VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

OPTIMALIZACE RYCHLOSTI VÝPOČTU KNIHOVNY PETNETSIM
PETNETSIM LIBRARY COMPUTATION PERFORMACE OPTIMIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Vojtěch Dražka
AUTHOR

VEDOUCÍ PRÁCE Ing. Ladislav Dobrovský
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Vojtěch Dražka**
Studijní program:	Aplikovaná informatika a řízení
Studijní obor:	bez specializace
Vedoucí práce:	**Ing. Ladislav Dobrovský**
Akademický rok:	2021/22

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Optimalizace rychlosti výpočtu knihovny PetNetSim

Stručná charakteristika problematiky úkolu:

Pro simulaci diskrétních systémů pomocí Petriho sítí je na VUT FSI vyvynuta knihovna PetNetSim. Současná implementace v čistém Pythonu s částečným využitím knihovny NumPy trpí poměrně nízkým výkonem. Vzhledem k nutnosti výpočtů více běhů pro statistiku stochastických jevů je i pro menší sítě motivace výkon zvýšit.

Cíle diplomové práce:

– Rešerše urychlení jazyka Python pomocí technik transkompilace, JIT anebo rozšiřujících modulů.
– Porovnání rychlosti alternativních interpretů a JIT (PyPy, Cinder, Pyston).
– Možnosti vývoje výpočetního jádra jako modul v systémovém jazyce (C, C++, Rust).
– Implementace zvoleného řešení.

Seznam doporučené literatury:

CRAPÉ, Arthur a Lieven EECKHOUT. A Rigorous Benchmarking and Performance Analysis Methodology for Python Workloads. In: IEEE International Symposium on Workload Characterization (IISWC). 2020, s. 83-93. ISBN 978-1-7281-7645-1. Dostupné z: doi:10.1109/IISWC50251.2020.00017.

Pybind11: Seamless operability between C++11 and Python [online]. 2021 [cit. 2021-10-21]. Dostupné z: https://github.com/wjakob/pybind11.

PyO3: Rust bindings for Python [online]. 2021 [cit. 2021-10-21]. Dostupné z: https://github.com/PyO3/pyo3.

PyPy: Features [online]. 2021 [cit. 2021-10-21]. Dostupné z: https://www.pypy.org/features.html

Cinder. [online]. [cit. 2021-10-21]. Dostupné z: https://github.com/facebookincubator/cinder.

DOBROVSKÝ, Ladislav: Simulační nástroj PetNetSim, dostupné z https://github.com/karna48/petnetsim.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2021/22

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

ABSTRAKT

Cílem diplomové práce byla optimalizace rychlosti výpočtu knihovny PetNetSim, která je momentálně implementovaná v jazyce Python. Smyslem práce bylo přistoupit k optimalizaci ze širšího hlediska a pokusit se Python urychlit pomocí různých technik. Práce se zabývá průzkumem dostupných řešení ve formě alternativních interpretů, rozšiřujících modulů, metody transkompilace a možnostmi vývoje výpočetního jádra coby modulu napsaném v systémovém jazyce. Praktická část popisuje snahu o implementaci těchto metod a jejich porovnání. Dále je implementována metoda multiprocessingu a nastíněn postup při vývoji výpočetního jádra coby rozšiřujícího modulu.

ABSTRACT

The aim of the thesis was to optimize the computation speed of the PetNetSim library, which is currently implemented in Python. The purpose of the thesis was to approach the optimization from a broader perspective and try to speed up Python using various techniques. The thesis explores the solutions available in the form of alternative interpreters, extending modules, trans-compilation methods and the possibility of developing the computational core as a module written in the system language. The practical part describes efforts to implement these methods and their comparison. Furthermore, the multiprocessing method is implemented and the procedure for developing the computational core as an extending module is outlined.

KLÍČOVÁ SLOVA

Optimalizace, alternativní interpret, rozšiřující modul, transpiler, tanskompilace, multiprocessing, Python, C++, Cython, PyPy, Pyston, Cinder, Pyjion

KEYWORDS

Optimization, alternative interpreter, extension module, transpiler, tanscompilation, multiprocessing, Python, C++, Cython, PyPy, Pyston, Cinder, Pyjion

ÚSTAV AUTOMATIZACE
A INFORMATIKY

2022

BIBLIOGRAFICKÁ CITACE

DRAŽKA, Vojtěch. Optimalizace rychlosti výpočtu knihovny PetNetSim. Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/139979. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky. Vedoucí práce Ladislav Dobrovský.

PODĚKOVÁNÍ

Tímto bych rád velice poděkoval svému vedoucímu práce Ing. Ladislavovi Dobrovskému za odborné rady a konzultace. Mé díky patří i celé mojí rodině, neboť nebýt jejího zázemí, nemohl bych dnes tato slova psát. Díky patří i všem mým přátelům, kteří jsou mi v životě oporou.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, vypracoval jsem ji samostatně pod vedením vedoucího práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury.

Jako autor uvedené práce dále prohlašuji, že v souvislosti s vytvořením této práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následku porušení ustanovení § 11 a následujících autorského zákona c. 121/2000 Sb., včetně možných trestně právních důsledků.

V Brně dne 20. 5. 2022 ……………………………………………………

Bc. Vojtěch Dražka

OBSAH

1 ÚVOD ... 15

2 PYTHON ... 17
2.1 Použití Pythonu ... 17
2.2 Slabiny Pythonu ... 17
2.3 Interpretace ... 17

3 URYCHLENÍ PYTHONU ... 19
3.1 Alternativní interpreti ... 19
3.1.1 JIT kompilace ... 19
3.1.2 Pyston ... 20
3.1.3 PyPy ... 20
3.1.4 Cinder ... 22
3.1.5 Jython ... 23
3.2 Rozšiřující moduly ... 24
3.2.1 Numba ... 25
3.2.2 Pyjion ... 27
3.3 Transpiler ... 28
3.3.1 Transpiler vs přepis kódu ... 28
3.3.2 Proč transpiler ... 28
3.3.3 Jak transpiler funguje ... 29
3.3.4 Cython ... 30
3.3.5 Pythran ... 32

4 IMPLEMENTACE JÁDRA V SYSTÉMOVÉM JAZYCE JAKO ROZŠIŘUJÍCÍ MODUL ... 35
4.1 Jazyk C a C++ ... 35
4.1.1 Boost.Python ... 36
4.1.2 Pybind11 ... 37
4.2 Jazyk Rust ... 38
4.2.1 PyO3 ... 38

5 METODIKA ... 41
5.1 Testované zrychlující metody: ... 41
5.2 Testované úlohy ... 41
5.3 Hardware & Software ... 41
5.4 Metodika testování a porovnání dat ... 42

6 IMPLEMENTACE ... 45
6.1 CPython ... 45
6.2 Pyston ... 46
6.3 PyPy ... 46
6.4 Cinder ... 48
6.5 Numba ... 49
6.6 Pyjion ... 50
6.7 Cython ... 51
6.8 Zhodnocení zrychlujících metod ... 54

7 OPTIMALIZACE STOCHASTICKÉ SÍTĚ ... 57
7.1 Základní stochastická síť ... 57

7.2 Rozsáhlejší stochastická síť 57
7.3 Multirpocessing 59

8 VÝVOJ MODULU V C++ 63

9 ZÁVĚR 67

10 BIBLIOGRAFIE 69

11 SEZNAM PŘÍLOH 73

1 ÚVOD

Petriho sítě jsou matematickým aparátem sloužícím k modelování a simulování diskrétních systémů (např. systémy hromadné obsluhy a další). Jednou z možností, jak tyto sítě modelovat, a tak tyto systémy simulovat, je knihovna PetNetSim, která je vyvíjena na Ústavu automatizace a informatiky na Fakultě strojního inženýrství v Brně. Tato knihovna, implementovaná v jazyce Python, ovšem trpí poměrně nízkým výpočetním výkonem, což je problém zejména v případě simulace stochastických jevů, u kterých je třeba provádět vyšší počet simulací pro statistické účely.

Abychom se ihned nemuseli uchýlit k poměrně razantnímu řešení, jako je přepisování knihovny do jiného jazyka, anebo úpravy kódu a algoritmů za účelem zvýšení výkonu této knihovny, je snaha se pokusit optimalizovat výpočetní rychlost alternativními metodami. Python totiž sám trpí poměrně nízkým výkonem a je snahou nemalé řady odborníků a vývojářů tuto nepříjemnou vlastnost co nejefektivněji zredukovat. Za tímto účelem bylo již vytvořeno několik alternativních interpretů jazyka Python, které si kladou za cíl být rychlejší než nativní interpret CPython. Dále existují nejrůznější rozšiřující moduly, jejichž implementace v kódu může mít značně pozitivní výsledek na celkovou rychlost programu, zejména pak v případě rozšiřujících modulů napsaných v jazyce C. Další alternativní možností, jak dosáhnout zvýšení výkonu, by mohla být kompilace programu (Python je jazyk interpretovaný). Poslední možností optimalizace se zdá být vývoj vlastního rozšiřujícího modulu napsaného v jednom ze systémových jazyků, který by sloužil jako výpočetní jádro našeho programu.

Cílem této práce byla rešerše alternativních interpretů, rozšiřujících modulů a tzv. transpilerů, které mají potenciál dosáhnout zvýšení výkonu knihovny PetNetSim, stejně jako prozkoumat možnost vývoje výpočetního jádra coby rozšiřujícího modulu napsaném v systémovém jazyce. Po zvážení možností bylo cílem implementovat zvolenou metodu za účelem urychlení výpočtů knihovny PetNetSim.

2 PYTHON

Programovací jazyk Python byl vytvořen v roce 1990 a jeho první verzi vytvořil holandský programátor Guido van Rossum [1]. Python je vysokoúrovňový, objektově orientovaný programovací jazyk s dynamickou typovou kontrolou, díky čemuž je využíván nejen jako skriptovací jazyk [2]. Python je uživatelsky přívětivý, má jednoduchou syntaxi, a je proto vhodný pro začátečníky.

2.1 Použití Pythonu

Programovací jazyk Python je dnes velice populární a používá se pro nejrůznější aplikace. Pomocí Pythonu dokážeme vytvořit jednoduché konzolové aplikace, ale také multiplatformní aplikace s grafickým prostředím [3]. Řada webových developerů dnes využívá Python nejen k tvorbě frontendu, ale i backendu. Velice silným a efektivním nástrojem se Python stává při aplikaci neuronových sítí a strojového učení. Python ovšem není všespásným řešením a pro některé aplikace není vhodný, ba dokonce nepoužitelný.

2.2 Slabiny Pythonu

Jelikož je Python vysokoúrovňový interpretovaný programovací jazyk, který využívá virtuálního stroje (viz kapitola 2.3), není zajisté nejvhodnější k ovládání hardwaru nebo k tvorbě ovladačů zařízení. Pro systémové aplikace a operační systémy je jeho použití absolutně nemyslitelné. Dalo by se tedy jednoduše říct, že Python není vhodné používat tam, kde je rychlost naší nejvyšší prioritou. Obecně vzato je Python pomalý jazyk, rozhodně v porovnání s jazyky C/C++. V dnešní době již však existují možnosti, jak Python urychlit.

2.3 Interpretace

Řada zdrojů [4] označuje Python za interpretovaný jazyk, dalo by se o něm ovšem říct, že je to také kompilovaný jazyk. Zdrojový kód je v interpretu kompilován do bajtkódu a následně provede jeho instrukce virtuálním strojem (VM), který v průběhu vykonání kódu deklaruje typy proměnných a překládá bajtkód do strojového kódu. Proces interpretace zachycuje následující schéma:

Obr. 1: Schéma interpretace Pythonu

Výhoda tohoto způsobu exekuce zdrojového kódu spočívá v tom, že je kód Pythonu multiplatformní, jelikož se o interpretaci pro potřebnou platformu postará interpret, respektive VM. Nevýhodou je ovšem fakt, že deklarace typů během exekuce kódu je výpočetně, a tedy i časově náročnější, než kdybychom spouštěli kompilovaný kód s datovými typy již deklarovanými, jako je např. jazyk C. [5]

3 URYCHLENÍ PYTHONU

Pokud je naším cílem zvýšit výkon Pythonu a urychlit tak náš program, existuje několik možností. Můžeme se pokusit použít jiný alternativní interpret než nativní CPython (kapitola 3.1), můžeme využít různé rozšiřující moduly, které nám urychlí části našeho kódu (kapitola 3.2), anebo využijeme metodu transpilace (kapitola 3.3), která převede náš program do jiného programovacího jazyka.

3.1 Alternativní interpreti

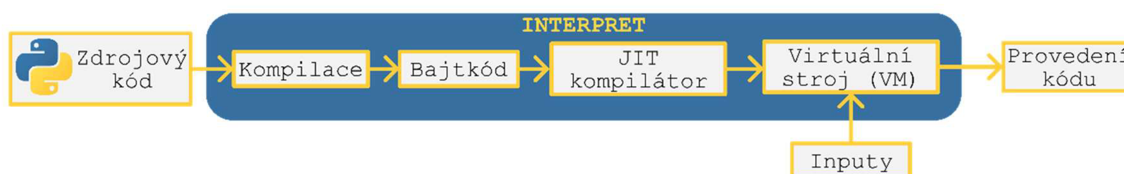
Tato část práce se věnuje popisu některých alternativních typů implementace jazyka Python. Představny jsou zde interpreti Pyston, PyPy, Cinder a Jython. Jelikož tři z nich využívají metodu JIT kompilace, představíme si tuto metodu samostatně a následně přikročíme již k alternativním interpretům.

3.1.1 JIT kompilace

JIT (Just-In-Time) je pojem známý spíše v ekonomii a logistice. [6] Jedná se o metodu, při které jsou potřebné materiály dodávány přesně v daný čas, kdy jich bude třeba (odtud pochází název JIT). Touto metodou eliminujeme hromadění materiálu a odpadu a snižujeme tak náklady na výrobu. Analogicky můžeme přemýšlet podobně i při vykonávání programů a procesů. Můžeme dodat veškerá potřebná data pro běh programu dopředu – AOT (Ahead-of-time) kompilace. Anebo je můžeme dodat až v momentě, kdy je program potřebuje – JIT kompilace.

Jak jsme si již vysvětlili v kapitole *2.3*, Python je interpretovaný jazyk, nikoli kompilovaný. JIT kompilace je jakýmsi skloubením obou světů interpretace a kompilace. Během procesu interpretace – tedy de facto při běhu (run-time) programu – JIT kompilátor pomocí různých analytických metod určuje, které části kódu jsou často využívané (např. cykly) a tyto kódové celky zkompiluje do strojového kódu. Tento proces má ten pozitivní výsledek, že se frekventované prvky kódu nemusí stále znovu překládat do strojového kódu, kdykoli je třeba je vykonat, ale jsou již díky JIT kompilátoru připravené. [7]

Pokud bychom poupravili schéma z ***Obr.1***, aby odpovídalo interpretu s implementovaným JIT kompilátorem, vypadalo by následovně:

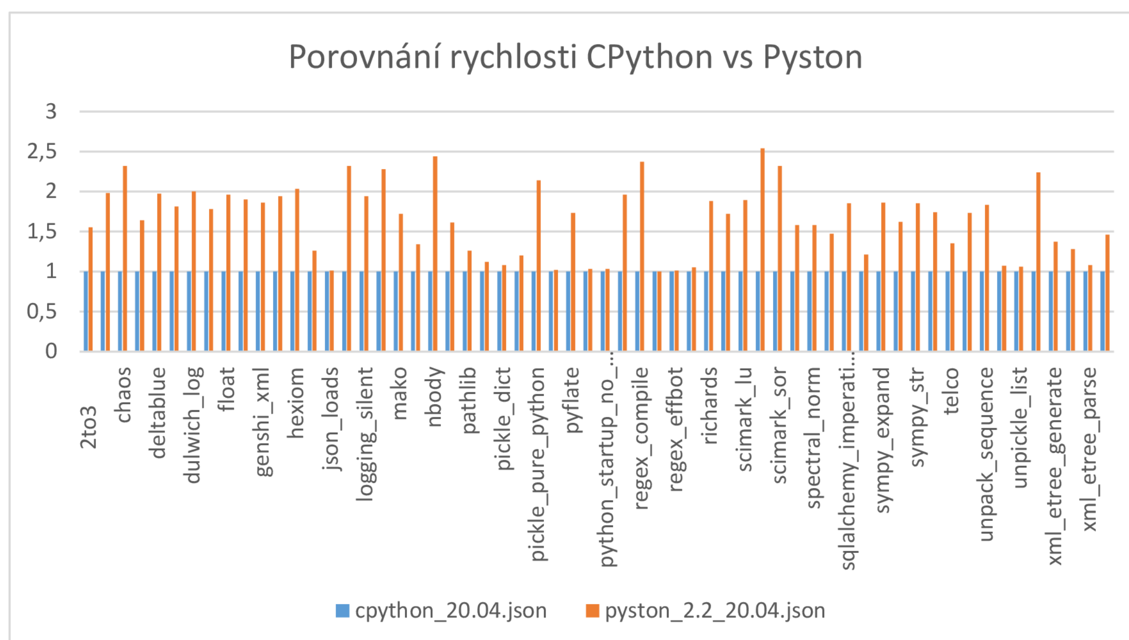


Obr. 2: JIT interpret

3.1.2 Pyston

Pyston je open-source implementace jazyka Python, která si klade za cíl být rychlejší než nativní CPython při vyvinutí minimálního úsilí ze strany uživatele. Na oficiálních stránkách se Pyston chlubí zrychlením implementace uživatelova kódu až o 30 % v případě webových stránek a jednodušší aplikace dokonce i více. Zároveň má být Pyston šetrnější ohledně požadavků na kapacitu serverů (snížení až o 25 %). [8]

Na stránkách GitHubu [9] prezentují developeři výsledky různých benchmarků, jež jsou zaneseny v grafu v následujícím obrázku 3. Z prezentovaných výsledků je vyextrahována hodnota relativního porovnání – kolikrát je Pyston rychlejší než CPython pro konkrétní test (vyšší hodnota je lepší). Graf má zde pouze ilustrativní účel.



Obr. 3: Porovnání rychlosti CPython vs Pyston

Pyston navazuje na CPython a využívá jeho optimalizační principy. Snaží se dosáhnout vylepšení a zrychlení oproti CPythonu pomocí metody JIT. Tvůrci také uvádí, že si dovolují být více „agresivní“ při procesu optimalizace oproti CPythonu, například větším vytížením „cache“ paměti za účelem zvýšení výkonu. Momentálně je Pyston podporován pouze na platformě Linux a podpora pro zařízení Mac a Windows je plánována pro nadcházející verze. [9]

3.1.3 PyPy

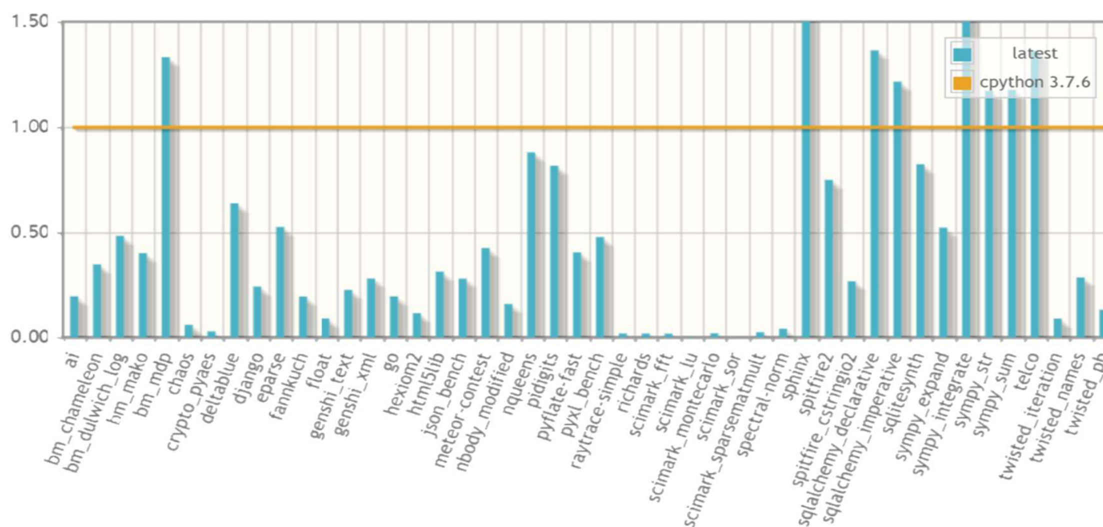
PyPy je jednou z dalších implementací jazyka Python, která je napsaná v RPythonu a která kompletně nahrazuje CPython. Účel je opět jasný – rychlost. PyPy implementuje Python 2.7.18 a Python 3.7.10. Stejně jako v případě implementace Pyston, PyPy dosahuje zrychlení metodou JIT. [10; 11]

PyPy využívá speciální metodu JIT kompilace, tzv. „meta-tracing JIT“ kompilaci. Tato metoda, na rozdíl od klasických JIT metod, neanalyzuje sémantiku samotného jazyka, aby v něm našla cykly a další opakující se sekvence programu. Místo toho analyzuje kód interpreta, který program vykonává. Tvůrci PyPy zvolili tuto metodu, aby se vyhnuli závislosti na programovacím jazyku. Aby tato metoda fungovala, bylo nezbytné vytvořit vlastního interpreta, který byl napsán v RPythonu (jedna z odnoží Pythonu). [12]

Jelikož je PyPy napsaný v RPythonu, PyPy jako interpret funguje nejlépe pro aplikace napsané čistě v Pythonu. Rozšiřující moduly napsané v jazyce C PyPy sice podporuje, nicméně dosahuje horších výsledků než při použití CPythonu. Řešením by mohlo být tyto moduly nahradit moduly napsanými v Pythonu, které by při použití CPythonu byly pomalé, v tomto případě by ovšem mohly dosahovat lepších výsledků. [11]

Stejně jako v případě dalších JIT interpretů, tato metoda dosahuje nejlepších výsledků pro dlouhé programy s větším množstvím cyklů. V případě krátkých programů bez cyklů nelze předpokládat zrychlení oproti použití nativní implementace CPython. [11]

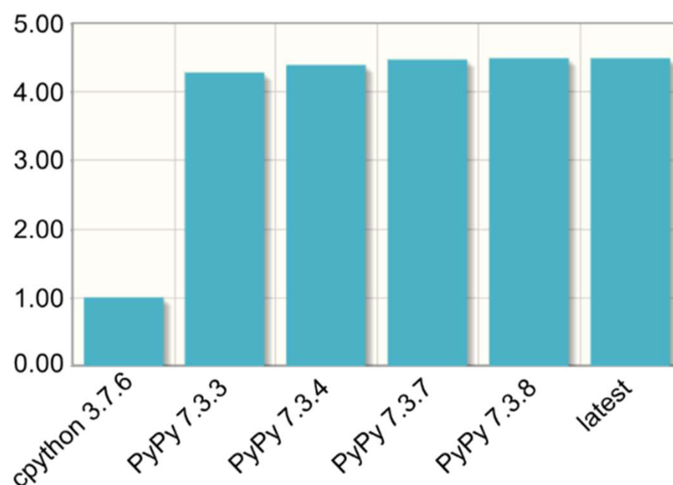
Na stránkách *speed.pypy.org* [13] lze vyčíst výsledky testů nejrůznějších programů. V některých případech je implementace PyPy takřka nepoužitelná (např. test „scimark_fft“ na Fourierovu transformaci a další, viz ***Obr.4***), v jiných případech (např. Sphinx – generátor dokumentace napsaný v Pythonu) dosahuje PyPy extrémně dobrých výsledků. Obrázek ***Obr.4*** zobrazuje porovnání časů (nižší hodnota je lepší) různých benchmarkových testů pro CPython a nejaktuálnější verzí PyPy.



Obr. 4: Porovnání časů testů pro PyPy a CPython [13]

Z ***Obr.4*** je zřejmé, že velice záleží na typu a rozsahu programu, která má PyPy optimalizovat. Geometrický průměr těchto testů znázorňuje, že lze s jistou mírou generalizace říct, že PyPy dosahuje cca 4.5 rychlosti CPythonu. Následující obrázek

Obr.5 znázorňuje geometrický průměr testů pro různé verze PyPy relativně k rychlosti CPython.



Obr. 5: Porovnání rychlostí PyPy vs CPython [13]

3.1.4 Cinder

Cinder je implementací jazyka Python navrženou speciálně pro aplikaci Instagram od společnosti Meta Platforms, Inc. (dříve Facebook, Inc.) a jedná se v podstatě o vylepšenou verzi CPythonu 3.8, která má za cíl zvýšení výkonu této nativní implementace. Tato implementace je na platformě GitHub veřejně dostupná, nejedná se ovšem o open-source platformu, a proto je její podpora ze strany jejích developerů značně omezená, nebo spíše – jak developeři sami uvádějí – žádná. Zveřejnění této platformy má sloužit pouze jako inspirace pro vylepšení nativního CPythonu a zamezení redundantní práce lidí pracujících na vylepšení výkonu CPythonu. [14]

Cinder dosahuje zvýšení výkonu několika způsoby. Mezi vyčtené patří „bytecode inline caching“, „eager evaluation of coroutines“ a „method-at-a-time JIT“.

- **Inline caching** [15] je metoda optimalizace rychlosti programu, která má za cíl ukládat do mezipaměti informace o datových typech a není proto nutné se na ně v každé instanci, kdy na ně program narazí, dotazovat. Tato metoda se v implementaci Cinder nazývá „*Shadowcode*“ [14] a pracuje tak, že analyzuje instrukční části strojového kódu (v publikacích často jako opcode – z ang. Operation code) a sekvence vhodné k optimalizaci nahrazuje speciálně navrženými částmi kódu.
- **Eager evaluation of corutines** se týká asynchronních funkcí. Jedná se o funkci, která je specifická dvojicí slov *async* a *await*. Tato funkce nám pomocí instrukce „await“ umožní vrátit řízení dalším procesům, takže je možné během doby čekání na vykonání dalších instrukcí asychronní funkce spracovávat další procesy programu. Příklad jednoduché asynchronní funkce napsané v Pythonu, která zobrazí pozdrav „Ahoj“ a o 5 sekund později zobrazí „kamaráde“ je na ***Obr.6***.

```
import asyncio

async def main():
    print("Ahoj")
    await asyncio.sleep(5)
    print("kamaráde")

asyncio.run(main())
```

Obr. 6: Příklad asynchronní funkce v Pythonu

V době 5 sekund se běh programu nepozastavuje a výpočetní čas je věnován dalším procesům programu. „Eager evaluation of corutines" analyzuje asynchrnonní funkce a v případě, že nalezne funkci, která dosáhne *return* bez potřeby *await*, je funkce okamžitě vykonána bez vytvoření obejktu corutine. Tato konkrétní optimalizace je vhodná zejména pro aplikaci Instagram, jejíž developeři hojně využívají asynchronní programování. V jejich případě ušetří optimalizace až 5% výkonu CPU.

- **Cinder JIT** – developeři Cinderu jejich JIT kompilátor nazývají method-at-a-time. Je napsán v C++ a podporuje téměř všechny instrukce Python bajtkódu. Podstata tohoto JIT kompilátoru je v tom, že v nativním režimu kompiluje všechny funkce programu. To může velice pravděpodobně způsobit, že náš program poběží pomaleji, neboť kompilátor dopředu kompiluje i funkce, které jsou volány jen zřídka nebo i jen jednou. Proto lze pomocí textového souboru jitlist.txt kompilátoru definovat, které funkce kompilovat má, ostatní nechá na interpretaci. Cinder JIT kompilátor je stále v počátcích vývoje a čeká ho mnoho vylepšení. I když už nabízí podstatné zlepšení v rychlosti (1,5-4x zvýšení rychlosti na mnoha výkonnostních benchmarcích), nabízí se stále mnoho optimalizačních možností a vylepšení.

Cinder coby alternativa k CPythonu je jistě nadějným kandidátem a jeho použití na celosvětově používané aplikaci Instagram je jistě dobrým předpokladem k výhodnosti jeho využívání. Momentálně je ovšem ve fázi interního vývoje a je vyvíjen a testován pouze na Linux x64.

3.1.5 Jython

Jython byl vytvořen v roce 1997 pod jménem JPython a jeho autorem je programátor Jim Hugunin. Barry Wasaw jej při vypuštění verze 2.0 v roce 1999 přejmenoval na Jython a tento název už zůstal. Jython je implementací jazyka Python na platformě Java a z toho vyplývá, že je tedy vhodný pro vývojáře, kteří chtějí nějakým způsobem implementovat jazyk Python při tvorbě systémů na platformě Java. [16]

Na rozdíl od dříve zmíněných alternativních interpretů, Jython dosahuje zvýšení efektivnosti jiným způsobem. Jython je velice vhodný pro následující úlohy [17]:

- **„Embedded scripting“** – programátoři mohou do svých Java systémů přidat knihovny napsané v Jythonu.
- **Interaktivní experimentování** – Jython disponuje interaktivním interpretem, s jehož pomocí lze interagovat s Java aplikacemi a programy. To umožňuje programátorům experimentovat a odstraňovat chyby jakýchkoli Java systémů pomocí Jythonu.
- **Rychlý vývoj aplikací** – Programy napsané v Pythonu jsou většinou podstatně kratší než programy se stejnou funkcí napsané v jiných jazycích. Jython umožňuje hladkou interakci mezi Javou a Pythonem, což může proces designu a vývoje systémů značně urychlit.

Jython bohužel nepodporuje moduly psané v C. Tyto moduly by bylo nutné upravit na moduly s JNI (Java Native Interface). Co se rychlosti oproti nativnímu CPythonu týče, záleží na JVM (Java Virtual Machine). Obecně se dá ale usoudit, že jsou si výkonově dosti podobné. [16]

Jelikož smyslem Jythonu není zvýšení výpočetního výkonu a urychlení Python programů, použití Jythonu pro náš účel nedává veliký smysl a tato kapitola má spíše informativní smysl, jelikož Jython by bylo jistě vhodné zmínit při výčtu alternativních interpretů.

3.2 Rozšiřující moduly

Rozšiřující moduly jsou ucelené kódové bloky, které lze do Pythonu (CPythonu) importovat a zahrnout je do našeho programu. Tyto rozšiřující moduly jsou převážně psané v C, C++ a dnes také často v jazyce Rust. Důvody pro použití rozšiřujících modulů jsou v podstatě dva: chceme-li použít existující knihovny nebo je třeba zvýšit výkon programu optimalizací funkcí a numerických operací. [18]

1. **Propojení s existujícími knihovnami:** V dnešní době je práce programátora závislá na používání knihoven všeho druhu, které mu značně urychlují tvorbu programů. Programátor si tak nemusí definovat základní matematické operace, interpretaci fyzikálních zákonů do simulovaného světa, práci s grafickým obsahem a spoustu dalších nástrojů, které se běžně používají. Abychom mohli v Pythonu používat knihovny napsané v C/C++, je nutné je s Pythonem propojit pomocí rozšiřujících modulů.
2. **Numerické operace a výkon:** Dalším důvodem, proč používat knihovny napsané v C/C++, je rychlost (jak již bylo nastíněno v druhé kapitole). Python může být v mnoha situacích dostatečně rychlý. Ovšem v momentě, kdy je rychlost a výkon zásadní a provádíme

operace s datovými poli (data arrays), je použití rozšiřujících modulů vhodné, ne-li nezbytné. Pravděpodobně nejpoužívanějším rozšiřujícím modulem pro numerické operace, zejména pak s maticemi, je rozšiřující modul *numpy*. Dále známe například scipy, pandas, numexpr, numba, a další.

Jelikož projekt, jehož optimalizace je jedním z cílů této práce, využívá rozšiřující modul NumPy, nebudeme si jej v této kapitole představovat, jelikož je již implementován a z hlediska další optimalizace by jeho rešerše byla zbytečná. Daleko zajímavější bude se blíže seznámit s rozšiřujícím modulem Numba a Pyjion.

3.2.1 Numba

Numba je just-in-time open-source kompilátor Python kódu. Nejlépe funguje pro cykly a stává se extrémně efektivním, používá-li program modul NumPy a jeho funkce a pole (arrays). Numba využívá dekorátor „*@jit*", kterým označíme funkce a části kódu, které má Numba optimalizovat. Numba bude JIT kompilovat danou část kódu do strojového kódu, přičemž zbytek programu poběží standardně. [19]

Numba je podporována pro:

- operační systémy Windows, OS X a Linux (32&64bit)
- Architekturu: x86, x86_64, ppc64le, experimentální na armv7l, armv8l
- GPU: Nvidia CUDA, experimentální na AMD ROCm
- CPython
- NumPy 1.15 nebo novější

Jak již bylo zmíněno v prvním odstavci, Numba je nejefektivnější při použití v programech, které hojně využívají cykly a s pomocí modulu NumPy provádí větší množství výpočtů, což je patrné z následující sekce *„Kód 3.1"*. Jde o příklad vhodného kódu, který autoři Numby prezentují ve své dokumentaci k tomuto modulu.

Kód 3.1: Numba – vhodný kód [19]

```
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent
to @njit
def go_fast(a): # Function is compiled to machine code when called the
first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

Naopak nevhodné je používat nepodporované moduly, jako např. *pandas*. Jelikož Numba nerozumí nepodporovaným modulům, nemůže jej optimalizovat, a proto nebude mít na výslednou rychlost programu žádný efekt. Autoři Numby uvádějí příklad kódu (*Kód 3.2*), kdy Numba nebude fungovat a její použití nemá smysl.

Kód 3.2: Numba – nevhodný kód [19]

```
from numba import jit
import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit
def use_pandas(a): # Function will not benefit from Numba jit
    df = pd.DataFrame.from_dict(a) # Numba doesn't know about pd.DataFrame
    df += 1                        # Numba doesn't understand what this is
    return df.cov()                 # or this!

print(use_pandas(x))
```

Dekorativní operátor „@jit“ operuje ve dvou módech – *nopython* a *object*. Nastavíme-li @jit(nopython=True) (viz ***Kód 3.1***), je to indikace pro Numbu, aby byla kompilace provedena zcela bez účasti interpreta Pythonu (CPythonu, nebo jiné alternativy, viz kapitola 3.1). Mód nopython lze také nastavit operátorem „@njit“ Pro dosažení nejlepších výsledků je doporučeno operovat v nopython módu.

V případě, že tento mód nenastavíme, nebo dojde k selhání při snaze jej použít, přepne se proces do *object* módu. V tomto módu Numba identifikuje cykly, které lze kompilovat a ty zkompiluje do strojového kódu. O chod zbytku programu se postará náš interpret Pythonu.

Výhoda modulu Numba spočívá v tom, že je z uživatelského hlediska snadno implementovatelná. Její instalaci lze provést v terminálu přes příkaz *„pip install numba“* a její použití pak přes dekorativní operátor @njit, jak již bylo zmíněno v předchozí části textu. Jelikož Numba dle dokumentace podporuje pouze CPython, nebude pravděpodobně možné ji kombinovat s nějakým alternativním JIT interpretem.

Zvýšení výkonu dost silně závisí na aplikaci. Velkou roli bude hrát i to, je-li možné problém paralelizovat (příkaz *@njit(parallel=True)*). Pokud ovšem můžeme předpokládat, že náš program obsahuje větší množství cyklů a jeho numerické operace jsou prováděny modulem NumPy, lze bezpečně předpovědět, že dosáhneme zvýšení výkonu o jeden až dva řády.

3.2.2 Pyjion

Dalším rozšiřujícím modulem z dílny společnosti Microsoft, který usiluje o zvýšení výkonu Pythonu pomocí technik JIT kompilace, je nástroj Pyjion [1]. Od ostatních nástrojů, jako je PyPy, Numba nebo Pyston se však liší zejména cílem, kterého chce dosáhnout.

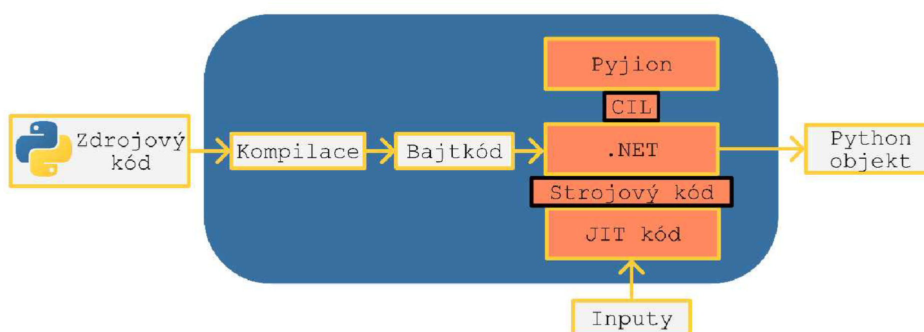
Vývojáři tohoto rozšíření totiž nechtějí být pouze konkurencí ve světě JIT kompilátorů pro Python. Jejich snahou je perfektní kompatibilita s nativním interpretem CPython, aby se v budoucnu mohl Pyjion stát součástí CPythonu a uskutečnit tak již dlouho toužené funkci nativního interpreta – možnosti JIT kompilace. [20]

Pro správné fungování Pyjion vyžaduje *CPython 3.10* a *.NET 6* [21]. Pokud máme tyto předpoklady splněné, lze Pyjion nainstalovat pomocí pip příkazu *„python -m pip install pyjion"*. Skript nebo program, který si přejeme optimalizovat, opatříme importem modulu *pyjion* a povolíme jej pomocí příkazu *„pyjion.enable()"*. Pyjion má 3 úrovně optimalizace (0-2) nastavitelné příkazem *„pyjion.config(level=2)"* a nativně používá druhou úroveň (úroveň 1). Pokud bychom si přáli testovat Pyjion na našem kódu s nastavením nejvyšší úrovně optimalizace, následující ukázkový kód znázorňuje, jak snadné to je:

Kód 3.3: Pyjion – aplikace

```
import pyjion;
pyjion.enable()
pyjion.config(level=2)

# náš kód, který si přejeme optimalizovat
```

Pyjion funguje tak, že nejprve vytvoří tabulku abstraktních typů v bajtkódu. V dalším kroku Pyjion kompiluje CPython bajtkód do IL (ECMA335 CIL)[2] instrukcí. Tento set instrukcí je následně zpracován .NET EE kompilátorem, který jej převádí do strojového kódu. Tento proces znázorňuje následující obrázek odvozený z www.trypyjion.com:



Obr. 7: Schéma funkce nástroje Pyjion [21]

---

[1] Výslovnost stejná jako u ang. „pigeon" - [pɪdʒ(ə)n]. [20]

[2] Standard definující infrastrukturu jazyků (Common Language Infrastructure – CLI) [38]. Více na stránkách *www.ecma-international-org.*

3.3 Transpiler

V následující kapitole si představíme pojem „*transpiler*“ — co tímto termínem rozumíme, jak funguje a proč jej využíváme. Pojem „transpiler“ zatím nemá český ekvivalent, a proto budeme v této práci pracovat s tímto termínem. Název „*transpiler*“ ovšem není jediný, se kterým je možné se setkat. Často můžeme narazit na pojmy jako „*source-to-source translator*“, „*source-to-source compiler*“, anebo „*transcompiler*“. Source-to-source translator je pravděpodobně nejvýstižnější název, jelikož přesně popisuje účel tohoto procesu. Pro stručnost ovšem budeme používat pojem *transpiler*, který je složeninou slov „*translator*“ a „*compiler*“. A co tedy transpiler je? Transpiler je program, který dokáže přeložit kód určitého programovacího jazyka do jiného programovacího jazyka [22], jako např. z Pythonu do C++.

3.3.1 Transpiler vs přepis kódu

Transpiler používáme tehdy, je-li náš záměr převést kód z jednoho programovacího jazyka do druhého. Možná nás napadne otázka, zdali není lepší kód napsat znovu v požadovaném jazyce. Na tuto otázku dává odpověď Joel Spolsky[3] ve svém článku „*Things You Should Never Do, Part I*“ [23]. Podle jeho slov je tento způsob práce ve většině případů nevhodný a firmy by se měly vyvarovat přepisování starých funkčních kódů a komplexních systémů. Sepsání fungujícího kódu je totiž časově náročný proces, při kterém developer tráví spoustu času vyhledáváním a opravováním „bugů“ — chyb v kódu. Přepisem kódu tak často ztrácíme znalosti o těchto chybách a celý proces je tak opět velice časově náročný, což ve výsledku může stát firmu nemalé částky a především riskují, že zůstanou pozadu za konkurencí. V neposlední řadě, píše Joel Spolsky, nemáme jistotu, že na konci tohoto procesu budeme mít k dispozici lepší řešení, než bylo to původní.

Je dobré zmínit, že v konkrétních případech může být skutečně výhodnější začít psát kód znovu, zejména v případě jednoduchých aplikacích. Ve většině případů je ale opravdu bezpečnější zvolit nějakou metodu transpilace, které v dnešní době dosahují slušných výsledků za vyvinutí minimálního úsilí. Různé metody transpilace budou představeny v pozdější kapitole této práce.

3.3.2 Proč transpiler

Dříve, než se pustíme do používání transpileru je vhodné si stanovit naši motivaci k jeho použití — proč chceme přepsat náš kód do jiného programovacího jazyka. Důvodů může být několik [22] :

1) **Migrace:** Máme-li kód, který chceme přenést do jiného jazyka, který má pro danou aplikaci lepší vlastnosti (např. disponuje větším množstvím knihoven) nebo který je uživatelsky přívětivější.

---

[3] Joel Spolsky je softwarový developer a v letech 2010-2019 působil jako CEO firmy Stack Overflow. Nyní zastává místo předsedy rady firmy Glitch.

2) **Kompatibilita:** Chceme kód převést pro jiný systém, nebo přecházíme na jinou verzi jazyka či standardu (např. z Python 2 do Python 3, ISO C++ 09/11/14/17/20…).

3) **Znalost jazyka:** Můžeme se ocitnou v situaci, kdy máme zdrojový kód v jazyce, který se dnes málo používá nebo je zastaralý a pro další práci s ním by bylo neefektivní zůstat u původního jazyka. V tomto případě je jistě snazší použít transpiler než najmout osobu specializovanou v tomto jazyce.

4) **Výkon:** Případ, kdy implementace jazyka nebo běhové prostředí (runtime) nedosahuje požadovaného výkonu a je tak motivací výkon aplikace zvýšit, je pravděpodobně jedním z hlavních důvodů, proč transpiler použít.

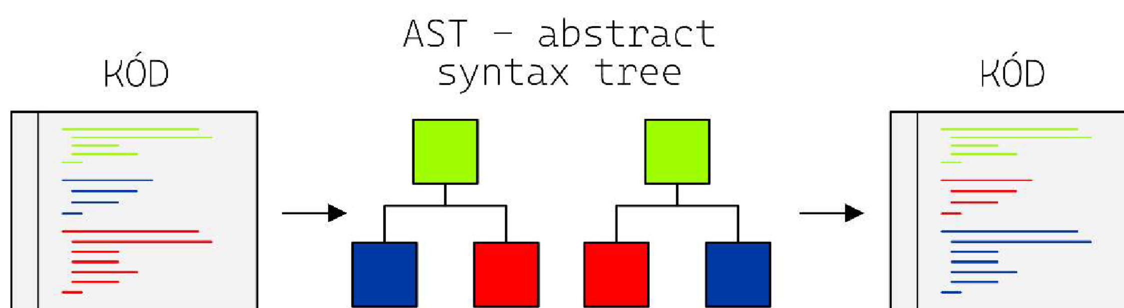
Náš případ je bod číslo 4, což vyplívá ze zadání a podstaty této práce. Python je mocný a uživatelsky přívětivý programovací jazyk a díky tomu je dnes hojně rozšířený jak mezi odborníky, tak i laiky a začátečníky. Jeho nevýhoda je ovšem jeho poměrně nízký výkon.

3.3.3 Jak transpiler funguje

Jak již víme, transpiler přeloží kód z původního programovacího jazyka do jiného. Jak k takovému úkolu ovšem přistoupit? Pojďme si stručně představit, jak tento proces funguje.

Úloha transpilace je řešena vytvořením abstraktního syntaktického stromu (AST) ze zdrojového kódu tak, že transpilátor rozebere zdrojový kód na elementární prvky jazyka – operátory, proměnné a klíčová slova jazyka. AST je reprezentací syntaxe zdrojového kódu, kde vnitřní uzly stromu jsou operátory a listy stromu jsou operandy. K jeho vytvoření je nutné znát gramatiku a syntaxi zdrojového programovacího jazyka.

Z tohoto stromu lze odvodit strom pro cílový programovací jazyk, opět za podmínky znalosti gramatiky a syntaxe cílového programovacího jazyka. Schéma tohoto procesu zobrazuje ***Obr.8***, který je odvozen z článku na webu *sitepoint.com* [24].



Obr. 8: Schéma transpileru [24]

3.3.4 Cython

Cython je programovací jazyk podobný jazyku Python, který vychází z jazyka Pyrex[4]. Cython je ale zároveň i kompilátorem, respektive *transpilerem*, neboť zdrojový kód přeloží do jazyka C/C++ a zkompiluje. To činí ze Cythonu mocný nástroj, který umožňuje psát programy (rozšiřující moduly) stejně pohodlně, jako jazyk Python, ale poskytuje výrazně rychlejší exekuci než nativní Python. Psaní rozšiřujících modulů v C pro Python totiž není triviální záležitostí a vyžaduje znalost Python/C API (Application Programming Interface). Cython tuto znalost nevyžaduje a Cython kód překládá do jazyka C/C++ a následně jej zkompiluje, aby jej bylo možné použít jako rozšiřující modul pro Python. [25; 26]

Jelikož je Cython odnoží Pythonu, je tak program napsaný v Pythonu více méně validním Cython kódem, a tak lze neupravený Python kód pomocí Cythonu kompilovat a již v této fázi dosáhnout znatelných zlepšení. Cython ale dále na rozdíl od Pythonu pracuje s několika operátory, které umožní lepší optimalizaci kódu. Jedním z nich je *cdef*, který umožní statické typování proměnných. [25]

Sestavení modulu

Na rozdíl od Pythonu, Cython musí být kompilován [27]. Soubor s příponou „*.py*" je Cythonem přeložen do jeho ekvivalentu v jazyce C. Výsledná soubor tak bude mít příponu „*.c*". Tento soubor obsahuje rozšiřující modul pro Python. Nyní jej musíme převést na takový soubor, aby jej mohl Python bez problému importovat. O to se postará modul setup.py (viz také ***Kód 4.3***). Představíme si jednoduchá příklad na funkci, která pozdraví uživatele po zadání jeho jména. ***Kód 3.3*** odpovídá sestrojení této funkce, pomocí ***Kód 3.4*** sestavíme rozšiřující modul.

Sestavenou funkci uložíme do souboru, např. „hello.py"

Kód 3.3: Vytvoření funkce

```
def say_hello_to(name):
    print("Hello %s!" % name)
```

Následně importujeme nástroj setup.py a cythonize.py. Pomocí příkazu *cythonize* zkompilujeme náš soubor „hello.py" do jazyka C/C++. Pro funkci setup vypíšeme její příslušené atributy a díky ní sestavíme náš rozšiřující modul.

Kód 3.4: Sestavení modulu

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name='Hello world app',
    ext_modules=cythonize("hello.py"),
    zip_safe=False,
)
```

[4] Pyrex je programovací jazyk příbuzný jazyku Python. Pyrex umožňuje kombinovat jazyk Python s datovými typy jazyka C [37].

Statické typování

Jak již bylo řečeno, Cython je kompilátorem, který dokáže kompilovat Python kód bez potřeby jakýchkoli úprav. Je-li naším cílem optimalizovat kód co nejlépe, Cython podává nejlepší výsledky, opatří-li uživatel kritické části kódu (numerické výpočty a cykly) statickými datovými typy, což může mít značný vliv na výslednou rychlost programu. K dispozici jsou všechny typy jazyka C. Tyto typy jsou deklarovány pomocí operátoru *cdef.* [28]

Uveďme si na příkladu, jak vypadá nativní kód funkce v Pythonu (***Kód 3.5***) v porovnání s funkcí napsanou v Cythonu (***Kód 3.6***) s deklarovanými typy. Jedná se o aplikaci, která integruje funkci x^2-x.

Kód 3.5: Funkce v Pythonu [28]

```
def f(x):
    return x ** 2 - x

def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

Pokud bychom funkci v kódu *3.5* pomocí Cythonu kompilovali, dosáhli bychom navýšení výkonu o cca 35 % [28].

Následující sekce ***Kód 3.6*** znázorňuje kód jazyka Cython, který pomocí operátoru *cdef* definuje datové typy jazyka C. Vidíme ale, že zbytek kódu je prakticky totožný s kódem jazyka Python.

Kód 3.6: Funkce v Cythonu [28]

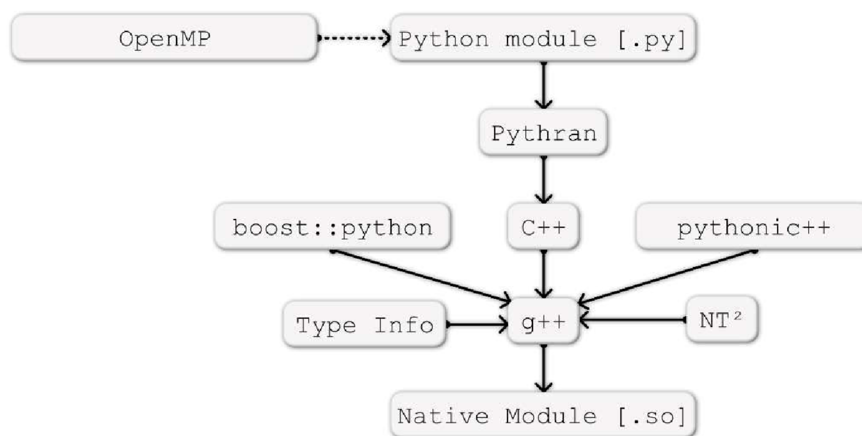
```
def f(double x):
    return x ** 2 - x

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s
    cdef double dx
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

Jelikož je iterační proměnná deklarována jako typ jazyka C, může Cython cyklus *for* zkompilovat. Stejně tak je důležité otypovat proměnné *a*, *s* a *dx*, jelikož se vyskytují uvnitř smyčky. Provedením této optimalizace dosáhneme zlepšení výkonu až o 400 %. [28]

3.3.5 Pythran

Pythran je dopředný kompilátor jazyka Python, jehož cílem je především optimalizace vědeckých programů a výpočtů. V současné verzi Pythran podporuje pouze Python 3. [29]

Pythran funguje tak, že vezme náš kód napsaný v Pythonu, který je nutné opatřit několika prvky Pythranu (viz dále). Pythran tento kód převede na kód C++, implementuje do něj potřebné knihovny a zkompiluje. Výsledkem je pak modul, který můžeme importovat do Pythonu. S minimálním úsilím tak dosáhneme zrychlení našeho programu. [29] Schéma tohoto procesu znázorňuje následující obrázek:



Obr. 9: Pythran schéma [30]

Autor projektu Serge Guelton ve své prezentaci na konferenci SciPy v roce 2013 představil svůj projekt veřejnosti a s ním uvedl i několik příkladů, jak jej použít. Jedna z demonstrací fungování Pythranu byla prezentována na Rosenbrockově funkci. ***Kód 3.7*** znázorňuje tuto funkci implementovanou v Pythonu:

Kód 3.7: Funkce v Pythonu [30]

```
import numpy as np
def rosen(x)
     return sum(100.*(x[1:]-x[:-1]**2.)**2.+(1-x[:1])**2.)
```

Aby mohl Pythran tuto funkci úspěšně optimalizovat, je třeba Pythranu deklarovat, s jakými datovými typy bude funkce pracovat. V tomto případě je to typ float. Tuto skutečnost definujeme pomocí příkazu *„#pythran export rosen(float[ ])“*. Tento příkaz vložíme kamkoli do našeho kódu. Pro přehlednost je vhodné jej umístit poblíž funkce, kterou si přejeme optimalizovat, viz ***Kód 3.8*** na následující stránce:

Kód 3.8: Pythran optimalizace [30]

```
import numpy as np
#pythran export rosen(float[]).
def rosen(x)
     return sum(100.*(x[1:]-x[:-1]**2.)**2.+(1-x[:1])**2.)
```

Tento kód nám vygeneruje rozšiřující modul *rosen.so*, který je následně možné importovat v Pythonu, nebo jej přímo spustit. Tato optimalizace dosáhla téměř třináctinásobného zvýšení výkonu oproti původnímu kódu.

4 IMPLEMENTACE JÁDRA V SYSTÉMOVÉM JAZYCE JAKO ROZŠIŘUJÍCÍ MODUL

Další možností, jak zvýšit výkon celého systému PetNetSim, je vytvořit jeho výpočetní jádro v jednom ze systémových jazyků. V této kapitole budou nastíněny možnosti, jak jádro napsané v různých systémových jazycích implementovat do Pythonu. Budeme se zabývat systémovými jazyky C, C++ a Rust.

4.1 Jazyk C a C++

CPython velice často využívá rozšiřující moduly napsané v C a C++. Podrobně popsat postup a metody tvorby těchto modulů by bylo nad rámec této práce. Přesto si na jednoduché funkci alespoň nastíníme, co tvorba vlastního modulu v jazyce C obnáší.

- Při tvorbě rozšiřujícího modulu je základem použít knihovnu „Python.h“. Ta obsahuje objekty jako *PyObject*, *Py_None*, *PyMethodDef*, *PyModuleDef* a další [31].
- Nyní můžeme začít psát naši funkci, v našem případě to bude funkce, která zobrazí „Hello, world“, viz ***Kód 4.1***:

Kód 4.1: Tvorba funkce v C [32]

```
#include <Python.h>

static PyObject* helloworld(PyObject* self, PyObject* args)
{
    printf("Hello World\n");
    return Py_None;
}
```

- Funkce je tímto hotová, proto postupujeme definicí metody a přidělením atributů našemu rozšiřujícímu modulu, viz ***Kód 4.2:***

Kód 4.2: Definice metody a modulu [32]

```
static PyMethodDef myMethods[] = {
    { "helloworld", helloworld, METH_NOARGS, "Prints Hello World" },
    { NULL, NULL, 0, NULL }
};
static struct PyModuleDef myModule = {
    PyModuleDef_HEAD_INIT,
    "myModule",
    "Test Module",
    -1,
    myMethods
};
PyMODINIT_FUNC PyInit_myModule(void)
{
    return PyModule_Create(&myModule);
}
```

- V tomto momentě ale ještě nemáme hotovo. Dříve, než budeme moci modul pomocí Pythonu importovat, je nutné jej nejdříve sestavit. K tomu slouží modul setup.py (viz ***Kód 4.3***), pomocí kterého definujeme název modulu, verzi a případně další atributy (jméno autora, kontaktní email autora, …).

Kód 4.3: Python setup.py [32]

```
from distutils.core import setup, Extension
setup(name = 'myModule', version = '1.0',  \
   ext_modules = [Extension('myModule', ['test.c'])])
```

- V dalším kroku (***Kód 4.4***) můžeme modul sestavit a nainstalovat

Kód 4.4: Python – sestavení & instalace [32]

```
$ python3.6 setup.py build
$ python3.6 setup.py install
```

- Pokud nám během sestavování a instalace nevyskočila žádná varovná zpráva, můžeme modul importovat a spustit funkci na vypsání našeho textu, viz ***Kód 4.5***:

Kód 4.5: Spuštění modulu v Pythonu [32]

```
>>> import myModule
>>> myModule.helloworld()
Hello World
```

Kód 4.1-4.5 zde mají spíše ilustrativní účel, neboť k plnému pochopení problematiky a schopnosti implementace je nutné investovat nemalé množství času. Jako schůdnější varianta se jeví použití knihovny Boost.Python, nebo knihovny PyBind11, viz dále.

4.1.1 Boost.Python

Boost.Python je součástí open-source knihovny Boost pro jazyk C++, která umožňuje provázání C++ kódu s kódem Pythonu a naopak. V konečném důsledku tak umožňuje integraci C++ tříd a funkcí do Pythonu, což je výhodné nejen z hlediska urychlení vývoje systémů a programů, ale může mít i pozitivní vliv na jejich rychlost. Knihovna Boost.Python je k dispozici zdarma i pro komerční užití. [33]

Na stránkách *www.boost.org* je uveden jednoduchý příklad, jak můžeme funkci napsanou v C++ použít v Pythonu:

- **Kód 4.6:** V C++ napíšeme funkci, v tomto případě nám funkce opět zobrazí „hello, world“:

Kód 4.6: C++ funkce [33]

```
char const* greet()
{
   return "hello, world";
}
```

- **Kód 4.7:** Pomocí Boost.Pythonu vytvoříme tzv. wrapper, který vygeneruje rozšiřující modul pro Python.

Kód 4.7: Boost.Python wrapper [33]

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
    def("greet", greet);
}
```

- **Kód 4.8:** V Pythonu načteme rozšiřující modul, který používá funkci napsanou v C++.

Kód 4.8: Python terminál [33]

```
>>> import hello_ext
>>> print hello_ext.greet()
hello, world
```

Podobným stylem lze provázat téměř jakékoli prvky C++ kódu s Pythonem a využívat tak silných stránek obou programovacích jazyků. Pro implementaci tohoto řešení je nutná C++ knihovna Boost, dostupná z *https://github.com/boostorg*.

4.1.2 Pybind11

Účel knihovny Pybind11 je stejný, jako v případě knihovny Boost.Python – umožnit komunikaci mezi C++ a Pythonem. Pybind11 navazuje svojí filozofií a provedením na knihovnu Boost.Python. Nevýhoda knihovny Boost.Python je ovšem v knihovně Boost. Knihovna Boost je značně rozsáhlá, jelikož se snaží být kompatibilní se všemi dostupnými kompilátory C++. Pybind11 je tak v podstatě odlehčenou verzí Boost.Python, která je zbavena všech neesenciálních prvků pro komunikaci mezi dnes rozšířeným C++11 a Pythonem. Implementace této knihovny je tedy dosti podobná, jako u knihovny Boost.Python. [20]

Uvedeme si opět krátký příklad, jak použití knihovny Pybind11 vypadá v praxi:

- **Kód 4.9:** V C++ definujeme funkci, která sčítá 2 uživatelem zadaná čísla. Pomocí knihovny Pybind11 vytvoříme rozšiřující modul pro Python.

Kód 4.9: C++ funkce & Pybind11 [21]

```
#include <pybind11/pybind11.h>
int add(int i, int j) {
    return i + j;
}
PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function that adds two numbers");
}
```

- **Kód 4.10:** V Pythonu importujeme vytvořený modul a spustíme funkci sčítání.

Kód 4.10: Python terminál [21]

```
>>> import example
>>> example.add(1, 2)
3
```

4.2 Jazyk Rust

[34] Programovací jazyk Rust sdílí mnoho podobností s jazykem C a C++. Rust je nižší programovací jazyk, který se vyvinul za účelem zvýšení bezpečnosti při přístupu k paměti, což je jeden z problémů jazyka C a C++. Rust také řeší problémy s paralelním zpracováním dat, kdy se část paměti snaží zaráz využít více procesů. Chceme-li tedy vyvíjet systém, který klade důraz na stabilitu a bezpečnost z hlediska operací s pamětí, ale nechceme dělat kompromisy ohledně výkonu, je Rust vhodným kandidátem. Ačkoli je Rust relativně novým programovacím jazykem, těší se nemalé oblibě nejen mezi studenty a developery, ale implementují jej i korporace jako Amazon nebo Figma.

4.2.1 PyO3

[35]Pro snadnou implementaci programů napsaných v jazyce Rust do Pythonu lze použít PyO3. Stejně jako v případě Boost.Python a Pybind11, PyO3 slouží ke generování nativních rozšiřujících modulů pro Python.

K tvorbě modulů pro Python s minimální potřebou konfigurace slouží nástroj *maturin*. Jako příklad uvedeme tvorbu modulu, který sečte hodnoty dvou textových řetězců. Postupujeme následovně [35]:

- **Kód 4.11:** Vytvoříme novou složku, jejíž název bude odpovídat názvu modulu. Složka bude obsahovat nové virtuální prostředí Pythonu a nainstalujeme do ní nástroj maturin.

Kód 4.11: Příprava nástroje maturin

```
# (string_sum odpovídá názvu modulu)
$ mkdir string_sum
$ cd string_sum
$ python -m venv .env
$ source .env/bin/activate
$ pip install maturin
```

- Nástroj *maturin* vygeneruje dva důležité soubory: *„Cargo.toml“* a *„lib.rs*, které budou vypadat zhruba tak, jako v kódu **4.12** a **4.13** na další stránce.
- V souboru *Cargo.toml* do položky „name“ pod sekcí [package] vyplníme název modulu. Položka „name“ pod sekcí [lib] odpovídá názvu knihovny. Pokud jej změníme, je nutné název změnit i v souboru *lib.rs*. Tento název

bude později použit k importování modulu v prostředí Python (import string_sum).

Kód 4.12: Cargo.toml

```
[package]
name = "string_sum"
version = "0.1.0"
edition = "2018"

[lib]
name = "string_sum"
crate-type = ["cdylib"]

[dependencies]
pyo3 = { version = "0.16.2", features = ["extension-module"] }
```

- V souboru *lib.rs* definujeme naši funkci k sečtení hodnot dvou textových řetězců. K označení této funkce slouží makro *#[pyfunction]*.
- V další části pod makrem *#[pymodule]* definujeme funkci k vytvoření modulu. Název této funkce musí být stejný, jako název v sekci [lib] v souboru Cargo.toml.

Kód 4.13: lib.rs

```
use pyo3::prelude::*;

#[pyfunction]
fn sum_as_string(a: usize, b: usize) -> PyResult<String> {
    Ok((a + b).to_string())
}

#[pymodule]
fn string_sum(_py: Python<'_>, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum_as_string, m)?)?;
    Ok(())
}
```

- Nyní máme vše připraveno k vygenerování modulu spustitelného pomocí Pythonu. Zbývá tedy jen spustit příkaz *maturin develop*, který modul vygeneruje a nainstaluje jej do našeho virtuálního prostředí Pythonu, které jsme vytvořili v prvním kroku. Po úspěšné instalaci můžeme náš rozšiřující modul otestovat jeho spuštěním. Tyto závěrečné kroky popisuje ***Kód 4.14***:

Kód 4.14: Sestavení & test modulu

```
$ maturin develop
# zobrazí se hlášky během procesu sestavení modulu
$ python
>>> import string_sum
>>> string_sum.sum_as_string(5, 20)
'25'
```

5 METODIKA

Dříve než přistoupíme k vlastní implementaci zvoleného řešení, popíšeme si základní principy při testování výkonu knihovny PetNetSim v původním a modifikovaném provedení a porovnání výkonu modifikované knihovny s původní knihovnou.

5.1 Testované zrychlující metody:

Vzhledem ke zkušenostem a možnostem autora práce byly zvoleny následující zrychlující metody určené k testování v praktické části této práce:

- **Alternativní interpreti:** Pyston, PyPy, Cinder
- **Rozšiřující moduly:** Numba, Pyjion
- **Transpiler:** Cython

Součástí testování je i nativní implementace CPython sloužící k referenčnímu porovnání výkonu zrychlujících metod.

5.2 Testované úlohy

Knihovna PetNetSim [36] obsahuje mimo jiné také dvanáct příkladů aplikací (petnetsim/samples), které demonstrují fungování a použití této knihovny. Jelikož jsou tyto příklady dílem autora knihovny, tvoří ideální testovací vzorek pro základní porovnání výkonu jednotlivých zrychlovacích metod (alternativních interpretů, rozšiřujících modulů a transkompilátorů). Některé z těchto úloh bylo nutné modifikovat zvětšením rozsahu testované sítě, jelikož jejich čas běhu byl příliš malý, aby byl spolehlivě měřitelný.

5.3 Hardware & Software

Testy byly prováděny na osobním stolním počítači s následujícími specifikacemi:

HARDWARE:

- **Procesor:** AMD Ryzen 5 3600
  - **Počet jader:** 6
  - **Počet vláken:** 12
  - **Takt:** 3 600 [MHz]
- **Paměť:** 16 GB (2x8 GB) 3 200 MHz, DDR4

SOFTWARE:
Následující tabulka obsahuje seznam použitého softwaru (zrychlujících metod) se specifikací jejich verzí a operačního systému, na kterém byly tyto metody testovány.

Název metody	Verze metody	Operační systém
CPython	CPython 3.10	Windows 11/ Ubuntu 20.04
Pyston	Pyston v2.3.3	Ubuntu 20.04
PyPy	PyPy v3.9	Windows 11
Cinder	Cinder 3.10	Ubuntu 20.04
Numba	Numba 0.55.1	Windows 11
Pyjion	Pyjion 1.2.3	Windows 11
Cython	Cython 0.29.28	Windows 11

Tab. 1: Seznam specifikace softwaru

5.4 Metodika testování a porovnání dat

Testování a měření:
Jak již bylo zmíněno v sekci 5.2 v této kapitole, předmětem porovnání bude dvanáct úloh, které aplikují PetNetSim a pokrývají její možnosti a funkce. Každá úloha bude podrobena testování v rámci příslušené zrychlující metody a nativní implementace CPython a pro statistické účely bude každá úloha spuštěna nejméně 20x po sobě[5]. Vykonání každé úlohy bude z hlediska časové náročnosti měřeno a zaznamenáno pro účely budoucího porovnání. Ukázka ze skriptu, který volá příslušné úlohy a zaznamenává čas každého běhu je zobrazena v sekci ***Kód 5.1***:

Kód 5.1: Ukázka z testovacího skriptu

```
cyc_num = 20
i=0
Time_001 = []
while i < cyc_num:
    start_time = time.perf_counter()

    sample_001_basic_edit.run()
    Time_001.append(time.perf_counter() - start_time)
    i = i+1
```

[5] Jak se později ukázalo, bylo 20 spuštění pro orientační poměření výkonu jednotlivých metod dostačující a více spuštění mělo na statistiku malý vliv. Proto byla každá úloha pro příslušné implementace měřena jen 20x.

Časy jednotlivých běhů pro všechny úlohy jsou tímto způsobem měřena a zaznamenávána do .csv tabulky, která bude sloužit jako zdroj dat k dalšímu statistickému vyhodnocení.

Porovnání dat:

Výstupem měření bude csv tabulka pro každou ze zrychlujících metod obsahující 12 řádků pro každou úlohu s 20 sloupci pro každý jeden běh dané úlohy. V jednotlivých buňkách tak nalezneme čas konkrétního běhů úlohy. Pro účely přehledného grafického zpracování budou data zpracována následovně:

- Časy jednotlivých úloh budou zprůměrovány a průměry poslouží jako hodnota reprezentující čas jednotlivých úloh.
- Průměrné časy úloh provedených interpretem CPython, coby nativním Python interpretem, poslouží jako referenční hodnota času jednotlivých úloh a průměrné časy úloh provedené každou ze zrychlujících metod budou porovnány relativně vůči hodnotám CPythonu. Matematický popis této metody zachycuje následující vztah (1):

$$P_{zm\,i} = \frac{100}{\frac{c_{zm\,i}}{c_{cpython\,i}}}; \qquad i = 1,2,3 \dots 12 \tag{1}$$

Ve vztahu (1) označuje i číslo úlohy, $c_{zm\,i}$ označuje průměrný čas příslušné úlohy a příslušné zrychlující metody (*zm*), $c_{cpython\,i}$ označuje průměrný čas úlohy i v CPythonu. Z rovnice je patrné, že jmenovatel vyjadřuje poměr času zrychlující metody vůči času CPythonu. Tím, že je tento poměr ve jmenovateli, dostáváme tak jednotky $P_{zm\,i} = \left[\frac{1}{s}\right]$. Vynásobením konstantou v čitateli tak $P_{zm\,i}$ udává poměr rychlosti alternativní zrychlující metody vůči CPythonu v procentech. Jinými slovy, pokud je $P_{zm\,i} > 100$, dosáhla alternativní metoda zrychlení, v opačném případě je metoda pomalejší.

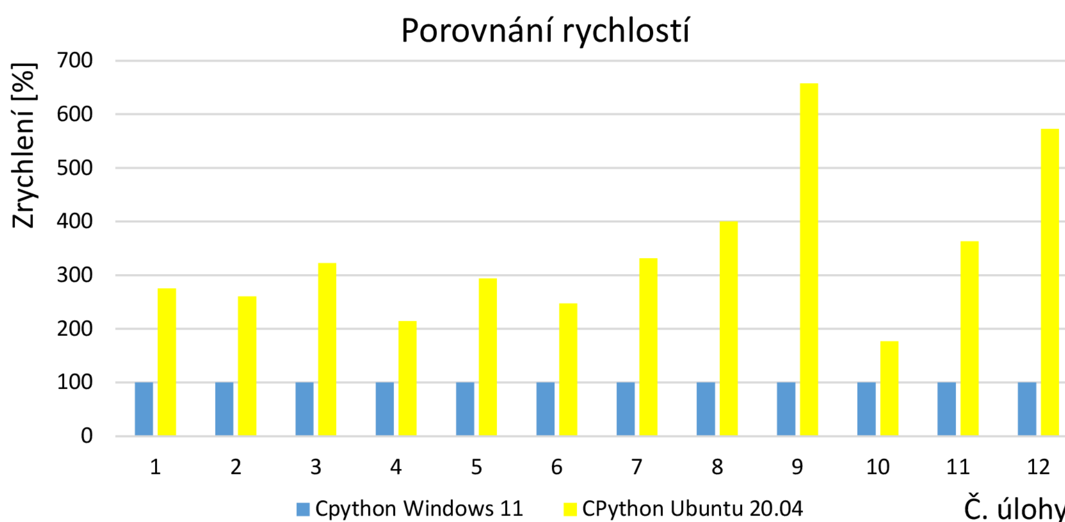
6 IMPLEMENTACE

V tento moment máme popsanou metodiku testování i zpracování dat a můžeme se tak pustit do samotné implementace jednotlivých metod, které mají za účelem zrychlit výpočet knihovny PetNetSim. Tabulky měření rychlosti jednotlivých metod jsou obsaženy v příloze, viz Seznam příloh

6.1 CPython

Nejprve otestujeme časy běhů 12 úloh dle metodiky popsané v předchozí kapitole pro nativní interpret CPython. Výstupem je tak tabulka, která udává časy jednotlivých běhů pro všech dvanáct úloh. Toto měření bylo provedeno jak na operačním systému Windows 11, tak na operačním systému Ubuntu 20.04 spouštěném ve virtuálním prostředí, jelikož alternativní interpreti Pyston a Cinder jsou podporovány pouze na platformě Linux.

Tyto testy dopadly s velice překvapivým výsledkem. Jelikož bylo Ubuntu 20.04 spouštěno na počítači, na kterém zároveň běžel i operační systém Windows 11 a pro systém Ubuntu byla vyhrazena jen 2 jádra a operační paměť 4 GB, byl předpoklad, že testy pro CPython na Ubuntu dopadnou z hlediska průměrných časů jednotlivých úloh spíš hůře než na Windows 11. Jak ale vypovídá ***Obr. 10***, opak je pravdou



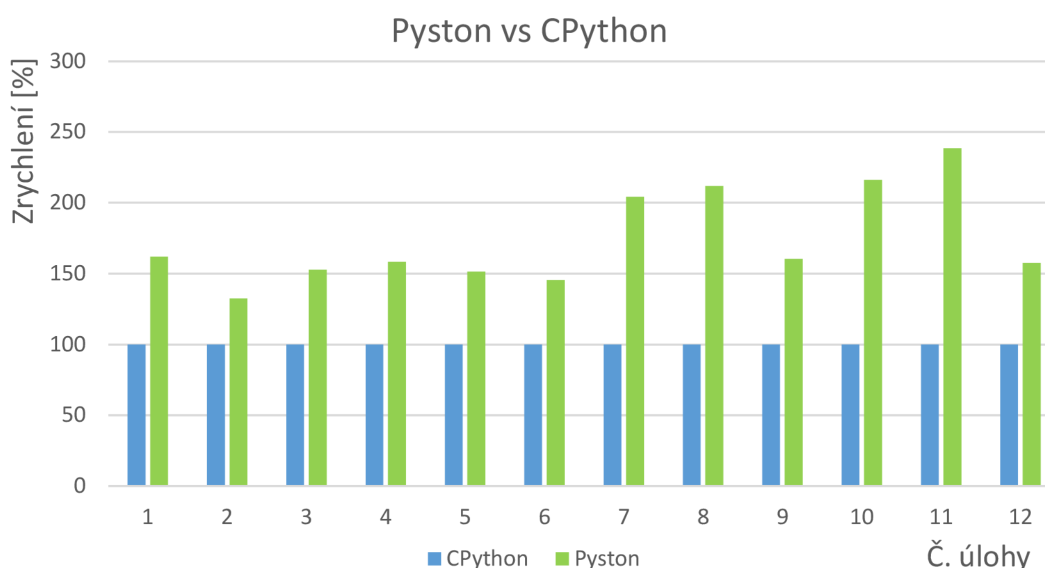
Obr. 10: Porovnání rychlosti CPython pro Windows a Ubuntu

Nakolik se jedná o záležitost tohoto konkrétního testu, anebo zda jde o vlastnost PetNetSim, není v tuto chvíli známo. Z hlediska porovnání alternativních interpretů to ani není tolik podstatné, jelikož alternativní metody pro Windows a Ubuntu budou porovnávány s výsledky CPythonu pro příslušný OS. Jedná se ovšem o velice zajímavý výsledek a důvod k těmto markantním rozdílu ve výkonu na dvou různých operačních systémech by stál za hlubší prozkoumání, které ovšem nebude součástí této práce.

6.2 Pyston

Jelikož je Pyston podporován pouze pro operační systém Linux, bylo nutné jej testovat ve virtuálním prostředí. K tomuto účelu jsem zvolil nástroj Hyper-V, který je součástí operačního systému Windows 11. Instalaci alternativního interpreta Pyston je možné provést několika způsoby. Nejsnazší variantou je použít vlastní instalační skript vytvořený vývojáři Pystonu dostupného na adrese: https://blog.pyston.org/. Tento skript zajistí instalaci potřebných nástrojů spolu s interpretem Pyston. Pokud instalace proběhne v pořádku a nevyskytnou se žádné chybové hlášky, je interpret připravený k použití. V našem vývojovém prostředí (Visual Studio, PyCharm…) vytvoříme virtuální prostředí a nastavíme Pyston jako interpreta jazyka Python.

Následně už je třeba jen postupovat dle metodiky popsané v kapitole 5 a výsledky měření shrnuje následující graf v ***Obr.11***:



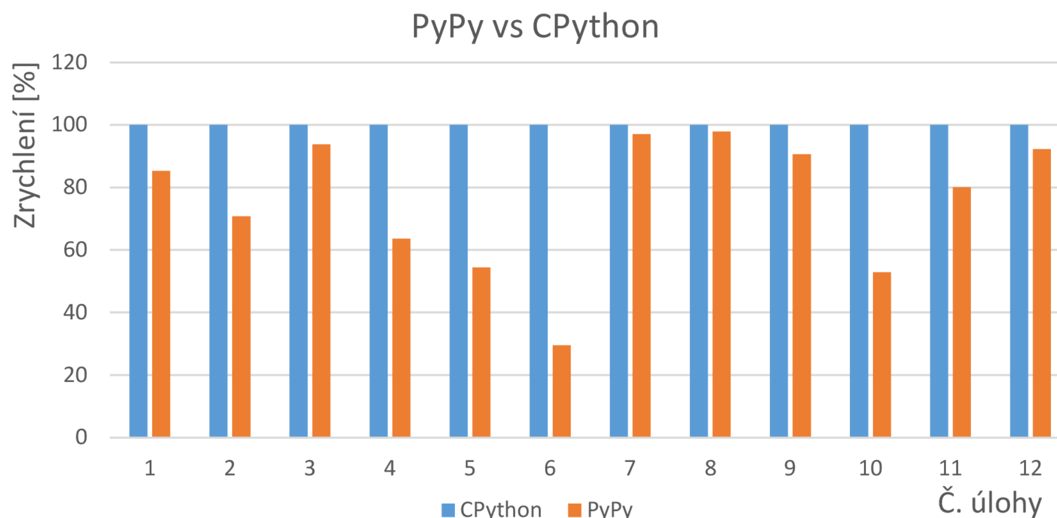
Obr. 11: Porovnání rychlosti Pyston vs CPython Ubuntu

Z grafu je jasně vidět, že testy dopadly úspěšně a Pyston dosáhl zrychlení oproti nativnímu CPythonu. V průměru na škále 12 testovaných úloh dosáhl Pyston navýšení rychlosti o 74%.

6.3 PyPy

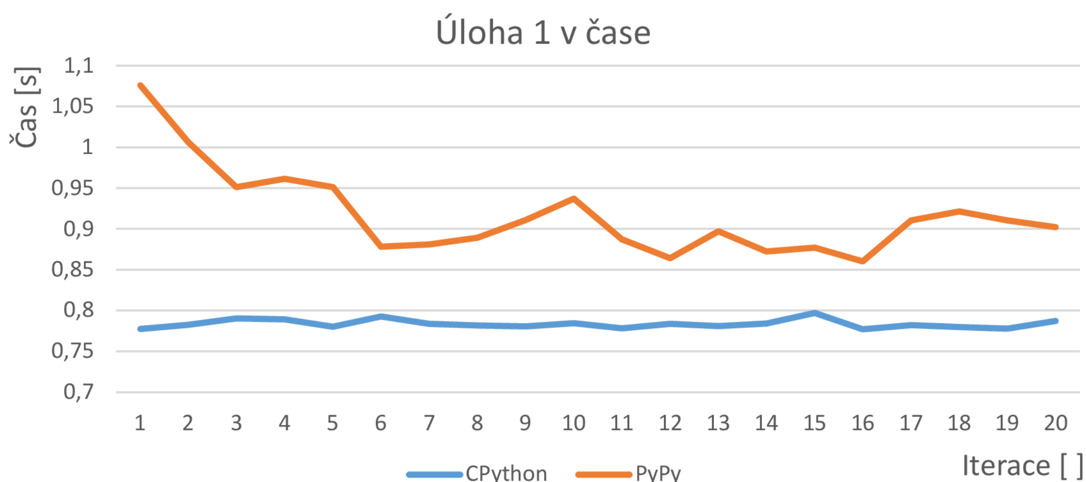
Implementace alternativního interpreta PyPy je poměrně jednoduchá. Vyžaduje pouze stažení již zkompilovaného interpreta ze stránek https://www.pypy.org/download.html. Jeho použití je nyní obdobné jako v případě Pystonu. Ve vývojovém prostředí stačí vytvořit nové virtuální prostředí a zadat cestu k alternativnímu interpretu. S použitím tohoto virtuálního prostředí je nyní možné spustit náš skript, který testuje jednotlivé úlohy

a dle metodiky zpracovat výstupní data tohoto skriptu. Graf, jehož vstupem jsou jednotlivé poměry rychlostí dle vztahu (1), je zobrazen na ***Obr. 12.***



Obr. 12: Graf porovnání rychlosti PyPy vs CPython

Z grafu je patrné, že PyPy v testech nedosahuje dobrých výsledků. Pokud výsledky pro jednotlivé úlohy zprůměrujeme, vychází nám, že PyPy dosahuje jen asi 75 % výkonu nativního CPythonu. Podívejme se blíže na analýzu testu první úlohy v grafu na následujícím obrázku:



Obr. 13: Analýza 1. úlohy v průběhu času

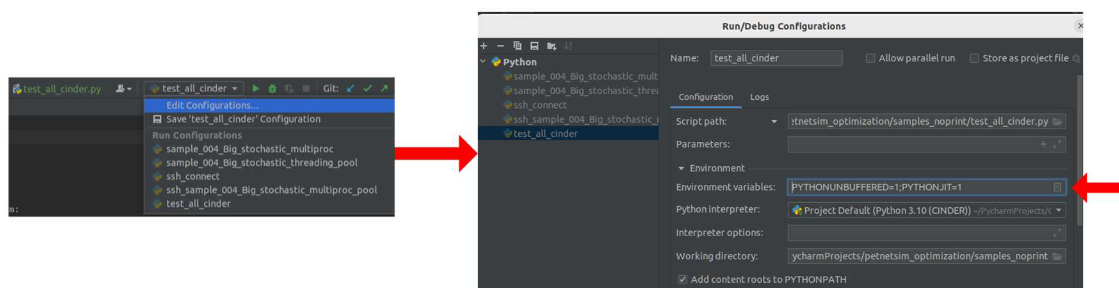
Z tohoto grafu je patrné, že JIT kompilátor v prvních iteracích pracuje pomaleji, jelikož je zatížen analyzováním kódu. V dalších iteracích se ale situace zlepšuje a čas se ustálí. Z tohoto faktu je zřejmé, že JIT kompilátor pracuje tak, jak má. Bohužel, nedosahuje takových výsledků, jaké bychom si přáli.

6.4 Cinder

Cinder je alternativní interpret z dílny společnosti Meta (dříve známé jako Facebook). Cinder je hnacím motorem aplikace Instagram a v rámci společnosti Meta je experimentálně využíván i k dalším Python aplikacím.

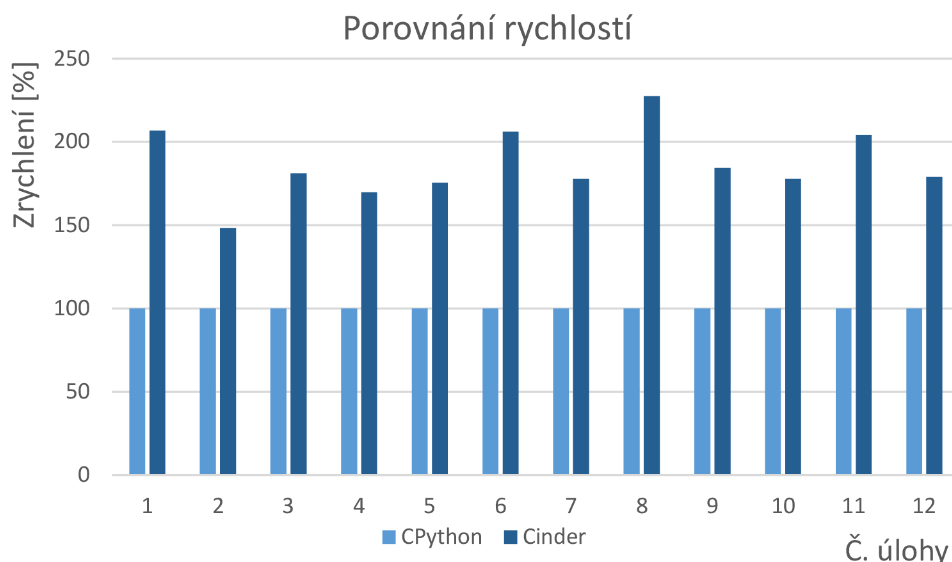
Cinder je sice veřejně dostupný, ne však jako nástroj, který by měl (zatím) nahradit CPython. Autoři sami upozorňují, že technicky vzato se jedná o experimentální alternativu k CPythonu a zveřejnění Cinderu má sloužit spíše ke kolaboraci vývojářů, kteří se snaží pracovat na urychlení CPythonu a minimalizovat tak dvojí úsilí. Dokumentace a podpora k Cinderu je minimální, nicméně stojí za to se pokusit jej implementovat. Nutno také zmínit, že Cinder je podporován jen pro Linux x64 a pokusy na jiných systémech pravděpodobně skončí neúspěchem.

Návod na sestavení interpreta Cinder je na stránkách GitHubu [14]. Instalace i následná kompilace se po odborné pomoci vedoucího této práce zdařila a metodu bylo možné otestovat. Je důležité zmínit, že funkce JIT u interpreta Cinder není nativně zapnutá a je nutné ji povolit v rámci zadaného parametru při spouštění skriptu přes Cinder, např. takto: *cinder3.10 -X jit mypythonfile.py*. Další možností je nastavit proměnnou prostředí PYTHONJIT = 1. V prostředí PyCharmu to lze pro náš testovaný skript snadno provést úpravou konfigurace spouštěného skriptu, kde nastavíme proměnnou prostředí, jak to znázorňuje následující obrázek:

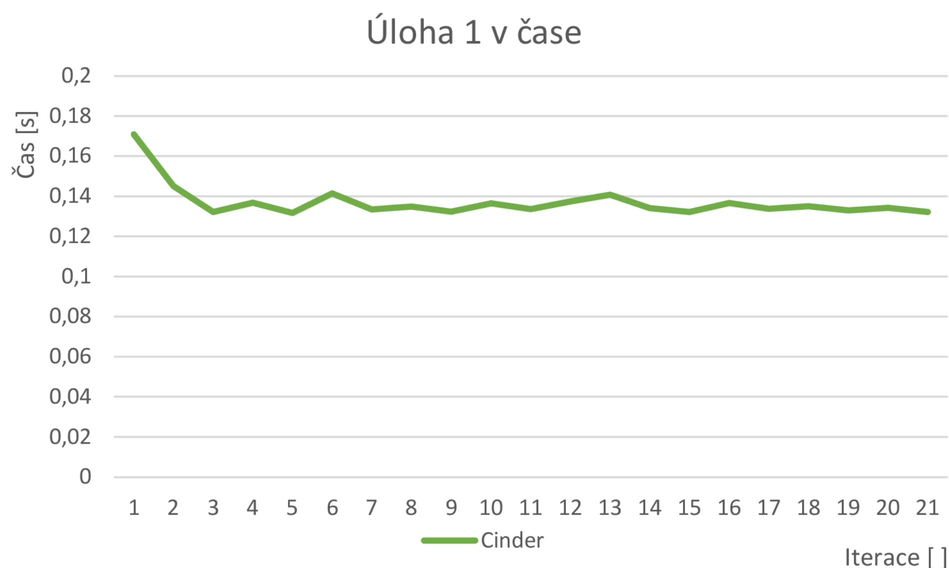


Obr. 14: Povolení funkce JIT pro Cinder

Výsledky měření pro 12 úloh znázorňuje ***Obr. 15*** na následující straně. Je z něj patrné, že si Cinder vedl velmi dobře v porovnání s nativním CPythonem. V průměru dosáhl 186 % rychlosti CPythonu a činí z něj tak nejúspěšnější zrychlující metodu zde testovanou. ***Obr. 16*** na následující straně pak opět znázorňuje časový rozbor první úlohy, ze kterého je evidentní, že Cinder používá metodu JIT kompilace.



Obr. 15: Porovnání rychlosti Cinder a CPython



Obr. 16: Analýza 1. úlohy v průběhu času

6.5 Numba

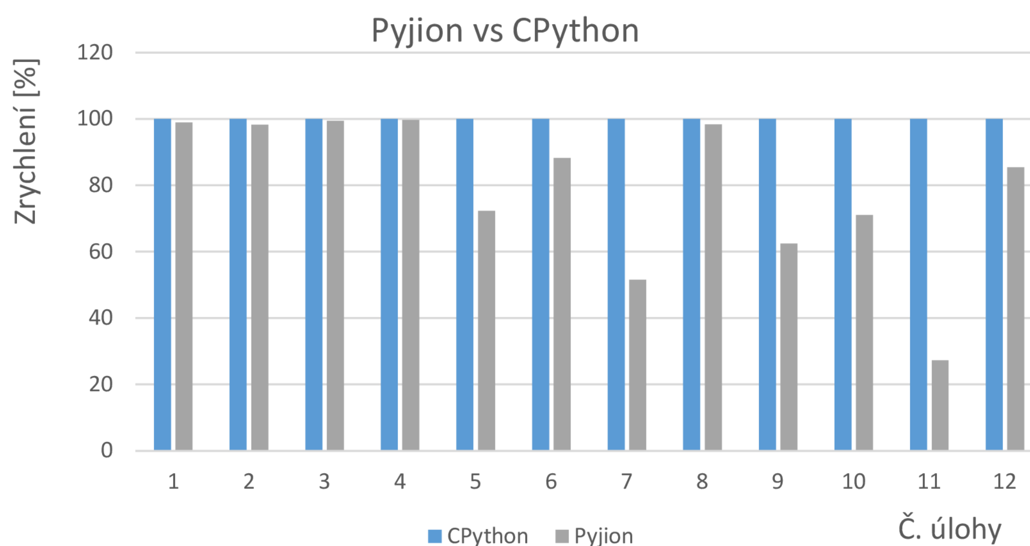
Numba je rozšiřující modul, který lze snadno nainstalovat pomocí správce balíčků, jako PyPi nebo Conda. Dosahuje optimalizace kódu metodou JIT kompilace a je určený k použití na smyčky a kód, který spoléhá na výpočetní kapacitu rozšiřujícího modulu Numpy. Na základě rešerše o tomto modulu v kapitole 3.2.1 je patrné, že Numbu nelze aplikovat na ledajaký kód, ale je použitelná jen na specifické funkce. V případě příznivých podmínek pro použití dosahuje Numba skvělých výsledků.

Použití modulu Numba na knihovnu PetNetSim nebylo v našem případě velice produktivní. Numbu nebylo možné aplikovat žádným způsobem tak, abychom dosáhli znatelných rozdílů ve výkonu. Implementace Numby by vyžadovala poměrně zásadní

zásah do struktury kódu, který by byl časově náročný a v tomto momentě i s nejasným výsledkem. Z tohoto důvodu bylo od použití Numby odstoupeno a testy nebyly provedeny. Je však třeba říct, že Numba je skvělým nástrojem pro optimalizaci (při již zmíněných podmínkách) a její použití vždy stojí za vyzkoušení, neboť její instalace a volání je z uživatelského hlediska velmi pohodlné.

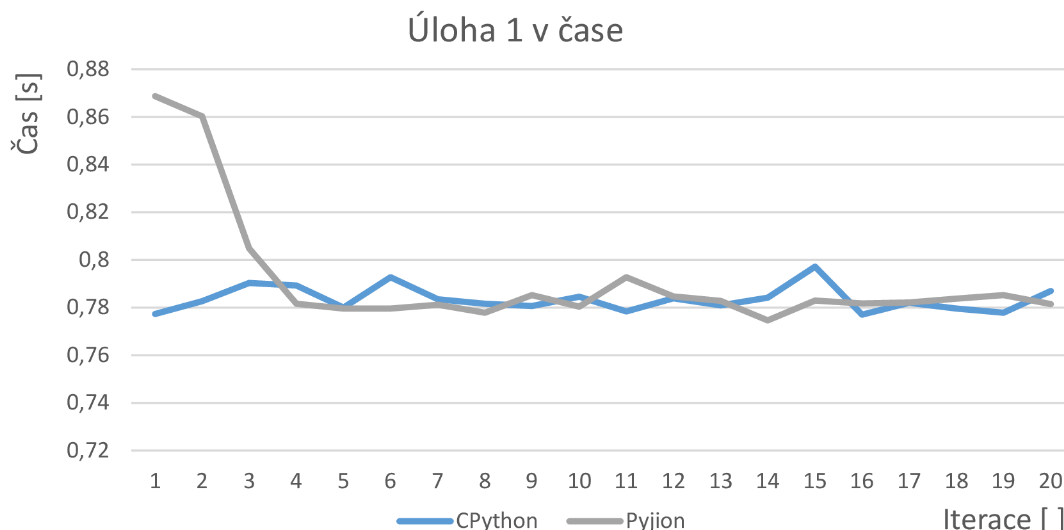
6.6 Pyjion

Pyjion je rozšiřující modul a jeho instalace probíhá stejně, jako v případě mnoha dalších modulů, jako je např. Numpy nebo Numba. Nejobecnější přístup, jako v případě mnoha dalších modulů, je použít příkaz pip *python -m pip install pyjion.* Tímto je náš rozšiřující modul nainstalovaný a připravený k použití. Skripty knihovny PetNetSim stačí tak už jen opatřit importem modulu Pyjion, povolit jej a nastavit požadovanou úroveň optimalizace, viz ***Kód 3.3.*** Výsledky porovnání rychlostí Pyjionu a CPythonu popsuje graf v následujícím obrázku:



Obr. 17: Graf porovnání rychlosti Pyjion vs CPython

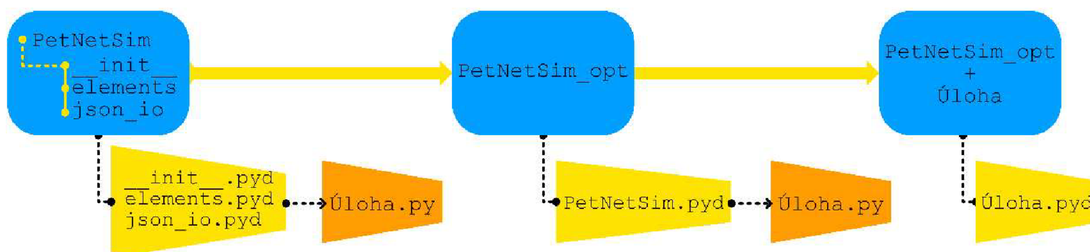
Je evidentní, že ani v případě Pyjionu nedošlo ke zlepšení napříč dvanácti testovanými úlohami. Pyjion dosáhl průměrně jen asi 80% rychlosti CPythonu, což je ovšem lepší výsledek než v případě PyPy. Časová analýza dvaceti běhů první úlohy na ***Obr.18*** opět krásně ukazuje, že JIT kompilace funguje jak má a čas běhu programu s počtem iterací klesá a ustálí se na podobné hodnotě, jako CPython.



Obr. 18: Analýza 1. úlohy v průběhu času

6.7 Cython

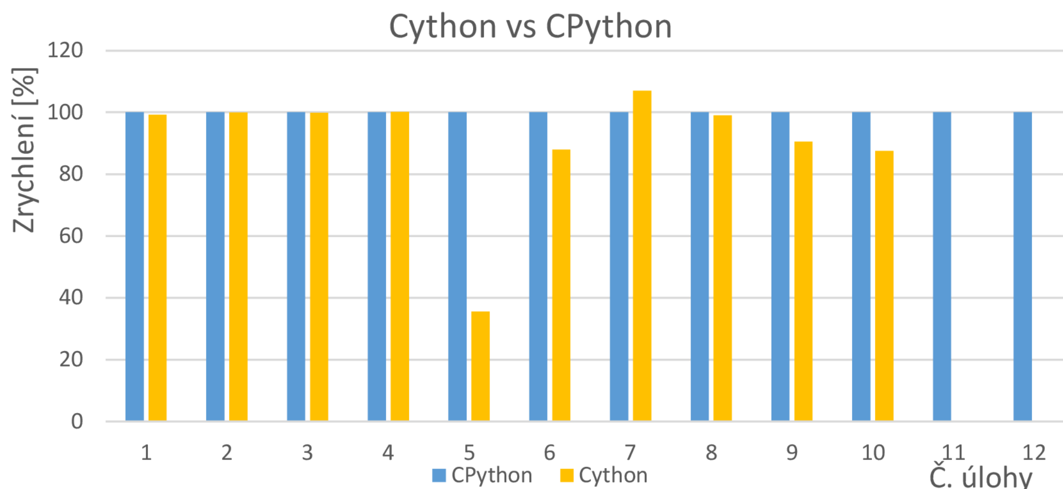
Rešerše, která předcházela použití nástroje Cython naznačovala poměrně snadné a bezproblémové nasazení tohoto nástroje na náš kód. V praxi se ovšem vyskytlo několik komplikací, a to zejména s provázáním importovaných knihoven a modulů. Alternativou tedy bylo sloučit všechny dílčí skripty do jednoho s názvem *PetNetSim_opt.py*. Bohužel ani tato metoda nebyla 100% odolná vůči problémům v procesu kompilace a spouštění kompilovaných modulů. Z časových důvodů bylo nutné se uchýlit k ne příliš elegantnímu řešení, nicméně funkčnímu. Každý skript, který importoval knihovnu PetNetSim a definoval úlohu, byl nyní rozšířen o celý kód knihovny, aby bylo zabráněno problémům s kompilací jednotlivých částí knihovny a výsledkem je 12 nativních modulů s koncovkou „*.so*“. Tento proces je pro názornost vyobrazen na ***Obr. 19:***



Obr. 19: Proces úpravy úloh před kompilací

Vstupem pro testování tedy bylo 12 nativních modulů, které proběhly procesem kompilace. Bohužel, ani tento proces neproběhl úspěšně na 100 % a v grafu na ***Obr. 20*** je zaznamenáno, že pro úlohy 11 a 12 nemáme žádná naměřená data, jelikož se nepovedlo tyto dvě konkrétní úlohy zprovoznit a z časových důvodů bylo nutné od jejich

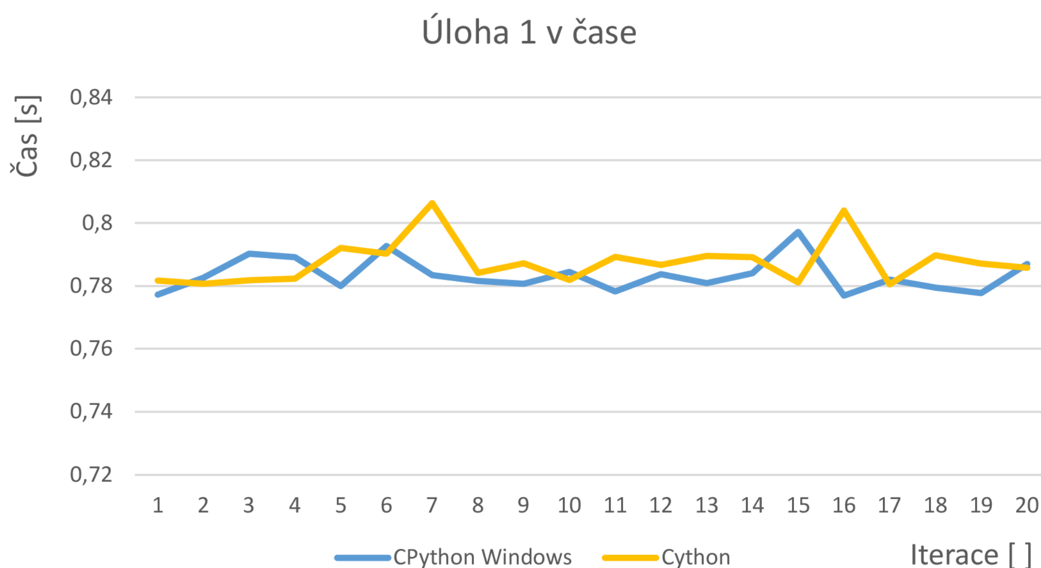
implementace ustoupit. Ostatní úlohy byly podrobeny stejnému testu, jako v případě dalších zrychlujících metod a výsledky shrnuje následující graf:



Obr. 20: Graf porovnání rychlosti Cython vs CPython

Z grafu je patrné, že samotná kompilace kódu nedosáhla výrazného zlepšení a v páté úloze vidíme dokonce markantní propad výkonu. Pokud zprůměrujeme porovnání rychlostí pro 10 naměřených úloh, je Cython v průměru na pouhých 90 % výkonu CPythonu.

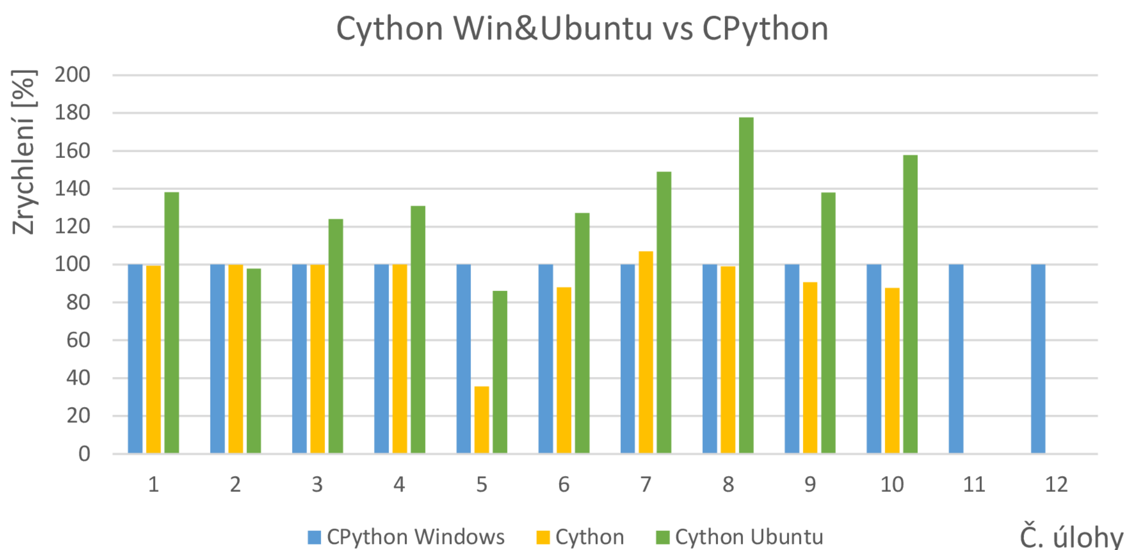
Opět se podívejme na analýzu první úlohy na následujícím obrázku:



Obr. 21: Analýza 1. úlohy v průběhu času

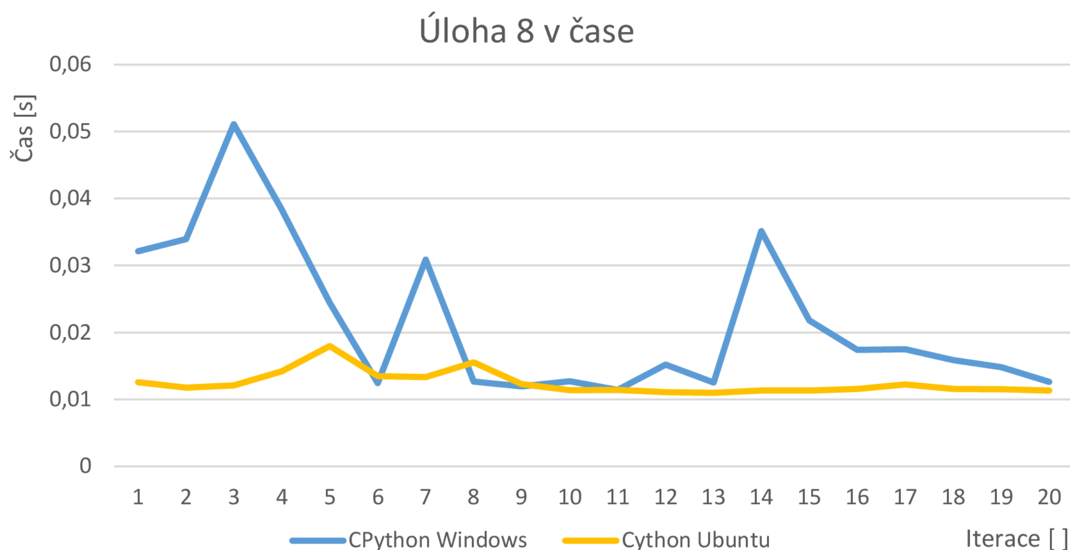
Je evidentní, že Cython už během jednotlivých iterací proces neoptimalizuje a dosahuje konzistentních výsledků napříč časem.

Následně byla kompilace provedena na systému Ubuntu a výsledky byly opět porovnány s CPythonem. Na Ubuntu se ukázala kompilovaná verze jako efektivnější a dosáhla lepších výsledků, viz následující graf:



Obr. 22: Cython na Windows a Ubuntu

Je poněkud podivné, že můžeme dosáhnout tak rozdílných výsledků pouze změnou systému. Náhled do reality nám může poskytnout analýza úlohy 8 na následujícím grafu:



Obr. 23: Analýza 8. úlohy v průběhu času

Z grafu je patrné, že nativní CPython je v porovnání s Cythonem časově méně stabilní, a to pak ze statistického hlediska značně ovlivňuje průměr. V následující kapitole, která se bude věnovat optimalizaci při testování na rozsáhlejší stochastické síti, bude vhodné toto chování ověřit.

Jak již bylo řečeno v kapitole 3.3.4, Cython dosahuje lepších výsledků, deklarujeme-li datové typy vytížených proměnných. Efektivní využití statického tipování by ovšem vyžadovalo poměrně rozsáhlou analýzu kódu a jelikož máme k dispozici slibnějšího kandidáta v podobě Pystonu, nebude tato snaha předmětem tohoto diplomového projektu.

6.8 Zhodnocení zrychlujících metod

V následujících řádcích si stručně shrneme výsledky této kapitoly – jakých výsledků bylo dosaženo pro jednotlivé zrychlující metody a jak náročná byla jejich samotná implementace. Graf porovnávající všechny testované alternativní metody je obsahem obrázku ***Obr. 24*** na konci této kapitoly.

Pyston:
Pyston je poměrně nový alternativní interpret, který momentálně běží pouze na operačním systému Linux. Jeho instalace a implementace do existujícího kódu byla z podstaty změny interpreta snadnou záležitostí. Pyston jako jediný z testovaných alternativních metod dosáhl zvýšení výkonu Pythonu a to v průměru na 174 % původní implementace v CPythonu. S použitím Pystonu se počítá v další snaze o optimalizaci výpočtu pro statistiky stochastických jevů.

PyPy:
PyPy je dominantním hráčem na poli alternativních interpretů pro jazyk Python. Funguje pro operační systémy Windows, Linux i MacOS. Z toho důvodu byl kladen předpoklad na pozitivní výsledky měření. Opak byl však pravdou. PyPy dosahoval v průměru jen asi 75 % výkonu nativní implementace v CPythonu. Jelikož tato práce nejde do takové hloubky, aby zkoumala mechanismy optimalizace jednotlivých alternativních metod, je důvod neuspokojivých výsledků neobjasněn. Implementace PyPy byla ovšem nejsnazší ze všech uvedených alternativních interpretů a jistě by byla chyba jej na základě nepříznivých výsledků pro naši aplikaci zatracovat.

Cinder:
Cinder je posledním alternativním interpretem, kterýž jsme se pokoušeli implementovat. Jeho instalace a následná kompilace se neobešla bez obtíží a byla nutná pomoc vedoucího práce. Výsledek ovšem stál za to. Při povolení funkce JIT dosáhl Cinder 186 % rychlosti nativního CPythonu a stal se tak nerychlejší implementací pro knihovnu PetNetSim. Cinder bude dál testován na rozsáhlejší úloze v kapitole 7.

Numba:
Numba je mocný JIT rozšiřující modul, který výborně funguje s knihovnou Numpy a funkcemi, které obsahují smyčky s mnoha cykly. Svým zaměřením a funkcionalitou je ovšem Numba velmi specifická, a proto se nepodařilo Numbu úspěšně aplikovat tak, aby došlo k znatelnému zvýšení výkonu. Její integrace do knihovny PetNetSim by znamenala nemalý zásah do struktury kódu.

Pyjion:

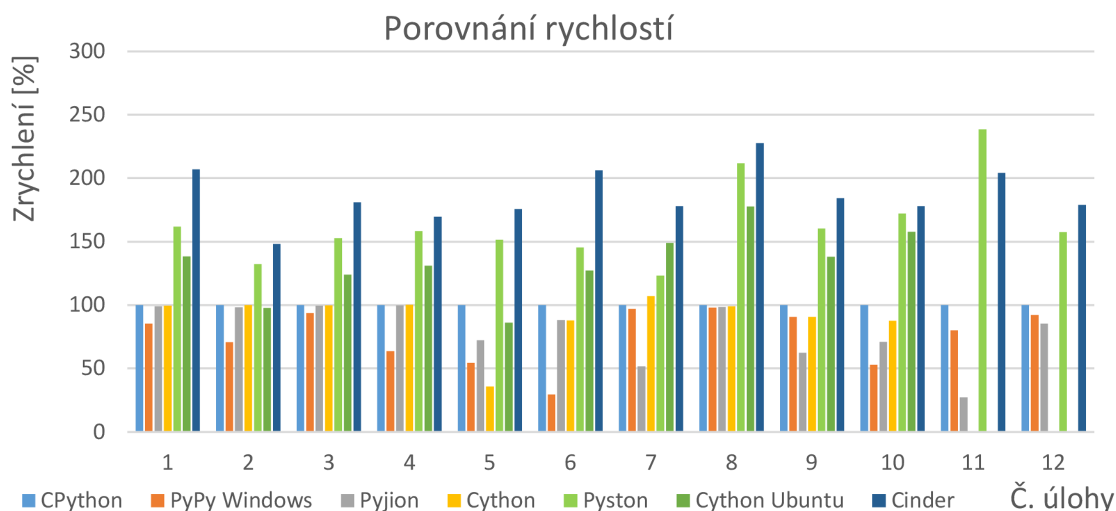
Pyjion je rozšiřující modul, který lze snadno nainstalovat a používat. Je dostupný pro Windows, Linux i MacOS. Stejně jako jiné zde zmíněné metody usiluje Pyjion o optimalizaci pomocí metody JIT kompilace. Přestože si v testech pro naši aplikaci nevedl nejlépe a dosáhl jen asi 80 % výkonu CPythonu, ambice Pyjionu jsou veliké a v budoucnu s ním bude nutno počítat jako s velkým konkurentem ve světě JIT kompilátorů. Vize vývojářu Pyjionu je totiž stát se přirozenou součástí CPythonu, a umožnit tak provádět JIT kompilaci bez nutnosti jakýchkoli rozšíření a kompromisů.

Cython:

Cython byl jediným testovaným představitelem tzv. transpilerů – metody překladu do jiného jazyka a následné kompilace. Cython je zároveň programovacím jazykem, který se velice podobá Pythonu a Python je defacto fungující verzí jazyka Cython. Proces kompilace je poměrně snadný pro menší funkce. Naším cílem bylo ovšem zkompilovat celou knihovnu a v tomto případě byl celý proces implementace poměrně časově náročný a vyžadoval nemalé úsilí. Kompilace byla provedena na systému Windows a systému Ubuntu. Zatímco pro Windows vykazoval Cython jen asi 90 % rychlosti CPythonu, na systému Ubuntu byla naměřena průměrná rychlost Cythonu odpovídající 130 % CPythonu. Tato měření mohou být ovšem ovlivněna velikostí sítě a kolísáním rychlosti CPythonu na systému Ubuntu a bude třeba tuto skutečnost ověřit v 7. kapitole této práce.

Závěr:

Následující graf v ***Obr.23*** shrnuje, jak si v testech jednotlivé zrychlující metody vedly. Je z něj patrné, že nejúspěšnějším kandidátem pro další optimalizaci je alternativní interpret Pyston a Cinder. Cython testovaný na systému Ubuntu dosáhl také solidních výsledků, je zde však jistá spekulace ohledně stability CPythonu na systému Linux pro menší úlohy, a proto bude Cython podroben dalším testům na rozsáhlejší síti.



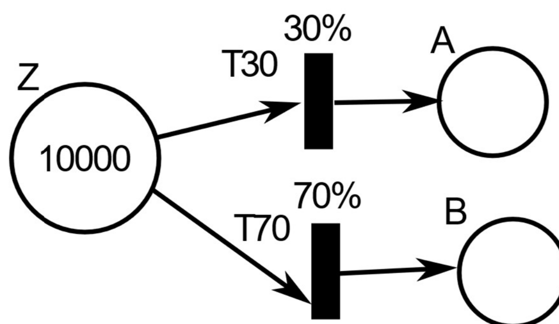
Obr. 24: Shrnutí výsledků zrychlujících metod

7 OPTIMALIZACE STOCHASTICKÉ SÍTĚ

Jelikož je hlavní motivací pro zvýšení výkonu knihovny PetNetSim nutnost více běhů stochastických jevů pro statistické účely, věnuje se tato kapitola výhradně optimalizaci při testování na stochastické síti. Proces optimalizace bude zahrnovat implementaci alternativního interpreta Pyston a Cinder, test na síti kompilované pomocí Cythonu a paralelizace výpočtů pro více běhů.

7.1 Základní stochastická síť

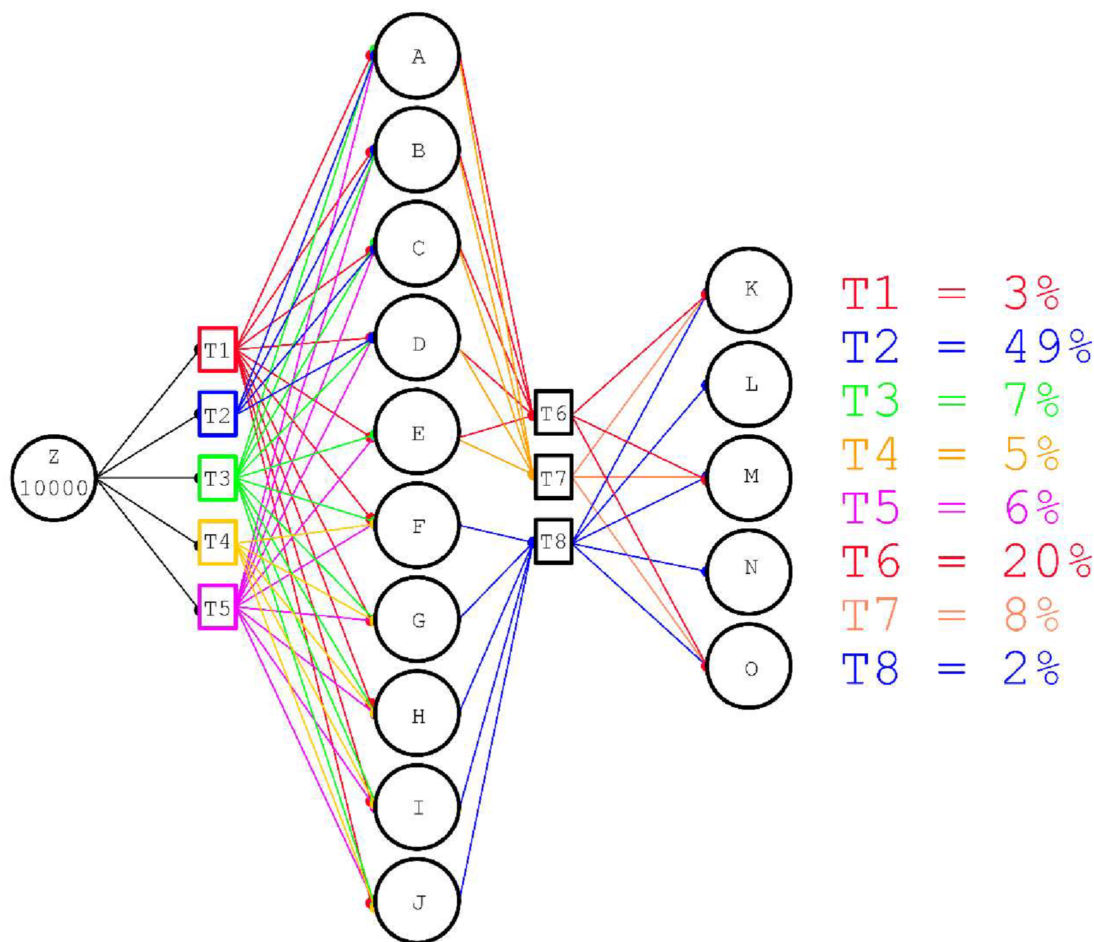
Úloha číslo 4, která byla jedna ze sady 12 testovaných úloh v předchozí kapitole je úloha, která aplikuje PetNetSim na základní stochastickou síť. Tato síť obsahuje 3 místa (jedno počáteční) a dva přechody s pravděpodobností přechodu 30 % a 70%. Síť je znázorněna na následujícím obrázku:



Obr. 25: Základní stochastická síť úlohy 4. [36]

7.2 Rozsáhlejší stochastická síť

Jelikož je síť na ***Obr. 25*** malá, je jen přirozené, že nás napadne chování zrychlujících metod testovat na větší síti. Nejen z důvodu, že stochastické jevy bývají často komplikovanější, ale je zde i šance, že se JIT kompilátor zapracuje lépe a složitější úlohy tak budou možno lépe optimalizovat. Proto byla připravena rozsáhlejší síť, která nyní obsahuje 16 míst a 8 přechodů. Zároveň je rozšířena počtem spojnic mezi přechody a místy – jeden přechod vede do více míst a více míst směřuje do stejného přechodu. Tuto síť i s rozložením pravděpodobností jednotlivých přechodů znázorňuje obrázek 26, který byl pro přehlednost umístěn na následující stránku.



Obr. 26: Rozsáhlejší stochastická síť

Následně byla tato síť testována stejnou metodou, jako jednotlivé úlohy v 6. kapitole. Síť byla spuštěna pro 20 běhů na systému Ubuntu, a to pro CPython, Pyston, Cinder a Cython. Cython byl testován z důvodu časové nestability CPythonu, o které je řeč v 6. kapitole. CPython bylo nutné otestovat k porovnání výkonu a Pyston a Cinder byly testován coby metody, které dosáhly v kapitole 6. nejlepších výsledků.

Během procesu optimalizace bylo zjištěno, že nativní přístup v úlohách testovaných v kapitole 6, a tedy de facto standardní postup při implementaci knihovny PetNetSim během výpočtu vypisoval mezivýsledky jednotlivých kroků. Jelikož je ale tento výpis z hlediska koncového stavu nepodstatný a dílčí informace by bylo možné poskytnout i jiným způsobem (např. zapsat jej do textového souboru, nebo do csv tabulky), byl výpis těchto dílčích mezivýsledků zakázán a ponechán jen výpis finálního stavu sítě. Tento počin nám urychlil výpočet sítě o 7 % na Ubuntu a téměř o 450 % [6] na Windows 11. V budoucí verzi knihovny PetNetSim by tedy bylo vhodné zvážit implementaci uložení dílčích informací do souboru, případně přidat funkci, který by dokázala na základě uživatelského vstupu výpis těchto informací povolit, či zakázat.

Cython na rozdíl od výsledků v kapitole 6 v tomto testu nevykázal žádné zlepšení a dosáhl v průměru 20 běhů jen asi 96,7 % výkonu nativního CPythonu. Z toho je zřejmé,

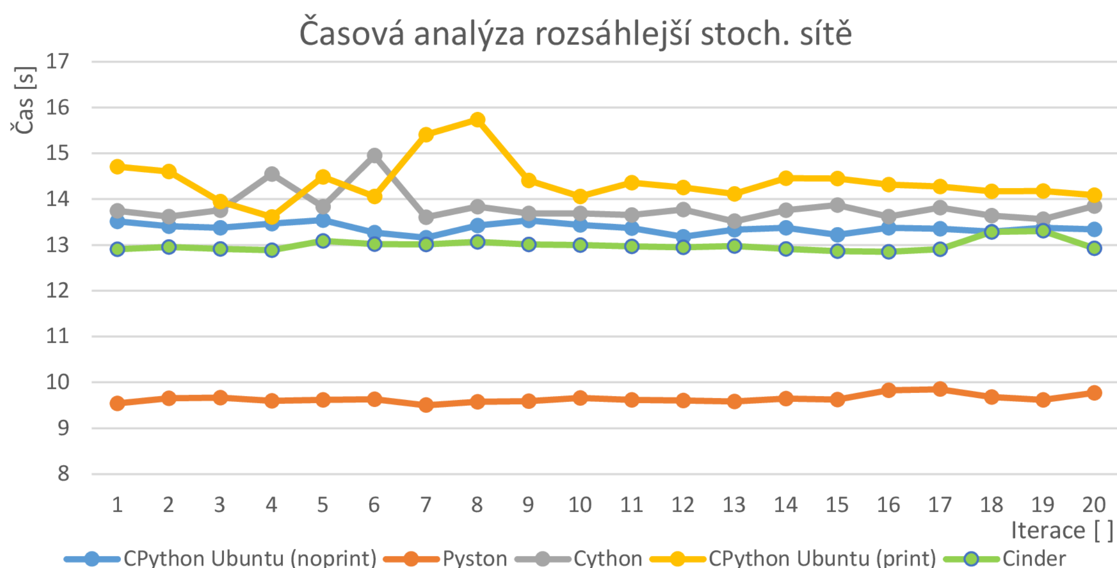
[6] Tento fakt by mohl souviset s rozdílem rychlostí CPythonu na Ubuntu a Windows, viz ***Obr. 10.***

že při delších časech běhu programů už naměřené kolísání výkonu CPythonu není tak relevantní a neovlivňuje statistiku tak silně. Díky tomu máme jasnější představu o výkonu Cyhonu na Ubuntu a jeho výkon více odpovídá implementaci Cythonu na Windows 11.

Pyston opět prokázal, že se jedná o rychlejší implementaci. Dosáhl 131 % výkonu CPythonu a z toho důvodu bude použit při další optimalizaci výpočtů stochastické sítě.

Cinder si naopak oproti předpokladům z předešlé kapitoly vedl v tomto případě hůře a dosáhl jen asi 103 % výkonu CPythonu. I přesto bude dále testován.

Porovnání jednotlivých implementací sítě (***Obr. 25***) znázorňuje graf na obrázku 27. Obsahuje informace o časech jednotlivých běhů pro nativní CPython (s výpisem i bez výpisu informací), Cython, Cinder a Pyston.



Obr. 27: Porovnání časů různých metod pro rozsáhlejší stoch. síť

7.3 Multirpocessing

Jak již bylo zmíněno, stochastické jevy je ze statistických důvodů nutné počítat pro větší počet běhů. Dalším krokem v optimalizaci času těchto výpočtů se tak nabízí paralelizace tohoto problému. Výběr vhodné metody paralelizace vyžaduje poměrně odbornou znalost této problematiky. Pro naše účely bude postačující (a z časového hlediska i efektivnější), když se pokusíme některé metody rovnou aplikovat a provedeme měření jejich efektivnosti při aplikaci na naši konkrétní úlohu.

V našem případě využijeme metodu multiprocessingu implementací dvou knihoven – *multiprocessing* a *concurrent.futures*. Příklad implementace těchto dvou technik multiprocessingu, kdy je předmětem paralelizace funkce *run()*, která spouští naši definovanou síť, znázorňují úseky ***Kód 7.1*** & ***Kód 7.2*** na další straně.

Kód 7.1: Multirpocessing - Process

```
import multiprocessing

cyc_num = 20

start_time = time.perf_counter()
processes = []
for _ in range(20):
    p = multiprocessing.Process(target=run)
    p.start()
    processes.append(p)

for process in processes:
    process.join()

print(time.perf_counter() - start_time)
```

Kód 7.2: Multirpocessing - Pool

```
import concurrent.futures

cyc_num = 20

start_time = time.perf_counter()

with concurrent.futures.ProcessPoolExecutor() as executor:
    results = [executor.submit(run) for _ in range(cyc_num)]

    for f in concurrent.futures.as_completed(results):
        print(f.result())

print(time.perf_counter() - start_time)
```

Obě metody byly testovány na dvou počítačích se systémem Ubuntu. Výsledky testů pro oba počítače jsou zaneseny v následující tabulce:

Počítač	**Process [s]**	**Pool [s]**
Ubuntu – virt. Prostředí	110,21569138199993	104,21673859419997
Ubuntu – UAI GPU[7]	19,901259629055858	19,457991925999522

Tab. 2: Porovnání metod multiprocessingu

Je tedy zřejmé, že postup dle kódu 7.2 dosahuje lepších výsledků, i když jsou obě metody srovnatelné.

Z podstaty metody multiprocessingu je problematické určit, jak velkého zrychlení dosáhneme, neboť je tato metoda závislá především na použitém hardwaru. Proto si uvedeme statistiky pro oba systémy, které porovnají celkový čas 20 běhů s a bez použití

[7] Výpočetní server UAI FSI Brno, AMD Ryzen 9 5900X

multiprocessingu a z nich se pokusíme učinit závěr. Systém na osobním počítači disponuje interpretem Pyston, proto bude statistika zahrnovat i použití tohoto alternativního interpreta. Měření na vzdáleném serveru UAI GPU bude provedeno za použití nativního interpreta CPython.

Počítač	Interpret	Bez multiprocessingu	S multiprocessingem
Ubuntu – virt. prostředí	CPython	264.79207669699827	152.12871197099957
Ubuntu – virt. prostředí	Pyston	194,76793864799765	108,47026662699864
Ubuntu – UAI GPU	CPython	196,45133323129267	16,902852120509257

Tab. 3: Porovnání bez/s multiprocessingem

Vidíme, že výsledky použití metody multiprocessingu jsou velice uspokojivé, máme-li k dispozici výkonný počítač jakým je UAI server. I přesto že Ubuntu na osobním počítači běželo ve virtuálním prostředí a byly mu přiděleny jen dvě jádra procesoru, dosáhla i tak metoda multiprocessingu velice uspokojivých hodnot.

Je tak jasným závěrem, že metoda multiprocessingu je velice efektivním způsobem optimalizace, za předpokladu že je možné úlohu paralelizovat a při výpočtu mnoha běhů stochastických sítí bude vhodné tuto metodu systematicky uplatňovat.

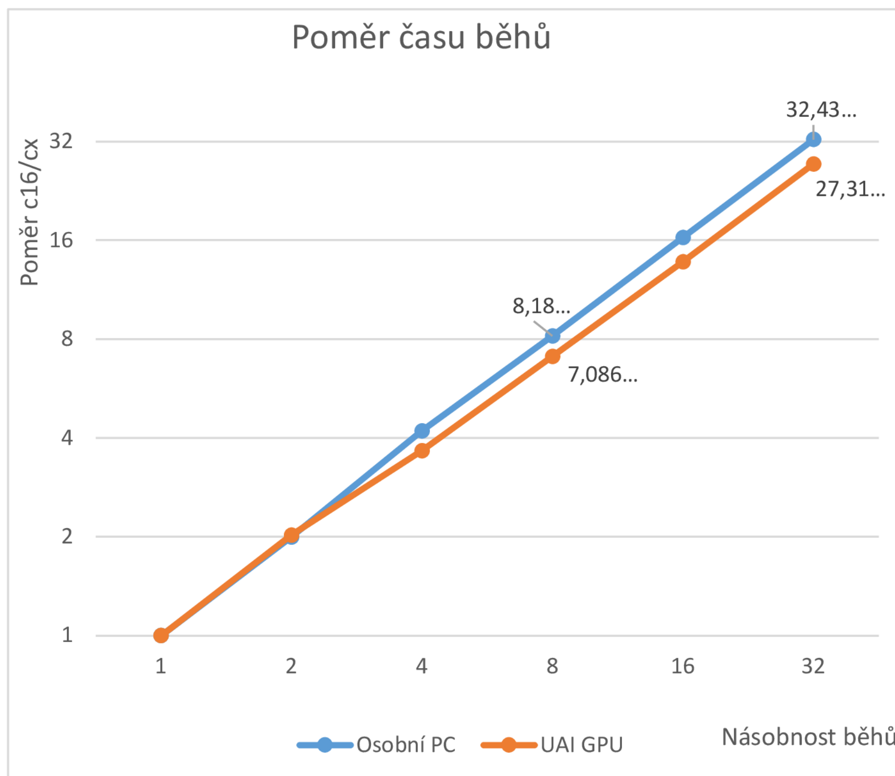
Jako poslední měření multiprocessingu provedeme měření pro různý počet spuštění v násobku počtu jader, konkrétně pro 16,32,64,128,256 a 512 a budeme monitorovat závislost času na počtu běhů. Z časových důvodů nebudeme pro tento počet běhů měřit metodu bez multiprocessingu. Jelikož není důvod pro CPython s počtem cyklů zrychlovat, trvalo by měření na osobním počítači velice dlouho (cca 13sec na jeden běh, 16+32+64+128+256+512 běhů => 13sec * 752 = cca 9700 sec = cca 2,7 hodin)

Touto metodou byly měřeny běhy PetNetSim pro síť z ***Obr. 26***. Měření probíhalo na osobním počítači (2 jádra) a výpočetním serveru UAI GPU (16 jader). Naměřené časy úloh pro 16, 32, … 512 běhů pro příslušné počítače jsou zaneseny v následující tabulce:

Násobnost běhů	1	2	4	8	16	32
Počet běhů	16	32	64	128	256	512
Časy UAI GPU	18,225	36,904	66,678	129,154	250,632	497,823
Časy Os. PC	116,07	231,719	487,8	949,547	1894,142	3765

Tab. 04: Měření linearity multiprocessingu

Je patrné, že abychom znázornili linearitu (či nelinearitu) naměřených dat, bude nutné je uvádět v logaritmickém měřítku. Pro porovnání obou počítačů je nutné na svislé ose uvést poměr prvního času (pro 16 běhů) ku příslušnému i-tému času. Výsledkem je pak graf na ***Obr. 28***, z kterého je zřejmé, že je-li např. úloha 4x složitější, zabere 4x více času. Čas strávený výpočty je tedy přímo úměrný počtu běhů simulace stochastické sítě.



Obr. 28: Posouzení linearity multiprocessingu v závislosti na počtu běhů

Závěr

Po zhodnocení výsledků měření v této kapitole vyplívá jasný závěr, že použití multiprocessingu na paralelizaci výpočtů více běhů za účelem statistiky je velice efektivní a v závislosti na použitém hardwaru můžeme dosáhnout až desetinásobné zrychlení. Z ***Tab. 3*** je patrné, že Pyston v tomto případě dosáhl urychlení o dalších 40 % a je tedy vhodné používat jako interpret Pyston místo nativního CPythonu.

8 VÝVOJ MODULU V C++

Pravděpodobně poslední možností, jak knihovnu PetNetSim optimalizovat, aniž bychom museli provádět razantní zásahy do samotného kódu, je vývoj výpočetního jádra coby modul napsaný v systémovém jazyce (viz 4. kapitola). Z časových důvodů se tato práce bohužel samotné implementaci z praktického hlediska věnovat nebude a nastíníme pouze příklad, jakým by bylo možné část kódu převést do systémového jazyka a provázat jej s jazykem Python. Motivací této kapitoly je nastínit možný postup pro budoucí snahy při optimalizaci této knihovny. Příklad bude prezentován na jednodušší třídě *Place*, pomocí které v PetNetSim definujeme a operujeme s místy[8].

Nejprve je nutné v jazyce C++ vytvořit ekvivalent třídy *Place*. Máme-li toto splněno, je možné přistoupit k wrappování – vytvoření kódu, který zaobalí naši třídu a umožní Pythonu přístup k parametrům, které definujeme. V našem případě tedy atributy třídy Place. K tomuto účelu využijeme knihovnu Pybind11, viz kapitola 4.1.2.

Pro jasnou demonstraci tohoto procesu a rozdílu mezi třídou napsanou v Pythonu a C++, jsou kódy třídy *Place* uvedeny v sekcích ***Kód 8.1*** & ***Kód 8.2.*** Pro větší přehlednost jsou tyto sekce kódů uvedeny na následujících stranách. Následuje ***Kód 8.3.***, který je součástí souboru obsahující třídu *Place* napsanou v C++ a opět je jen pro přehlednost uveden na samostatné straně.

Třída *Place* v C++ implementaci byla napsána v prostředí Visual Studio od společnosti Microsoft, který na svých stránkách uvádí i návod, který s použitím knihovny Pybind11 zajistí po sestavení kódu výstup jako modul s příponou *.pyd*, který lze v Pythonu importovat. Tento návod, který je důležitý předně z hlediska nastavení některých parametrů projektu, je dostupný na stránkách Microsoftu ve článku *Vytvoření rozšíření C++ pro Python* [37].

Tímto jsme úspěšně vytvořili modul napsaný v C++ obsahující třídu *Place.* Tento čin samotný nám ovšem nezajistí zrychlení knihovny PetNetSim, neboť těžištěm tohoto snažení by měl být modul, který se postará o výpočetně náročné operace. Nyní máme však dobrý základ k tomu, abychom tak učinili. Jak bylo ovšem zmíněno v úvodu kapitoly, tato práce se touto problematikou zabývat nebude a může být předmětem jiného diplomového projektu.

Následující stránky budou obsahovat už jen sekce kódu ***Kód 8.1***, ***Kód 8.2*** & ***Kód 8.3*** s drobným komentářem.

[8] Petriho sítě, které PetNetSim simuluje, operují s místy, přechody a hranami.

Kód 8.1: Třída Place v Pythonu

```
class Place:
    INF_CAPACITY = 0

    def __init__(self, name=None, init_tokens=0, capacity=INF_CAPACITY,
context=default_context()):
        if name is None:
            self.name = 'P_'+str(context['counters']['P'])
            context['counters']['P'] += 1
        else:
            match = re.fullmatch(r'P_(\d+)', name)
            if match is not None:
                context['counters']['P'] = int(match.group(1)) + 1

            self.name = name
        self.capacity = capacity
        self.init_tokens = init_tokens
        self.tokens = init_tokens

    def can_add(self, n_tokens):
        return self.capacity == Place.INF_CAPACITY or self.tokens +
n_tokens <= self.capacity

    def can_remove(self, n_tokens):
        return self.tokens - n_tokens >= 0

    def add(self, n_tokens):
        self.tokens += n_tokens

    def remove(self, n_tokens):
        self.tokens -= n_tokens

    def reset(self):
        self.tokens = self.init_tokens

    def clone(self, prefix):
        p = copy(self)
        p.name = prefix+p.name
        return p
```

Tato stránka znázorňuje původní část kódu knihovny PetNetSim. Jedná se o třídu Place, která definuje a umožňuje práci s místy, což je jeden ze základních prvků Petriho sítí, které knihovna PetNetSim simuluje.

Kód 8.2: Třída Place v C++

```
#include <iostream>

const unsigned INF_CAPACITY = 0;

class Place
{
    friend class PetriNet;
    unsigned int capacity;
    unsigned int init_tokens;
    unsigned int tokens;
    std::string name;

public:
    Place(
        const std::string name_,
        unsigned int init_tokens_ = 0,
        unsigned int capacity_ = INF_CAPACITY
    ) :
        name(name_), init_tokens(init_tokens_), capacity(capacity_) {
    }
    void setName(const std::string& name_)
    {name = name_;}
    const std::string& getName() const
    {return name;}

    bool can_add(unsigned int n_tokens)
    {return capacity == INF_CAPACITY || tokens + n_tokens <= capacity;}

    bool can_remove(unsigned int n_tokens)
    {return tokens - n_tokens >= 0;}

    void add(unsigned int n_tokens)
    {tokens += n_tokens;}

    void remove(unsigned int n_tokens)
    {tokens -= n_tokens;}

    void reset()
    {tokens = init_tokens;}

    void setTokens(unsigned int tokens)
    {tokens;}
    const unsigned int& getTokens() const
    {return tokens;}

    Place clone(std::string prefix)
    {
        Place p = Place(this->name, this->init_tokens, this->capacity);
        p.name = prefix + p.name;
        return p;
    }
};
```

Tato stránka zobrazuje kód třídy *Place* napsaném v jazyce C++. Z hlediska funkce je tato implementace ekvivalentní s předchozím kódem napsaným v Pythonu.

Kód 8.3: Wrapper pro třídu Place pomocí Pybindu11

```
#include <pybind11/pybind11.h>

namespace py = pybind11;
using namespace std;

PYBIND11_MODULE(Place, m)
{
    // define all classes
    py::class_<Place>(m, "Place")
        .def(py::init<string, int, int>()) // constructor
        .def("can_add", &Place::can_add)
        .def("can_remove", &Place::can_remove)
        .def("add", &Place::add)
        .def("remove", &Place::remove)
        .def("reset", &Place::reset)
        .def("clone", &Place::clone)
        .def_property("name", &Place::getName, &Place::setName)
        .def_property("tokens", &Place::getTokens, &Place::setTokens);
    m.doc() = "C++ module for PetNetSim, class Place";
}
```

Tato část kódu je součástí souboru obsahující i ***Kód 8.2.*** Jedná se o použití wrapperu Pybind11, který umožňuje Pythonu přistupovat k jednotlivým atributům třídy *Place*, které jsme sami definovali. Pomocí příkazu *m.doc()* je možné (ne nutné) zadat popis našeho modulu. Výsledkem je tak rozšiřující modul Place.pyd, který je možné ve stávající verzi knihovny PetNetSim použít a je součástí přílohy.

9 ZÁVĚR

Cílem této práce byla rešerše alternativních interpretů jazyka Python využívajících JIT kompilaci, dále rešerše rozšiřujících modulů, které by mohly zvýšit výkon kódu napsaném v Pythonu a rešerše tzv. transpilerů, díky kterým by bylo možné kód kompilovat. Dále bylo cílem prozkoumat možnosti vývoje výpočetního jádra coby rozšiřujícího modulu napsaném v jednom ze systémových jazyků. Po provedení rešerše bylo cílem implementovat zvolené řešení za účelem zvýšení výkonu kniovny PetNetSim.

Teoretická část této práce se zabývala porovnáním tří alternativních interpretů s funkcí JIT kompilace (Pyston, PyPy, Cinder), dvou rozšiřujících modulů (Numba, Pyjion), dvou nástrojů pro převedení kódu do jazyka C s následnou kompilací kódu (Cython, Pythran) a implementací výpočetního jádra napsaném v jednom ze systémových jazyků coby rozšiřující modul (C++ - Boost.Python, Pybind11, Rust – PyO3).

Pro dosažení nejlepších výsledků bylo přistoupeno k otestování co největšího počtu alternativních zrychlujících metod za účelem výběru té nejvhodnější pro naši konkrétní aplikaci. V 6. kapitole tak byly provedeny testy na 12 úlohách pokrývajících aplikační schopnosti knihovny PetNetSim a pro statistické účely bylo provedeno 20 běhů pro každou úlohu pro každou z testovaných metod a z naměřených dat byly vyvozeny závěry. Testovány byly: Pyston, PyPy, Cinder, Pyjion, Cython (Numba byla zavržena z důvodu obtížné aplikovatelnosti na naši konkrétní aplikaci). Z těchto metod vyšla jako nerychlejší a nejspolehlivější zrychlující metoda alternativní interpret **Pyston**, která dosahovala zrychlení 30 – 40 %. Nad rámec zadání byla dále implementována metoda multiprocessingu na rozsáhlejší stochastickou síť. Ta se ukázala jako velice účinná a v případě nutnosti výpočtu více běhů je tato metoda, která je popsaná v kapitole 7, silně doporučena.

Během měření bylo zjištěno, že další optimalizace lze dosáhnout zakázáním výpisu informací v jednotlivých krocích. Na systému Ubuntu tato metoda dosáhla zvýšení výkonu jen o cca 7 %, v případě Windows 11 dosáhla v průměru 12 úloh zvýšení výkonu o cca 450 %. Tento výsledek je poměrně zajímavý a je pravděpodobně důvodem rozdílů výkonů nativního interpreta CPython na Ubuntu a Windows 11. Při konstrukci úloh v každém případě nelze vypisování mezivýsledků doporučit, jelikož zpomalují běh úloh a neposkytují důležité informace. Pokud by tyto informace bylo z nějakého důvodu důležité zaznamenat, bude lepší je ukládat např do .csv souborů, což by mohl být námět na vylepšení pro další verzi knihovny PetNetSim.

Závěrečná kapitola se věnuje nastínění vývoje výpočetního jádra coby rozšiřující modul napsaný v systémovém jazyce. Tato metoda je demonstrována na jedné ze tříd knihovny PetNetSim a tento rozšiřující modul je součástí přílohy. Vývoj výpočetního jádra by mohl být předmětem jiné závěrečné práce.

10 BIBLIOGRAFIE

[1] *Python Software Foundation. History and License.* [online]. [cit. 2021-12-12]. Dostupné z: https://docs.python.org/3/license.html

[2] What is Python? Executive Summary. In: *Python* [online]. [cit. 2021-12-12]. Dostupné z: https://www.python.org/doc/essays/blurb/

[3] *What is Python? Powerful, intuitive programming* [online]. USA, 2019 [cit. 2021-12-12]. Dostupné z: https://www.infoworld.com/article/3204016/what-is-python-powerful-intuitive-programming.html

[4] SUBASI, Abdulhamit. *Practical Machine Learning for Data Analysis Using Python*. Jeddah, Saudi Arabia, 2020. ISBN 978-0-12-821379-7.

[5] LUTZ, Mark. *Learning Python*. 2. USA: O'Reilly Media, Inc., 2009. ISBN 978-0-596-15806-4.

[6] LAI, Kee-hung a T.C. CHENG. *Justi-In-Time Logistics*. 1. New York, USA: Routledge, 2016. ISBN 978-0-566-08900-8.

[7] *Just in Time Compilation Explained* [online]. 2020 [cit. 2022-04-11]. Dostupné z: https://www.freecodecamp.org/news/just-in-time-compilation-explained/

[8] *Pyston: Enhance your Python* [online]. 2021 [cit. 2022-01-26]. Dostupné z: https://www.pyston.org/

[9] Pyston Github. In: *Github* [online]. [cit. 2022-01-26]. Dostupné z: https://github.com/pyston/pyston

[10] *PyPy* [online]. [cit. 2022-01-27]. Dostupné z: https://www.pypy.org/

[11] *PyPy Documentation* [online]. [cit. 2022-01-27]. Dostupné z: https://doc.pypy.org/en/latest/

[12] BOLZ, Carl Friedrich, Antonio CUNI, Maciej FIJALKOWSKI a Rigo ARMIN. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. New York, NY, USA: Association for Computing Machinery, **2009**, 18–25. ISSN 9781605585413. Dostupné z: doi:10.1145/1565824.1565827

[13] *PyPy Speed* [online]. [cit. 2022-02-24]. Dostupné z: https://speed.pypy.org/

[14] *Cinder GitHub* [online]. 2022 [cit. 2022-03-20]. Dostupné z: https://github.com/facebookincubator/cinder

[15] *Inline Caching: Max Bernstein* [online]. 2021 [cit. 2022-03-20]. Dostupné z: https://bernsteinbear.com/blog/inline-caching/

[16] *Jython Wiki* [online]. 2015 [cit. 2022-03-03]. Dostupné z: https://wiki.python.org/jython/FrontPage

[17] *Jython* [online]. [cit. 2022-03-03]. Dostupné z: https://www.jython.org/

[18] *C-Extention Tutorial: What is an Extension Module?* [online]. Joe Jevnik, 2017 [cit. 2022-03-28]. Dostupné z: https://llllllllll.github.io/c-extension-tutorial/what-is-an-extension-module.html

[19] *Numba* [online]. Anaconda, Inc. and others, 2020 [cit. 2022-03-28]. Dostupné z: https://numba.pydata.org/numba-doc/latest/index.html

[20] *Pybind11 GitHub* [online]. 2022 [cit. 2022-03-29]. Dostupné z: https://github.com/pybind/pybind11

[21] *Pybind11* [online]. 2017 [cit. 2022-03-29]. Dostupné z: https://pybind11.readthedocs.io/en/latest/index.html

[22] How to write a transpiler. In: *Strumenta.com* [online]. [cit. 2021-12-12]. Dostupné z: https://tomassetti.me/how-to-write-a-transpiler/

[23] *Things You Should Never Do, Part I* [online]. 2000 [cit. 2021-12-12]. Dostupné z: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

[24] Understanding ASTs by Building Your Own Babel Plugin. In: *Sitepoint* [online]. 2016 [cit. 2021-12-12]. Dostupné z: https://uploads.sitepoint.com/wp-content/uploads/2016/04/1459949808babel.png

[25] *Cython: An Overview* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://cython.readthedocs.io/en/latest/src/quickstart/overview.html

[26] *Cython official website* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://cython.org/

[27] *Cython: Building a Cython code* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://cython.readthedocs.io/en/latest/src/quickstart/build.html

[28] *Cython: Faster code via static typing* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://cython.readthedocs.io/en/latest/src/quickstart/cythonize.html

[29] *Pythran: Official Website* [online]. 2014 [cit. 2022-04-05]. Dostupné z: https://pythran.readthedocs.io/en/latest/

[30] *Pythran: SciPy 2013 Presentation* [online]. 2013 [cit. 2022-04-05]. Dostupné z: https://youtu.be/KT5-uGEpnGw

[31] *Build Python C-Extension module* [online]. 2022 [cit. 2022-03-29]. Dostupné z: https://realpython.com/build-python-c-extension-module/

[32] FORBES, Elliot. *Creating Basic Python C Extensions - Tutorial* [online]. 2017 [cit. 2022-03-29]. Dostupné z: https://tutorialedge.net/python/python-c-extensions-tutorial/

[33] *Boost Python* [online]. [cit. 2022-03-29]. Dostupné z: https://www.boost.org/

[34] *Rust: Programming Language* [online]. 2022 [cit. 2022-04-02]. Dostupné z: https://www.rust-lang.org/

[35] *PyO3: User Guide* [online]. 2022 [cit. 2022-04-04]. Dostupné z: https://pyo3.rs/v0.16.2/

[36] *PetNetSim: Github* [online]. 2022 [cit. 2022-05-07]. Dostupné z: https://github.com/karna48/petnetsim

[37] *Microsoft: Vytvoření rozšíření C++ pro Python* [online]. 2022 [cit. 2022-05-18]. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2022&viewFallbackFrom=vs-2022.

[38] *Pyrex - Python Wiki* [online]. 2016 [cit. 2022-04-04]. Dostupné z: https://wiki.python.org/moin/Pyrex

[39] *ECMA International: ECMA-335 - Common Language Infrastructure (CLI)* [online]. 6. Ženeva: ECMA International, 2016 [cit. 2022-05-07]. Dostupné z: https://www.ecma-international.org/publications-and-standards/standards/ecma-335/

11 SEZNAM PŘÍLOH

- Tabulka časů úloh pro CPython Windows 11
- Tabulka časů úloh pro CPython Ubuntu 20.04
- Tabulka časů úloh pro Pyston
- Tabulka časů úloh pro PyPy
- Tabulka časů úloh pro Pyjion
- Tabulka časů úloh pro Cython
- Tabulka časů úloh pro Cinder
- Excel soubor Tabulky.xlsx obsahující tabulky časů.
- Rozšiřující modul Place.pyd

PŘÍLOHY

CPython Windows 11

úloha/iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0,77727730	0,78261960	0,79023550	0,78916620	0,77996880	0,79272660	0,78346330	0,78158860	0,78066010	0,78451440	0,77827470	0,78377180	0,78091130	0,78408370	0,79725110	0,77704310	0,78199060	0,77950670	0,77779970	0,78697220
2	0,54589260	0,54448730	0,59792560	0,55794060	0,54425970	0,54835390	0,54217910	0,53161550	0,54719800	0,54632350	0,54532740	0,56830840	0,54840370	0,53439340	0,54904340	0,54628800	0,56011910	0,53162700	0,54096730	0,54840100
3	2,23961530	2,22771290	2,25259140	2,23458430	2,23712890	2,23178540	2,23291670	2,23690770	2,24803890	2,22992450	2,22861720	2,24611070	2,23631030	2,23577370	2,23401800	2,22861050	2,23622410	2,23704520	2,22937220	2,24060090
4	2,00069140	2,04652400	2,02080690	1,99091920	2,00160820	2,00718830	1,99213050	1,99983900	2,00694230	2,00690460	1,99898260	2,00080520	1,99996150	2,00381000	1,99925600	2,01831840	1,99144120	1,99597910	1,99447240	1,99948970
5	0,00284160	0,01320180	0,00278600	0,01244570	0,00286460	0,01286100	0,00280310	0,01326670	0,00280840	0,01324640	0,00275170	0,01294840	0,00284860	0,01315310	0,00295140	0,01297930	0,00274840	0,01284870	0,00283990	0,01324350
6	0,01290790	0,00387570	0,01413460	0,01330350	0,00418020	0,01369950	0,01348390	0,00369130	0,01438100	0,01313910	0,01377650	0,00385110	0,01412090	0,01377800	0,00342950	0,01415720	0,01415230	0,00349080	0,01457630	0,01415960
7	0,00056640	0,00046210	0,00044530	0,00043740	0,00043260	0,01281430	0,00049350	0,00051880	0,00062120	0,00048300	0,01275520	0,00071900	0,00061240	0,00044720	0,00043960	0,00051060	0,01338670	0,00047520	0,00043820	0,00044270
8	0,09228140	0,09400120	0,07878160	0,09315710	0,09277120	0,08108380	0,09209910	0,09298450	0,08002070	0,09277590	0,09192440	0,07983820	0,09350850	0,09094860	0,07987060	0,09529310	0,09280230	0,08089060	0,09409720	0,09265450
9	0,00146570	0,01505290	0,01489430	0,00219880	0,01387760	0,01622830	0,01519410	0,00156860	0,01473300	0,00159090	0,01456320	0,01461610	0,00154730	0,01411080	0,01586560	0,01526570	0,01463250	0,00166290	0,01582960	0,01508540
10	0,00231650	0,01256320	0,00229960	0,00226580	0,01184010	0,00244680	0,01118380	0,00259180	0,00233560	0,01111820	0,00227610	0,01250340	0,00243280	0,00239670	0,01111370	0,00226180	0,01278730	0,00234390	0,00231210	0,01144850
11	0,01572850	0,01633060	0,11849500	0,00759430	0,01443900	0,01589830	0,01450220	0,01543110	0,01589290	0,01509550	0,02634370	0,01572290	0,00434460	0,02106200	0,01036520	0,01605470	0,01579660	0,01604730	0,01617250	0,01443190
12	0,03080790	0,04477430	0,03199780	0,03085010	0,03241350	0,03164160	0,04626560	0,03187170	0,03133890	0,03222220	0,03226640	0,04567760	0,03180600	0,03204750	0,03131770	0,03198690	0,04572230	0,03200260	0,03155260	0,03036200

CPython Ubuntu 20.04

úloha/iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0,34120787	0,32481833	0,34194559	0,23680153	0,23142256	0,24925548	0,27215198	0,27450789	0,25403642	0,26093593	0,36095591	0,41419091	0,24299634	0,34270263	0,25105186	0,28022249	0,23998493	0,24893781	0,23289189	0,21473432
2	0,26708689	0,24374273	0,19735044	0,18836100	0,23602511	0,24167777	0,23376602	0,21232459	0,19797568	0,17885770	0,20376909	0,18678168	0,20417089	0,20985524	0,19969261	0,21539539	0,20940884	0,19050545	0,19135157	0,18172924
3	0,56697435	0,54120685	0,58903719	0,64123630	0,68013907	0,72826326	0,71899634	0,70574996	0,87259978	0,63184621	0,60030839	0,63271822	0,69986173	0,74109032	0,79652320	0,61336837	0,71063288	0,94722568	0,74901866	0,85949635
4	0,97278391	1,05219264	1,21683514	1,15118206	1,04901343	0,85665376	0,79177792	0,75771075	0,71083667	0,82975800	0,75030796	0,97054204	0,77755655	0,82110028	0,82283049	0,91671296	1,18389853	0,97202224	1,14739701	0,91904914
5	0,00237796	0,00233838	0,00349589	0,00497536	0,00207463	0,00256866	0,00223704	0,00206812	0,00235056	0,00379485	0,00223040	0,00216874	0,00230705	0,00199032	0,00203864	0,00205907	0,00398148	0,00194090	0,00242033	0,00219290
6	0,00265456	0,00268476	0,00423930	0,00273829	0,00264285	0,00272760	0,00313167	0,00271245	0,00278895	0,00468271	0,00269967	0,00282811	0,00261106	0,00420708	0,00380097	0,01444877	0,00792958	0,00269846	0,00943132	0,00454160
7	0,00103815	0,00549429	0,00042255	0,00117131	0,00040968	0,00047388	0,00029345	0,00034485	0,00035687	0,00058165	0,00064875	0,00042333	0,00028576	0,00028258	0,00045708	0,00047237	0,00047167	0,00029438	0,00027937	0,00039114
8	0,03210810	0,03394637	0,05110866	0,03832567	0,02439857	0,01241806	0,03087469	0,01265001	0,01193630	0,01270177	0,01146567	0,01519112	0,01254181	0,03511616	0,02179865	0,01737514	0,01750805	0,01586312	0,01482502	0,01261243
9	0,00104953	0,00152339	0,00097246	0,00106418	0,00126178	0,00101890	0,00255791	0,00309253	0,00128702	0,00326444	0,00142143	0,00174765	0,00128304	0,00129439	0,00179197	0,00134039	0,00224944	0,00196686	0,00096470	0,00095364
10	0,00193159	0,00199428	0,00180960	0,00190863	0,00185139	0,00183501	0,01082752	0,00189070	0,00220323	0,00226829	0,00264537	0,00989466	0,00276699	0,00807438	0,00216045	0,00179206	0,00226650	0,00263295	0,00229901	0,00170968
11	0,00436735	0,00353711	0,00306804	0,00341707	0,00282607	0,00277276	0,00426775	0,00826922	0,01264303	0,02474073	0,00313261	0,00681260	0,00654088	0,00317655	0,00315312	0,00600920	0,00333073	0,00277850	0,00287848	0,00362979
12	0,00529204	0,00633077	0,00591897	0,00532564	0,00404117	0,00517559	0,00567995	0,02108027	0,00399794	0,00693416	0,00622456	0,00598562	0,00523271	0,00466367	0,00694686	0,00398372	0,00389166	0,00419913	0,00406571	0,00393633

Pyston

úloha/iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0,20571946	0,16960728	0,17941110	0,18424060	0,16743844	0,15609567	0,17508686	0,17322660	0,19178729	0,18752320	0,16249738	0,17488724	0,15894615	0,14871621	0,16765815	0,17072438	0,21145862	0,18473373	0,16685163	0,16723104
2	0,15678591	0,14763864	0,18589191	0,17634667	0,15029886	0,13463578	0,19256295	0,13656754	0,14134171	0,13830185	0,14808518	0,13586524	0,17102845	0,21536906	0,15309777	0,18299115	0,16661072	0,16478777	0,12989273	0,12782336
3	0,42831351	0,48556368	0,43981449	0,46509357	0,47552333	0,42936161	0,43311223	0,43010362	0,57636057	0,45367152	0,49179212	0,41129281	0,43651862	0,47018009	0,43964619	0,43063057	0,42202830	0,42879743	0,46915915	0,45633054
4	0,56510599	0,59509013	0,59132605	0,57765023	0,56546995	0,58268285	0,58047943	0,56444199	0,55773906	0,55471630	0,57126536	0,58144130	0,59022646	0,74061065	0,60516232	0,65642888	0,57921883	0,58511429	0,57270522	0,58045136
5	0,00166531	0,00161546	0,00160334	0,00152803	0,00155142	0,00166898	0,00154172	0,00181191	0,00169583	0,00162893	0,00168197	0,00170083	0,00155803	0,00278955	0,00197211	0,00175173	0,00158436	0,00167870	0,00162225	0,00157809
6	0,00249594	0,00239950	0,00277290	0,00346686	0,00287612	0,00283802	0,00325543	0,00335581	0,00349763	0,00346839	0,00389319	0,00411115	0,00322796	0,00255483	0,00258536	0,00226236	0,00258028	0,00215660	0,00234017	0,00254245
7	0,00022666	0,00020943	0,00022283	0,00027097	0,00022404	0,00020884	0,00030845	0,00039109	0,00021604	0,00020555	0,00029938	0,00022156	0,00020862	0,00020475	0,00022347	0,00112278	0,00094045	0,00038244	0,00086579	0,00137163
8	0,01167844	0,00929233	0,00987715	0,01003176	0,01236174	0,00923180	0,01040642	0,01110856	0,01430619	0,01014592	0,01085153	0,00900351	0,00896842	0,00893296	0,00926771	0,00906646	0,00889267	0,00981517	0,01615237	0,00993850
9	0,00141856	0,00112098	0,00103797	0,00096260	0,00119798	0,00153282	0,00096333	0,00092631	0,00080566	0,00078150	0,00075314	0,00114275	0,00115550	0,00075875	0,00073188	0,00070639	0,00142604	0,00101192	0,00100099	0,00096878
10	0,00138715	0,00155932	0,00182929	0,00315389	0,00201863	0,00197294	0,00133387	0,00128596	0,00129579	0,00129638	0,00128359	0,00129498	0,00126763	0,00135117	0,00131929	0,00128755	0,00159895	0,00135520	0,00128217	0,00204023
11	0,00509690	0,00355952	0,00269126	0,00212893	0,00230684	0,00225956	0,00205439	0,00189176	0,00208434	0,00206832	0,00236838	0,00195488	0,00192045	0,00223753	0,00193057	0,00191904	0,00210381	0,00207866	0,00252311	0,00203442
12	0,00304593	0,00311817	0,00371057	0,00393471	0,00376757	0,00774275	0,00344174	0,00300789	0,00295183	0,00431205	0,00307949	0,00333790	0,00499505	0,00482960	0,00371846	0,00320744	0,00399203	0,00327394	0,00351240	0,00384201

PyPy

úloha/iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1,07601260	1,00601260	0,95127380	0,96127380	0,95127380	0,87808520	0,88097550	0,88930410	0,91089440	0,93703140	0,88727870	0,86396280	0,89702830	0,87227910	0,87687240	0,86047750	0,91033000	0,92133000	0,91033000	0,90193370
2	1,02831760	0,90204140	0,78657470	0,84379760	0,78608530	0,74319510	0,74408680	0,75101400	0,73229880	0,73005390	0,71048110	0,74053730	0,76008490	0,73401250	0,78322040	0,73630670	0,73284380	0,74967760	0,74228110	0,73658770
3	2,41001950	2,35487640	2,47537970	2,70545210	2,35446670	2,34565070	2,36470050	2,37299960	2,45563190	2,32650320	2,38502730	2,35184230	2,37933570	2,32828980	2,32354590	2,35372440	2,33975750	2,34410250	2,33328220	2,31878620
4	3,06618180	2,92462200	2,95648030	2,97638360	3,17456720	3,30062690	3,76189250	3,22897460	3,42378140	3,18663770	3,25795620	3,13621950	3,03558290	3,10935140	3,08879270	3,07757340	3,08063880	3,03997470	3,04357070	3,10583110
5	0,01233060	0,02103050	0,01792650	0,01151850	0,01446690	0,01435760	0,01077740	0,00984850	0,02109220	0,01045410	0,01024470	0,01326690	0,00926970	0,01627420	0,00997740	0,00982100	0,00949690	0,01099790	0,03362370	0,01457300
6	0,01856670	0,01489480	0,01767250	0,32868250	0,01502950	0,04262360	0,01542470	0,03669790	0,02114940	0,01559380	0,01449560	0,01363100	0,02551080	0,01356400	0,01338010	0,02340680	0,02341050	0,01498690	0,01819450	0,01371960
7	0,00141480	0,00144710	0,00134620	0,00146830	0,00784460	0,00145350	0,00132670	0,00163480	0,00131730	0,00879220	0,00135700	0,00144310	0,00170530	0,00240470	0,00150000	0,00706660	0,00160850	0,00149440	0,00187260	0,00186280
8	0,09335770	0,09031130	0,09225010	0,10685070	0,08411940	0,08859140	0,09264120	0,08478590	0,09090230	0,09149820	0,09193460	0,08004410	0,09210520	0,09453640	0,09160870	0,08507040	0,09914000	0,08355060	0,09251080	0,09023110
9	0,00479780	0,01322770	0,01411220	0,00436500	0,01203790	0,02618840	0,02007360	0,00447860	0,01371690	0,01330730	0,00436780	0,01309720	0,01324750	0,01727310	0,01357050	0,00631190	0,01160460	0,01266600	0,00781090	0,00952960
10	0,00807420	0,01163430	0,01316820	0,00806320	0,01417460	0,00967280	0,02739520	0,01294600	0,01781270	0,00851640	0,00961410	0,00783660	0,01046050	0,00739610	0,01218960	0,00790750	0,00829680	0,00782080	0,00760260	0,00890770
11	0,02898850	0,02771820	0,03518870	0,02359280	0,03003690	0,02156560	0,02240010	0,02142450	0,05394070	0,01887820	0,01846820	0,01915920	0,03091650	0,02397840	0,02221240	0,02577210	0,01792710	0,01777980	0,02841420	0,01765120
12	0,04052210	0,04209350	0,04777920	0,03813850	0,04686990	0,04249790	0,03260650	0,03308740	0,03361460	0,04265260	0,03172560	0,03250040	0,03513600	0,04206410	0,03204570	0,03271470	0,03303690	0,03238300	0,04229090	0,03276200

Pyjion

úloha/iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0,86876070	0,86024150	0,80471570	0,78156240	0,77957840	0,77953570	0,78121340	0,77784110	0,78514990	0,78032990	0,79272580	0,78467500	0,78278880	0,77467210	0,78287060	0,78163340	0,78214880	0,78376440	0,78520230	0,78147210
2	0,58753460	0,57943420	0,54602130	0,55542290	0,56541830	0,56139480	0,55484340	0,55793300	0,54545980	0,54870660	0,55716690	0,55571030	0,56564740	0,55723440	0,53451480	0,56262620	0,57297730	0,54886470	0,56354780	0,55133010
3	2,27665020	2,26392640	2,25281330	2,25967720	2,27102470	2,26294020	2,23609160	2,23302720	2,25005960	2,22332610	2,23031570	2,28258440	2,25022240	2,25092260	2,23806830	2,26339750	2,24283020	2,23139720	2,22746810	2,23420870
4	2,01899010	1,98963640	2,00567280	2,01096610	2,00483240	2,05000510	1,99680730	1,99829740	2,00526860	2,01725720	1,99871040	2,02471010	2,00895580	1,99714740	2,00825570	2,00616240	2,01608500	2,00979620	2,02605820	2,01091330
5	0,06171260	0,00819370	0,00377810	0,01199920	0,00460850	0,01166780	0,00385430	0,01199420	0,00377620	0,01218840	0,00418010	0,01208420	0,00433320	0,01136240	0,00435160	0,01163030	0,00472140	0,01031170	0,00402550	0,01105840
6	0,03921800	0,01097280	0,01386190	0,00485870	0,01377780	0,01416790	0,00444920	0,01395690	0,01365430	0,00458900	0,01312970	0,01318540	0,00423680	0,01457140	0,01407450	0,01379440	0,00435840	0,01366390	0,00456850	0,01246370
7	0,04293420	0,00068160	0,00053900	0,00056170	0,00055560	0,01370090	0,00080810	0,00054940	0,00053730	0,00056450	0,01322520	0,00086310	0,00084650	0,00055500	0,00065820	0,00053920	0,01203210	0,00063650	0,00055440	0,00054720
8	0,12297230	0,09117400	0,09085920	0,08172990	0,09458090	0,09087970	0,08319560	0,09072710	0,09290980	0,08141120	0,09224040	0,09147130	0,08225570	0,09275940	0,08963690	0,08306020	0,09252740	0,09165520	0,08072330	0,09177180
9	0,06052010	0,01712250	0,01513350	0,01551050	0,01448640	0,01690620	0,01396880	0,01549630	0,01613200	0,01552010	0,00181270	0,01487230	0,01668810	0,01773330	0,02851420	0,01634490	0,00251160	0,01435750	0,01468300	0,01598210
10	0,04482110	0,00324980	0,00292730	0,01170120	0,00283390	0,00274120	0,01236000	0,00297250	0,01032460	0,00293650	0,00271720	0,01040780	0,00276970	0,01192680	0,00292580	0,00273160	0,01185590	0,00275720	0,01181210	0,00340840
11	1,00625620	0,16352460	0,01446560	0,01593920	0,01733240	0,01457670	0,01493040	0,01589100	0,01473070	0,01622820	0,01558610	0,01678460	0,01616310	0,01575700	0,01623310	0,01459700	0,01606370	0,01498570	0,01600470	0,01513570
12	0,14984000	0,03104380	0,03148250	0,04481030	0,03139730	0,03166090	0,03053330	0,03021470	0,04627830	0,03048170	0,03043020	0,03139650	0,03256190	0,04614680	0,03140560	0,03168250	0,03270560	0,04423000	0,03240280	0,03257340

Cython

úloha/iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0,78176410	0,78070270	0,78178750	0,78234980	0,79205900	0,79027010	0,81640910	0,78414310	0,78721010	0,78193570	0,78926470	0,78668740	0,78957180	0,78919370	0,78111650	0,82403660	0,78057020	0,78971950	0,78712030	0,78583120
2	0,54826000	0,55303670	0,53752870	0,54875960	0,55131260	0,55220650	0,56028830	0,53527080	0,53466660	0,54853770	0,57624000	0,54988340	0,53333250	0,54705300	0,54249380	0,55607640	0,55371970	0,55629630	0,54929140	0,55975900
3	2,46397290	2,22474450	2,25414530	2,22882240	2,21208370	2,22134890	2,22822850	2,22990770	2,24119830	2,22694260	2,22274180	2,22226520	2,22097290	2,22093130	2,22228380	2,23612450	2,23384360	2,22864250	2,22206730	2,22492400
4	1,99592510	1,98612220	2,00115500	2,01875790	1,99702860	1,99609590	1,98994130	1,99970710	1,99689960	1,98266550	1,98810030	2,01235160	2,03776320	1,99422970	1,98679630	1,99965390	1,99736980	1,99477090	2,05202210	2,00467720
5	0,26612580	0,01315920	0,00280440	0,00265660	0,01370490	0,00287210	0,01255980	0,00271250	0,01350600	0,00268120	0,01299460	0,00264000	0,01337680	0,00264200	0,01330820	0,00276280	0,01228480	0,00266070	0,01237030	0,00266920
6	0,03488750	0,00838670	0,01415130	0,00360590	0,01442930	0,01447000	0,00411140	0,01384870	0,01306440	0,00427400	0,01462820	0,01256340	0,00424550	0,01396540	0,01369770	0,00388270	0,01457890	0,01419980	0,01290990	0,00408330
7	0,00304000	0,00040500	0,00043280	0,00040160	0,00039860	0,00844100	0,00047760	0,00040980	0,00038150	0,00039110	0,01327540	0,00043130	0,00042600	0,00038450	0,00044620	0,00038050	0,01306000	0,00042100	0,00038880	0,00038940
8	0,10836020	0,09238940	0,09481570	0,07815430	0,09496640	0,09240450	0,08000430	0,09388940	0,09097490	0,08107600	0,09493260	0,09320400	0,07914200	0,09315630	0,09462300	0,07780560	0,09308190	0,09205330	0,08009640	0,09398000
9	0,00780860	0,00132400	0,01507100	0,00129910	0,02964000	0,00134670	0,01536230	0,01497270	0,01500610	0,01522300	0,01663970	0,00159350	0,01471450	0,01490490	0,00131780	0,01470820	0,01538590	0,01550770	0,01446040	0,00212140
10	0,01605310	0,00249520	0,01194900	0,00229560	0,00225380	0,01299810	0,00253510	0,01188900	0,00279870	0,00236990	0,01151780	0,00237260	0,01079660	0,00247590	0,00239390	0,01271250	0,00229680	0,01243060	0,00258900	0,00256100
11	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
12	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA

Cinder

úloha/iterace	1	2	3	4	5	6	7	8	9	10
1	0,17098927	0,14506125	0,13220263	0,13671970	0,13173747	0,14142895	0,13342261	0,13484478	0,13228440	0,13642359
2	0,13555646	0,13837481	0,14916587	0,14133263	0,13669682	0,14390278	0,13637495	0,16340661	0,16976118	0,14018130
3	0,39107847	0,37920356	0,37636232	0,37156773	0,45401573	0,39428687	0,38028479	0,37418389	0,37604785	0,40168905
4	0,55150008	0,54598880	0,55132365	0,54968143	0,55328465	0,55285835	0,56469202	0,54799438	0,55342627	0,55107999
5	0,00169086	0,00144625	0,00155187	0,00145626	0,00147390	0,00144267	0,00148273	0,00143290	0,00146174	0,00148439
6	0,00206137	0,00231957	0,00210261	0,00194836	0,00196385	0,00206923	0,00210857	0,00201297	0,00208569	0,00211310
7	0,00024462	0,00022840	0,00027561	0,00023508	0,00026011	0,00030661	0,00033259	0,00022769	0,00022149	0,00026131
8	0,00940967	0,00981498	0,00989151	0,00967336	0,00955391	0,01061487	0,00972581	0,00958347	0,00947976	0,00949073
9	0,00110364	0,00070953	0,00074649	0,00071526	0,00069785	0,00070262	0,00067782	0,00126243	0,00133848	0,00107360
10	0,00147676	0,00144958	0,00140905	0,00140882	0,00164390	0,00144577	0,00146174	0,00149202	0,00142694	0,00157976
11	0,00362968	0,00274086	0,00267553	0,00287032	0,00269675	0,00304389	0,00286651	0,00272727	0,00273824	0,00286937
12	0,00338650	0,00301003	0,00306296	0,00313210	0,00317049	0,00334430	0,00314593	0,00307083	0,00304365	0,00325727

úloha/iterace	11	12	13	14	15	16	17	18	19	20
1	0,13363218	0,13740015	0,14071274	0,13412428	0,13206005	0,13662505	0,13369250	0,13496208	0,13287520	0,13413286
2	0,14491796	0,13950038	0,13424706	0,14055610	0,13795567	0,13742399	0,13371706	0,13949132	0,14166498	0,14348269
3	0,37672591	0,37729406	0,37953019	0,37332892	0,37741971	0,37451887	0,37085032	0,37021065	0,37383175	0,37251449
4	0,54875159	0,54759717	0,54771900	0,54737258	0,54347277	0,54770708	0,54315877	0,56808734	0,54836178	0,55136967
5	0,00146651	0,00153327	0,00150537	0,00145626	0,00148392	0,00142646	0,00151467	0,00143075	0,00141048	0,00147295
6	0,00206995	0,00216722	0,00214911	0,00201440	0,00213122	0,00202942	0,00221848	0,00194097	0,00208616	0,00195694
7	0,00022721	0,00023699	0,00022316	0,00023699	0,00024891	0,00023293	0,00030732	0,00027847	0,00023460	0,00022316
8	0,00954008	0,00970626	0,00963473	0,01015234	0,00987458	0,00985765	0,00985312	0,01002789	0,00964165	0,01041985
9	0,00069857	0,00067186	0,00070214	0,00141239	0,00065875	0,00123334	0,00070477	0,00110197	0,00069141	0,00073814
10	0,00146270	0,00143051	0,00143647	0,00155497	0,00140643	0,00144506	0,00168991	0,00143361	0,00158811	0,00133681
11	0,00266600	0,00275612	0,00256371	0,00258660	0,00288367	0,00268745	0,00253081	0,00268626	0,00253177	0,00259447
12	0,00365520	0,00311804	0,00316668	0,00313878	0,00317502	0,00328279	0,00402832	0,00570941	0,00337815	0,00328898