časy podobné referenční metodě. Délka výpočtu jednovláknových naivních metod je výrazně vyšší než refenční metody. Přesné výsledky jsou uvedeny v tabulce 6.1.

6.2 Podle počtu vláken

Rychlost paralelních metod je ovlivněna počtem jader procesosu a počtem vláken, které metoda používá pro výpočet. Referenční procesor obsahuje čtyři jádra. Počet vláken je proměnný a je předmětem této podkapitoly.

Metoda	Délka výpočtu (ms)
vícevláknová naivní v OpenMP	11,7
overlap-save Rust	24,2
konvoluční teorém	25,4
vícevláknová naivní Rust	26,3
jednovláknová naivní C	51,3
jednovláknová naivní Rust	71,5

Tabulka 6.1: Délky výpočtu



Obrázek 6.1: Délka výpočtu paralení naivní metody pomocí OpenMP v závisloti na počtu vláken.

Naivní metoda pomocí OpenMP dosahuje nejvyšší rychlosti při použití čtyř vláken, kdy je výsledný signál spočítán za 11,7 milisekund. Doba výpočtu je nejvyšší při použití jednoho vlákna a snižuje se s vyšším počtem vláken až do minima při využití čtyř procesů. Při použití pěti až šestnácti vláken kolísá mezi 16,7 a 18,4 ms, to je výrazně více než minimum. Závislost délky výpočtu a počtu vláken je zobrazena v grafu 6.1.

Naivní metoda v jazyce Rust vypočítá výsledný signál nejrychleji při použití sedmi vláken a to za 26,3 ms. Doba výpočtu je nejvyšší při použití jednoho vlákna a snižuje se s vyšším počtem vláken až do tří. Při použití tří až šestnácti vláken kolísá doba výpočtu mezi 26,3 a 31 ms, to jsou hodnoty mírně vyšší než minimum. Grafické znázornění je v grafu 6.2.

Výpočet pomocí metody Overlap-save byl nejrychlejší při použití čtyř vláken. Jeho délka byla 24,2 ms. Doba výpočtu je nejvyšší při použití jednoho vlákna a snižuje se s vyšším počtem vláken až do minima při použití čtyř vláken. Při využití pěti až šestnácti vláken se délka výpočtu pohybovala mezi 24,7 a 28,1 ms, to jsou hodnoty mírně vyšší než minimum. Závislost délky výpočtu a počtu vláken zobrazuje graf 6.3.

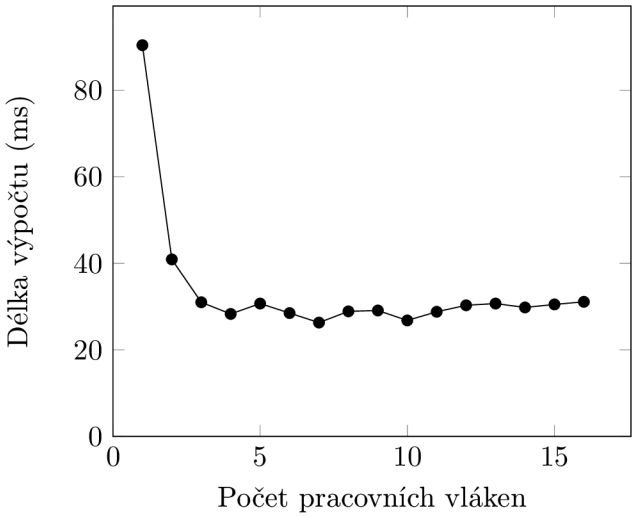
6.3 Podle počtu zpráv odeslaných mezi vlákny

Komunikace mezi vlákny má vliv na celkový výkon programu. Čím více zpráv metoda použije tím více času je vyžadováno na jejich zpracování. V této podkapitole porovnám počet zpráv použitých v paralelních metodách jazyka Rust.

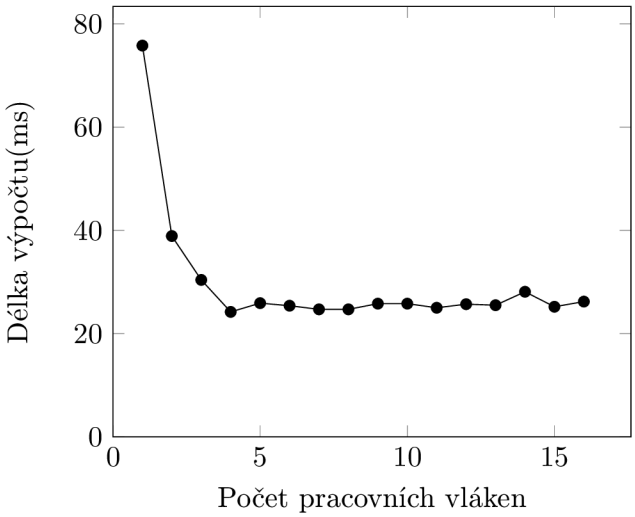
Paralení naivní metoda používá jednu zprávu pro zadání výpočtu jednoho prvku a jednu zprávu pro obdržení výsledku práce. Z toho vyplívá, že pro výpočet jednoho prvku výsledného signálu potřebuji dvě zprávy. Počet zpráv na výpočet celého signálu je $p_{naive} = 2n_y$, kde n_y je délka výsledného signálu.

Metoda overlap-save používá jednu zprávu pro obdržení segmentu výsledného signálu. Počet segmentů je roven počtu vláken, který je zadán uživatelem. Počet zpráv pro výpočet celého výstupního signálu je tedy roven počtu vláken.

Při měření výkonu je používán vstupní signál o rozměrech 128x128 prvků. Kernel má



Obrázek 6.2: Délka výpočtu paralení naivní metody v jazyce Rust v závisloti na počtu pracovních vláken.



Obrázek 6.3: Délka výpočtu metody overlap-save v jazyce Rust v závisloti na počtu pracovních vláken.

rozměry 16x16 prvků. Na výpočet budou použity čtyři vlákna. Výstupní signál má rozměry 144x144 prvků, to znamená, že tvořen 20736 prvky. Naivní metoda použije pro výpočet 41472 zpráv. Metoda overlap-save použije 4 zprávy. V tomto případě metoda overlap-save použije 10368krát méně zpráv než naivní metoda.

6.4 Z pohledu programátora

Hlavním rozdílem při implementaci algoritmů mezi OpenMP a Rustem jsou různé přístupy pro paralelizaci programů. Jazyk Rust používá speciální konstrukce jazyka pro vyjádření paralelizace a komunikace mezi procesy. Jazyk C takové konstrukce neobsahuje, a proto se používají směrnice pro překladač, které se vkládají do sekvenčního kódu.[7] Tyto dva přístupy ssebou přinášejí výhody i nevýhody.

Výhodou OpenMP je to, že program funguje i bez direktiv pro překladač. Program tak může být napsán a otestován jako jednovláknový. Paralelizace tak může být provedena později, například podle výsledků profilování. Z toho i vychází druhá výhoda a ta je, že paralelní verze algoritmu se od neparalelní liší jen direktivami. Konkrétně v praktické části této práce je paralelizace provedena pomocí dvou řádků. Největší nevýhodou jen to, že ne všechny funkce standardní knihovny jazyka C jsou bezpečné pro použití ve více vláknových aplikacích.[1] Další nevýhodou je závislot na externí knihovně, která není standardní součástí jazyka. Mezivláknová komunikace v OpenMP je řešena pomocí sdílené paměti. Přístup k ní musí být ošetřen a špatné ošetření sdílené paměti může vést k uváznutí nebo špatnému fungování programu.

Hlavní výhodou jazyka Rust je, že byl navržen jako jazyk pro paralelní programování. To znamená, že všechny konstrukce potřebné pro tvorbu paralelních programů jsou součástí standardní knihovny, která je bezpečná pro použití ve vícevláknových programech.[4] Sdílení dat je zjednodušeno neměnitelnými proměnnými. Rust obsahuje mechanizmy zvyšující bezpečnost jazyka jako například neměnitelné proměnné a kontrola indexů při práci s vektory. Největší nevýhodou jazyka Rust je jeho nízký věk a ním spojená nestálost. Nové verze přinášejí nové konstrukce a odstraňují staré, takže nejsou vzájemně kompatibilní. Oficiálním návodům chybí kapitoly a neoficiální jsou obvykle psané pro starší verze. Nevýhodou zvýšení bezpečnosti při práci s vektory je zvýšení doby práce s vektory.

Kapitola 7

Závěr

V této práci jsem implementoval celkem šest metod pro výpočet konvoluce v jazyce Rust, jazyce C s použitím OpenMP a pomocí knihovny FFTW. Metody jsem mezi sebou porovnal pomocí několika kritérií.

Nejrychlejší byla vícevláknová metoda pomocí OpenMP, která spočítala výsledek za 11,7 ms. Druhá nejrychlejší byla metoda overlap-save v jazyce Rust, ta spočítala výsledný signál za více než dvounásobek času nejrychlejší metody - 24,2 ms. Při porovnání podle počtu vláken jsem zjistil, že metoda používající OpenMP je nejrychlejší při použití počtu vláken, který se rovná počtu jader procesoru. Ideální počet vláken má výrazný vliv na tuto metodu. V jazyce Rust je počet vláken méně důležitý. Pokud je počet vláken rovný nebo vyšší než počet jader, dosahují metody podobné rychlosti. Jednovláknové naivní metody se liší o 20,2 ms, tento výrazný rozdíl demonstruje jak je výkon stejného algoritmu ovlivněn jazykem implementace. Doba výpočtu vícevláknových metod v jazyce Rust se lišila pouze o 1,9 ms. Tento rozdíl byl způsoben odlišným počtem zpráv, které tyto metody používají.

Z pohledu programátorské přívětivosti je na tom lépe OpenMP. Jazyk Rust má sice mnoho vlastností, které ho předurčují k tomu, aby byl velmi dobrým jazykem pro tvorbu paralelních programů. Ty jsou ale zastíněny jeho nevyzrálostí.

Vícevláknová metoda v pomocí OpenMP dosáhla výrazně lepších výsledků než metoda používající knihovnu FFTW, proto si myslím, že tato metoda je vhodná pro použití ve výzkumu či praxi. Metody implementované v jazyce Rust můžou být po vydání nové verze aktualizovány a využity pro porovnávání výkonu mezi starou a novou verzí překladače jazyka. Jednovláknová naivní metoda může sloužit pro porovnání optimalizace sekvenčního kódu. Rozdíl mezi vícevláknovou naivní a overlap-save je ovlivněn účinností práce s velkým počtem zpráv u první metody, toto může být použito pro testování efektivity práce se zprávami.

Literatura

[1] POSIX.1-2008. Technická zpráva, The IEEE and The Open Group.

[2] The Rust Language. `http://www.rust-lang.org/`, 2013-02-18 [cit. 2013-02-18].

[3] The Rust Language Tutorial. http://static.rust-lang.org/doc/0.10/tutorial.html, 2013-04-26 [cit. 2013-04-26].

[4] The Rust Reference Manual. http://static.rust-lang.org/doc/0.10/rust.html, 2013-04-26 [cit. 2013-04-26].

[5] DevIL. http://openil.sourceforge.net/, 2013-05-03 [cit. 2013-05-03].

[6] FFTW Home Page. http://www.fftw.org/, 2013-05-03 [cit. 2013-05-03].

[7] Chandra, R.; Dagun, L.; Kohr, D.; aj.: *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[8] Dvořák, V.: *Parallel Systems Architecture and Programming*. 2008.

[9] Madisetti, V.: *The Digital Signal Processing Handbook*. CRC Presss, 1997.

[10] Naghizadeh, M.; Sacchi, M. D.: Multidimensional convolution via a 1D convolution algorithm. *The Leading Edge*, 2009.

[11] OpenMP Architecture Review Board: *OpenMP Application Program Interface*. July 2013.

[12] Press, W.: *Numerical recipes in Pascal : the art of scientific computing*. Cambridge University Press, 1989.

[13] Rabiner, L. R.: *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.

[14] Walton, P.: Removing Garbage Collection From the Rust Language. http://pcwalton.github.io/blog/2013/06/02/removing-garbage-collection-from-the-rust-language/, 2013-04-26 [cit. 2013-06-02].

[15] Welsh, N.: The Rust Language. http://lambda-the-ultimate.org/node/4009, 2013-07-08 [cit. 2013-02-18].

Příloha A

Obsah CD

/inputs Ukázky vstupních signálů.

/kernels Ukázky kernelů.

/outputs Složka pro výstupní signály.

/report Zdrojové soubory tohoto repotu.

/src Zdrojové soubory praktické části této práce.

Příloha B

Požadavky

- GCC 4.7.0 nebo vyšší
- DevIL 1.7.8 nebo vyšší
- FFTW 3.3.3 nebo vyšší
- rustc 0.10

Příloha C

Popis rozhraní

Rozhraní obou programů jsou si velmi podobná. Liší se pouze seznamem podporovaných metod. Podporované metody a k nim odpovídající hodnota přepínače `-m` jsou zapsány v tabulce C.1. Následující výčet popisuje jednotlivé přepínače programů.

-h Vypiš nápovědu.

-i FILENAME Vstupní signál je uložen v souboru FILENAME.

-k FILENAME Kernel je uložen v souboru FILENAME.

-o FILENAME Výstupní signál bude uložen do souboru FILENAME.

-a Vstupní signál je uložen po sloupcích.

-b Kernel je uložen po sloupcích.

-c Výstupní signál bude uložen po sloupcích.

-m METHOD Pro výpočet použij metodu METHOD podle tabulky C.1.

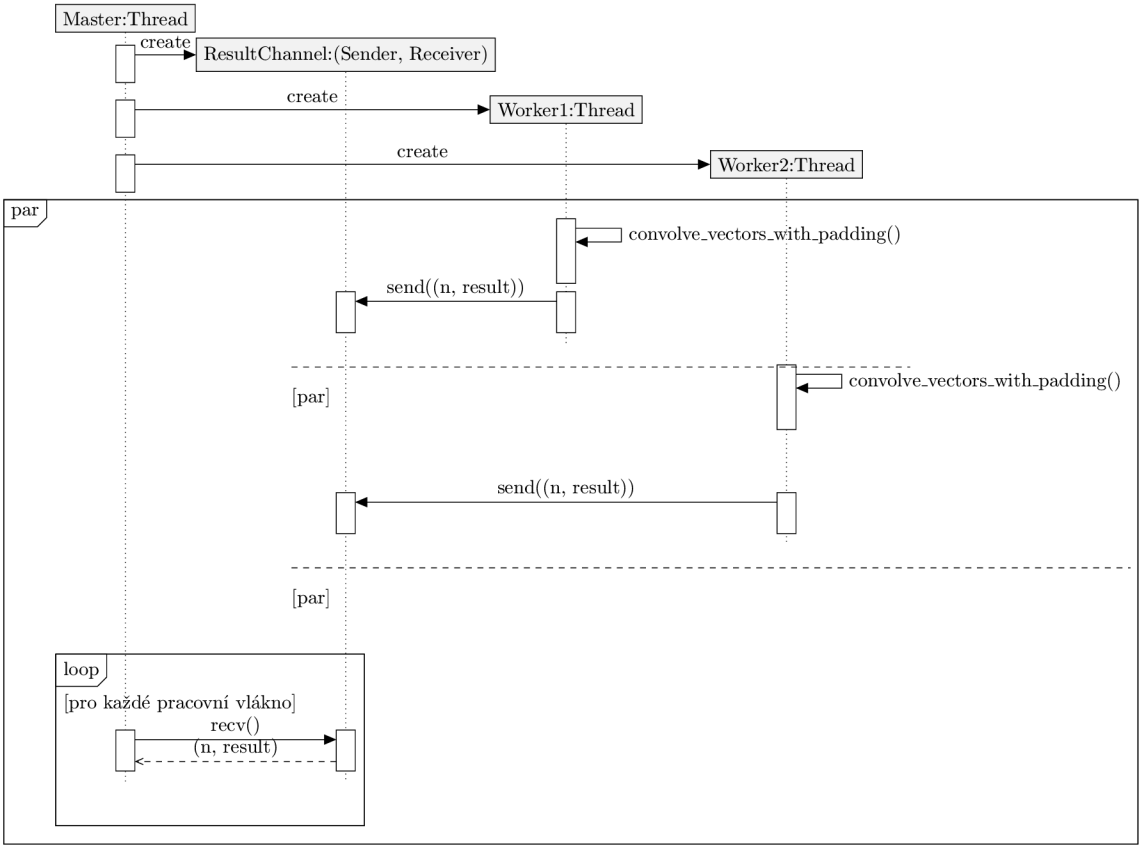
Pokud je zadán přepínač `-h` všechny ostatní se ignorují. Pokud nejsou zadány přepínače `-i`, `-k` a `-o` je program ukončen kvůli chybě na příkazovém řádku. Když není zadán přepínač `-m` je použita jednovláknová naivní metoda.

Metoda	Parametr	C	Rust
konvoluční teorém	`fourier`	Ano	Ne
jednovláknová naivní	`single-naive`	Ano	Ano
vícevláknová naivní	`multi-naive`	Ano	Ano
overlap-save	`overlap-save`	Ne	Ano

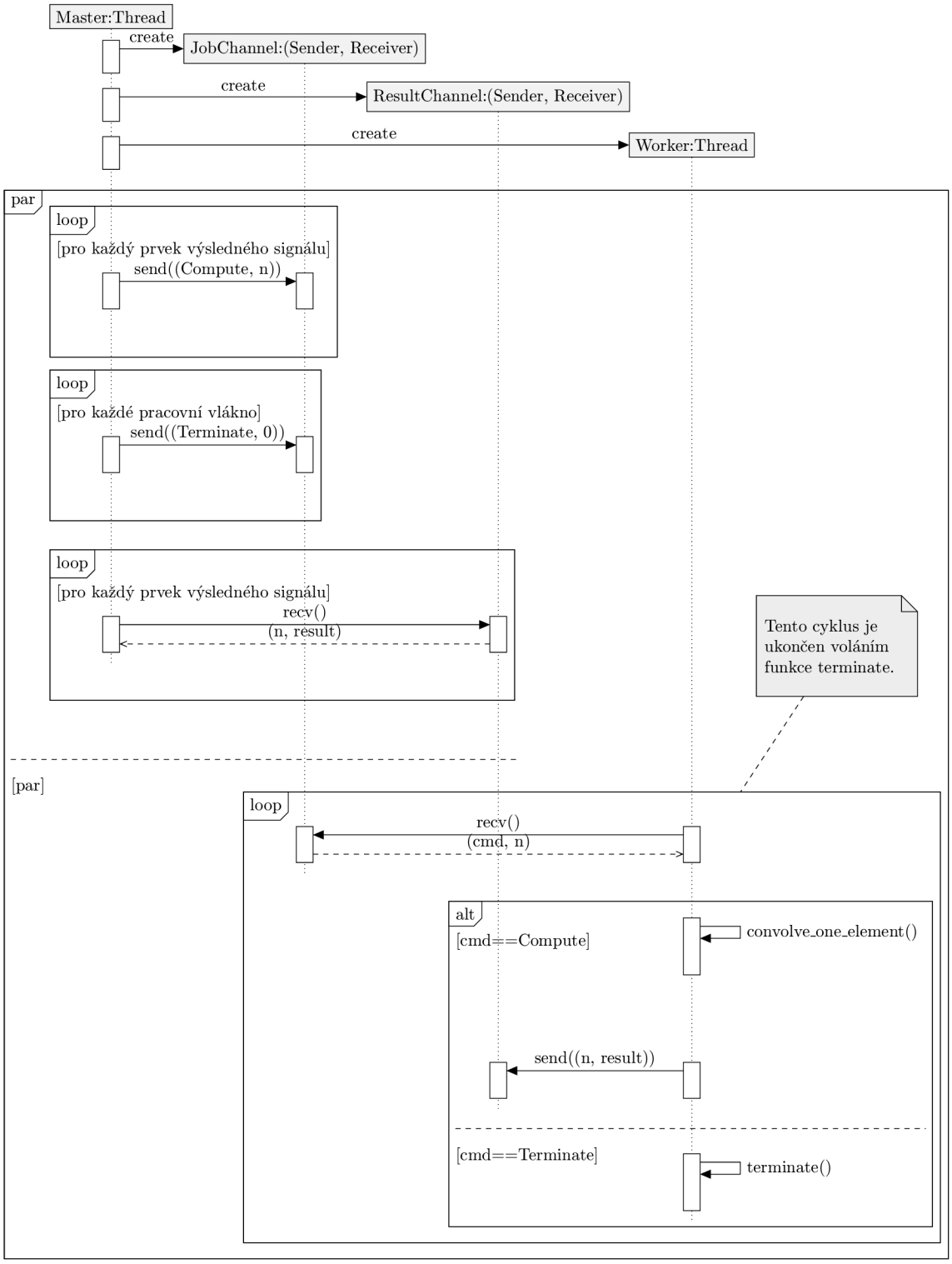
Tabulka C.1: Podporované metody

Příloha D

Diagramy



Obrázek D.1: Metoda overlap-save v jazyce Rust.



Obrázek D.2: Vícevláknová naivní metoda v jazyce Rust.