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

MODERNÍ PROGRAMOVACÍ JAZYK JULIA

MODERN PROGRAMMING LANGUAGE JULIA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PAVEL FOJTÍK
AUTHOR

VEDOUCÍ PRÁCE Ing. VOJTĚCH NIKL
SUPERVISOR

BRNO 2015

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Fojtík Pavel**
Obor: Informační technologie
Téma: **Moderní programovací jazyk Julia**
Modern Programming Language Julia
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s programovacím jazykem Julia, který kombinuje kompaktnost zdrojového zápisu dynamických jazyků typu Python, Perl, Matlab atd. a výkon statických jazyků typu C/C++.
2. Navrhněte sadu benchmarků, které porovnají výkon Julie s v současnosti nejvíce používanými jazyky.
3. Benchmarky implementujte.
4. Naměřte výkon implementovaných kódů.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Podle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- První a druhý bod zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Nikl Vojtěch, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato práce popisuje dynamický programovací jazyk Julia. Nejprve uživatele seznámí s jeho syntaxí a implementací. Dále popisuje základní pravidla pro efektivní psaní kódu a optimalizaci. Tento dokument také uvádí některé příklady použití ve vědeckých pracích. Nakonec je v experimentální části provedeno porovnání Julie s jazykem Python a C, kteří byli vybráni jako zástupci nejpoužívanějšího statického a dynamického jazyka.

Abstract

This work describes dynamic programming language Julia. Firstly, user is introduced to syntax and implementation of this language. Next there are advices for writing effective code and his optimalization. Also some examples of using Julia in scientific projects are described. Comparison between Julia, C and Python is in experimental part. Python and C were chosen as examples of statically and dynamically typed languages.

Klíčová slova

Julia, C, benchmark, Python, porovnání jazyků, numerické výpočty

Keywords

Julia, C, benchmark, Python, language comparison, numerical computing

Citace

Pavel Fojtík: Moderní programovací jazyk Julia, bakalářská práce, Brno, FIT VUT v Brně, 2015

Moderní programovací jazyk Julia

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Vojtěcha Nikla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Pavel Fojtík
17. května 2016

Poděkování

Chtěl bych poděkovat panu Vojtěchu Niklovi za vedení a odbornou pomoc při psaní této práce.

Obsah

Kapitola 1

Úvod

Programovací jazyky se stále vyvíjejí a stále vznikají nové. Je proto stále těžší se v nich orientovat a vybrat si ten vhodný pro daný problém. Ve škole se většinou setkáváme s těmi nejvíce používanými, což není špatně, ale člověku to může snižovat rozhled a místo toho, aby svůj projekt napsal v jazyce, který je vhodný, tak spíše použije jazyk, který zná. V důsledku toho může dojít k horšímu výkonu, horší udržovatelnosti kódu, atd.

Cílem tohoto dokumentu je představit poměrně nový, méně známý programovací jazyk Julia. Jsou představeny funkce, které jsou méně známé, ale mohou být z všeobecného hlediska užitečné. Popsány jsou i vlastnosti známé z jiných programovacích jazyků, s případným vysvětlením rozdílu. Tato práce nedokáže pokrýt všechny aspekty jazyka, ale může sloužit jako rozcestník.

V kapitole 2 je základní popis jazyka Julia s krátkým úvodem do syntaxe. Také jsou zde uvedeny některá pravidla a rady pro použití jazyka.

Kapitola 3 pojednává o využití Julie v praxi. Jedna se o výtahy z prací dalších autorů zabývajících se stejným tématem.

Kapitola 4 popisuje jazyky použité pro porovnání s Julia. Jsou zde také popsány základní rozdíly v implementaci nebo v použití.

Experimentální část je popsána v kapitole 5. Jsou zde uvedeny informace o testovacím prostředí, popisy jednotlivých testů, jejich výsledky s grafy a nakonec i souhrné výsledky.

Kapitola 2

Julia

Julia [4] je nový dynamický programovací jazyk zaměřený na numerické a technické výpočty. Ve vývoji je od roku 2009 [3], přičemž veřejně byla vydána v březnu 2012. Jedná se o opensourcový projekt skupiny NumFocus[1] (projekt je šířen pod MIT licencí). Julia se dostává do popředí zájmu mezi programátory díky kompaktnosti zápisu zdrojového kódu, srovnatelného s jazyky typu Python nebo Matlab, a vysoké výkonnosti srovnatelné s jazyky C/C++, které dosahuje díky JIT (just-in-time) kompilátoru založeném na LLVM[2] (viz Obr. 2.1).

2.1 Jádro Julie

Jádro Julie se skládá z následujících komponent:

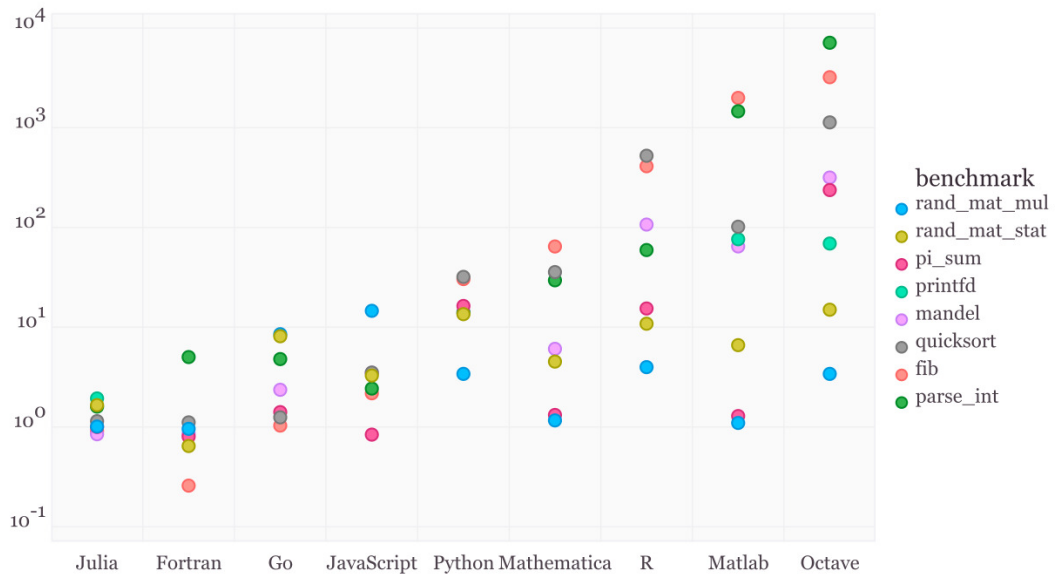
- Syntaktická vrstva, pro přeložení syntaxe do odpovídající vnitřní reprezentace.
- Symbolický jazyk a odpovídající datové struktury pro reprezentaci určitých druhů typů a implementace lattice operátorů (`meet`, `join`, ...) pro tyto typy.
- Implementace generických funkcí a dynamických multimetod pro tyto typy.
- Vnitřní funkce kompilátoru pro přístup k objektovému modelu.
- Vnitřní funkce kompilátoru pro základní aritmetiku, bitové řetězcové operace a volání C a Fortran funkcí.
- Mechanismy pro bindování top-level jmen.

2.2 Syntaxe a sémantika

2.2.1 Proměnné

Názvy proměnných v Julii využívají znakové sady Unicode, je tedy možné jim přiřadit i jiný než alfanumerický znak. To může vést k přehlednějšímu zápisu například matematických vzorců a umožňuje pojmenování proměnných i v jiných druzích písma než je latinka:

[1] http://www.numfocus.org/
[2] http://llvm.org/



Obrázek 2.1: Benchmark výkonu oproti C (převzato z [3]).

```
π = 3.14
S = 4*π*r^2
```

2.2.2 Typový systém

Obecně dělíme jazyky na staticky a dynamicky typované. Staticky typované jazyky, jako je například C nebo Fortran, se vyznačují vyšší výkonností, ale nižší produktivitou při vývoji software. Dynamicky typované, mezi něž patří Python, Ruby a další, mají problém opačný. Při programování v Julii je umožněno vybrat si obě možnosti. Odvozovací systém zachovává dobré výsledky i s dynamickým typovacím systémem. Další problémem u dynamicky typovaných jazyků je, že mají velké výkonnostní rozdíly mezi vestavěnnými daty a uživatelsky definovanými. V Julii je tento rozdíl minimalizován.

2.2.3 Numerické datové typy

Programátor má přístup k obvyklým datovým typům jako je integer (8–128 bitů) a double (16–64 bitů). Také zde patří datový typ bool (8 bitů). Pro integer jsou k dispozici také bez znaménkové varianty. Při nespecifikováni datového typu se celým číslům přiřadí datový typ na základě dané platformy, buď 32 nebo 64 bitů (daná hodnota se dá vyčíst z proměnné `WORD_SIZE`). Pokud by se číslo nevešlo do 32 bitů, tak se vždy přiřadí 64 bitů.

Čísla v plovoucí řádové čárce jsou standardně 64 bitová, pokud nejsou jinak deklarována. Tato čísla obsahují dvě nulové hodnoty, kladnou a zápornou lišící se binární reprezentací. Mezi čísla s plovoucí čárkou patří také tři speciální hodnoty. Kladné nekonečno (`Inf`), záporné nekonečno (`-Inf`) a `NaN`.

Julia podporuje také výpočty nad čísly s libovolnou přesností. Programátorovi je umožněna práce s datovými typy `BigInt` a `BigFloat`. Oba tyto typy obsahují konstruktory pro

převod z primitivních datových typů nebo je možné využít funkci `parse()` pro konverzi z abstraktního řetězce.

```
julia> parse(BigFloat, "1.23456789012345678901")
1.234567890123456789010000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.459565876494606882133333333333333333333333333333333333333333333333344e+19
```

2.2.4 Kompozitní datové typy

Julia nemá klasický objektový model jako například Python, ale definicí se podobá více jazyku C s výjimkou toho, že je možné definovat konstruktor:

```
julia> type animal
      name
      age::Int32
    end
klokan = animal("Jack", 15)
```

Je možné vytvořit i neměnitelné datové typy, pokud `type` nahradíme klíčovým slovem `immutable`. Tyto typy po instancování zůstávají neměnné, mohou ovšem obsahovat typy, které stále zůstávají měnitelné (`mutable`).

2.2.5 Řetězce

V Julii je podobně jako v jiných jazycích řetězec [4] konečnou posloupností znaků. Základní pravidla:

- Řetězce se skládají ze znaků (datový typ `char`, skládající se z 32 bitů).
- Řetězce jsou neměnitelné (`immutable`). Při změně se konstruuje nový řetězec.
- Řetězcové literály jsou vždy ASCII nebo UTF-8, přičemž může být podporováno jiné kódování z externích zdrojů.
- Znaky jsou přetypovatelné na celá čísla (32 nebo 64 bitů, dle architektury). Při přetypování z celých čísel ale není prováděna kontrola validity daného znaku (je možné provést funkcí `isvalid()`).
- Řetězce jsou uvozeny dvěma nebo třemi dvojitými uvozovkami. Druhá možnost umožňuje psaní víceřádkových řetězců.
- Řetězce jsou podobně jako v jiných jazycích indexovatelné. Obsahují také speciální index `end`, který vrací poslední znak v řetězci.
- Řetězce jsou lexikograficky porovnávatelné.
- Pokud je před řetězcem znak `'r'`, tak se s sním zachází jako s regulárním výrazem.

2.3 Operátory

Julie poskytuje klasickou sadu operátorů známých z jiných programovací jazyků (+ ,- ,*, /, atd.). Obsahuje, ale také operátor \ pro inverzní dělení nebo operátor ^ pro mocninu. Mezi další operátory patří také operátor zlomku //:

```
julia> 4\12
3.0
julia> 3//6 + 1
3//2
```

Další věcí, která umožňuje přehlednější matematický zápis, je možnost před proměnou vložit číselný literál. Tato operace poté implikuje násobení. Stejně tak je možné vložit složitější výraz ohraničený závorkami:

```
julia> y = 5
julia> 2(y-1)y
40
```

Mezi bitovými operátory můžeme nalézt and (&), not (∼), or (|), xor ($), logický posun vpravo (>>>), aritmetický posuv vpravo (>>) a logicky/aritmetický posuv vlevo (<<). Porovnávací operátory jsou klasické jako v ostaních programovacích jazycích s tím, že některé operátory jsou nahraditelné za zkráceny výraz. Možné je také operátory řetězit (jako např. v Pythonu):

```
julia> 3≤2
false
julia> a < b < c
true
```

2.4 Numerické výpočty

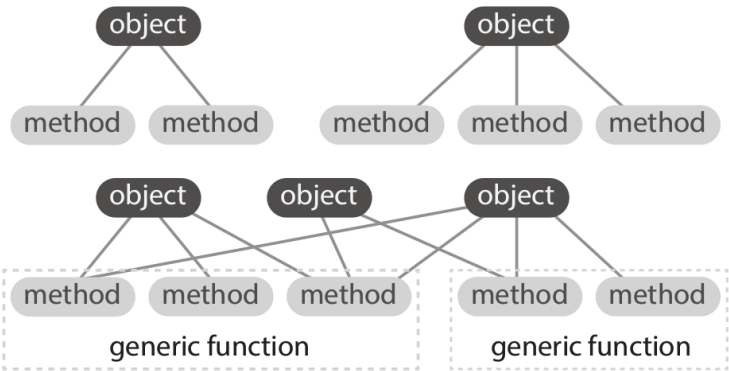
Julia má velké množství vestavěných matematických metod, je schopná pracovat s racionálními čísly, ale stále se jedná o poměrně nový jazyk, který zatím neposkytuje takovou funkcionalitu, jako ostatní vyspělejší jazyky. Pro tyto případy je možné volat funkce Pythonu použitím PyCall[3] balíčku nebo funkce jazyka C bez využití wrapperů nebo speciálních API. Pro vytváření grafů je zde připravena knihovna Gadfly[4]. S tímto nástrojem je možné vytvářet rozmanité grafy, v různých formátech (svg, pdf, atd.).

2.5 Funkce

Julia umožňuje dva styly zápisu funkcí, klasický (jako je např. v C) a zkrácený:

```
julia> plus(x,y) = x+y
julia> plus(1,2)
3
```

[3] https://github.com/stevengj/PyCall.jl
[4] https://github.com/dcjones/Gadfly.jl



Obrázek 2.2: Rozdíl mezi třídními metodami a generickými funkcemi (převzato z [2]).

Funkce stejně jako proměnné mohou také obsahovat Unicode znaky.

2.5.1 Návratové hodnoty

Návratová hodnota může být určena jako poslední příkaz funkce nebo klíčovým slovem `return`. Můžeme vracet najednou také více hodnot, v tomto případě se návratové hodnoty uloží jako `tuple`.

2.6 Control Flow

Jednotlivé příkazy jsou odděleny koncem řádku nebo středníkem. Při zápisu se středníkem je možné vložit více příkazu na jeden řádek:

```
julia> z = (a = 1; b = 3; b*a)
3
```

Julia obsahuje klasické metody řízení toku (`if, for, while, try, catch`, atd.). For cykly mohou ale obsahovat i více proměnných. Podporuje také koprogramy, metody, které jsou schopny za pomoci funkce `produce()` přerušit činnost a vrátit řízení nadřazenému podprogramu.

2.7 Generické metody

Generické metody [2] (nebo také multimetody) jsou objektově orientované paradigma, ve kterém jsou metody definovány nad kombinací dat místo toho, aby byly zapouzdřeny ve třídách (viz Obr. 2.2). Tyto funkce jsou poté seskupeny do generických funkcí. Při volání těchto metod se poté vybere ta, která nejvíce odpovídá dané signatuře. Za pomocí této vlastnosti je můžeme definovat nebo přetížit funkci, aby pracovala nad námi definovanými daty:

```
add(a::Int64...)
add(a::Float64...)
```

... notace umožňuje, aby daná funkce přijala i volání s blíže nespecifikovaným počtem následujících parametrů. Tuto notaci je možná použít pouze na konci signatury.

2.8 Implementace polí

Základním datovým typem pole v Julii je `AbstractArray{T,N}`. `T` určuje typ prvku a `N` počet dimenzí. Od tohoto datového typu jsou poté odvozeny všechny typy polí a typy podobné polím (např. matice). Typy odvozené od `AbstractArray` by měly minimálně implementovat metody `size(A)`, `getindex(A, i)` a `getindex(A, i1, ..., iN)`.

Jedním z odvozených typu je `DenseArray`, který zahrnuje druhy polí uložených v paměti podle daného offsetu, tudíž mohou být předávány do funkcí jazyka C nebo Fortranu. Specifickou instancí toho typu je typ `Array` (také `Vector` a `Matrix`, což jsou aliasy pro jednorozměrná a dvourozměrná pole). Matice jsou poté v paměti uloženy po sloupcích.

2.9 Paralelní výpočty

Princip paralelního programování v Julii je podobný jako v jazyce C. Pokud chceme `for` cyklus paralelizovat, je potřeba před klíčové slovo `for` přidat makro `@parallel`, to ale nezajistí synchronizaci vláken, pokud je potřeba. Na to je nutné přidat ještě makro `@sync`:

```
@sync @parallel for d = 2:m - 1
   for c = 2:m - 1
   ⋮
```

A následné spuštění:

```
$ julia -p 2 heat.jl
```

Přepínačem `-p` upřesňujeme počet vláken.

2.10 Optimalizace

Základní pravidla pro rychlý běh algoritmu v Julii:

- Vyhýbat se globálním proměnným. Kód, který je kritický pro algoritmus, by měl být umístěn do funkce.
- Vyhýbat se polím s abstraktním typem parametru (např. `Real[]`)
- Při vytváření vlastních datových typů deklarovat datové typy proměnných.
- Funkce by měla vracet vždy stejný datový typ.
- Neměnit datový typ proměnné (psát typově stabilní kód).
- Přistupovat k matici po sloupcích.

Julia umožňuje použití maker, které mohou v některých případech urychlit běh algoritmu:

```
@simd for i = 1:length(x)
   @inbounds s += x[i]*y[i]
end
```

`@inbounds`, umožňuje vypnout kontrolu hranic pole. `@simd` je experimentální funkce umožňující vektorizaci, platí pro ní ale několik pravidel. Musí se jednat o nejníže zanořený cyklus. Iterace na sobě musí být nezávislé. Cyklus má dané kroky (ne náhodné). Cyklus nesmí obsahovat `break`, `continue` a `@goto`. Dalším makrem je `@fastmath`, které optimalizuje operace s čísly v plovoucí řádové čárce, ale výsledky poté nemusí odpovídat standardu `IEEE`.

Pokud se vyskytne problém s rychlostí, tak je možné využít makra:

- `@code_warntype` vygeneruje vnitřní reprezentaci kódu.
- `@allocated` vrací množství paměti alokované daným výrazem.
- `@profile` makro pro zobrazení mapy volání funkcí.
- `@time` vrací celkový čas běhu výrazu, jeho počet alokací a množství alokované paměti.

Tyto funkce mohou pomoci nalézt problematické části kódu (např. častým ukazatelem bývá nepřiměřená velikost alokované paměti).

Kapitola 3

Julia v praxi

3.1 Distribuované výpočty

Julia nativně poskytuje podporu pro víceprocesové distribuované výpočty [5] založené na jednostranném předávání zpráv.

- Funkce `remotecall()` vytvoří neblokující vzdálené volání funkce.
- Při volání funkce je navrácen vzdálený ukazatel.
- Za pomocí blokující funkce `fetch()` je poté získána návratová hodnota.

Tento postup je možné zjednodušit předpřipravenými makry `@spawn` a `@spawnat`. Použitím tohoto přístupu můžeme poté definovat práci funkce nad vzdálenými daty:

```
*(r1::RemoteRef, r2::RemoteRef)
```

3.1.1 Experiment

V [5] je ukázán experiment násobení náhodných matic s Gaussovými vstupy o velikosti n = 4096. Na tomto příkladu je demonstrován graf zrychlení (viz Obr. 3.1) při použití distribuovaných výpočtu, ale při zachování kompaktnosti zápisu.

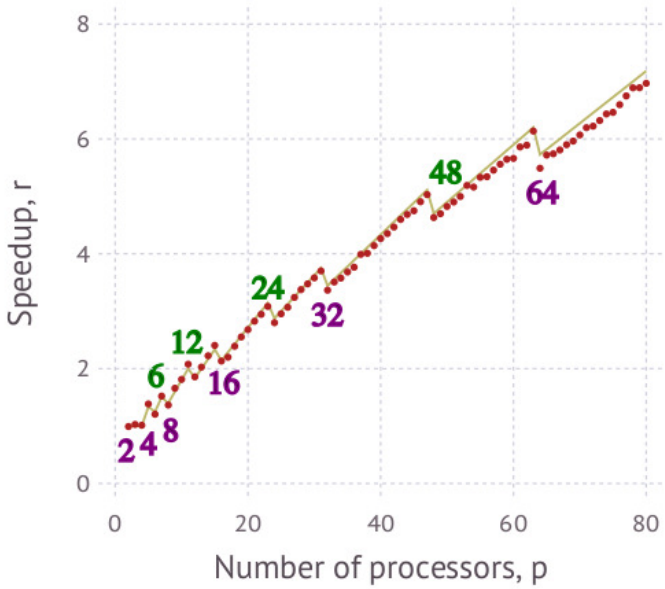
3.2 Convergent cross mapping

CCM (*Convergent cross mapping*)[12] je metoda pro rozpoznání kauzalit v dlouhodobě sbíraných datech. Tato metoda může být použita v medicínském výzkumu pro rozpoznání dlouhodobých závislostí. Například při dlouhodobém výzkumu je možné ze vzorků sestavit určitý vzor změn.

3.3 CauseMap

CauseMap[1] je knihovna implementující CCM v jazyce Julia. Julia byla vybrána pro svůj výkon, jednoduchost použití a nezávislosti na platformě. Efektivnost algoritmu můžeme vyčíst z tabulky 3.1.

[1] https://github.com/cyrusmaher/CauseMap.jl



Obrázek 3.1: Graf zrychlení výpočtu při zvyšování počtu procesorů (převzato z [5]).

Time series length	Runtime(s)
71	10.2
142	40.4
213	116.6
284	317.2
355	534.7
426	1080.5

Tabulka 3.1: Tabulka časů pro implementaci CauseMap (převzato z [12]).

3.4 Další použití

- Automatizované řešení parciálních diferenciálních rovnic za použití globální spektrální metody [15]. Vyřešení rovnice o více než milionu stupních volnosti vykonná Matlab do 60 sekund. Julia to samé zadání zvládne do 10 sekund. Experimentální implementace je dostupna v balíčku `ApproxFun.jl`[2].

- Výpočet kořenů realných polynomů s rozdílnými kořeny [14]. Metoda je implementovaná v knihovně `Arrowhead.jl`[3].

[2] https://github.com/ApproxFun/ApproxFun.jl
[3] https://github.com/ivanslapnicar/Arrowhead.jl

Kapitola 4

Srovnání s vybranými jazyky

V této kapitole budou popsány dva jazyky, proti kterým bude Julia srovnávána - C a Python.

4.1 Programovací jazyk C

4.1.1 Popis

C [8] je programovací jazyk pro obecné použití, je vhodný jak pro vysokoúrovňové programy, tak i pro nízkoúrovňové. Někdy bývá nazýván „system programming language“ kvůli tomu, že se hodí pro psaní kompilátorů a operačních systémů. C je staticky typovaný jazyk. Jeho základní datové typy jsou znaky, celá čísla a čísla s plovoucí řádovou čárkou. Poté existují datové typy odvozené, jako je pole, ukazatele, struktury a uniony. Proměnné mohou být lokální pro určitou funkci nebo pro jeden zdrojový soubor, ale také viditelné pro celý program (globální). Poskytuje také základní konstrukce pro řízení programu jako je `if-else, switch, while, for, do, break` a další.

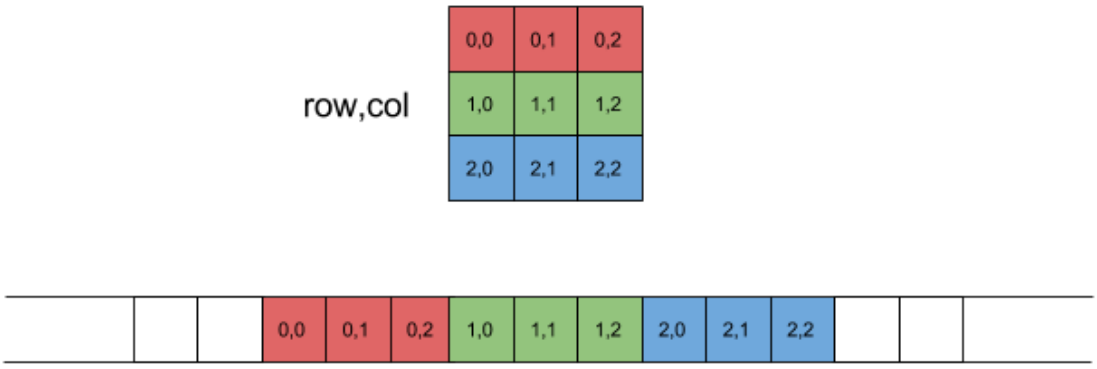
C neposkytuje metody pro přímou práci s kompozitními objekty jako jsou seznamy, pole, atd. Neposkytuje ani garbage collector nebo operace pro práci se soubory. Tyto funkce mohou být využity přes explicitně volané funkce (některé tyto funkce, které kompilátor musí poskytovat, jsou definovány standardem). Tento přístup ale zachovává jednoduchost a přístupnost jazyka. Jazyk C byl pro tuto práci vybrán pro jeho vysoký výkon a jako zástupce staticky typovaného jazyka.

4.1.2 Implementace polí

Staticky alokovaná jednorozměrná pole v jazyce C jsou implementovány [1] jako homogenní posloupnost prvků o dané velikosti (důležité pro ukazatelovou aritmetiku). Matice jsou ukládány po řádcích, viz Obr. 4.1. Dynamicky alokovaná pole si uživatel musí definovat manuální alokací paměti na hromadě. Matice jsou poté alokovány jako pole polí.

4.1.3 Paralelní výpočty

Pro paralelní výpočty je v jazyce C využita knihovna `OpenMP`. Jedná se o knihovnu pro práci s vlákny, mezi jejíž největší přínosy patří jednoduchost použití a efektivnost. Příklad použití knihovny `OpenMP`:



Obrázek 4.1: Uložení matic v jazyce C (převzato z [1]).

```
#pragma omp parallel for private(c,d)
for (c = 1; c < m-1; c++) {
   for (d = 1; d < m-1; d++) {
      ⋮
```

Použitím makra `#pragma omp parallel for` se dosáhne toho, že iterace daného cyklu jsou rozděleny mezi jednotlivá vlákna a prováděny paralelně. Pro určení počtu vláken můžeme v unixové prostředí nastavit proměnnou `OMP_NUM_THREADS` nebo v programu přidáním makra `num_threads(n)`(n je počet vláken).

4.1.4 Vybrané rozdíly oproti Julii

- Indexování začíná od 0, místo od 1 (Julia).
- Julia nepracuje s 1 a 0 jako s boolovskými hodnotami, tudíž nelze napsat např. `if(1)`.
- Matice jsou v Julii ukládány po sloupcích narozdíl od C, kde jsou ukládány po řádcích.
- V Julii '//' funguje jako operátor.
- V Julii '^' funguje jako operátor mocniny.

4.2 Programovací jazyk Python

4.2.1 Popis

Python [6] je vysokoúrovňový jazyk pro obecné užití (původně myšlen jako skriptovací). Tento jazyk má vyšší míru abstrakce, což usnadňuje psaní kódu (programy obecně bývají kratší), snadněji se také čtou. Python se řadí mezi interpretované jazyky (viz Obr. 4.2), protože jeho kód je spouštěn přímo interpretem. V testech je použit interpret CPython [13]. Jedná se o implementaci Pythonu v jazyce C. Zdrojový kód je nejprve zkompilován do byte kódu a poté interpretován virtuálním strojem. Daný interpret pracuje ve dvou módech. Prvním je interaktivní mod, vhodný pro krátké funkce (podobný principu příkazové řádky).

Obrázek 4.2: Schéma interpretovaných jazyků (převzato z [6]).

Druhým módem je spuštění celého skriptu. Python je silně objektový jazyk, všechno je objektem (řetězce, funkce, moduly, atd.). S funkcí můžeme pracovat jako s proměnnou:

```
>>> def foo(a,b):
...     return a + b
>>> funkce = foo
>>> funkce(1,2)
3
```

Python byl vybrán pro porovnání s Julii proto, že se také jedná o objektový jazyk a jeho schopnosti abstrakce jsou mnohem rozsáhlejší než v případě jazyka C, na druhou stranu ale neposkytuje zpravidla takový výkon.

4.2.2 Implementace polí

V Pythonu jsou využity dva druhy polí. Prvním jsou seznamy. Jejich implementace [11] záleží na daném interpretu. V CPythonu jsou seznamy implementovány jako struktury jazyka C:

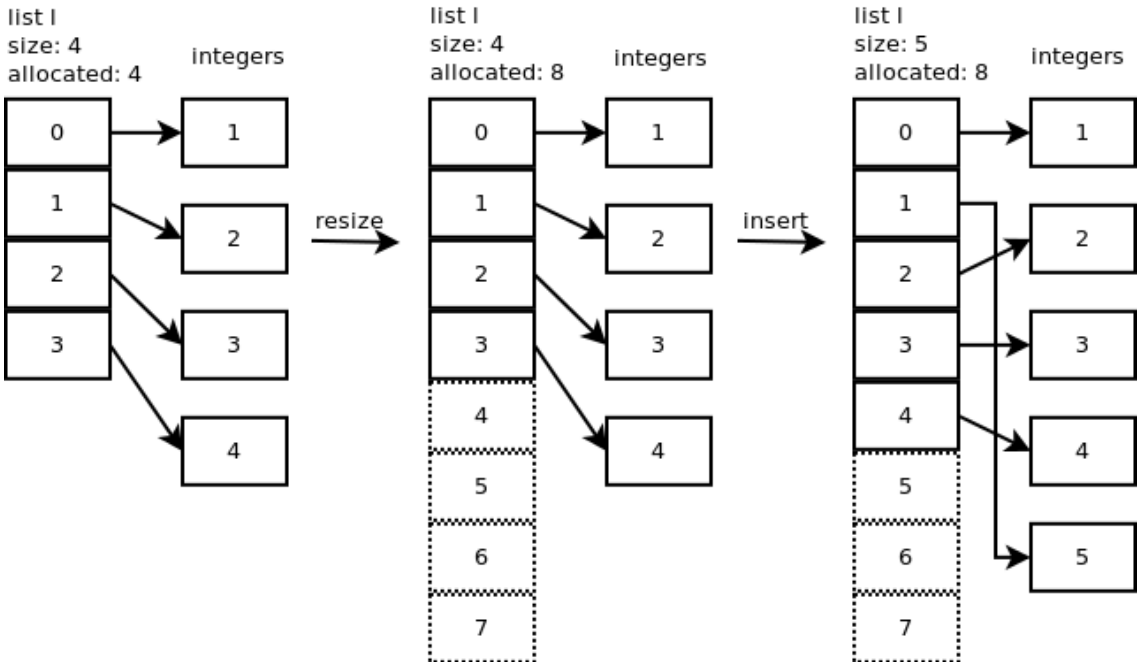
```
typedef struct {
    PyObject_VAR_HEAD;
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

Délka seznamu ale nemusí odpovídat alokované paměti. Interpret provádí alokace vždy po větších blocích, aby zmenšil počet operací `list_resize`. Velikost alokovaného bloku se řídí vzorem (0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...). Operace `insert` nad seznam znázorněna v Obr. 4.3. Druhou možností je využití polí z knihovny `Numpy`[1]. V testech (kapitola 5) konkrétně využívám funkci `numpy.empty()`[2], alokující nenaplněné pole, o velikosti dané parametrem. Tato funkce, kromě určení datového typu, také umožňuje zvolení způsobu ukládání vícerozměrného pole v paměti. První způsob je podle jazyka C, tedy ukládání po řádcích. Druhou možností je ukládání po sloupcích podle jazyka Fortran.

4.2.3 Paralelní výpočty

V Pythonu není prováděno měření paralelních operací, kvůli chybějící podpoře vláken v interpretu `CPython`. Tento interpret obsahuje GIL (Global Interpreter Lock), takže i přesto, že Python má knihovnu pro práci s vlákny, tak v jeden čas má k interpretu přístup pouze jedno vlákno.

[1] http://www.numpy.org/
[2] http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.empty.html



Obrázek 4.3: Operace insert v Python seznamu (převzato z [11]).

4.2.4 Vybrané rozdíly oproti Julii

- Indexování začíná od 0, místo od 1 (Julia)
- Zkrácený zápis inicilizace pole (*list comprehension*) podporuje klauzuli `if`.
- Julia nemá podporu záporných indexů.
- Matice jsou v Julii ukládány po sloupcích, narozdíl od Pythonu, kde jsou ukládány po řádcích (defaultně v `Numpy`).
- Julia vyhodnocuje defaultní parametry funkcí po každém volání. Python pouze při prvním.
- Operátor `%` je v Julii operátor pro zbytek. V Pythonu se jedná o modulo.

Kapitola 5

Mikrotesty

5.1 Obecně

Nejdříve byla vytvořena jednoduchá sada benchmarků pro Julii, C a Python. C a Python byly zvoleni pro porovnání s Julií jako populární zástupci staticky a dynamicky typovaných jazyků. Testy jsou psány tak, aby byl zápis algoritmu ve všech jazycích stejný nebo co nejvíce podobný, zároveň ale zohledňoval některá výkonová omezení daného jazyka (např. ukládání matic po řádcích vs. po sloupcích). Je také minimalizováno využívání vestavěných funkcí pro jejich netransparentnost nebo využívání specializovaných knihoven (BLAS, OpenBLAS např. pro násobení matic). Základní implementační kostra je tak vždy stejná a poskytuje srovnání základního výkonu jazyka.

5.2 Testovací prostředí

Julie je verze 0.4.2. Python byl použit ve verzi 3.4.3. Pro kompilaci jazyka C byl použit nástroj `gcc` verze 5.3.1 s danými parametry:

```
-std=c99 -Wall -O3 -msse -pedantic -g -lpthread -lrt -fopenmp
```

Jako testovací prostředí byl zvolen operační systém Fedora Workstation 23. Daný hardware pro sériové algoritmy viz tab. 5.1. Pro paralerní algoritmy jsem využil školní server z důvodu vyššího počtu jader, viz tab. 5.2.

5.3 Měřicí funkce

Pro větší přesnost určení času jednotlivých běhů běží algoritmy ve smyčce s takovým počtem iterací, aby se dosáhlo času alespoň 5–10 sekund. Výsledný čas se pak získá jako naměřený

Tabulka 5.1: Sériové testy - hardware.

PC	ASUS K70IC
Procesor	Intel(R) Core(TM)2 Duo CPU T6600,@ 2.20GHz
RAM	4GB DDR2 800 MHz SDRAM
GPU	NVIDIA GeForce GT 220M

Tabulka 5.2: Paralelní testy - hardware.

Server	Supermicro 7048GR-TR
Základní deska	Supermicro X10DRG-Q
Procesor	2x Intel Xeon E5-2620v3
RAM	DDR4-2133 64GB (4 channels)
SSD	Crucial 250GB
GPU	NVidia GTX 980 4GB GDDR5
Ostatní	Intel Xeon Phi

čas podělený počtem iterací. Pokud není specifikováno jinak, tak je vždy měřena hlavní část algoritmu, tj. není měřena alokace paměti, načtení dat, výpis, atd.

Pro první měření byla použita linuxová funkce `time` [10]. Tato metoda se ale neukázala jako vhodná. Prvním důvodem je přesnost. Druhým důvodem je ten, že nebylo možné konkrétně specifikovat část algoritmu, která má být měřena. Tato funkce byla použita pro orientační ověření správnosti vestavěných měřících funkcí daného jazyka.

Pro měření času v jazyce C je použita funkce `clock_gettime()` [9] dostupná z knihovny `time.h`. Tato funkce zaznamená čas na začátku prováděného úseku kódu a poté na konci. Výsledný čas se vypočítá jako rozdíl mezi koncovým a počátečním stavem. Přesnost této metody je v řádu nanosekund.

Čas v Pythonu je měřen za pomocí funkce `time()` z modulu `time` [7]. Čas je nejprve zaznamenán na začátku měřeného úseku a poté znovu na konci. Výsledný čas je poté rozdíl těchto časů.

Čas v Julii je měřen pomocí profilovacího makra `@time`[1]. Společně s uplynulým časem vypíše i množství alokované paměti. To je hlavně užitečné při profilování kódu. Pokud je velikost neúměrně velká, jednou z možností může být, že dochází k typové nestabilitě. To se projevuje mnohonásobným zpomalením výkonu interpretu (může se jednat i o více než stonásobné zpomalení).

5.4 Fibonacci

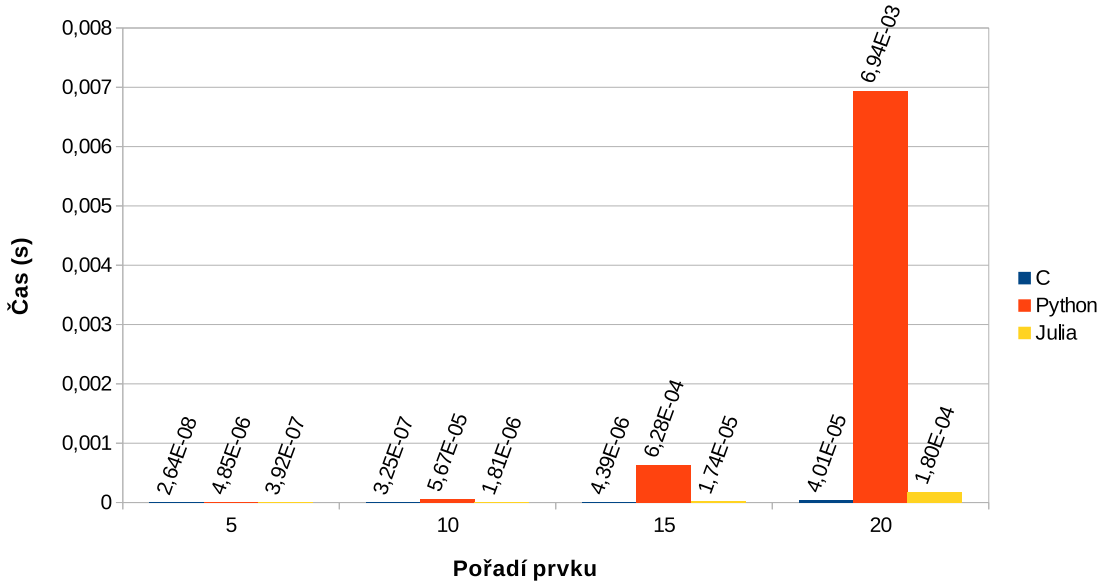
5.4.1 Popis testu

Tento algoritmus počítá hodnotu prvku na dané pozici ve Fibonacciho posloupnosti (viz Alg. 1). Postupně měřená funkce bere jako parametr hledaný prvek. Použita je jeho rekur-

Algoritmus 1: Fibonacciho algoritmus.

```
fib(n):
if n < 2 then
 | return n
end
else
 | return fib(n − 1) + fib(n − 2)
end
```

[1] http://docs.julialang.org/en/release-0.4/manual/performance-tips/



Obrázek 5.1: Výpočet daného prvku Fibonnacciho posloupnosti (pořadí 5–20).

zivní verze a měřeno je přímé zavolání této funkce. Časová náročnost je $O(n)$. Test slouží pro demonstraci výkonu při volání funkcí, rekurzi a skokových instrukcí.

5.4.2 Výsledky

Výsledky jsou v grafu 5.1 a 5.2. C má předpokládaný nejvyšší výkon. Je ale velký rozdíl mezi výkonem Julie a Pythonu. Při výpočtu 45. prvku dosahuje Julia rychlejšího času než jazyk C. Julia byla průměrně 5× pomalejší než C. Oproti Pythonu byla průměrně 164× rychlejší.

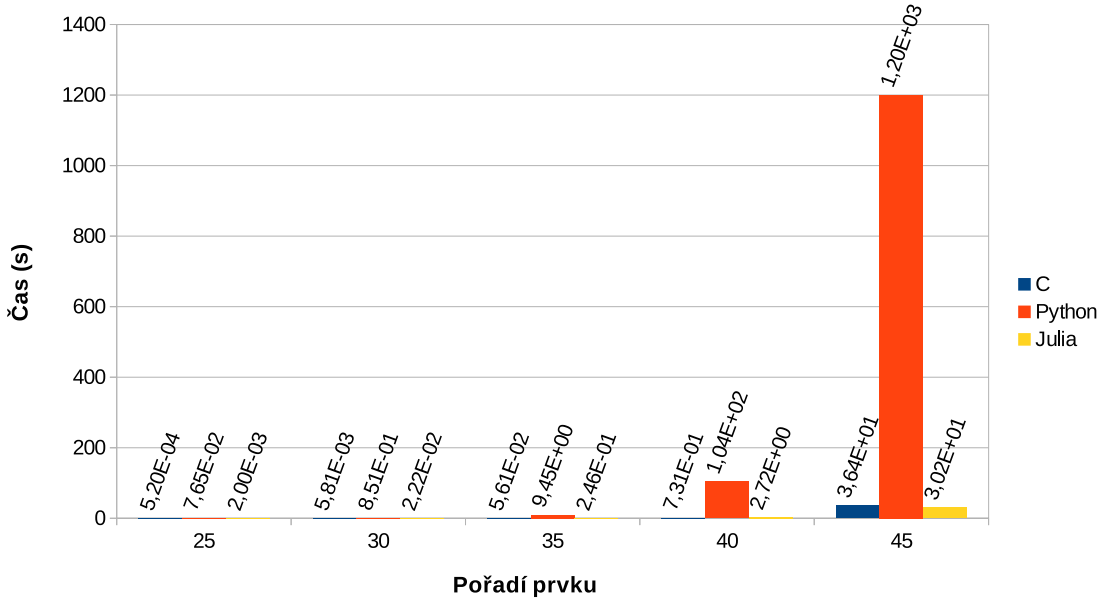
5.5 Násobení matic v jednorozměrném poli

5.5.1 Popis testu

Tento test měří čas násobení matic v jednorozměrném poli (viz Alg. 2) a je zaměřen na výkon při práci s většími poli a operace sčítání a násobení. Zároveň je porovnána jednorozměrná a dvourozměrná indexace z pohledu výkonu.

5.5.2 Výsledky

Výsledky jsou v grafu 5.3. C má viditelně rychlejší výkon než Julia a Python. To může být způsobeno nevhodností použití jednorozměrné indexace v obou maticích. Python je ale pomalejší než Julia. C bylo oproti Julii průměrně 114× rychlejší. Julia byla oproti Pythonu průměrně 2× rychlejší.

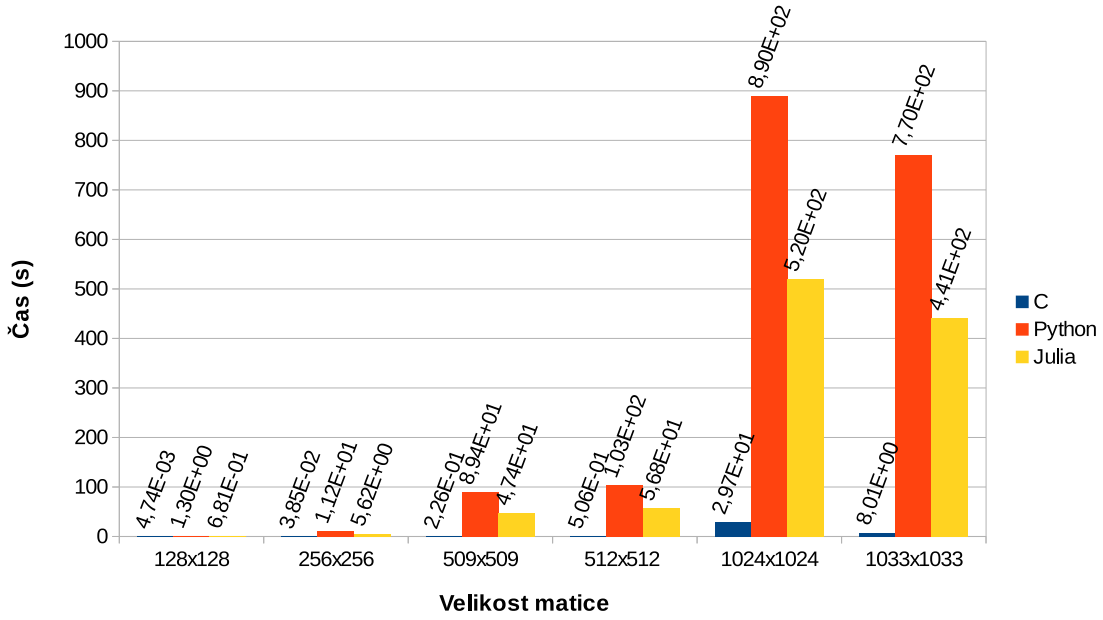


Obrázek 5.2: Výpočet daného prvku Fibonnacciho posloupnosti (pořadí 25–45).

Algoritmus 2: 1D násobení matic.

Data: FirstMatRows, SecondMatCols, SecondMatRows, first, second, multiply, sum

```
sum ← 0
for c ← 0 to FirstMatRows do
    for d ← 0 to SecondMatCols do
        for k ← 0 to SecondMatRows do
            sum ← sum + first[n × c + k] × second[q × k + d]
        end
        multiply[q × c + d] ← sum
        sum ← 0
    end
end
```



Obrázek 5.3: Výsledky násobení matic v jednorozměrném poli.

5.6 Násobení matic v dvourozměrném poli

5.6.1 Popis testu

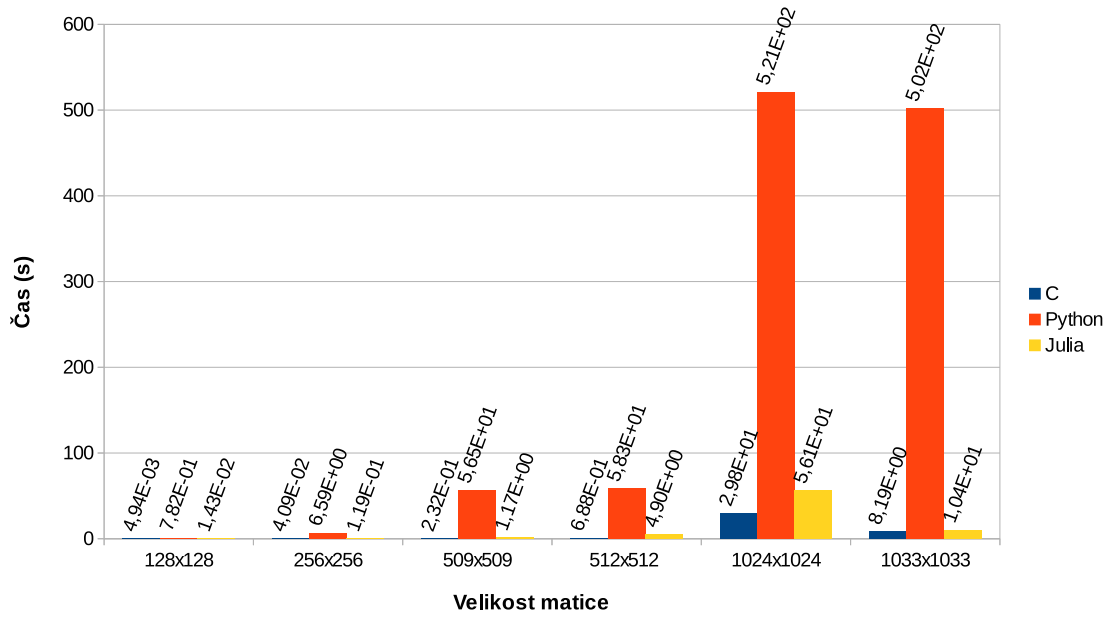
Pro srovnání byl proveden test pro násobení matic v dvourozměrném poli (viz Alg. 3). Test slouží pro porovnání doby přístupu mezi jednorozměrnou a dvourozměrnou indexací.

Algoritmus 3: 2D násobení matic.

```
Data: FirstMatRows, SecondMatCols, SecondMatRows, first, second, multiply, sum
sum ← 0
for c ← 0 to FirstMatRows do
    for d ← 0 to SecondMatCols do
        for k ← 0 to SecondMatRows do
            sum ← sum + first[c][k] × second[k][d]
        end
        multiply[c][d] ← sum
        sum ← 0
    end
end
```

5.6.2 Výsledky

Výsledky jsou v grafu 5.4. C vykazuje stejné výsledky jako při práci s jednorozměrným polem. U zbývajících jazyků lze pozorovat znatelný nárůst výkonu oproti jednorozměrné indexaci. C je rychlejší než Julie, ale výsledky jsou časově srovnatelné. Python je naproti tomu mnohokrát pomalejší. C bylo oproti Julii průměrně 3,5× rychlejší. Julia byla oproti Pythonu průměrně 38× rychlejší.



Obrázek 5.4: Výsledky násobení matic v dvourozměrném poli.

5.6.3 Vestavěné funkce

Pro porovnání byla naměřena i doba vykonávání vestavěných funkcí. V Pythonu je použita funkce `dot()` funkci z knihovny `numpy`. V Julii je se jedná o přetížení operátoru pro násobení (toto řešení přispívá k čitelnosti kódu). V C je použit algoritmus z původního testu. Výsledky testu jsou v grafu 5.5.

5.7 Aproximace π

5.7.1 Popis testu

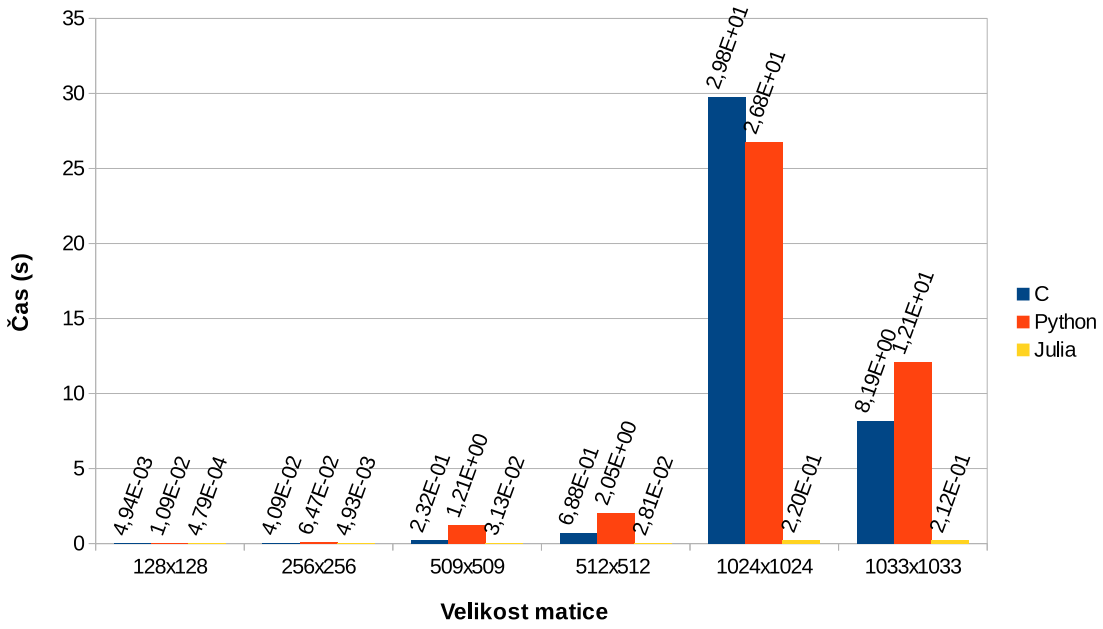
V tomto testu je zkoumána iterativní aproximace čísla π pomocí Taylorova rozvoje (viz Alg. 4). Tento test demonstruje rychlost práce s aritmetikou čísel v plovoucí řádové čárce v iterativním cyklu. Měřena je část provádění výpočtu.

Algoritmus 4: Aproximace π.

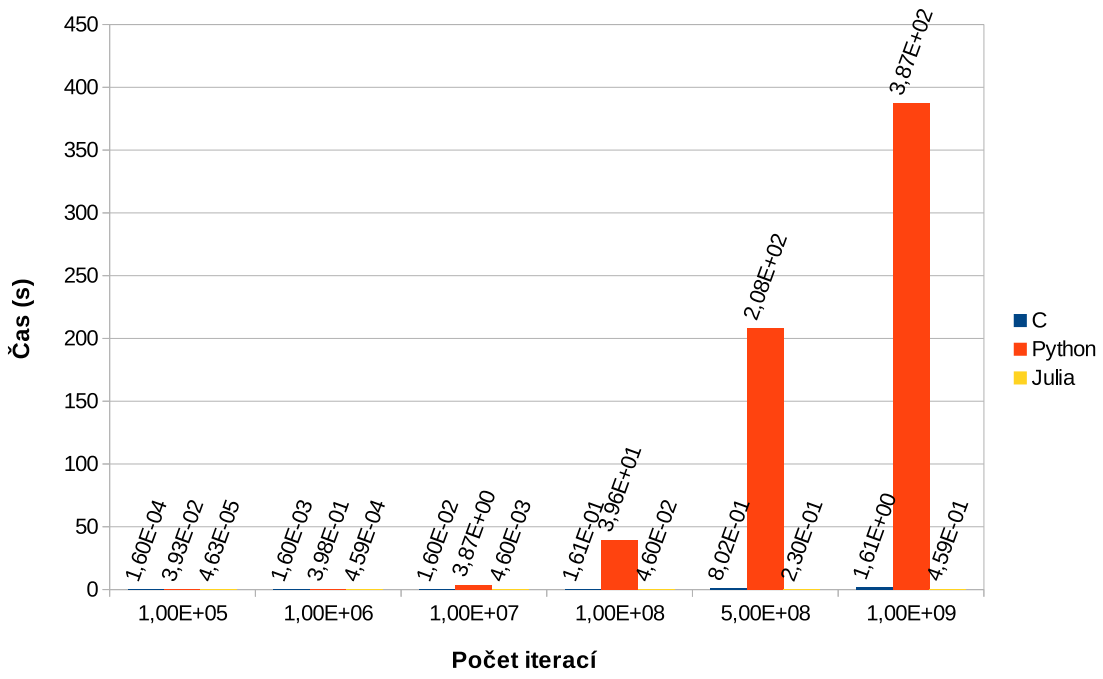
Data: m
$sum \leftarrow 1.0$
for $i \leftarrow 2.0$ *to* m **do**
  $sum \leftarrow sum + (1.0/(i \times i))$
end

5.7.2 Práce s globálními proměnnými

Při testování tohoto skriptu se vyskytl problém, který Julii znatelně zpomaloval (zpomalení v řádu stovek násobků). Problém byl v tom, že se s globální proměnou pracovalo uvnitř



Obrázek 5.5: Výsledky vestavěného násobení matic v dvourozměrném poli.



Obrázek 5.6: Výsledky výpočtu π.

funkce. Pro výrazné zrychlení stačilo proměnou předat jako parametr funkce. Důvod byl ten, že docházelo k typové nestabilitě, což vede ke zpomalení kódu.[2]

5.7.3 Výsledky

Výsledky jsou v grafu 5.6. V tomto měření je nejrychlejší Julia ze všech tří jazyků, přičemž Python je několikanásobně pomalejší. Julia byla oproti C průměrně 3,5× rychlejší. Python byl oproti Julii průměrně 861× pomalejší.

5.8 Quicksort

5.8.1 Popis testu

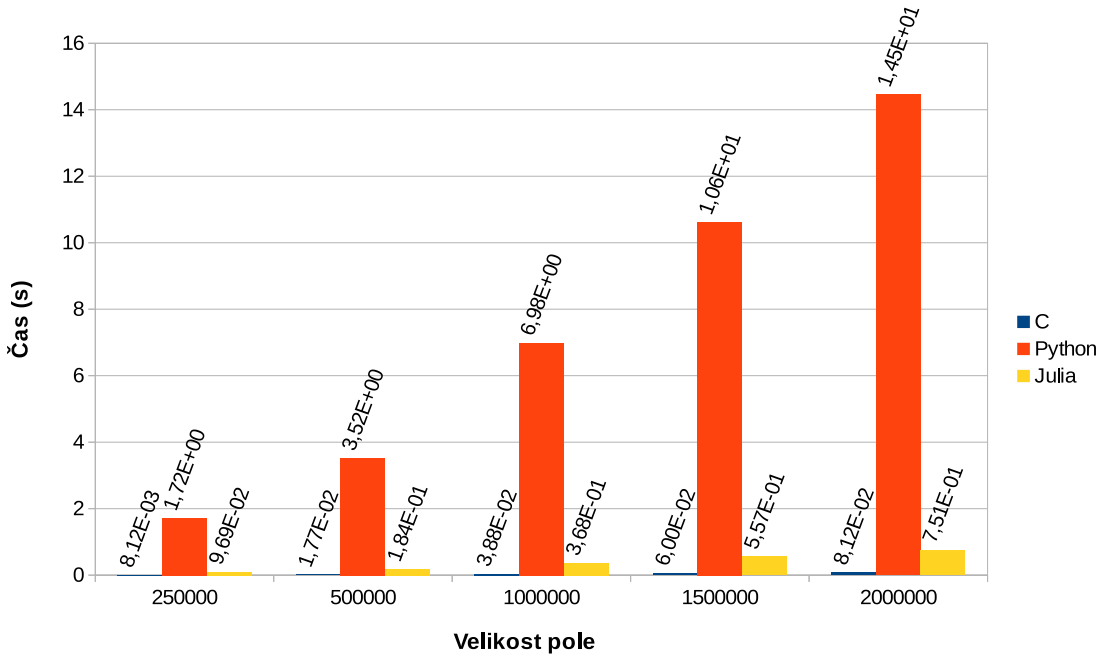
V tomto testu, který je zaměřen na rychlost průchodu polem, je měřena rychlost řadicího algoritmu quicksort (viz Alg. 5). Pole obsahuje 32-bitová celá čísla seřazená v opačném pořadí.

Algoritmus 5: Quicksort.

```
Quicksort(array, left, right):
boundary ← Partition(array, left, right)
if left < boundary − 1 then
  | Quicksort(array, left, boundary − 1)
end
if boundary < right then
  | Quicksort(array, boundary, right)
end

Partition(array, left, right):
pivot ← array[(left + right)/2]
i ← left
j ← right
while i ≤ j do
  | while array[i] < pivot do
  |   | i ← i + 1
  | end
  | while array[j] > pivot do
  |   | j ← j − 1
  | end
  | if i ≤ j then
  |   | Swap(array, i, j)
  |   | i ← i + 1
  |   | j ← j − 1
  | end
end
return i
```

[2] Více o daném problému zde: http://docs.julialang.org/en/release-0.4/manual/performance-tips/



Obrázek 5.7: Výsledky quicksort algoritmu.

5.8.2 Výsledky

Výsledky jsou v grafu 5.7. C je v tomto testu nejrychlejší. C bylo oproti Julii průměrně 10× rychlejší. Julia byla oproti Pythonu průměrně 19× rychlejší.

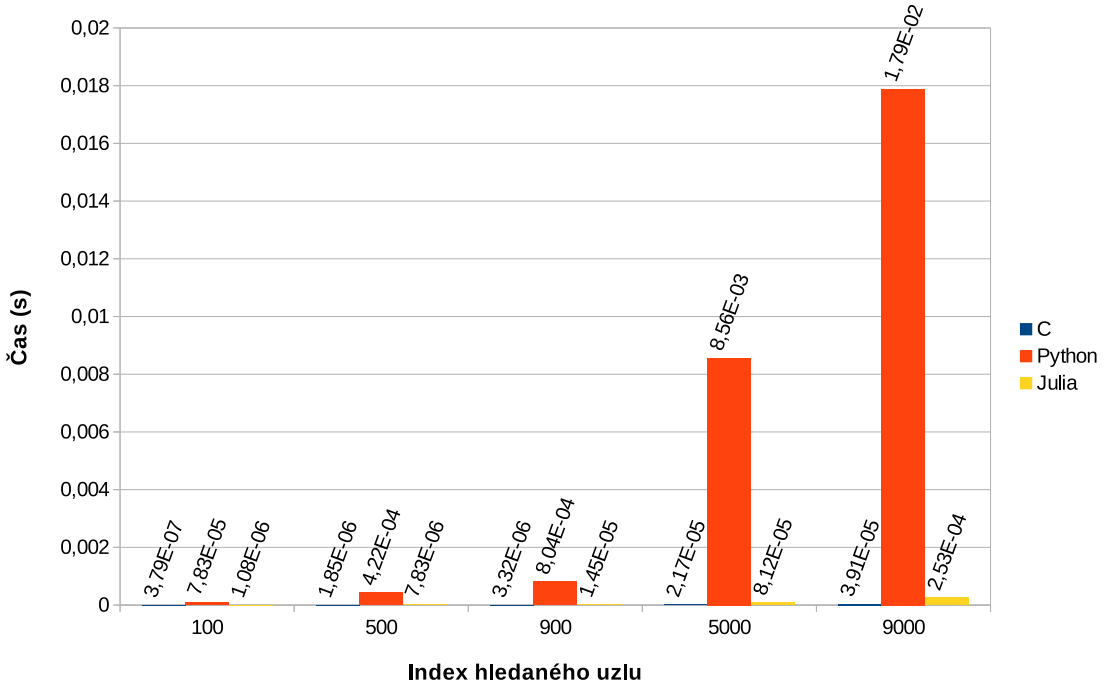
5.9 Binární vyhledávací strom

5.9.1 Popis testu

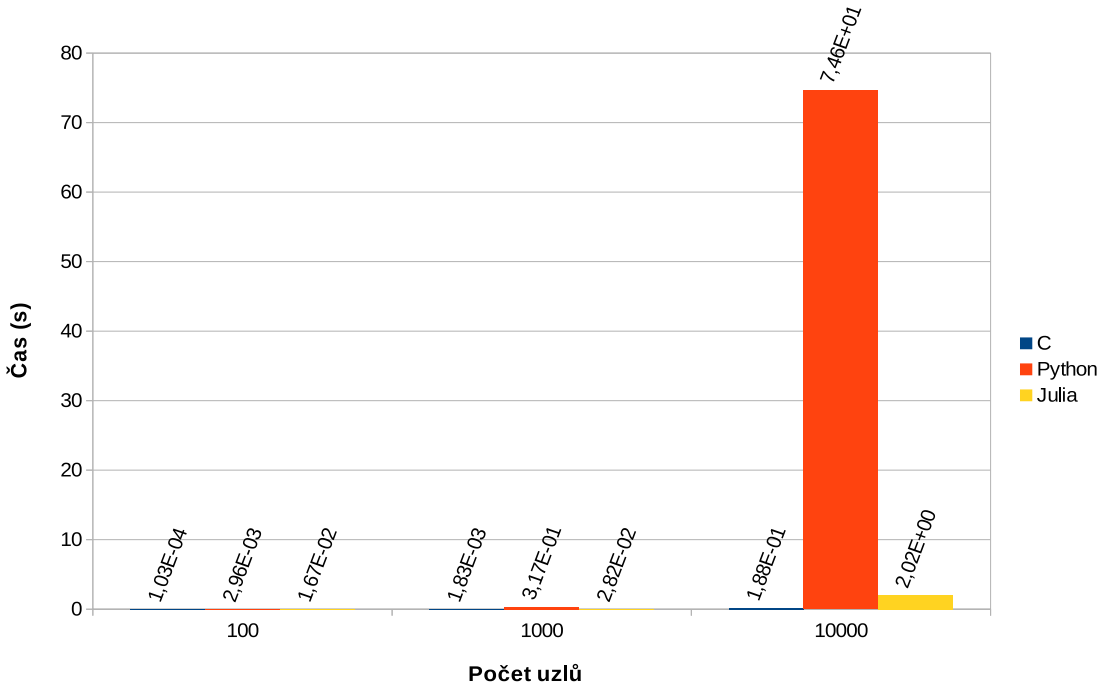
V tomto testu je měřena rychlost rekurzivního vyhledávání v binárním vyhledávacím stromě (viz Alg. 6). Tento test je zaměřen na práci s objekty (strukturami) a čas alokace těchto struktur.

Algoritmus 6: BST

```
Search(key, leaf):
if key == leaf.key then
  | return leaf
end
else if key < leaf.key then
  | return Search(key, leaf.left)
end
else
  | return Search(key, leaf.right)
end
```



Obrázek 5.8: Vyhledávání v binárním vyhledávacím stromě.



Obrázek 5.9: Alokace binárního vyhledávacího stromu.

5.9.2 Výsledky

Výsledky jsou v grafu 5.8. C je v tomto testu nejrychlejší. C bylo oproti Julii průměrně 4× rychlejší. Julia byla oproti Pythonu průměrně 70× rychlejší. Výsledky alokace viz. graf 5.9

5.10 Šíření tepla

5.10.1 Popis testu

V tomto testu je měřena rychlost primitivního algoritmu pro šíření tepla (viz Alg. 7). Algoritmus prochází polem reprezentujícím plochu a každý bod se vypočítá jako průměr čtyř sousedních bodů a jeho samotného.

Algoritmus 7: Šíření tepla.

```
Data: matrixSize, firstMatrix, secondMatrix
for i ← 0 to iter do
    for row ← 1 to matrixSize − 1 do
        for column ← 1 to matrixSize − 1 do
            secondMatrix[row][column] ← (firstMatrix[row][column] +
            firstMatrix[row + 1][column] + firstMatrix[row − 1][column] +
            firstMatrix[row][column + 1] + firstMatrix[row][column − 1])/5.0
        end
    end
    pom ← first
    first ← second
    second ← pom
end
```

5.10.2 Výsledky

Výsledky vyhledávání jsou v grafu 5.10. Výsledky Julie a jazyka C byly srovnatelné, naproti tomu byl Python pomalejší. Jazyk C byl oproti Julii 2,5× rychlejší. Python byl oproti Julii průměrně 240× pomalejší.

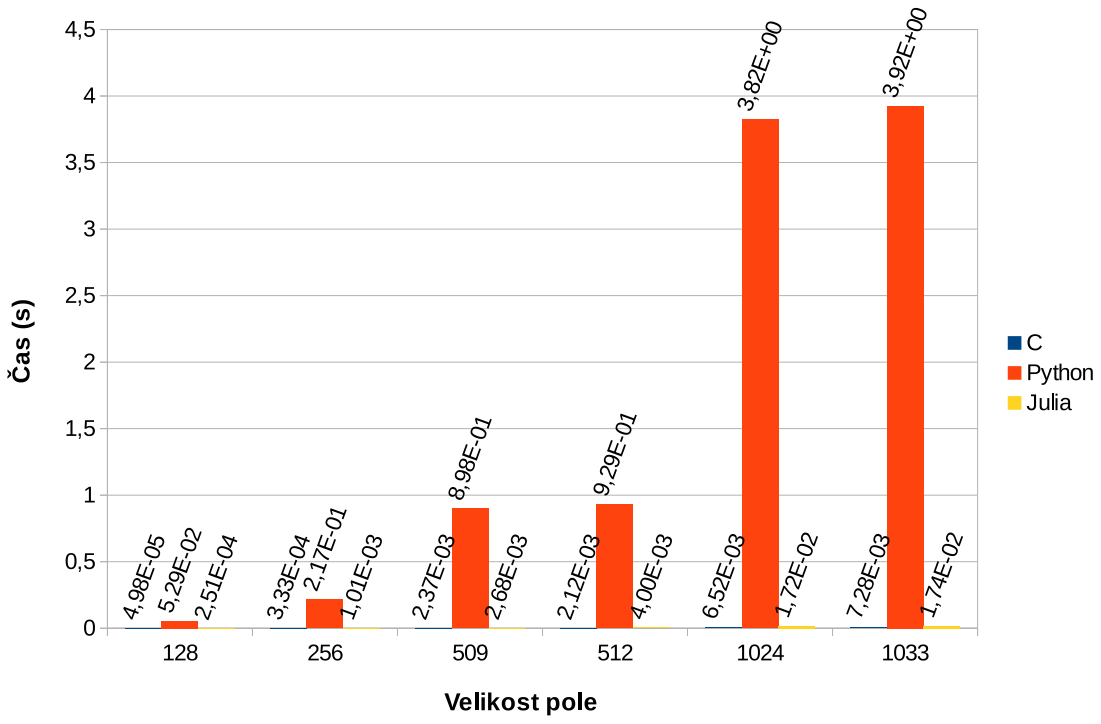
5.11 Test paralelního algoritmu šíření tepla

5.11.1 Popis testu

Algoritmus je upraven tak, že se výpočty jednotlivých řádků provádějí paralelně. Aby nedocházelo k nedeterministickému chování a chybným výpočtům, kdy by některá vlákna mohla přistupovat k již aktualizovaným bodům, jsou použita dvě pole, kde jedno slouží jako zdrojové a druhé jako cílové, a po každé iteraci se jejich role obrací (viz Alg. 5.11).

5.11.2 Výsledky

Výsledky jsou v grafu 5.12. C je v tomto testu průměrně 8× rychlejší než Julie.

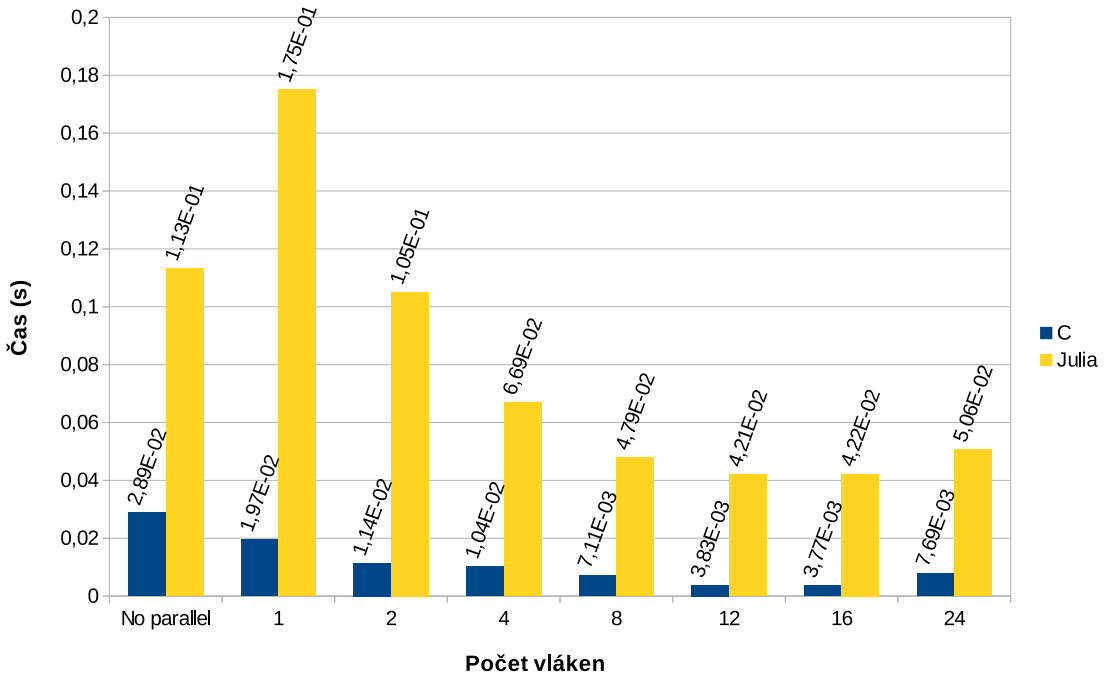


Obrázek 5.10: Algoritmus pro šíření tepla.

```
//C
#pragma omp parallel for private(c,d)
for (c = 1; c < m-1; c++) {
   for (d = 1; d < m-1; d++) {
      second[c][d] = (first[c][d] + first[c+1][d] + first[c-1][d] +
          first[c][d+1] + first[c][d-1]) / 5.0f;
         ⋮

//Julia
@sync @parallel for d = 2:m - 1
   for c = 2:m - 1
      @inbounds second[c,d] = (first[c,d] + first[c+1, d] + first[c-1,
          d] + first[c, d+1] + first[c, d-1]) / 5.0;
         ⋮
```

Obrázek 5.11: Paralelní implementace algoritmu pro šíření tepla.



Obrázek 5.12: Paralelní algoritmus šíření tepla.

5.12 Test paralelního násobení matic

5.12.1 Popis testu

Algoritmus je upraven tak, že jednotlivé řádky matice jsou postupně distribuovány mezi vlákny (viz Alg. 5.13).

5.12.2 Výsledky

Výsledky jsou v grafu 5.14. C je v tomto testu průměrně 38× rychlejší než Julie.

5.13 Souhrnné výsledky

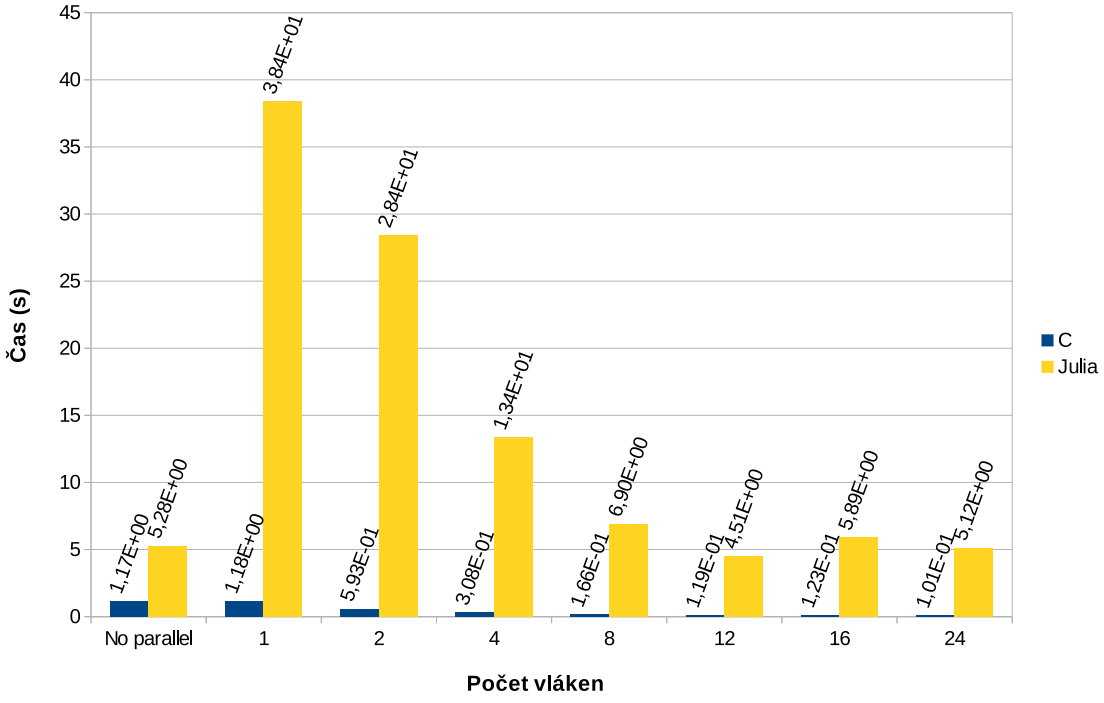
Souhrné výsledky všech testů se nachází v grafu 5.15 a 5.16.

Z výsledku testu Fibonacciho posloupnosti, jde vidět, že Julia pracuje poměrně rychle s rekurzivním voláním s hlubokým stupněm zanoření, v některých případech i rychleji než jazyk C. Z výsledků testů na jednorozměrné násobení matic, lze usuzovat, že Julii tato indexace velmi zpomaluje. Nejlepších časů bylo dosaženo v aproximaci čísla π, tedy v testech s čísly v plovoucí řádové čárce. Prohledávání binárního vyhledávacího stromu, bylo poměrně rychlé, ale nejvíce času zabrala alokace daných struktur. V testech paralelních algoritmů jde vidět, že při větším počtu vláken, lze dosáhnout zrychlení, ale v některých situacích i zpomalení.

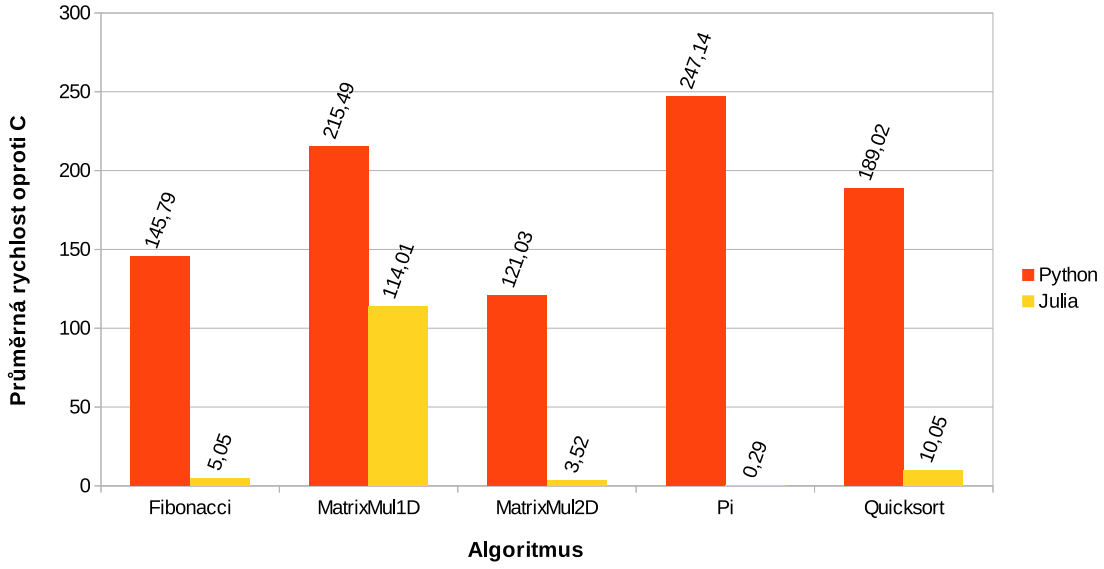
```
//C
#pragma omp parallel for private(c, d, k)
for (c = 0; c < m; c++) {
   for (d = 0; d < q; d++) {
      for (k = 0; k < p; k++) {
         ⋮

//Julia
@sync @parallel for c = 1:m
   for d = 1:q
      for k = 1:p
         ⋮
```

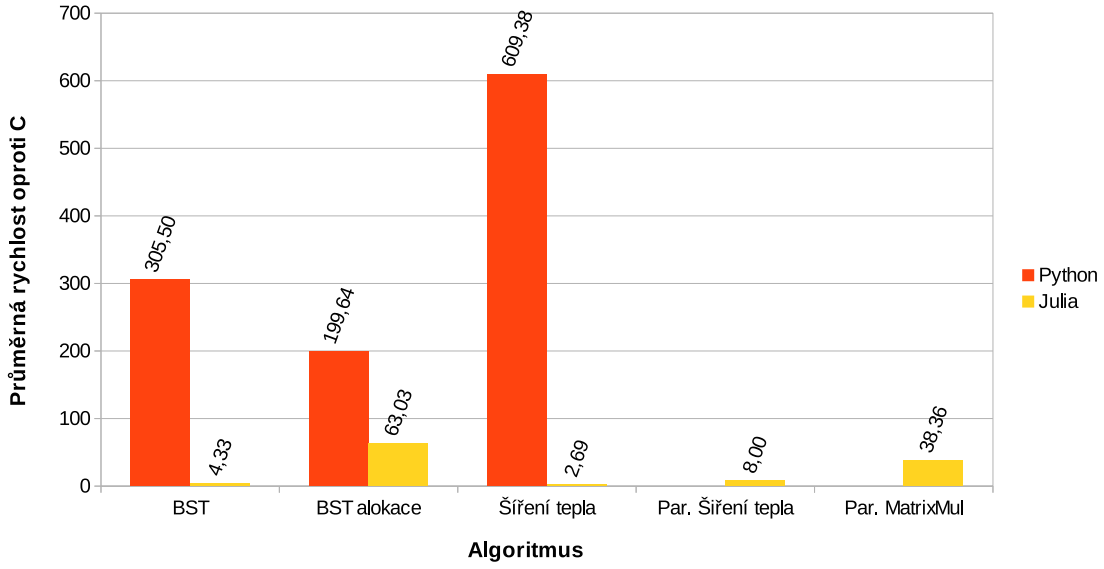
Obrázek 5.13: Paralelní algoritmus násobení matic.



Obrázek 5.14: Paralelní algoritmus násobení matic.



Obrázek 5.15: Souhrnné výsledky 1. část.



Obrázek 5.16: Souhrnné výsledky 2. část.

Kapitola 6

Závěr

Cílem této práce bylo seznámení s programovacím jazykem Julia a porovnání jejího výkonu s dalšími vybranými jazyky. Pro porovnání výkonu byla vytvořena testovací sada snažící se pokrýt co nejvíce aspektů daného jazyka. Jako jazyky pro porovnání byly zvoleny Python a jazyk C.

V teoretické části této práce je popsaná základní syntaxe jazyka Julia, její implementace a použití. Obsahuje také základní informace o Pythonu a jazyku C, včetně některých vybraných rozdílů, jak implementačních, tak syntaktických.

V experimentální části se nachází popis testovacího prostředí, obecné způsoby tohoto měření a výsledky. V popisu každého testu je uveden daný algoritmus (pseudokódem nebo slovně). Výsledky dané testu jsou zobrazeny v grafu a popsány slovně. Na závěr jsou zde souhrné výsledky všech testů.

Julia se osvědčila jako rychlý programovací jazyk pro numerické výpočty. Kdo potřebuje maximální výkon, tak pro něj je stále nejlepší volbou jazyk C, ale pokud potřebujete vyšší míru abstrakce a nevadí vám o něco pomalejší běhový čas, tak je Julia ideální volbou. Nevýhodou může být menší přístupnost jazyka, vzhledem k jeho malé rozšířenosti. V této práci provádím jen výkonové porovnání, což nepokrývá všechny aspekty jazyka. Vzhledem, k tomu, že v Julii se nejedná o klasický třídní model a tudíž na ní nelze aplikovat klasické návrhové vzory, tak další výzkum by se proto mohl zabývat použitím Julie ve větších projektech.

Literatura

[1] BENDERSKY, E.: Memory layout of multi-dimensional arrays. September 2015, [cit. 2016-05-11].
URL http://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/

[2] BEZANSON, J.; CHEN, J.; KARPINSKY, S.; aj.: Array operators using multiple dispatch: a design methodology for array implementations in dynamic languages. In *ARRAY'14 Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, New York, NY, USA: ACM, 2014, s. 56–61, doi:10.1145/2627373.2627383, 1407.3845.

[3] BEZANSON, J.; EDELMAN, A.; KARPINSKY, S.; aj.: Julia: A Fresh Approach to Numerical Computing. November 2014, 1411.1607.

[4] BEZANSON, J.; KARPINSKY, S.; SHAH, V.; aj.: Julia Language Documentation. May 2016, [cit. 2016-05-02].
URL https://media.readthedocs.org/pdf/julia/latest/julia.pdf

[5] CHEN, J.; EDELMAN, A.: Parallel Prefix Polymorphism Permits Parallelization, Presentation & Proof. In *HPTCDL'14 Proceedings of the 1st Workshop on High Performance Technical Computing in Dynamic Languages*, New York: ACM, 2014, s. 47–56, doi:10.1109/HPTCDL.2014.9, 1410.6449.
URL http://jiahao.github.io/parallel-prefix

[6] DOWNEY, A.: *Think Python*. Needham, Massachusetts: Green Tea Press, 2012.
URL http://www.greenteapress.com/thinkpython/thinkpython.pdf

[7] Foundation, P. S.: 16.3. time - Time access and conversions. January 2016, [cit. 2016-05-05].
URL https://docs.python.org/3.4/library/time.html

[8] KERNIGHAN, B. W.; RITCHIE, D. M.: *The C programming language. 2nd ed.* Englewood Cliffs, N.J.: Prentice Hall, 1988, iSBN 0131103628.

[9] KERRISK, M.: `CLOCK_GETRES(2)`. December 2015, [cit. 2016-05-05].
URL http://man7.org/linux/man-pages/man2/clock_gettime.2.html

[10] KERRISK, M.: TIME(7). March 2016, [cit. 2016-05-05].
URL http://man7.org/linux/man-pages/man7/time.7.html

[11] LUCE, L.: Python list implementation. March 2011, [cit. 2016-05-11].
URL http://www.laurentluce.com/posts/python-list-implementation/

[12] MAHER, M. C.; HERNANDEZ, R. D.: CauseMap: Fast inference of causality from complex time series. 2015, doi:10.7287/peerj.preprints.583v2, 3:e1053.

[13] REITZ, K.: Picking an Interpreter. 2016, [cit. 2016-05-15].
URL http://docs.python-guide.org/en/latest/starting/which-python/

[14] STOR, N. J.; SLAPNICAR, I.: Forward stable computation of roots of real polynomials with only real distinct roots. 2015: s. 1–15, 1509.06224.

[15] TOWNSEND, A.; OLVER, S.: The automatic solution of partial differential equations using a global spectral method. 2014, 1409.2789.