VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ODHAD DOBY BĚHU ALGORITMU POMOCÍ STROJOVÉHO UČENÍ

ESTIMATION OF ALGORITHM EXECUTION TIME USING MACHINE LEARNING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARTIN BUCHTA**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

148327

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Buchta Martin, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Počítačové sítě
Název:	**Odhad doby běhu algoritmu pomocí strojového učení**
Kategorie:	Umělá inteligence
Akademický rok:	2022/23

Zadání:

1. Seznamte se s technikami strojového učení v oblasti predikce a interpolace.
2. Prostudujte chování a škálování distribuované implementace softwarového balíku k-Wave v závislosti na typu a velikosti vstupních dat a množství přidělených výpočetních zdrojů.
3. Vytvořte datovou sadu obsahující doby běhu pro různé velikosti vstupních dat a množství přidělených výpočetních prostředků.
4. Navrhněte metodu pro odhad doby běhu programu k-Wave pro danou velikost vstupních dat a množství přidělených výpočetních zdrojů.
5. Navržené řešení implementujte.
6. Otestujte přesnost predikce pro různé scénáře.
7. Zhodnoťte dosažené výsledky a diskutujte přínos práce pro další směrování vývoje balíku k-Wave.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	31.10.2022

Abstrakt

Cílem této práce je vytvořit model pro odhad doby běhu ultrazvukové simulace k-Wave na základě dané velikosti domény. Program využívá MPI a může být spuštěn na více uzlech superpočítače. Predikční modely byly vytvořeny s využitím symbolické regrese a následně porovnány s modely založenými na neuronových sítích. Tyto modely byly natrénovány na zaznamenaných datech. Výsledky ukazují, že modely překonávají stávající řešení. Model se symbolickou regresí dosáhl průměrné relativní odchylky 5,64 % u vhodných úloh. Model neuronové sítě dosáhl průměrné relativní odchylky 8,25 % na neznámých doménách včetně těch, které nejsou optimalizované pro simulaci k-Wave. Tato práce přináší nový, přesnější model pro předpovídání doby běhu a porovnává chybovost neuronových sítí a symbolické regrese pro tento konkrétní typ regresní úlohy. Celkově tyto modely mají potenciál praktického využití při spouštění a plánování simulací k-Wave.

Abstract

This work aims to predict the execution time of k-Wave ultrasound simulations on supercomputers based on a given domain size. The program uses MPI and can be run on multiple nodes. Prediction models were developed using symbolic regression and neural networks, both of which trained on captured data and compared against each other. The results demonstrate that the models outperform existing solutions. Specifically, the symbolic regression model achieved an average error of 5.64 % for suitable tasks, while the neural network model achieved an average error of 8.25 % on unseen domain sizes and across all tasks, including those not optimized for k-Wave simulations. This work contributes a new, more accurate model for predicting execution time, and compares the effectiveness of neural networks and symbolic regression for this specific type of regression problem. Overall, these findings suggest that new models will have important practical applications in the field of k-Wave ultrasound simulations.

Klíčová slova

odhad doby běhu programu, regrese, interpolace, extrapolace, spline, neuronová síť, strojové učení, symbolická regrese, evoluční algoritmy, predikce, superpočítač, simulace, k-wave, ultrazvuk, HeuristicLab

Keywords

estimation of execution time, regression, interpolation, extrapolation, spline, neural network, machine learning, symbolic regression, evolutionary algorithm, prediction, supercomputer, simulation, k-wave, ultrasound, HeuristicLab

Citace

BUCHTA, Martin. *Odhad doby běhu algoritmu pomocí strojového učení.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Odhad doby běhu algoritmu pomocí strojového učení

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Jiřího Jaroše, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Buchta
15. května 2023

Poděkování

Tímto bych chtěl poděkovat panu doc. Ing. Jiřímu Jarošovi, Ph.D. za jeho čas a odborné rady, které mi pomohly v dokončení této diplomové práce.

Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy České republiky prostřednictvím e-INFRA CZ (ID:90140).

Obsah

Kapitola 1

Úvod

Cílem této práce je vyvinout metodu pro odhad doby běhu ultrazvukové simulace k-Wave na superpočítačích z infrastruktury IT4Innovations. Ultrazvuk je akustické vlnění, které člověk nemůže slyšet svým uchem. Nicméně v praxi se setkáváme s použitím ultrazvuku často. Využíváme jeho vlastností např. v medicíně, pro odhad hloubky pomocí sonaru, nebo pro přenos energie na dálku, čímž můžeme cíleně zabít rakovinové buňky. [27]

Před samotným použitím ultrazvuku si musíme být jistí, co dané vyzařování ultrazvukových vln způsobí. K tomu nám může posloužit simulace ultrazvukového šíření. Pro takovou simulaci se často používá nástroj k-Wave[1]. K-Wave je balíček pro Matlab, který umožňuje provádět simulaci ultrazvukového šíření v 1D, 2D i 3D.

Provádět detailní simulace ultrazvuku s tímto nástrojem v Matlabu je ovšem časově velmi náročné, proto má balíček k-Wave i verzi implementovanou v C++, která je zároveň paralelizovaná pomocí MPI. Tato verze je vhodná pro spuštění na superpočítačích, protože umožňuje spuštění na mnoha procesorech najednou a tím dokáže výrazně snížit dobu běhu, která je potřeba k provedení simulace. V této práci se budeme zabývat právě odhadem této doby, jak dlouho potrvá, než se daná simulace na superpočítači provede.

Cílem této práce není jenom provést odhad doby, jak dlouho bude daný uživatel čekat na výsledek. Pro samotné spuštění úlohy na superpočítači je kromě jiného potřeba zadat i maximální čas, po který může daná úloha být spuštěna. Platí pak, že čím menší čas je zadán, tím dříve je úloha spuštěna. Ovšem neskončí-li úloha v tomto časovém období, pak je úloha ukončena a znamená to ztrátu rozpracované simulace. Pro nezkušeného uživatele balíčku k-Wave na superpočítači toto může být překážkou v používání, a tato práce má za cíl mu tento problém usnadnit.

Během řešení této práce jsem se musel vypořádat s tím, že data, která potřebuji k natrénování takového modelu, se obtížně získávají a je jich málo. Získávání těchto dat je časově i finančně náročné, protože je potřeba spustit různé náročné simulace na různě výkonných prostředcích. K vytvoření modelů pro odhadnutí doby běhu ultrazvukové simulace jsem se rozhodl vyzkoušet dva různé modely strojového učení pro odhad spojité veličiny. Použil jsem symbolickou regresi trénovanou pomocí genetického programování a neuronové sítě.

Výsledkem této práce je model symbolické regrese, který dokáže odhadnout dobu běhu programu k-Wave s průměrnou relativní odchylkou 5,64 % pro domény, které jsou vhodné k simulaci k-Wave (velikost domény v každé z os má vhodný rozklad na prvočísla pro knihovnu FFTW). Tato metoda dokáže spolehlivě určit dobu, která je potřeba ke spuštění ultrazvukové simulaci k-Wave na superpočítačích IT4Innovations. Pro odhad doby běhu

[1] http://www.k-wave.org/

simulace pro všechny velikosti domén (i ty, které nemají vhodný rozklad na prvočísla) lépe performovaly neuronové sítě, které měly průměrnou relativní odchylku 8,25 % na neviděných doménách během trénování.

Tato práce v kapitole 2 pojednává o superpočítačích, představuje jejich architektury a to, jak na takovém superpočítači můžeme spouštět naše výpočty. Kapitola 3 představuje balíček k-Wave a jeho spouštění na superpočítačích. Kapitola 4 shrnuje současný stav poznání v oblasti predikce, vysvětluje interpolaci, regresi, symbolickou regresi i genetické algoritmy a neuronové sítě. Kapitola 5 představuje datovou sadu, na které byla metoda trénovaná, jak tato data byla získaná a na jakých datech se ověřovala funkčnost. Kapitola 6 popisuje současný stav problému a navrhuje, jak bude touto prací vylepšen. V kapitole 7 je popsána implementace tohoto modelu a v kapitole 8 je popsáno, jak byl tento model otestován a jak byla ověřena funkčnost.

Kapitola 2

Superpočítače

Superpočítače se od klasických počítačů liší tím, že mají výrazně vyšší výpočetní výkon oproti strojům určeným k domácímu použití. Pro účely této práce budou nejdůležitější superpočítače v Česku (Karolina, Barbora), které jsou spravovány inovačním centrem IT4Innovations. V této kapitole představuji superpočítače. Kapitola 2.1 pojednává o jejich výkonu, kapitola 2.2 popisuje jejich architektury, kapitola 2.3 krátce pojednává o plánovačích, které jsou podstatné pro dělení zdrojů superpočítače a kapitola 2.4 popisuje superpočítače v IT4I, které budu v rámci této práce používat.

2.1 Výkon superpočítačů

Pro měření výkonu superpočítače se obvykle používá jednotka FLOPS, která udává počet operací v pohyblivé řádové čárce za vteřinu, které je daný superpočítač schopen vykonat. U dnešních superpočítačů se setkáme s předponami tera (10^{12}), peta (10^{15}) a nově i exa (10^{18}). Měření výkonu pomocí FLOPS je dobrým měřítkem, protože se blíží reálnému použití superpočítačů a nezohledňuje rozdíly v architekturách, které by např. počet instrukcí za sekundu (IPS) zohledňoval. Protože výkon superpočítače je ovlivněn mnoha faktory, nejen výkonem procesoru, je zvykem měřit výkon pomocí benchmarků, které řeší nějakou předem specifikovanou úlohu. Je třeba totiž počítat např. s výpadky cache pamětí a načítání dat z pamětí do procesoru. Často se pro měření výkonu superpočítačů používá Linpack benchmark[8], který řeší soustavu lineárních rovnic. Výsledkem měření je pak počet FLOPS dosažených během výpočtu. Žebříček TOP500[26] pravidelně sestavuje seznam nejvýkonnějších superpočítačů na světě. Úryvek z tohoto seznamu obsahuje tabulka 2.1. Nejvýkonnější superpočítač je Frontier, který dosáhl maximálního výkonu během Linpack benchmarku 1 102 PFlop/s. České superpočítače mu sice nemohou konkurovat v tomto benchmarku, ale Karolina se stále drží v první stovce.

Podle seznamu TOP500 jsou všechny superpočítače v tomto seznamu poháněny operačním systémem z rodiny Linux. Podobně výrazně dominuje taky architektura x86-64.

2.2 Architektury superpočítačů

Tato sekce krátce představuje architektury superpočítačů a jejich vývoj v čase. [19] Nejprve je zde popsán vektorový procesor, pak masivně paralelní procesor a na závěr architektura cluster. Z pohledu této práce nás nejvíce zajímá architektura cluster, na níž jsou posta-

Pořadí	Název	Počet jader	Rmax (PFlop/s)
1	Frontier (USA)	8 730 112	1 102,00
2	Supercomputer Fugaku (Japonsko)	7 630 848	442,01
3	LUMI (Finsko)	2 220 288	309,10
4	Leonardo (Itálie)	1 463 616	174,70
5	Summit (USA)	2 414 592	148,60
6	Sierra (USA)	1 572 480	94,64
7	Sunway TaihuLigh (Čína)	10 649 600	93,01
8	Perlmutter (USA)	761 856	70,87
9	Selene (USA)	555 520	63,46
10	Tianhe-2A (Čína)	4 981 760	61,44
...			
85	Karolina, GPU partition (Česko)	71 424	6,75
...			
226	Karolina, CPU partition (Česko)	92 160	2,84

Tabulka 2.1: Výčet ze seznamu nejvýkonnějších superpočítačů na světě podle žebříčku TOP500[26] k listopadu 2022 s důrazem na superpočítače v Česku.

veny superpočítače Karolina a Barbora, a která také dominuje v seznamu nejvýkonnějších superpočítačů TOP500. [26]

2.2.1 Vektorový procesor

Vektorové procesory jsou takové procesory, jejichž instrukce pracují s celým polem dat (vektorem) v jednom okamžiku. Na rozdíl od skalárních procesorů, které pracují právě s jednou datovou jednotkou v jeden okamžik. Vektorové procesory jsou úzce spojené s hlavní pamětí počítače. Vektorové procesory umožňují najednou spočítat velké množství výpočtů, čehož se v superpočítačích s vektorovým procesorem využívalo. [25] Vektorové procesory dominovaly na poli superpočítačů v 70. a 80. letech minulého století. V 80. letech také vznikly paralelní vektorové procesory (PVP), které ještě zlepšily výkonnost vektorových procesorů pomocí paralelního zpracování. Tyto procesory používají několik vektorových procesorů, které jsou propojené vysoce propustnou mřížkou, která spojuje procesory s hlavní pamětí. PVP typicky nepoužívají paměti cache, ale mají mnoho vektorových registrů. Vektorové procesory byly po mnoho let prakticky synonymem pro superpočítače. S koncem 80. let se ale vývoj superpočítačů s vektorovým procesorem prakticky zastavil, protože už nebylo možné dál zvyšovat taktovací frekvenci procesorů. Protože se vektorové procesory často dělaly na zakázku přímo pro superpočítač, byla jejich pořizovací cena vysoká. Technologický vývoj se však nezastavil a dnes můžeme prvky těchto procesorů vidět u moderních superpočítačů právě v podobě SIMD instrukcí na skalárních procesorech, nebo taky v grafických kartách (GPU).

2.2.2 Masivně paralelní procesor (MPP)

V oblasti superpočítačů na začátku 90. let minulého století byly vektorové procesory (respektive PVP) nahrazeny masivně paralelními procesory (MPP). Architektura MPP má stále zastánce v seznamu 500 nejlepších superpočítačů TOP500, ačkoliv dominují superpočítače s architekturou cluster.

Superpočítač s architekturou MPP používá mnoho procesorů se svou vlastní pamětí, které fungují paralelně a jsou propojeny vysokorychlostními sběrnicemi na základní desce. MPP systémy mají fyzicky distribuovanou paměť a některé používají distribuovaný vstup/-výstup. Jednotlivé procesory nejsou schopny fungovat plně samostatně a jsou mezi sebou propojeny např. multidimenzionální kostkou. Jednotlivé části superpočítače jsou pak mezi sebou úzce spojeny, což je hlavní rozdíl mezi architekturou MPP a architekturou cluster.

2.2.3 Cluster

Na konci 90. let minulého století začínala být cluster architektura populární díky rapidnímu vývoji mikropočítačů a síťových technologií. Dnes je architektura cluster nejpoužívanější architekturou mezi superpočítači ze seznamu TOP500.

Architektura cluster se skládá z jednotlivých uzlů, které dokáží pracovat samostatně. Tyto uzly jsou propojené vysokorychlostní počítačovou sítí, která umí rychle předávat data mezi jednotlivými uzly a tyto uzly tak mohou pracovat paralelně.

Superpočítače s cluster architekturou používají běžně dostupný hardware, software i síťové komponenty, narozdíl od MPP, které používá vlastní výpočetní hardware. Proto cluster superpočítače bývají cenově dostupnější, nepotřebují totiž nákladný vývoj vlastních komponent, ale poskládají se z cenově dostupnějších komponent běžně dostupných na trhu. Clustery se taky dají snáze škálovat a přidávat nové uzly. Běžně se uzly dělí do několika kategorií:

- **Login uzel** – Uzel, na který se uživatelé přihlašují a ze kterého přidávají a spravují své požadavky na výpočty na ostatních uzlech. Obvykle se nejedná o výpočetně výkonné uzly a proto bývá zakázáno provádět zde jakékoliv výpočetně náročnější operace.
- **Výpočetní uzel** – Jedná se o běžný výpočetní uzel obvykle s několika procesory, svou pamětí a často i s úložištěm.
- **Výpočetní uzel s akcelerátorem** – Oproti běžnému výpočetnímu uzlu má navíc nějaký typ hardwarového akcelerátoru, jako např. GPU pro provádění velkého množství paralelních výpočtů nebo TPU pro trénování neuronových sítí.
- **Tlustý uzel** – Jde o výpočetní uzel s velkým množstvím dostupné paměti pro výpočty, které jsou na paměť extrémně náročné.

2.3 Plánovač

Plánovač je software, který se stará o alokaci zdrojů pro jednotlivé výpočetní úkoly. Na superpočítačích s architekturou cluster obvykle najdeme nějaký plánovač běžet. Úlohy se posílají plánovači, který je umístí do některé z front a jakmile to bude možné, úlohu spustí. Proto je nutné se s plánovačem seznámit, abychom byli schopni superpočítač používat. Superpočítače obvykle neslouží jen jednomu účelu nebo uživateli, proto je plánovač důležitý, aby rozhodoval, kdo bude mít přidělené výpočetní zdroje a na jak dlouho. Pro spuštění úlohy jsou důležité tyto faktory:

- **Priorita úlohy** – Jedná se o číselný údaj, který je dán prioritou fronty, do které byla úloha zařazena. Priorita má největší vliv na to, kdy bude úloha spuštěna.

- **Čas, který je potřeba k vyřešení úlohy** – Je to odhad doby, jak dlouho bude trvat výpočet daného problému. Odhad provádí uživatel během odesílání požadavku do fronty. Čím větší je odhadovaný čas, tím později bude úloha spuštěna, protože bude obtížnější naplánovat její spuštění. Na druhou stranu není dobré tento údaj podhodnotit, protože pokud se nestihne úloha v tomto časovém intervalu dokončit, pak bude ukončena plánovačem a to může vézt ke ztrátě dat.

- **Množství vyčerpaných zdrojů uživatelem** – Snaží se rozprostřít výpočetní zdroje mezi všechny uživatele. Penalizuje uživatele, pokud spotřebovali v poslední době hodně výpočetního času, naopak pomáhá uživatelům, kteří za poslední období nepočítali.

Následující části této kapitoly obsahují krátký popis plánovače SLURM v podsekci 2.3.1 a popis plánovače PBS v podsekci 2.3.2.

2.3.1 SLURM

Slurm Workload Manager, dříve taky Simple Linux Utility for Resource Management, je open-source plánovač pro architekturu cluster. [31] SLURM je naprogramován v jazyce C, ačkoli původně byl určen pouze pro Linux, je přenositelný na ostatní Unixové operační systémy. SLURM se těší velké popularitě mezi univerzitami, laboratořemi i firmami po celém světě a jedná se o nejpoužívanější plánovač mezi superpočítači ze seznamu TOP500. Architektura plánovače SLURM se skládá z těchto částí: [17]

- **Centrální kontrolér** – Démon jménem *slurmctld*, který spravuje stav klastru. Po startu systému si tento kontrolér načte nastavení z konfiguračního systému. Během běhu periodicky zapisuje stav kontroléru na disk kvůli zajištění odolnosti vůči výpadkům. V případě výpadku je stav kontroléru obnoven z posledního zápisu.

  Kontrolér se mimo jiné periodicky dotazuje každého uzlu v klastru na jeho stav, stará se o synchronizaci konfigurace napříč všemi uzly, přijímá požadavky na spuštění úloh od uživatelů a ukládá je do prioritních front. Periodicky je také spouštěn správce úloh a pokouší se spustit úlohu s největší prioritou. Jakmile zjistí, že byla úloha na některém uzlu dokončena, spustí čištění toho uzlu, upraví stav clusteru a pokouší se pro daný uzel najít další úlohu.

- **Sekundární centrální kontrolér** – Jedná se o kopii slurmctld, která je v normálním režimu neaktivní a jen kopíruje stav centrálního kontroléru. V případě jeho selhání jej okamžitě nahradí a začne řídit cluster. SLURM se tak snaží o odolnost vůči výpadkům. V případě výpadku se aktivní a neaktivní kontrolér vymění a po selhání se snaží zotavit do konzistentního stavu.

- **Databázový démon** – Jedná se o démona jménem *slurmdbd* (SLURM database daemon), který zapisuje statistiky o používání výpočetních prostředků do databáze. Jedná se o užitečnou funkcionalitu pro zpoplatnění výpočetního času nebo pro zamezení tomu, aby jeden uživatel nebo projekt zabraly celý výpočetní výkon po příliš dlouhou dobu.

- **Lokální démon** – Démon jménem *Slurmd*, který běží na každém uzlu. Po načtení konfigurace (a případné obnovy po selhání) komunikuje s centrálním kontrolérem,

informuje ho o svém stavu, o své aktivitě, čeká na práci a tuto práci následně spouští. Musí běžet s root pravomocemi, protože spouští úlohy pro ostatní uživatele.

Lokální démon komunikuje s centrálním kontrolérem a informuje ho o stavu výpočtu na svém uzlu. Také se stará o spouštění a monitorování úloh, které mu jsou předány k běhu od centrálního kontroléru. Po dokončení práce čistí stav uzlu. Lokální démon zajišťuje přesměrování streamů stdin, stdout a stderr. Stream stdin může být přesměrován z nějakého souboru, z procesu *srun* (viz níže), nebo může být prázdný. Streamy stdout a stderr mohou být přesměrovány do souboru, nebo poslány procesu *srun*, v obou případech jsou bufferovány, aby se systém nebrzdil čekáním na vstupně-výstupní operace. Lokální démon umožňuje asynchronní komunikaci pomocí propagování signálů nebo požadavkem k ukončení.

- **Nástroje příkazového řádku** – Jedná se o uživatelské rozhraní k SLURM plánovači, které umožňuje uživatelům spravovat jejich úlohy a administrátorům umožňuje měnit nastavení systému. Jedná se především o `srun` pro odeslání úlohy do fronty, `scancel` pro ukončení již běžící úlohy nebo smazání čekající úlohy z fronty, `scontrol` pro administrátorskou správu clusteru, `sinfo` pro získání agregovaných statistik o jednotlivých uzlech a `squeue` pro vypsání fronty čekajících úloh.

2.3.2 PBS

Portable batch system (PBS) je plánovač pro architekturu cluster a je dostupný pro Unixové systémy stejně jako pro Windows. [23] PBS byl původně vyvinut v 90. letech minulého století pro účely NASA. Nyní je PBS plánovač dostupný v několika implementacích:

- OpenPBS
- PBS Professional
- TORQUE

Plánovače PBS a SLURM řeší stejný problém a lze najít hodně analogií v tom, jak jej řeší. [23].

Pro vytvoření požadavku na zařazení výpočetní úlohy do fronty úloh slouží příkaz `qsub`. Jakmile jsou volné výpočetní prostředky pro danou úlohu, je daná úloha spuštěna na prvním dostupném uzlu. Pro správné přijetí úlohy je potřeba specifikovat frontu, do které se úloha má zařadit, počet výpočetních uzlů, které chceme pro úlohu alokovat, maximální výpočetní čas, identifikátor projektu a nakonec pracovní script. Pro debugování programů lze místo pracovního scriptu použít přepínač `-I`, který spustí úlohu v interaktivním režimu. Po spuštění úlohy pak uživatel může výpočetní prostředky ovládat přímo ze svého terminálu. V případě použití pracovního scriptu mohou být požadované parametry pro spuštění úlohy uvedeny přímo v pracovním scriptu. Příklad takového pracovního scriptu se specifikací parametrů ke spuštění je na výpisu 2.1. Odeslání požadavku ke spuštění takové úlohy pak lze jednoduše udělat pomocí příkazu na výpisu 2.2.

Plánovač PBS dále mimo jiné poskytuje tyto příkazy pro práci s již existujícími požadavky. Příkaz `qdel` odstraní požadavek z fronty nebo ukončí již běžící úlohu. Pro uložení průběžného stavu můžeme využít zpoždění, které můžeme specifikovat parametrem `-W`. To způsobí, že bude zpoždění mezi signálem `SIGTERM` a `SIGKILL` a během tohoto času můžeme ukládat průběžné výpočty. Dále příkaz `qsig` pošle signál dané úloze. Příkaz `qhold` pozastaví vykonávání úlohy a příkaz `qrerun` obnoví běh pozastavené úlohy. Příkaz `qmove` může

přesunout čekající úlohu do jiné fronty. Příkaz `qstat` nám ukáže stav všech úloh nebo stav vybrané úlohy. Vidíme pak především, je-li daná úloha ve frontě nebo již běží, případně jak dlouho běží nebo v jaké frontě čeká. Pomocí příkazu `pbsnodes` můžeme získat informace a aktuální stav všech uzlů nacházejících se v clasteru.

```
#!/bin/bash
#PBS -q qprod
#PBS -N ModelFit
#PBS -l select=4
#PBS -A OPEN-24-47

ml TensorFlow

python ./model-fit.py
```

Výpis 2.1: Příklad pracovního scriptu pro plánovač PBS, který specifikuje frontu `qprod`, název úlohy pro snazší identifikaci uživatelem, výpočetní prostředky jsou 4 uzly a název projektu je `OPEN-24-47`. Dále je načten modul Tensorflow, který je nainstalován administrátory clusteru a je zpravidla optimalizován pro danou architekturu. A nakonec je spuštěn script v Pythonu.

```
qsub ./job-mode-fit.sh
```

Výpis 2.2: Ukázka odeslání požadavku k spuštění úlohy plně specifikovanou pracovním scriptem z výpisu 2.1.

2.4 Superpočítače IT4I

IT4Innovations[1] (dále jen IT4I) je národní superpočítačové centrum při VŠB – Technické univerzitě Ostrava. IT4I provozuje superpočítače Barbora[2] a Karolina[3]. V této práci se budeme zabývat především superpočítačem Barbora. Dříve IT4I provozovalo ještě superpočítače Anselm a Salomon, oba ale ukončily svou činnost v roce 2021. Superpočítač Anselm byl v provozu od roku 2013 a měl teoretický výkon 94 TFlop/s. Superpočítač Salomon byl spuštěn v roce 2015 a měl teoretický výkon 2 PFlop/s, jednalo se tehdy o 40. nejvýkonnější superpočítač podle seznamu TOP500[26].

2.4.1 Barbora

Superpočítač Barbora byl uveden do provozu na podzim roku 2019 a má teoretický výkon 849 TFlop/s. Je postaven na architektuře cluster (viz sekce 2.2.3). Pro rozdělení výpočetního výkonu mezi uživatele je použit plánovač PBS Professional (viz sekce 2.3.2). Běží na něm operační systém CentOS. Uzly clusteru jsou navzájem propojené pomocí neblokující InfinityBand sítě[21]. Na Barboře je nainstalováno mnoho softwarových balíčků, které jsou

[1] https://www.it4i.cz/
[2] https://www.it4i.cz/infrastruktura/barbora
[3] https://www.it4i.cz/infrastruktura/karolina

nainstalované právě pro použití na této architektuře a jejich použití usnadní vědeckou práci. Login uzly jsou dostupné pod adresou `barbora.it4i.cz`.

Superpočítač Barbora je tvořen 192 standardními výpočetními uzly, každý z nich má 2 18jádrové procesory Intel Cascade Lake 6240 a má k dispozici 192 GB paměti RAM. Dále je dostupných 8 výpočetních uzlů s akcelerátory typu GPU. Každý z těchto uzlů má 2 12jádrové procesory Intel Skylake Gold 6126, operační paměť RAM 192 GB a 4 akcelerátory GPU typu NVIDIA Tesla V100 s vnitřní pamětí 16 GB. Dále je pak dostupný 1 tlustý uzel, ten má 8 16jádrových procesorů Intel Skylake 8153 a operační paměť RAM 6 TB. [12]

Barbora má 3 úložiště, která jsou dostupná ze všech uzlů clusteru, dále pak má každý uzel vlastní úložiště a paměť RAM. Úložiště *HOME* je připojeno na adresář `/home`. Jedná se o sdílené úložiště, které je dostupné ze všech uzlů. Každý uživatel má svůj adresář dostupný pod `/home/username`. Pro toto úložiště má každý uživatel (až na výjimky určené administrátorem) kvótu 25 GB paměti. Úložiště je zálohované, zálohy však slouží pouze pro obnovení po nějaké kritické události. Nejedná se o zálohy, které by mohly být obnoveny na přání uživatele. Další dostupné úložiště je *SCRATCH*. Toto úložiště je také dostupné ze všech uzlů. Scratch úložiště je připojeno do adresáře `/scratch`. Kvóta pro toto úložiště je 10 TB pro každého uživatele. Jedná se o limit nastavený proto, aby se zabránilo nechtěnému zabrání celého dostupného prostoru některým z uživatelů. Propustnost je 5 GB/s. Toto úložiště je vhodné pro ukládání dočasných dat, zejména pak výsledků uživatelských výpočtů a pro úlohy náročné na vstupně-výstupní operace. Data na tomto úložišti jsou automaticky mazána po 90 dnech od posledního přístupu k nim. Poslední sdílené úložiště, které je dostupné ve všech uzlech clusteru, je úložiště *PROJECT*. Toto úložiště slouží jako primární úložiště pro data potřebná pro projekty, které na superpočítači běží. Zároveň je toto úložiště dostupné i ze všech superpočítačů IT4I. Data jsou automaticky mazána po vypršení platnosti daného projektu. Výchozí kvóta pro každý projekt je nastavena na 20 TB. Adresář, kam je úložiště daného projektu připojeno, je možné zjistit příkazem `it4i-get-project-dir OPEN-XX-XX`, kde `OPEN-XX-XX` je název projektu. Informace o dostupných úložištích pro daného uživatele lze získat příkazem `it4ifsusage`.

Výpočetní zdroje na superpočítači Barbora jsou dostupné pomocí několika prioritních front. Nejmenší jednotkou, kterou lze alokovat, je obvykle jeden uzel. Níže je uveden výčet několika základních front:

- `qcpu` – Fronta pro produkční použití standardních uzlů bez akcelerátoru.
- `qgpu` – Fronta pro použití uzlů s akcelerátorem GPU.
- `qcpu_exp` – Expresní fronta pro spuštění malých nebo testovacích úloh. Maximálně lze alokovat 8 uzlů na uživatele. Mohou být alokovány i samostatné procesory, nejen celé uzly. Největší priorita. Vhodné pro interaktivní režim PBS.
- `qgpu_exp` – Expresní fronta pro spuštění malých nebo testovacích úloh na uzlu s dostupným akcelerátorem GPU. Největší priorita, vhodné pro interaktivní režim.
- `qcpu_preempt` – Fronta pro výpočetní výkon zdarma. Jedná se o frontu s nejmenší prioritou. Spuštěné úlohy mohou být zabity, pokud existuje úloha s vyšší prioritou, která čeká na dostupné uzly.
- `qgpu_preempt` – Fronta pro uzly s akcelerátorem GPU zdarma. Nejmenší priorita, úlohy mohou být zabity.
- `qcpu_long` – Fronta pro dlouhé produkční běhy. Maximální doba běhu je 144 hodin.

- `qfat` – Fronta pro přístup k tlustému uzlu.

2.4.2 Karolina

Superpočítač Karolina byl instalován v roce 2021. Teoretický výkon tohoto superpočítače je 15,7 PFlop/s. Tento superpočítač byl navržen tak, aby umožnil řešit nejen klasické numerické simulace, ale taky rozsáhlé datové analýzy nebo využití umělé inteligence. Podobně jako na superpočítači Barbora (viz sekci 2.4.1), i Karolina je postavena na architektuře cluster (viz sekci 2.2.3). Cluster je složen z 831 uzlů. Tyto uzly jsou navzájem propojené pomocí sítě InfiniBand[21]. I na Karolině funguje operační systém CentOS. Pro dělbu výpočetního výkonu mezi uživatele je použitý plánovač PBS Professional (viz sekci 2.3.2). Na superpočítači Karolina se také nachází mnoho softwarových modulů, které jsou optimalizovány pro tuto architekturu. Tyto moduly jsou instalovány administrátory superpočítače a jsou snadno použitelné v uživatelských úlohách.

Cluster Karolina má 720 univerzálních výpočetních uzlů, z nich má každý 2 64jádrové procesory AMD Zen 2 EPYC 7H12 a má k dispozici 256 GB paměti RAM. Dále se v clusteru nachází 72 uzlů s akcelerátory typu GPU, každý z nich má 2 64jádrové procesory AMD Zen 3 EPYC 7763, operační paměť RAM 1024 GB a 8 akcelerátorů GPU typu NVIDIA A100 s vnitřní pamětí 40 GB. Dále je dostupný 1 tlustý uzel pro zpracování velkých dat, ten má k dispozici 32 24jádrových procesorů Intel Xeon-SC 8268 a operační paměť RAM o velikosti 24 TB. Zbývající uzly slouží k poskytování cloudových služeb, správě sítě a k poskytování datového úložiště.

Na superpočítači Karolina jsou, obdobně jako na Barboře, dostupné 3 úložiště ze všech uzlů clusteru. Každý uzel pak má k dispozici vlastní lokální paměť a operační paměť RAM. Úložiště *HOME* je limitováno na 25 GB pro uživatele a je dostupné pod `/home/username`. Toto úložiště se sdílené mezi všemi uzly na Karolině. Další dostupné úložiště je `SCRATCH`, které má omezení na 20 TB na projekt. Toto úložiště je dostupné pod `/scratch`. Scratch úložiště je vhodné pro data generovaná nebo načítaná ve výpočtech náročných na I/O operace. Data jsou automaticky mazána po 90 dnech od posledního přístupu k nim. Poslední dostupné sdílené úložiště je `PROJECT`, které je sdílené napříč všemi uzly v IT4I, nejen na Karolině. Umístění Project úložiště k danému projektu lze zjistit příkazem `it4i-get-project-dir`. Limit je nastaven na 20 TB pro projekt.

I zde jsou výpočetní zdroje dostupné pomocí několika prioritních front. Nejmenší alokovatelná jednotka je obvykle celý uzel, s výjimkou uzlů s GPU akcelerátorem, kde se dá alokovat minimálně 1/8 uzlu (tj. 16 jader CPU a 1 GPU). Jinak jsou fronty velice podobné frontám na superpočítači Barbora (viz sekci 2.4.1).

Kapitola 3

Simulace ultrazvuku s nástrojem k-Wave

V této kapitole popisuji samotný problém, jehož časovou náročnost se snažím odhadnout vytvořeným modelem. Tato kapitola popisuje samotný balíček k-Wave, proč vznikl, co řeší a hlavně, jak to řeší. Je zde představeno, jak se nástroj k-Wave používá nejenom v MATLABU, ale jak lze simulaci urychlit s použitím superpočítačů popsaných v kapitole 2. Zároveň se zde diskutuje o tom, co vše ovlivňuje časovou náročnost prováděných simulací a jak to můžeme ovlivnit.

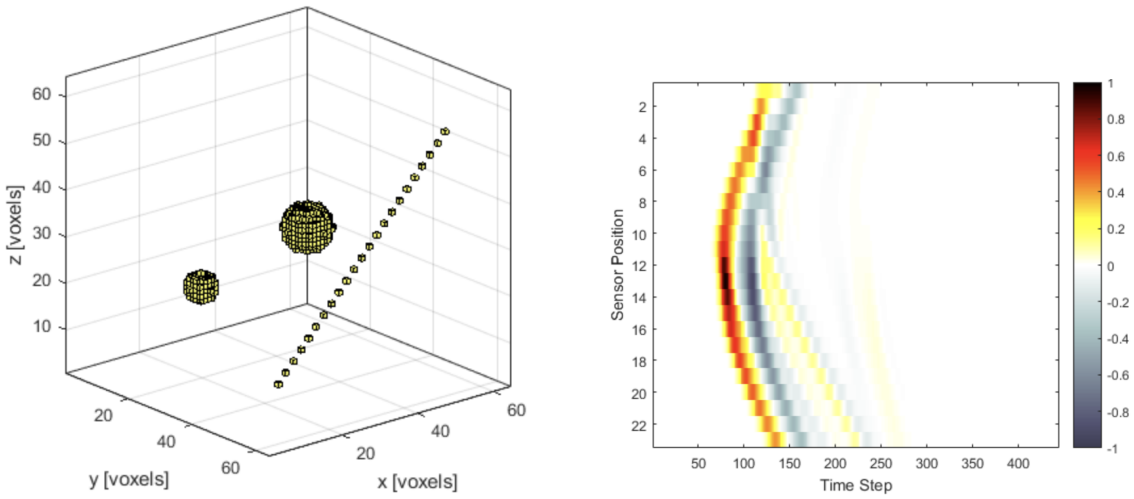
3.1 Balíček k-Wave

K-Wave [4] je open-source nástroj pro MATLAB, který je navržen pro simulování šíření akustických vln v 1D, 2D a 3D prostoru. Balíček byl vytvořen na University College London v roce 2009 pod vedením Bradley Treeby a Ben Cox. Prvotně k-Wave řešil simulaci a rekonstrukci fotoakustických vln v bezztrátovém médiu. Postupem času byl balíček rozšířen o časově proměnlivé zdroje vlnění, absorpci, nelineárně šířené vlny, elastické materiály a další vlastnosti. Později byla vydána taky verze implementovaná v C++ vyvinutá Jiřím Jarošem a verze v CUDA pro použití GPU akcelerace.

Pro simulaci reálného světa se nejčastěji používá 3D simulace. Ta je ale nejvíce náročná na výpočetní zdroje a taky na čas. Verze k-Wave implementovaná v MATLABU je uživatelsky přívětivá, ovšem pro provádění rozměrných 3D simulací je nevhodná právě z důvodu časové náročnosti výpočtu. Tento problém řeší verze implementovaná v CUDA, která využívá výpočetní zdroje grafické karty. Pro rozsáhlou paralelizaci na mnoha procesorech CPU je vhodná verze implementovaná v C++, která pro komunikaci mezi procesy používá knihovnu *Open MPI*[1]. Touto verzí se budeme zabývat ve zbytku této práce, protože to je verze vhodná pro použití na superpočítačích s mnoha dostupnými CPU procesory.

Obrázek 3.1 zobrazuje, jak si můžeme přestavit typickou simulační úlohu k-Wave ve 3D. V této úloze máme definován 3D prostor rozdělený na 64 bodů v každém směru, počáteční tlak, který se v prostoru vyskytuje, množinu senzorů a heterogenní médium. Výsledkem simulace je tlak v bodech se senzory v každém časovém kroku simulace.

[1] https://www.open-mpi.org/



Obrázek 3.1: Vlevo je příklad použití k-Wave pro spuštění úlohy s počátečním tlakem a sadou senzorů ve 3D prostoru. Vpravo je výsledek simulace znázorňující tlak v senzorech s měnícím se časem. Převzato z [29].

3.2 Simulační úloha

Jako vstup do k-Wave simulace je potřeba zadat několik parametrů, které specifikují simulační prostor, médium, zdroj vlnění a senzory, které snímají vlny. Tyto parametry jsou znázorněny na obrázku 3.2. V toolboxu k-Wave existují 3 simulační funkce: `kspaceFirstOrder1D`, `kspaceFirstOrder2D` a `kspaceFirstOrder3D`. Všechny tyto funkce dostávají jako parametry níže popsané 4 struktury (diskrétní prostor, médium, zdroj a senzor). Tyto funkce umí úlohu spustit a simulovat šíření vln, zároveň ale taky umí exportovat úlohu pro spuštění pomocí verzí k-Wave pro implementovaných v C++ nebo CUDA. Výpis 3.1 ukazuje spuštění simulace přímo v MATLABu a poté export úlohy do souboru, který slouží jako vstup pro k-Wave C++ nebo CUDA.

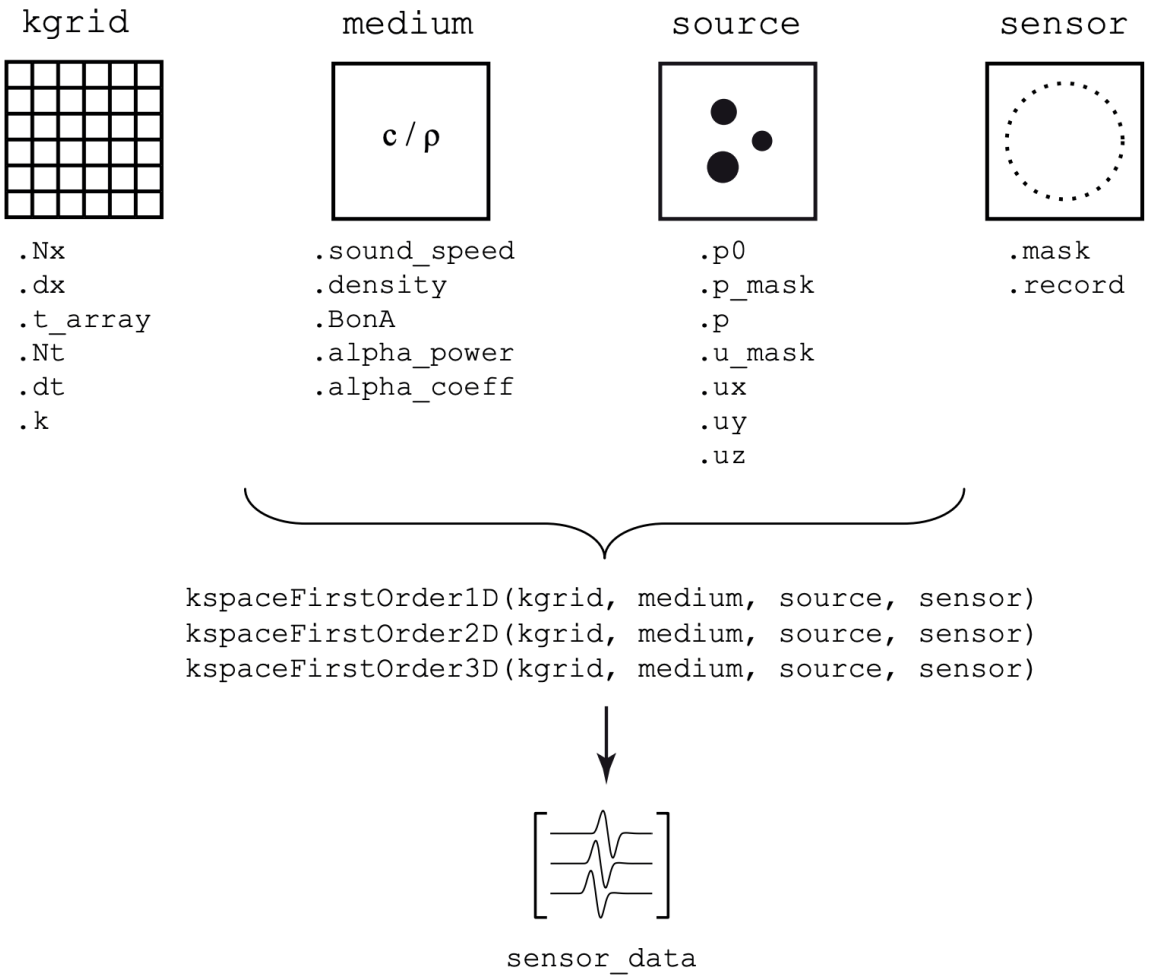
```
% Run the simulation
sensor_data = kspaceFirstOrder3D(kgrid, medium, source, sensor);

% Export the simulation task for running in C++ or CUDA k-Wave version
filename = simulation_input.h5;
kspaceFirstOrder3D(kgrid, medium, source, sensor, SaveToDisk, filename);
```

Výpis 3.1: Spuštění simulační úlohy přímo a export úlohy do souboru pro spuštění ve verzi k-Wave pro C++ nebo CUDA.

3.2.1 Prostor

Nejdříve je potřeba specifikovat strukturu *kGrid*. Ta definuje prostor a čas simulace. Obsahuje zejména hodnoty `Nx`, `Ny` a `Nz`. Ty definují počet diskrétních bodů v každém směru. Hodnoty `dx`, `dy` a `dz` definují vzdálenost mezi jednotlivými body v každém směru. Hodnoty `Nt` a `dt` definují počet kroků simulace a časový interval mezi každým krokem. Příklad definování prostoru je na výpisu 3.2.



Obrázek 3.2: Parametry, které je potřeba specifikovat, abychom mohli spustit k-Wave simulaci. Převzato z [4]. Je třeba specifikovat prostor, ve kterém se simulace odehrává, médium, ve kterém se vlny šíří, zdroje akustického vlnění a senzory, které poté vlnění zaznamenávají.

3.2.2 Médium

Médium definuje materiálové vlastnosti média v každém diskrétním bodě. Parametr média `sound_speed` určuje rychlost šíření zvuku v našem médiu, atribut `density` definuje hustotu, `BonA` určuje nelineární parametr, absorpční koeficient `alpha_coeff` a absorpční exponent `alpha_power`. Médium může být buď homogenní nebo heterogenní. Homogenní médium je v každém bodě stejné a je definováno skalárními hodnotami. Zatímco heterogenní médium se může v různých bodech lišit a jeho atributy jsou definovány jako pole obsahující hodnotu pro každý diskrétní bod. To může ovlivňovat rychlost simulace. Příklad definování heterogenního média je na výpisu 3.3.

3.2.3 Zdroj

Zdroj definuje vlastnosti a umístění akustického zdroje v médiu. Můžeme použít 3 typy zdrojů: počáteční tlak, časově proměnlivý zdroj tlaku a časově proměnlivý zdroj rychlosti částic. Příklad definování počátečního zdroje pro simulaci je na výpisu 3.4.

```
% Create the comutational grid
Nx = 1024;
dx = 1e-3;
kgrid = kWaveGrid(Nx, dx, Nx, dx, Nx, dx);
```

Výpis 3.2: Definování prostoru pro simulaci k-Wave. Prostor je 3D a je rozdělen na 1024 diskrétních bodů v každém směru. Mezi jednotlivými body v každém směru je vzdálenost 1 mm.

```
% Heterogeneous material
medium.sound_speed = 1500 * ones(Nx, Ny, Nz);
medium.sound_speed(1:Nx/2, :, :) = 1700;
medium.density = 1000 * ones(Nx, Ny, Nz);
medium.density(:, Ny/4:Ny, :) = 800;
```

Výpis 3.3: Definování heterogenního média pro simulaci k-Wave. Heterogenní médium je definováno maticemi s rychlostí zvuku a hustotou.

3.2.4 Senzor

Posledním vstupem pro definování úlohy je struktura `sensor`. Ta určuje vlastnosti a umístění senzorů v diskrétním prostoru, které zaznamenávají akustický tlak v daném bodě v každém kroku simulace. Parametr `sensor.mask` určuje pozici senzorů. Ta může být zadána jako binární matice, která přímo udává body v prostoru, kde jsou senzory umístěné. Další způsoby, jak definovat umístění senzoru, jsou souřadnice mřížky dvou protilehlých rohů, nebo jako pole kartézských souřadnic. Příklad definování senzoru jako pole kartézských souřadnic je na výpisu 3.5.

3.3 Spuštění simulace

Pokud máme úlohu uloženou v souboru (podle návodu v sekci 3.2), můžeme simulaci provést pomocí verze k-Wave implementovanou v C++ nebo CUDA. Dále se budeme zabývat jen verzí pro C++ (MPI). Příklad spuštění simulace pomocí této verze je na výpisu 3.6. Pro spuštění simulace na superpočítači je nutné nejdříve požádat o výpočetní prostředky (popsáno v kapitole 2) a až poté můžeme simulaci spustit.

Na dobu výpočtu mají vliv zejména:

- Velikost prostoru
- Počet časových kroků
- Homogenita média
- Linearita
- Absorpce

Velikost prostoru a počet simulovaných časových kroků přímo ovlivňují časovou náročnost simulace. Simulace homogenního média snižuje čas simulace o 5-10 % podle benchmarku. [13] To je způsobeno tím, že stačí pracovat s jednou skalární hodnotou, která

```
% Initial pressure distribution using makeBall
source.p0 = makeBall(Nx, Ny, Nz, Nx/2, Ny/2, Nz/2, radius);
```

Výpis 3.4: Definování počátečního zdroje pro simulaci k-Wave.

```
% Defining 3D Cartesian sensor mask
x = -10:2:10;
y = -10:2:10;
z~= -10:2:10;
sensor.mask = [x; y; z];
```

Výpis 3.5: Definování senzoru pro simulaci k-Wave jako pole kartézských souřadnic.

se snadno udržuje v paměti cache, na rozdíl od 3D matice reprezentující každý bod prostoru a tím způsobenými častějšími výpadky cache. Simulace bezztrátového média je o 30 % rychlejší oproti simulaci absorpčního média (podle stejného benchmarku). To je způsobeno sníženým počtem počítaných rychlých Fourierových transformací (10 namísto 14).

Velmi důležité pro dobu počítání simulace je to, zda je vhodně zvolený rozměr simulačního prostoru. K-Wave používá k simulování Fourierovu transformaci v implementaci FFTW[10]. Ta funguje dobře, pokud délka pole (kGrid) může být rozložena na prvočísla 2, 3, 5 a 7. Pokud tomu tak není, používá se pomalejší implementace, což má zásadní vliv na celkovou rychlost simulace k-Wave.

```
#!/bin/bash

mpirun -np 128 \
    --map-by node \
    /home/xbucht28/k-Wave-Fluid-MPI/kspaceFirstOrder3D-MPI \
    -i /scratch/project/open-25-25/input_1024.h5 \
    -o /scratch/project/open-25-25/output_1024.h5 \
    --fftw_planner_flag FFTW_EXHAUSTIVE \
    > /home/xbucht28/Data/run-log/output-1024.log
```

Výpis 3.6: Spuštění C++ verze k-Wave na 128 procesorech komunikujících pomocí MPI. Zadání úlohy je vytvořeno v MATLAB verzi k-Wave. Výstup této simulace se v ní dá otevřít a získat výsledky. Pro počítání fourierovy transformace je použita knihovna FFTW s flagem `FFTW_EXHAUSTIVE`, který má rychlejší výpočetní dobu, ale pomalejší inicializaci.

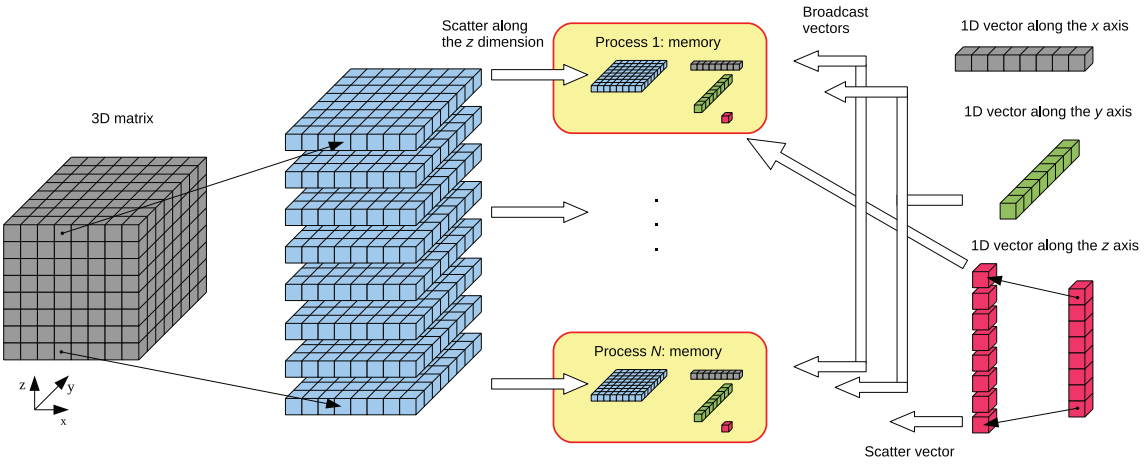
3.4 Popis algoritmu

Tato sekce popisuje 3D algoritmus k-Wave implementovaný pomocí MPI v C++. Popisuje se zde hlavně simulace samotná, dekompozice prostoru, komunikace a časová a paměťová složitost.

Činnost simulace k-Wave můžeme rozdělit do 3 částí: *pre-processing*, *simulace*, *post-processing*. V první fázi se načítají vstupní data pro danou úlohu a iniciuje se knihovna *FFTW* pro Fourierovu transformaci. Tato iniciace v režimu *FFTW_EXHAUSTIVE* může trvat i několik minut. Po simulaci samotné následuje poslední část, ve které se analyzují

výstupy simulace, překládají se do člověkem čitelného formátu a data se ukládají na disk. Tato práce se soustředí pouze na prostřední (simulační) část. Zbylé dvě fáze (pre-processing a post-processing) tato práce ignoruje. [13]

Simulační fáze na začátku provede jednodimenzionální dekompozici 3D prostoru podél osy Z, čímž vytvoří několik 2D řezů (podle velikosti osy Z). Tyto řezy jsou distribuovány jednotlivým procesorům (na různých uzlech superpočítače) společně s 1D vektory podél osy Z. Ostatní 1D vektory (podél osy X a Y) jsou broadcastem poslána všem procesům. Tento proces je znázorněn na obrázku 3.3.



Obrázek 3.3: Dekompozice 3D prostoru v programu k-Wave MPI. Převzato z [13]. Prostor je rozdělen na 2D řezy podle osy Z. Tyto řezy jsou spolu s 1D vektory distribuovány jednotlivým procesorům.

Během simulace se počítá 3D rychlá Fourierova transformace (dále jen FFT), po které následuje inverzní 3D FFT. Každá 3D FFT se spočítá sérií 2D FFT na 2D řezu s využitím lokálních dat na daném uzlu. Poté následuje komunikace typu každý s každým, kdy se transponuje v osách Z-Y. Po této transpozici následuje série 1D FFT následovaná další transpozicí v osách Y-Z. Inverzní FFT se počítá předem uvedeným postupem s obráceným pořadím operací.

Z hlediska časové náročnosti jsou operace prováděné na jednom uzlu do jisté míry předvídatelné. Je-li prostor dostatečně malý a vejde-li se do L1 cache, můžeme odhadnout simulační čas s velkou mírou přesnosti prostým spočtením počtu potřebných taktů procesoru a podělením frekvencí procesoru. Náročnější z hlediska odhadu to začíná být, pokud se všechna data nevejdou do L1 cache. Úzkým hrdlem celého procesu je ovšem komunikace mezi uzly, kdy každý uzel může komunikovat s každým a snadno může dojít k přetížení propojovací sítě. Celý proces pak může strávit i většinu času pouze komunikací. [33]

Simulační část je založena na 3D FFT, jejíž časová složitost je $O(z \cdot y \cdot x \cdot log(x) + z \cdot x \cdot y \cdot log(y) + x \cdot y \cdot z \cdot log(z))$. Paměťová náročnost roste s velikostí simulované domény, klesá ale s počtem dostupných uzlů (do jisté míry počtu uzlů). [13]

Kapitola 4

Predikce doby běhu programu pomocí strojového učení

Tato kapitola se zabývá problémem odhadu hodnoty funkce v bodě, kde její hodnota není známá. Vycházíme tak ze znalostí okolních bodů a podle nich se snažíme předpovědět hodnotu, kterou může funkce nabývat v bodě, kde opravdovou hodnotu neznáme. Pomocí metod uvedených v této kapitole můžeme přistupovat i k našemu problému, kdy se snažíme odhadnout čas, který bude trvat ultrazvuková simulace na superpočítači. Parametry této funkce mohou být všechny příznaky, které výsledný čas ovlivňují (např. velikost domény, počet uzlů a počet procesorů...). Hodnota této funkce může být právě čas potřebný k simulaci ultrazvukového šíření.

Tato kapitola se nejdříve v sekci 4.1 věnuje interpolaci, extrapolaci a spline. Následuje sekce 4.2 pojednávající o regresi. V sekci 4.3 se popisuje symbolická regrese a v následuje sekce 4.4 o genetických algoritmech, které se ukázaly jako vhodný způsob optimalizace symbolické regrese. Na závěr je pro úplnost pojednání o neuronových sítích (4.5), které jsem v rámci experimentů také vyzkoušel.

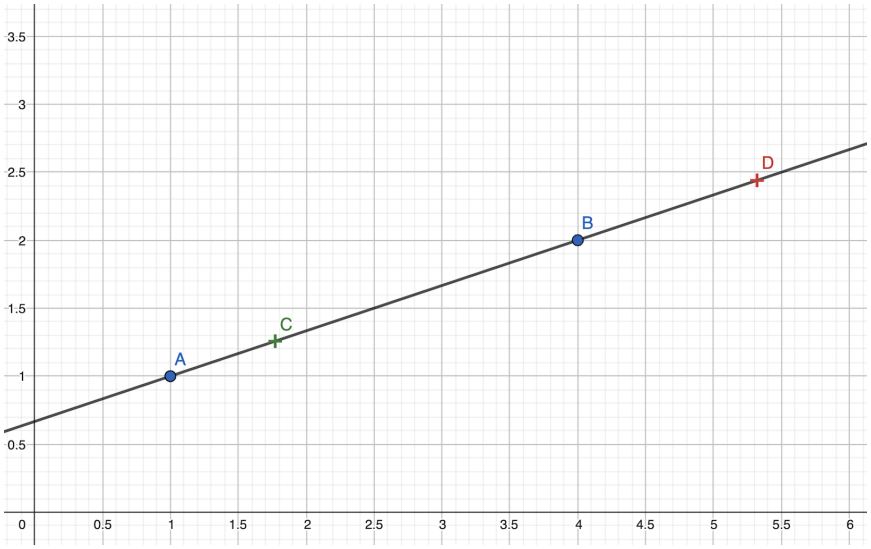
4.1 Interpolace, extrapolace, spline

Interpolace se snaží pro danou množinu známých bodů vytvořit takovou funkci, která je všechny prokládá. [9] Jedná se tedy o aproximaci funkce, která požaduje přesnou shodu ve všech uzlových bodech, které známe. [20] V rámci interpolace zkoumáme interval ohraničený známými krajními body, zatímco v rámci extrapolace se zajímáme o intervaly mimo tento známý prostor. Rozdíl mezi interpolací a extrapolací je znázorněn na obrázku 4.1. Pomocí interpolace můžeme:

- Najít neznámý předpis funkce, u níž známe jenom konečné množství funkčních hodnot (uzlových bodů).
- Zjednodušit předpis funkce, která popisuje nějaký jev.

4.1.1 Aproximace polynomem

Při interpolaci polynomem hledáme právě takový polynom, který prochází uzlovými body a jehož stupeň je maximálně o jedna menší, než je počet uzlových bodů. Zkonstruovat takový



Obrázek 4.1: Rozdíl mezi interpolací a extrapolací. Máme dva známé body A, B. Ty protneme přímkou (aproximujeme lineárním polynomem), bod C je interpolován, protože leží mezi body A a B. Na druhou stranu bod D je extrapolován.

polynom můžeme několika způsoby, níže uvedu Lagrangeův tvar interpolačního polynomu. Hledáme-li interpolační polynom pro množinu bodů (body jsou navzájem různé), které mohou procházet jednou funkcí (mají rozdílné x-ové souřadnice), nalezneme vždy interpolační polynom, který těmito body prochází a zároveň existuje právě jeden polynom, který danou množinou bodů prochází. [9] Výsledný polynom bude mít stupeň nanejvýš $n-1$, kde n je počet uzlových bodů, které známe. Existence interpolačního polynomu pro každou množinu bodů je dokázána níže (uvedením postupu, jak hledaný polynom zkonstruovat).

Jednoznačnost interpolačního polynomu je dokázána sporem. Předpokládejme, že máme 2 interpolační polynomy pro nám známých n bodů. Označme je jako polynomy $P_n(x)$ a $Q_n(x)$. Tyto polynomy mají stupeň nejvýš $n-1$. Vytvoříme další polynom vzniklý rozdílem předchozích dvou polynomů: $R_n(x) = P_n(x) - Q_n(x)$. Nový polynom $R_n(x)$ je také stupně nejvýše $n-1$ a v námi známých uzlových bodech nabývá hodnot 0. Tento polynom má pak alespoň n kořenů, přičemž jeho stupeň je nejvýše $n-1$. To je možné pouze, pokud je tento polynom všude roven 0. Pak jsou tedy 2 původní interpolační polynomy $P_n(x)$ a $Q_n(x)$ stejné.

Lagrangeův tvar interpolačního polynomu je lineární kombinace

$$P_n(x) = f_i l_i(x), \tag{4.1}$$

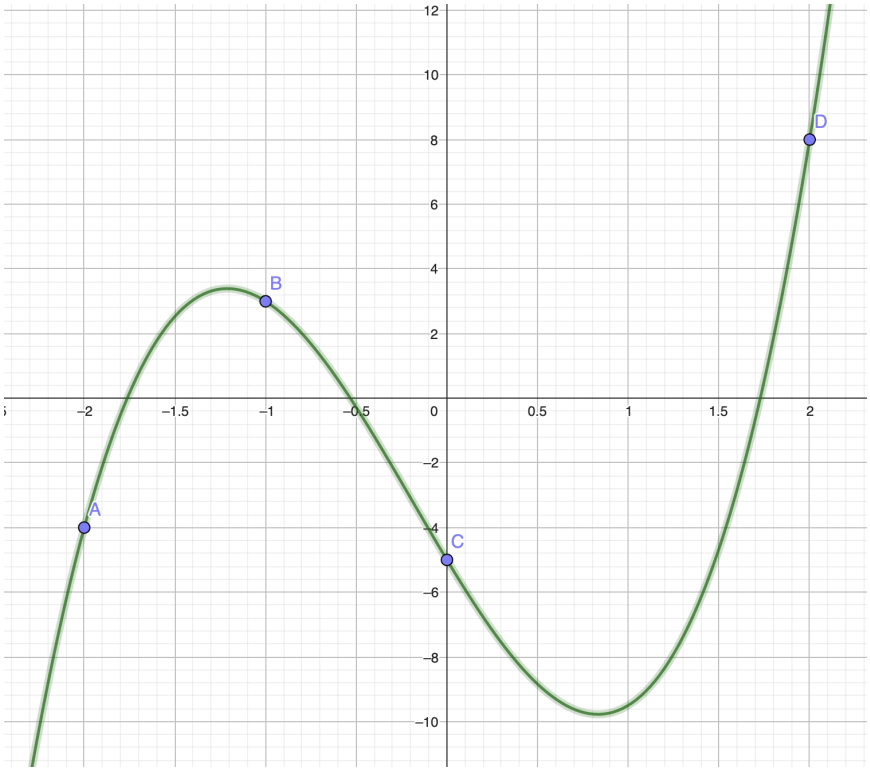
kde hodnoty f_i jsou hodnoty z uzlových bodů a $l_i(x)$ jsou pomocné polynomy, které podle potřeb nabývají hodnoty 0 nebo 1 v uzlových bodech. Polynomy $l_i, i = 0, \ldots, n$ mají tvar

$$l_i(x) = \frac{(x-x_0)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_n)}{(x_0-x_1)(x_i-x_0)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_n)} \tag{4.2}$$

Výsledný interpolační polynom $P_n(x)$ pak má tvar

$$P_n(x) = f_0 \frac{(x-x_1)(x-x_2)\dots(x-x_n)}{(x_0-x_1)(x_0-x_2)\dots(x_0-x_n)} + f_1 \frac{(x-x_0)(x-x_2)\dots(x-x_n)}{(x_1-x_0)(x_1-x_2)\dots(x_1-x_n)} + \dots$$
$$\dots + f_n \frac{(x-x_0)(x-x_1)\dots(x-x_{x-1})}{(x_n-x_0)(x_n-x_1)\dots(x_n-x_{x-1})} \tag{4.3}$$

Znázorněný interpolační polynom získaný postupem uvedeným výše pro 4 uzlové body je na obrázku 4.2. Nevýhodou tohoto postupu je, že pro přidání jednoho uzlového bodu je třeba výrazně upravit celý polynom. Také lze vidět, aproximace polynomem je vhodnější pro interpolaci než pro extrapolaci.

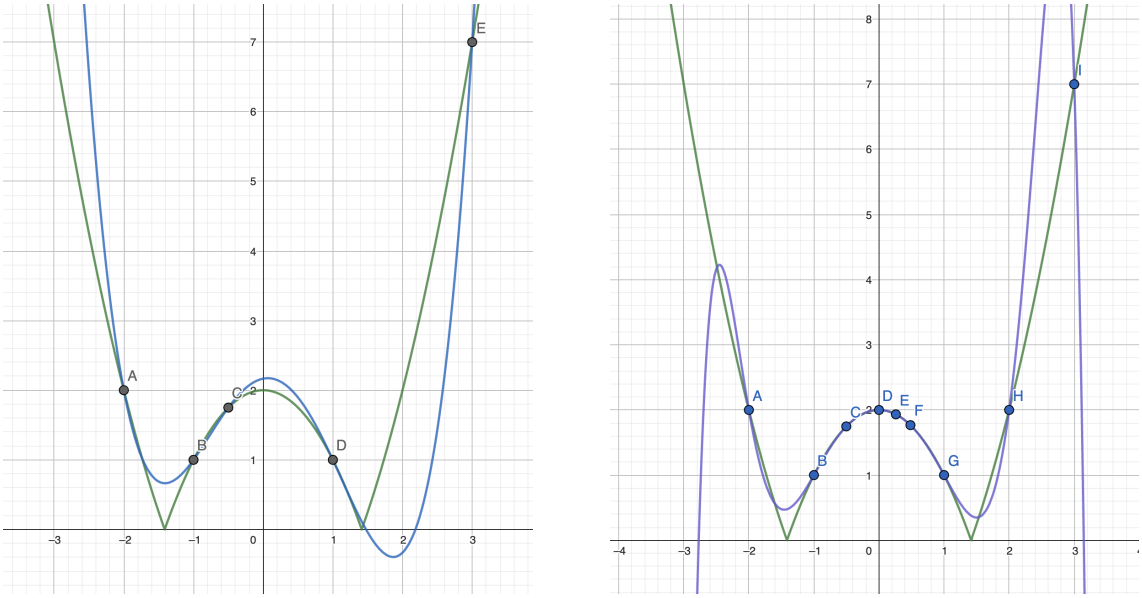


Obrázek 4.2: Příklad interpolačního polynomu získaného Lagrangeovým postupem stvořeného pro uzlové body $A = [-2, -4], B = [-1, 3], C = [0, -5], D = [2, 8]$.

Na obrázku 4.3 je znázorněna funkce $y = |x^2 - 2|$ a interpolace polynomem této funkce udělaná pomocí 5 bodů a pomocí 9 bodů. Tento obrázek ukazuje, že při použití interpolačního polynomu je potřeba být obezřetný. Často je výsledná aproximace vhodná pouze za určitých podmínek nebo na určitých intervalech (nebo v malém okolí uzlových bodů). Je dobré taky vycházet ze znalostí o zkoumaném jevu. Dále je potřeba zvážit, kolik známých bodů budeme interpolovat. Velký počet těchto bodů sice může zpřesnit aproximaci v určitých intervalech, zároveň ale zvyšuje řád výsledného polynomu, což často způsobí oscilaci.

4.1.2 Splajn

Dalším způsobem, jak můžeme aproximovat neznámou funkci se znalostí několika uzlových bodů, jsou splajny. Splajny se snaží řešit problém s oscilací, který mají interpolační poly-



Obrázek 4.3: Znázornění aproximace funkce $y = |x^2 - 2|$ (zelená) aproximačním polynomem (fialová) se znalostí 5 uzlových bodů (vlevo) a se znalostí 9 uzlových bodů (vpravo). Dále jsou znázorněny uzlové body, které byly pro interpolaci použity.

nomy. Na rozdíl od aproximačních polynomů se splajny nesnaží najít pouze jeden funkční předpis pro celý interval našeho zájmu. Splajny hledají funkci, která je na různých intervalech definována různě.

Splajn vzniká rozdělením intervalu od nejmenšího uzlového bodu do největšího uzlového bodu. Tento interval se rozdělí do několika podintervalů, kde každý podinterval je interval mezi dvěma sousedními uzlovými body. Na každém z těchto podintervalů se dále hledá aproximace původní funkce. Chceme přitom, aby výsledný splajn splňoval tyto body:

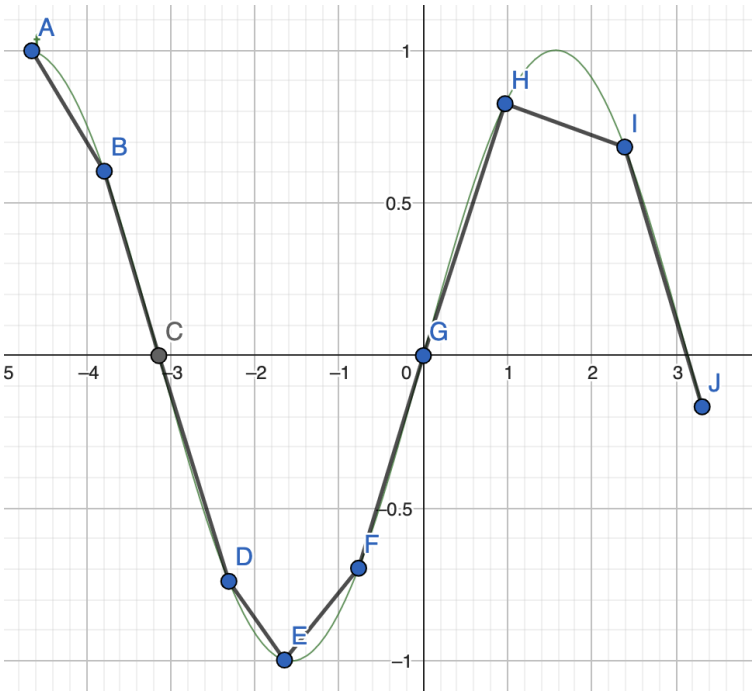
- Splajn musí procházet všemi uzlovými body (stejně jako interpolační polynom popsán v sekci 4.1.1).
- Všechny části splajnu jsou polynomy stejného typu. Některé z koeficientů, včetně toho u členu s největší mocninou, mohou vyjít nulové.
- Sousedící funkce musí na sebe navazovat. Pro všechny uzlové body x_u pak musí platit, že derivace zleva v bodě x_u se rovná derivaci zprava v bodě x_u. Tato podmínka platí až do derivace daného řádu.

Výhoda použití splajnu spočívá v tom, že se celková aproximace skládá z polynomů nižšího stupně. Sami si zvolíme řád polynomu, jímž budeme interpolovat. Tento řád nezávisí na počtu známých uzlových bodů. Můžeme tedy zpřesňovat aproximaci tím, že budeme přidávat nové známé uzlové body, a zachováme jednoduchost použitých polynomů. Polynomy nižšího řádu mají menší tendenci k výrazné oscilaci, která byla znázorněna na obrázku 4.3. V praxi se často používá lineárních a kubických splajnů.

Lineární splajn (splajn řádu 1) je funkce, která je na každém intervalu mezi uzlovými body aproximována úsečkou. U lineárního splajnu se nepožaduje návaznost derivací. Je požadována pouze spojitost funkce. Spajn je na každém intervalu $\langle x_i, x_{i+1} \rangle, i = 0, \ldots, n-1$ určen úsečkou, jejíž rovnice je

$$S_i(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i), \quad x \in \langle x_i, x_{i+1} \rangle, \tag{4.4}$$

kde $f(x_i)$ je hodnota uzlového bodu x_i. Lineární spajn je znázorněn na obrázku 4.4. Jde vidět, že znalost více uzlových bodů zpřesní aproximaci lineárním splajnem.



Obrázek 4.4: Interpolace funkce $y = sin(x)$ pomocí lineárního splajnu se znázorněnými uzlovými body.

V technické praxi se nejvíce používají **kubické splajny**, tj. splajn řádu 3. Pro sestrojení kubického splajnu musíme na podintervalech mezi sousedními uzlovými body $\langle x_i, x_{i+1} \rangle$, $i = 0, 1, \ldots, n-1$ hledat polynom tvaru

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{4.5}$$

Z podmínek se můžeme dostat k rovnicím pro vypočítání koeficientů pro jednotlivé kubické polynomy, abychom dostali přirozený kubický splajn. Pro odvození těchto rovnic viz [9]. Koeficienty a získáme přímo jako funkční hodnoty ze známých uzlových bodů.

$$a_i = f(x_i), i = 0, 1, \ldots, n-1 \tag{4.6}$$

Dále potřebujeme vypočítat pomocné hodnoty h_i a Δf_i. Tyto hodnoty opět určíme pomocí uzlových bodů jako vzdálenosti sousedních uzlových bodů a jako rozdíl funkčních hodnot v nich.

$$h_i = x_{i+1} - x_i, \quad \Delta f_i = f(x_{i+1}) - f(x_i), \quad i = 0, \ldots, n-1 \tag{4.7}$$

První a poslední parametry c_n jsou nulové, $c_0 = 0$ a $c_n = 0$. Všechny ostatní parametry c_i, $i = 1, \ldots, n-1$ vypočítáme jako řešení soustavy lineárních rovnic uvedených níže.

Můžeme použít libovolnou metodu pro řešení soustavy lineárních rovnic, např. Gaussovu eliminační metodu.

$$\begin{aligned}
2(h_0+h_1) + \quad & h_1c_2 && = 3\left(\frac{\Delta f_1}{h_1} - \frac{\Delta f_0}{h_0}\right)\\
h_1c_1 \quad & + 2(h_1+h_2)c_2 + h_2c_3 && = 3\left(\frac{\Delta f_2}{h_2} - \frac{\Delta f_1}{h_1}\right)\\
& \ddots && \vdots\\
& h_{n-2}c_{n-2} \quad + 2(h_{n-2}+h_{n-1})c_{n-1} && = 3\left(\frac{\Delta f_{n-1}}{h_{n-1}} - \frac{\Delta f_{n-2}}{h_{n-2}}\right)
\end{aligned}$$

Zbývající koeficienty b_i a d_i dopočítáme pomocí koeficientů spočtených v předchozích krocích.

$$b_i = \frac{f(x_{i+1}) - f(x_i)}{h_i} - \frac{c_{i+1} + 2c_i}{3}h_i, \qquad i = 0, \ldots, n-1 \tag{4.8}$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \qquad i = 0, \ldots, n-1 \tag{4.9}$$

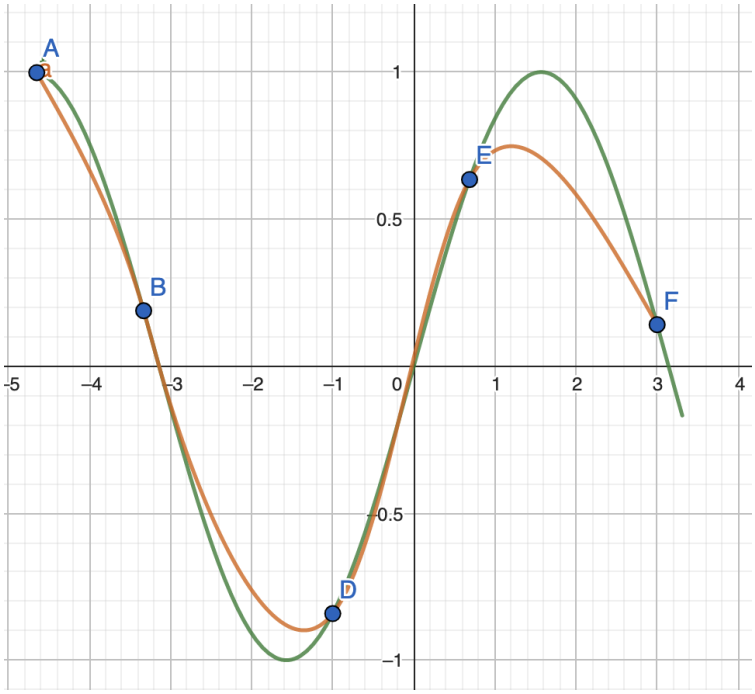
Příklad kubického splajnu je vidět na obrázku 4.5. Podobně jako aproximace polynomem, ani splajny nejsou vhodné pro extrapolaci. Výhodou splajnů je ale možnost použití více uzlových bodů, které nevedou k nechtěným oscilacím, jako tomu bylo u aproximaci pomocí polynomu. Pro odhad doby běhu programu na superpočítači se tyto metody dají použít. Můžeme např. aproximovat funkci udávající čas běhu pro vybranou velikost domény, jejímž vstupním parametrem je počet procesorů.

4.2 Regrese

V předchozích částech této kapitoly se pojednává o interpolačních polynomech a splajnech. Obě tyto metody mají společné to, že hledají funkci, která danou skupinu známých uzlových bodů protíná. To je ale v praxi často příliš striktní podmínka pro aproximaci dat. Pokud totiž naše data pocházejí z nějakého měření, obsahují vždy nějakou chybu. Tuto chybu poté přenášíme i do naší aproximace. Další nevýhodou dříve uvedených metod je, že neumožňují mít více dat v jednom uzlovém bodě. V praxi se ale často stává, že provádíme jedno měření vícekrát a pokaždé naměříme mírně odlišné výsledky. Tyto metody s tím pracovat neumí, je potřeba mít pro jeden uzlový bod jenom jednu hodnotu, případně můžeme použít průměr těch dat. Dále tyto metody neumí zohlednit naší možnou znalost o procesu, který data generuje. Můžeme mít například tušení vycházející ze znalosti procesu, že data budou mít lineární či kvadratickou závislost. Regrese oproti tomu řeší tento problém tak, že se snaží najít funkci, která prochází k známým bodům co nejblíže.

4.2.1 Metoda nejmenších čtverců

Pro určení, která funkce prochází k známým bodům co nejblíže, můžeme použít mnoho různých přístupů. Jedním z nejpoužívanějších je metoda nejmenších čtverců. Tato metoda vyčísluje chybu, kterou daná funkce prokládá data, jako sumu druhých mocnin rozdílu mezi odhadovanou hodnotou a mezi hodnotou skutečnou:



Obrázek 4.5: Interpolace funkce $y = sin(x)$ (zelená) pomocí kubického splajnu (červená) se znázorněnými uzlovými body.

$$\rho^2 = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - f(x_i))^2, \tag{4.10}$$

kde n je počet uzlových bodů, y_i je hodnota známého bodu a $f(x_i)$ je aproximační funkce v bodě x_i.

4.2.2 Lineární regrese

Lineární regrese je taková regrese, která aproximuje data přímkou. Před použitím lineární regrese je důležité si uvědomit, jestli data, která aproximujeme, jsou vhodná pro tento druh regrese. Pokud mohly být vygenerovány systémem s lineární závislostí, pak ano. Pokud ne, je potřeba data aproximovat jinou funkcí. Princip ale zůstane stejný, jako je níže popsaný pro lineární regresi.

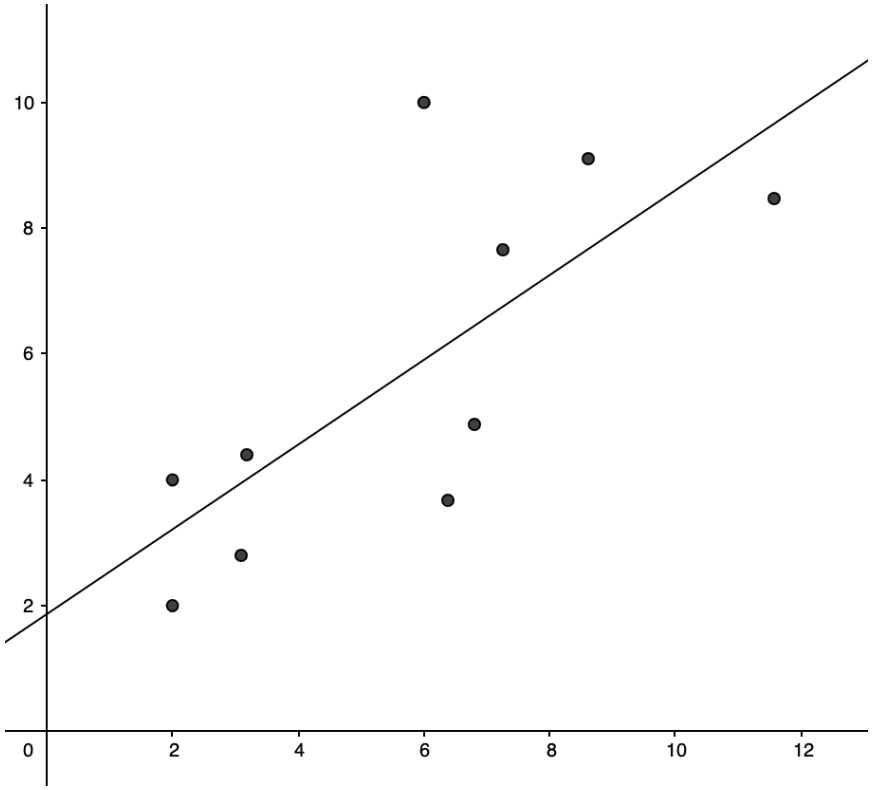
Máme známe body x_i, $i = 0, \ldots, n$ a funkční hodnoty v těchto bodech y_i. Lineární regrese hledá přímku s rovnicí

$$y = c_0 + c_1 x, \tag{4.11}$$

která aproximuje data pomocí metody nejmenších čtverců. Parametry přímky c_0 a c_1 můžeme spočítat analyticky. Budeme přitom minimalizovat chybu uvedenou v rovnici 4.10. Koeficienty c_0 a c_1 jsou řešením soustavy rovnic:

$$c_0(n+1) + c_1 \sum_{i=0}^{n} x_i = \sum_{i=0}^{n} y_i$$
$$c_0 \sum_{i=0}^{n} x_i + c_1 \sum_{i=0}^{n} x_i^2 = \sum_{i=0}^{n} x_i y_i,$$

kde počet známých uzlů je $n+1$. Pro odvození viz [9]. Příklad lineární regrese pro vyznačené známé uzly je vidět na obrázku 4.7.



Obrázek 4.6: Lineární regrese pro známé body.

4.2.3 Polynomiální regrese

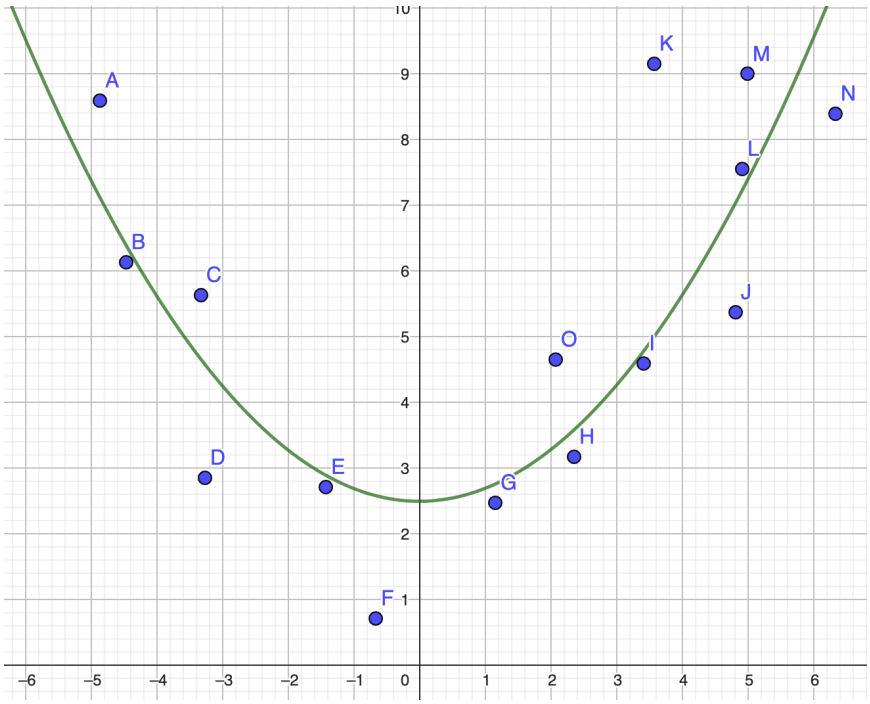
Polynomiální regrese aproximuje data pomocí polynomu předem daného stupně m:

$$P_m(x) = c_0 + c_1 x + \cdots + c_m x^m \tag{4.12}$$

Postupujeme pak stejně jako u lineární regrese. Analyticky vypočítáme koeficienty $c_0, \ldots, c_m$ regresního polynomu. Příklad polynomiální regrese polynomem stupně 2 je na obrázku 4.7.

Obecně můžeme dělat regresi libovolným typem funkce. Některé (např. výše uvedené regrese polynomem) můžeme snadno vypočítat analyticky, na jiné je potřeba použít iterační metody pro nalezení dostatečně dobrého řešení. Vždy je však třeba určit, jakým typem funkce budeme data aproximovat a toto rozhodnutí je potřeba dělat na základě znalosti

systému, který data generuje. Pokud je v datech periodická složka, můžeme použít trigoniometrické polynomy, pokud je závislost v datech lineární, použijeme polynom stupně 1 atd...



Obrázek 4.7: Polynomiální regrese pro známé body. Stupeň polynomu jsme určili 2 (parabola). Znázorněna je výsledná parabola (zelená) a známé uzlové body.

4.3 Symbolická regrese

Symbolická regrese [24] je metoda regresní analýzy, která se snaží data aproximovat pomocí matematického výrazu, který nejlépe odpovídá datové sadě. Příklad takového modelu, který může vygenerovat symbolická regrese, je na obrázku 4.8. Na rozdíl od metod popsaných v sekci o klasické regresi (viz 4.2), symbolická regrese může vytvářet libovolně komplexní výrazy. Netrpí tak nedostatky, které má klasická regrese. Ta totiž má tendenci reálný problém zjednodušovat. Mnoho problémů z reálného světa zkrátka nemůžeme popsat např. polynomem, ale je nutné data aproximovat složitějším výrazem. Oproti neuronovým sítím (viz 4.5) se symbolická regrese nechová jako černá skříňka, ale výraz, kterým jsou data aproximována v symbolické regresi, je pro člověka dobře čitelný a snadno interpretovatelný.

Zatímco ostatní regresní metody obvykle optimalizují váhy v předem definovaném modelu (např. určování koeficientů polynomu, trénování neuronové sítě), symbolická regrese navíc hledá i optimální strukturu modelu. Symbolická regrese se snaží kombinovat příznaky, matematické operace a konstanty do jedné funkce. K hledání této funkce můžeme použít různé přístupy. Tato práce se zaměřuje na trénování symbolické regrese pomocí genetického programování (viz 4.4).

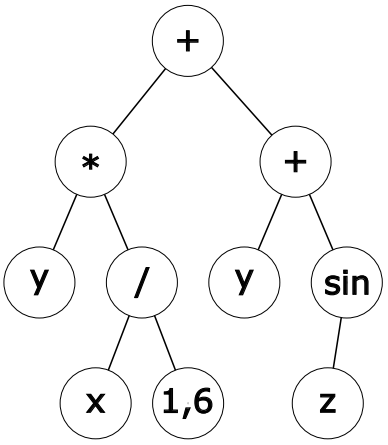
Ukazuje se, že symbolická regrese je vhodná spíše pro řešení problémů, kdy máme malou datovou sadu. [30] Z toho můžeme profitovat při řešení problému odhadu doby běhu programu na superpočítači, protože sběr dat pro tuto úlohu je finančně náročný. Pro každý

záznam v datové sadě je potřeba spustit program na superpočítači s danými výpočetními zdroji. Proto výběrem metody, která dokáže lépe fungovat s malou datovou sadou, můžeme ušetřit mnoho peněz.

Postup trénování modelu symbolické regrese obvykle začíná vygenerováním množiny náhodných syntaktických stromů a postupným upravováním těchto modelů pomocí mutace, křížení a reprodukce, dokud není trénování ukončeno, protože byly splněny požadavky na přesnost modelu, nebo bylo trénování omezeno časem nebo počtem generací.

Model symbolické regrese je reprezentován jako syntaktický strom, kde vnitřní uzly jsou matematické operace a listové uzly jsou konstanty nebo příznaky z datové sady. Syntaktické stromy jsou přitom generovány tak, aby výsledné modely měly podobu smysluplného matematického výrazu. Z každé generace modelů se vybere podmnožina modelů, které se dostanou do další generace. Přitom jsou na ně aplikovány s jistou pravděpodobností náhodné úpravy. Mutace může provést náhodnou změnu jednoho uzlu (např. změna hodnoty konstanty v listovém uzlu, náhrada uzlu za matematickou operaci...) nebo celé větve stromu (např. náhrada větve stromu vstupním příznakem nebo konstantou). Křížení může náhodně vyměnit některou větev za jinou větev z jiného modelu v dané generaci. Detailnější vysvětlení genetického programování je v kapitole 4.4.

Další metody trénování modelů symbolické regrese používají např. algoritmus zpětné propagace k optimalizovaní konstant v listových uzlech syntaktického stromu. [28] Tato metoda pak může zjistit základní strukturu syntaktického stromu pomocí genetického programování, zatímco přesné hodnoty použitých konstant jsou doladěny algoritmem zpětné propagace. Klasická metoda trénování pomocí genetického programování nechává výběr hodnot konstant na náhodné mutaci, což nemusí být ten nejlepší přístup. Jiné se snaží modely optimalizovat nejen pro jejich chybovost vůči trénovacím datům, ale do ohodnocení modelu zahrnují komplexnost daného modelu, přičemž se snaží, aby výsledný model byl co nejjednodušší a měl co nejmenší chybu. [3]



Obrázek 4.8: Příklad modelu vygenerovaného pomocí symbolické regrese. Strom reprezentuje výraz $(y * \frac{x}{1,6}) + (y + sin(x))$.

4.4 Genetické algoritmy, genetické programování

Genetické algoritmy [5] jsou optimalizační algoritmy, které jsou inspirovány biologií (evolucí). Genetické algoritmy fungují dobře tam, kde by jinak hledání optimálního řešení bylo

časově příliš náročné nebo by bylo potřeba zkusit všechny možnosti pro vyhodnocení té nejlepší, např. problém batohu [7]. Pro pochopení fungování genetických algoritmů jsou podstatné tyto pojmy:

- **Jedinec** – Kandidát na řešení daného problému. Každý jedinec má svůj chromozom se zakódovaným řešením problému.
- **Chromozom** – Zakódované kandidátní řešení problému. Je potřeba řešení vhodně zakódovat, obvykle se setkáme s polem bitů pevné délky.
- **Generace** – Skupina jedinců. Každá generace vychází z generace předchozí po aplikaci křížení a mutací.
- **Přirozený výběr** – Výběr je v genetických algoritmech důležitý pro určení jedinců, kteří budou vybráni k reprodukci.
- **Chybová funkce** – Vyčísluje, jak daleko je daný jedinec k optimálnímu řešení (trénovací datům). Často se setkáme se střední kvadratickou chybou nebo střední absolutní chybou.
- **Křížení** – Pro vytvoření nového jedince se zkříží zakódovaná genetická informace dvou rodičů. Křížení probíhá tak, že se náhodně vybere část prvního rodiče, která se zdědí do genetické informace potomka. Zbytek genetické informace potomka se zdědí od druhého rodiče. Znázorněno na obrázku 4.9.
- **Mutace** – Na nového jedince vytvořeného křížením se poté aplikuje náhodná úprava každého bitu v genetické informaci.

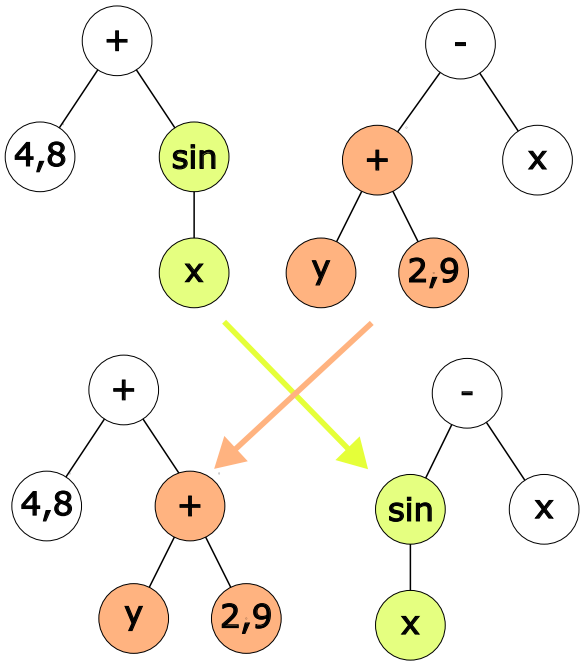
Běh genetického algoritmu se zahajuje vytvořením první generace, která se skládá z n náhodně vytvořených jedinců. Různorodost těchto jedinců je ze začátku velká. Později mají genetické algoritmy tendenci tuto různorodost zmenšovat. Po vytvoření první generace je spočítána chybová funkce pro každého jednice. Na základě ní jsou vybráni jedinci, kteří budou určeni k reprodukci. Obvyklým způsobem k vybrání jedinců je ruleta. Metoda náhodně vybírá jedince, ovšem bere v potaz hodnotu chybové funkce, kdy větší pravděpodobnost výběru mají ti jedinci, kteří mají chybu menší. [16] Po výběru dvou jedinců se provede křížení, tj. náhodně se vybere místo v jejich chromozomech. Nový jedinec si část nalevo od tohoto místa vezme od prvního vybraného jedince a zbytek od druhého. Po křížení nastává mutace, ta s předem danou pravděpodobností mění hodnotu náhodně vybraného bitu (nebo bitů) v chromozomu. [22] Následně je nový jedinec zařazen do nové generace. Proces selekce, křížení a mutace se opakuje, dokud není nová generace naplněna předem specifikovaným počtem jedinců.

Další generace se vytvářejí stejně, jako je popsáno výše. Po vyhodnocení chybové funkce u všech jedinců v každé generaci se porovná chybová funkce nejlepšího jedince z nové generace s nejlepším jedincem v celé evoluci. Je-li chyba menší, nalezli jsme novou elitu (nejlepšího známého kandidáta). Nastane-li podmínka pro ukončení genetického algoritmu (maximální počet generací, omezení časem nebo dostatečně nízká chyba...), algoritmus vrátí nejlepšího známého kandidáta v dané evoluci. Pseudokód běhu genetického algoritmu je zobrazen v algoritmu 1.

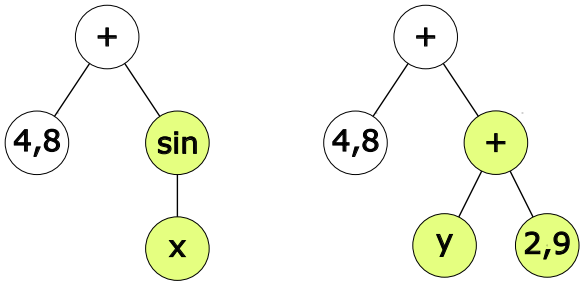
Vstup : Velikost generace, n
Maximální počet generací, MAX
Výstup: Nejlepší globální řešení, Y_{bt}
Začátek
$t \leftarrow 0$
Vygeneruj počáteční populaci n chromozomů Y_i (i = 1,2,. . . ,n)
Spočítej chybovou funkci pro všechny chromozomy
while $t < MAX$ **do**
Vyber páry chromozomů z poslední generace na základě jejich chyby
Aplikuj křížení na každý pár
Aplikuj mutaci s danou pravděpodobností
Vytvoř novou generaci z právě vzniklých jedinců
$t \leftarrow t + 1$
end
return Jedinec s nejmenší chybou Y_{bt}
Konec

Algoritmus 1: Genetický algoritmus [5]

Genetické programování [18] je specifickým případem použití genetických algoritmů. Liší se hlavně reprezentací jedince, kdy jedinec má podobu stromové struktury s proměnlivou délkou. Tito jedinci mohou reprezentovat matematickou funkci nebo počítačový program. Pro problém řešený v této práci, odhad doby běhu programu na superpočítači, můžeme využít genetického programování k hledání vhodného předpisu funkce (symbolické regresi) určující potřebný čas (viz sekce 4.3). Selekce pak probíhá stejně, jako je popsáno výše. Křížení může promíchat různé větve od rodičů (viz obrázek 4.9). Mutace může náhodně upravit uzel nebo celou větev (viz obrázek 4.10).



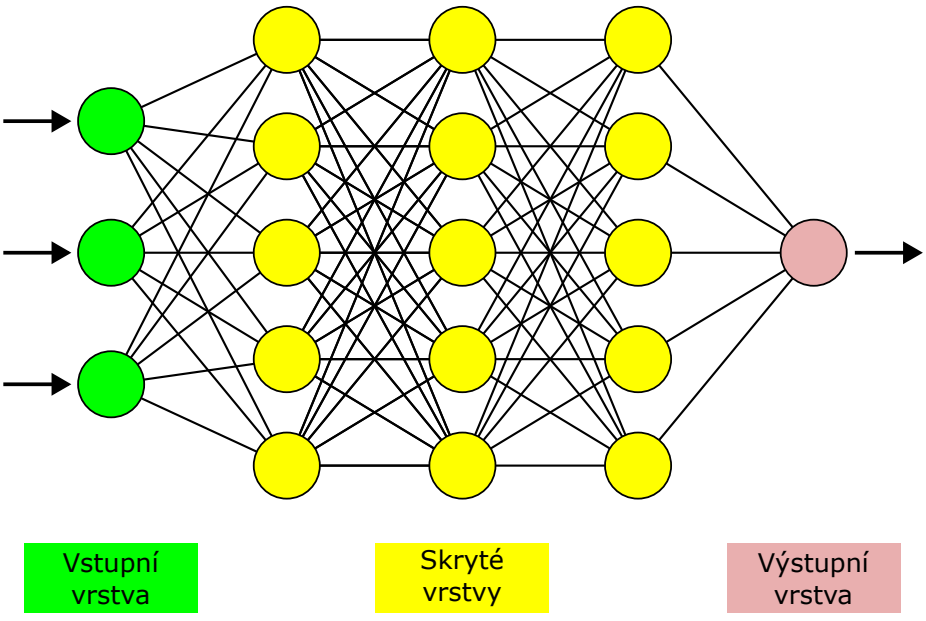
Obrázek 4.9: Křížení dvou chromozomů vybraných jedinců z jedné generace (nahoře) v genetickém programování a vytvoření potomci (dole).



Obrázek 4.10: Mutace aplikovaná na původního jedince (vlevo) a výsledný jedinec (vpravo).

4.5 Neuronová síť

Jedním z typů modelů, který jde použít pro řešení regresního problému, jsou neuronové sítě. [32] Neuronové sítě se skládají z vrstev neuronů. Každý neuron je propojen na vstupu s výstupy předchozí vrstvy. Výstupní hodnota neuronu se spočítá jako suma vstupních hodnot vynásobená odpovídajícími vahami a přičtením hodnoty *bias*. Pro zbavení linearity se obvykle na výstup neuronu aplikuje nelineární funkce (ReLU, Sigmoid, Softmax. . .). [2] Pro vyhodnocení přesnosti neuronové sítě se používá chybová funkce. Ta je často *Mean Squared Error*, protože s větší absolutní chybou roste i derivace této funkce, z čehož profituje algoritmus trénování *Gradientní sestup*. Tento algoritmus iterativně hledá minimum chybové funkce pomocí posouvání se ve směru opačném gradientu funkce.



Obrázek 4.11: Schéma neuronové sítě s vyznačenou vstupní vrstvou, skrytými vrstvami a výstupní vrstvou.

Neuronové sítě pro odhad hodnoty spojité veličiny (regrese) obvykle mají ve vstupní vrstvě tolik neuronů, kolik je vstupních příznaků. Ve výstupní vrstvě pak obvykle bývá jeden neuron s lineární aktivací. Mezi vstupní a výstupní vrstvou se pak vyskytuje několik skrytých vrstev, které mají daný počet neuronů s nelineární aktivační funkcí (např. ReLU). [1] Schéma takové neuronové sítě je na obrázku 4.11.

Kapitola 5

Data k predikci doby běhu k-Wave

V této kapitole popisuji data, která jsem použil pro řešení odhadu doby běhu programu k-Wave na superpočítači. Nejlepší datová sada, která by k tomuto problému mohla být, by obsahovala všechny možné kombinace výpočetních zdrojů na superpočítači, všechny možné velikosti a atributy simulované úlohy, a výsledek odhadu bychom mohli rovnou z této sady brát. Protože je získávání dat do této datové sady náročné na výpočetní zdroje superpočítače, je to taky náročné na finanční prostředky. I proto jsme limitování při sběru dat a je třeba je získávat s rozvahou. Množství nasbíraných dat taky následně ovlivňuje výběr metody strojového učení, protože některé metody jsou lepší než jiné při menším počtu nasbíraných dat. Tato kapitola popisuje v sekci 5.1 sběr dat a v následující sekci (5.2) je popsána struktura dat.

5.1 Získávání dat

Data, která jsem použil v této práci, jsou kombinací dat sesbíraných doc. Ing. Jiřím Jarošem Ph.D. a mnou. Data byla získávána spuštěním simulační úlohy na superpočítačích Barbora a Karolina. Simulace byla omezena na prvních 100 simulačních kroků, aby se zamezilo dlouhým běhům k-Wave. Čas, který nás zajímá, je čas, kdy byla spuštěna samotná simulace. K-Wave před samotným simulováním ultrazvuku provádí vstupně-výstupní operace závislé na diskovém poli a taky se iniciuje knihovna FFTW, která v módu *FFTW_EXHAUSTIVE* provádí optimalizaci výpočtu Fourierovy transformace. Tyto operace jsou z výsledného času vyjmuty, stejně jako finální ukládání výsledku na disk. Výsledný čas je počet milisekund na jeden simulační krok.

Typ simulační úlohy byl pokaždé stejný. Měnily se pouze počty diskrétních bodů prostoru. Ve všech simulovaných úlohách se jedná o heterogenní médium a nelineární vlnění. Dále je definován počáteční tlak `p0` a senzory definovány binární maticí. Tato úloha byla spouštěna na různých velikostech simulačního prostoru, především na krychlích, ale taky jsem měřil hranoly a prostory v každé ose jinak dlouhé. Na superpočítači Barbora jsem naměřil 39 různých velikostí simulační úlohy.

Na Barboře jsem naměřil tyto velikosti simulačního prostoru:

- **krychle:** 512^3, 576^3, 640^3, 645^3, 648^3, 650^3, 672^3, 675^3, 704^3, 736^3, 748^3, 756^3, 768^3, 800^3, 810^3, 813^3, 832^3, 840^3, 864^3, 896^3, 900^3, 960^3, 1024^3, 1280^3, 1536^3, 2048^3, 3072^3

- **hranoly:** $256 \times 256 \times 1024$, $256 \times 256 \times 1536$, $256 \times 256 \times 2048$, $384 \times 384 \times 1024$, $384 \times 384 \times 1536$, $384 \times 384 \times 2048$, $432 \times 432 \times 1728$, $512 \times 512 \times 1024$, $512 \times 512 \times 1536$, $512 \times 512 \times 2048$
- **kvádry:** $384 \times 256 \times 1536$, $384 \times 512 \times 1536$

Na obrázku 5.1 jsou vyobrazeny simulační časy pro krychle od velikosti 512^3 do 1024^3 s krokem 64. Svislá osa používá logaritmickou stupnici pro lepší znázornění. V grafu stojí za všimnutí místa, kdy se přidáním jednoho uzlu sníží výpočetní čas o mnohem více, než jinde. Jako příklad lze zmínit simulační prostor 1024^3, kdy takový skok nacházíme mezi 14 a 15 uzly. Důvod většího poklesu času mezi těmito dvěma uzly je ten, že pro 15 uzlů již můžeme použít 512 ranků pro výpočet, zatímco pro 14 uzlů můžeme použít maximálně 504 ranků ($14 * 36$), protože jeden uzel na Barboře má 36 procesorů (viz kapitola 2.4.1). Jenže použití 504 ranků by z hlediska distribuce výpočtu nemělo žádný přínos oproti použití 342 ranků (1024/3), protože nově přidané ranky by musely velkou část běhu programu jenom čekat na ostatní. Proto se po přidání 15. uzlu může objem výpočtu snížit pro všechny pracující procesory a tím se zamezí tomu, aby některý procesor čekal na jiný procesor, kterému se objem práce nesnížil. Popsaný skok je detailně zobrazený v Tabulce 5.1.

Počet uzlů	Dostupných procesorů	Počet ranků	Čas [ms/krok]
13	468	342	962,0
14	504	342	916,3
15	540	512	753,2
16	576	512	704,1

Tabulka 5.1: Silné škálování simulace o rozměru 1024^3 okolo skoku z 14 na 15 uzlech. Přidáním výpočetního uzlu se sníží čas i přes zachování počtu ranků, protože se zvýší agregovaná propustnost sítě a paměti.

5.2 Struktura dat

Ze spouštěných simulací jsem zaznamenával tato základní data:

- **Velikost osy X** – Celočíselný údaj, který popisuje počet diskrétních bodů simulovaného prostoru v ose X.
- **Velikost osy Y** – Celočíselný údaj, který popisuje počet diskrétních bodů simulovaného prostoru v ose Y.
- **Velikost osy Z** – Celočíselný údaj, který popisuje počet diskrétních bodů simulovaného prostoru v ose Z.
- **Počet uzlů** – Celočíselný údaj popisující počet uzlů superpočítače, které byly k simulování alokovány a použity.
- **Počet ranků** – Celočíselný údaj, který udává počet procesů, které byly použity pro simulaci programem k-Wave. Jedná se o parametr spuštění MPI, nejedná se o počet všech dostupných jader na dostupných uzlech.

- **Čas simulace** – Jedná se o desetinné číslo, které udává počet milisekund, které trvala simulace jednoho kroku. Toto číslo je průměrem prvních 100 kroků simulace.

Tato základní data jsem následně rozšířil o další data, která jsem vypočítal z původních dat uvedených v seznamu výše. Motivací pro rozšíření těchto dat o níže uvedené bylo, aby se algoritmy strojového učení mohly snáze optimalizovat a mohly do dat dostat expertní znalost vycházející ze znalosti programu k-Wave a superpočítače Barbora (Karolina). Data jsem rozšířil o následující příznaky:[1]

- **Factor X** – Jedná se o největší prvočíslo ze seznamu prvočísel rozkládajících počet diskrétních bodů ve směru osy X. Motivací k tomuto atributu bylo zavézt informaci o tom, že knihovna FFTW funguje nejlépe pro pole, jejichž délka se dá rozložit na prvočísla do 2, 3, 5 a 7.
- **Factor Y** – Stejné jako *Factor X*, ale v ose Y.
- **Factor Z** – Stejné jako *Factor X*, ale v ose Z.
- **Podíl simulujících procesorů** – Jedná se o podíl počtu procesorů, které jsou použity k simulaci, ku celkovému počtu dostupných procesorů na uzlech zapojených do úlohy. Motivací zavedení tohoto atributu byla domněnka, že čím více dostupných procesorů bude použito k simulaci, tím rychlejší výpočet bude. V implementaci je pojmenováno *ranksRatio*, viz kapitolu 7. Přidáním tohoto příznaku do algoritmů strojového učení se snažím napomoci odhadnout skoky v čase, které jsou popsané v kapitole 5.1. Hodnotu tohoto příznaku v měřeních okolo skoku z 14 na 15 uzlech pro prostor 1024^3 jde vidět v tabulce 5.2. Hodnota se může pohybovat v intervalu $\langle 0; 1 \rangle$. Čím blíže je 1, tím více dostupných procesorů je zapojených do simulace a tím nižší bude výpočetní čas.
- **Podíl procesorů s dostatečným množstvím dat** – Jedná se o ukazatel, který říká, jak velký je podíl procesorů, které jsou během simulace aktivní po celou dobu, protože mají vždy přidělenou určitou část simulačního prostoru. V implementaci je nazváno *fullWorkX*, *fullWorkY* a *fullWorkZ*. Motivací pro vytvoření toho příznaku bylo, aby výsledný model umět predikovat čas simulace i v podmínkách, kdy simulace nebude spuštěna s vhodnými parametry a bude mít k dipozici ranky, které nebudou mít v jistý okamžik co dělat. Jedná se spíše o číselný ukazatel, než o přesné vyčíslení počtu ranků... Vzoreček pro výpočet tohoto ukazatele je uveden níže:

$$f = \frac{\frac{d}{r}}{roundUp(\frac{d}{r})}, \tag{5.1}$$

kde f je hodnota tohoto příznaku, d je počet diskrétních bodů v dané ose, r je počet ranků a $roundUp$ je funkce zaokrouhlení nahoru. Správně nastavené úlohy mají hodnotu tohoto ukazatele 1 nebo velmi blízko 1. Čím je hodnota nižší, tím hůře je úloha nastavena. Obrázek 5.2 znázorňuje důležitost správně nastaveného počtu ranků pro výpočetní úlohu.

[1] Protože se výpočetní prostor distribuuje v osách Y a Z, teoreticky bychom přidané příznaky *Factor X* a *fullWorkX* mohli vynechat. Nicméně jejich přítomnost by neměla ničemu vadit a kvůli použitelnosti i v budoucích verzích k-Wave, které mohou výpočet distribuovat jinak, jsem se rozhodl je ponechat.

Počet uzlů	Dostupných procesorů	Počet ranků	Podíl simulujících procesorů	Čas
13	468	342	0,73	962,0
14	504	342	0,68	916,3
15	540	512	0.95	753,2
16	576	512	0,89	704,1

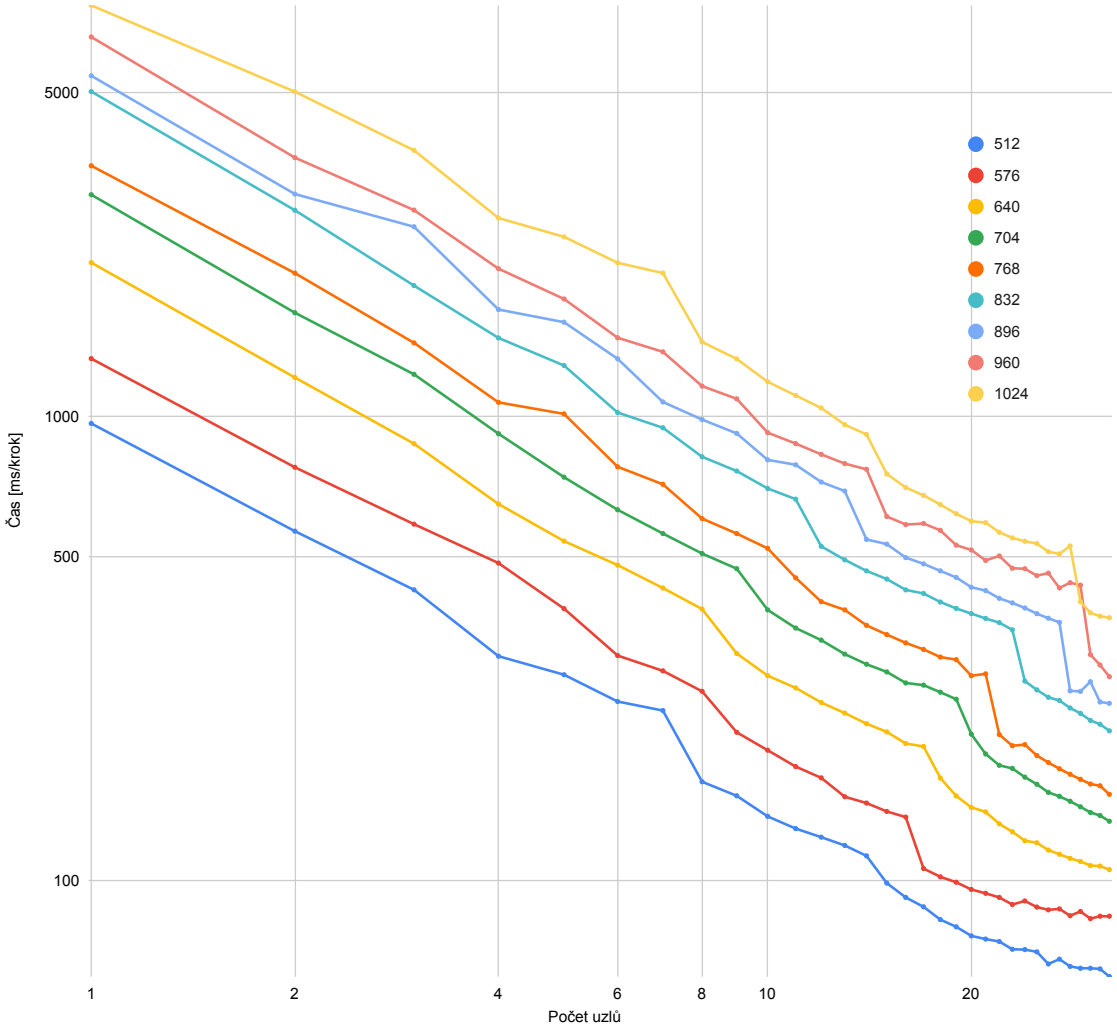
Tabulka 5.2: Rozšířená tabulka 5.1 o hodnoty příznaku *podílu simulujících procesorů*. Škálování simulace o rozměru 1024^3 okolo skoku z 14 na 15 uzlech. Z tohoto příznaku jde snáze odhadnout místo velkého snížení času.

5.3 Nasbíraná data

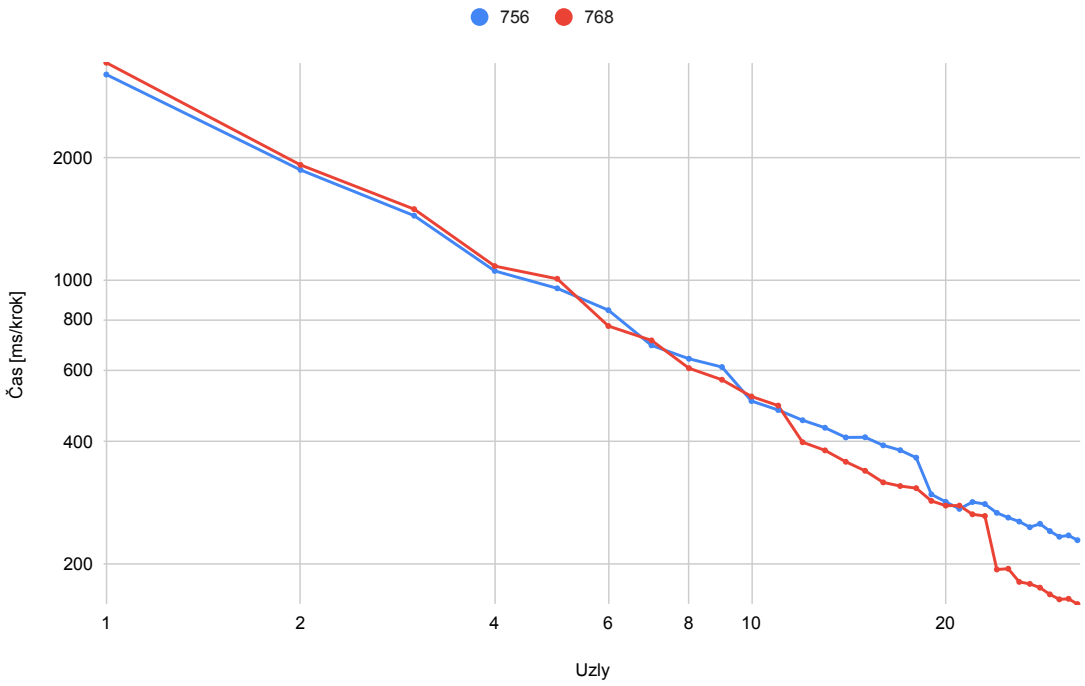
Dohromady jsem nasbíral data o 2 080 spuštěných instancí simulace k-Wave, z toho 1 813 je na superpočítači Barbora a 267 je na superpočítači Karolina.

Na superpočítači Barbora jsem naměřil celkem 39 různých velikostí simulačního prostoru, z toho je 27 krychlí, 10 hranolů a 2 obecné kvádry. V těchto naměřených datech je 7 rozměrů prostoru se špatným rozkladem na prvočísla (*factor* větší než 7).[2] V naměřených datech na Barboře je 414 měření, kde úloha nebyla ideálně distribuovaná napříč dostupnými uzly (příznak *fullWork* je menší než $0,95$).

[2]Měřil jsem nejen úlohy, které jsou vhodné pro k-Wave simulaci. Zkoušel jsem i úlohy, které nejsou vhodné (kvůli rozkladu na prvočísla) nebo jsou vhodné, ale nejsou ideálně spuštěny (špatný počet ranků). V dalších částech této práce budu filtrovat tato neideální měření a zjišťovat, jaký mají vliv na přesnost odhadů.



Obrázek 5.1: Graf znázorňuje závislost času jednoho kroku simulace na počtu dostupných uzlů pro simulování prostoru. Jsou znázorněny krychle od velikosti 512^3 do velikosti 1024^3 s krokem 64.



Obrázek 5.2: Graf znázorňuje důležitost správného nastavení počtu ranků pro simulační úlohu. Dvě podobně velké úlohy s vhodným rozkladem na pročísla, simulace krychle 756^3 je nastavena správně, příznak *fullWork* je stále 1, zatímco simulace krychle 768^3 není nastavena ideálně, příznak *fullWork* začíná na 1 a postupně klesá až na $0,56$. Pomocí hodnoty ukazatele *fullWork* jsme schopni odhadnout, jak dobře je výpočet nastaven a jestli jsou dostupné výpočetní zdroje správně využity. Zatímco větší krychle (správně distribuovaná) má na 1 uzlu simulační dobu o více než 200 ms/krok větší, na 32 uzlech ji má o 70 ms/krok menší, což odpovídá poklesu o $30,4\,\%$.

Kapitola 6

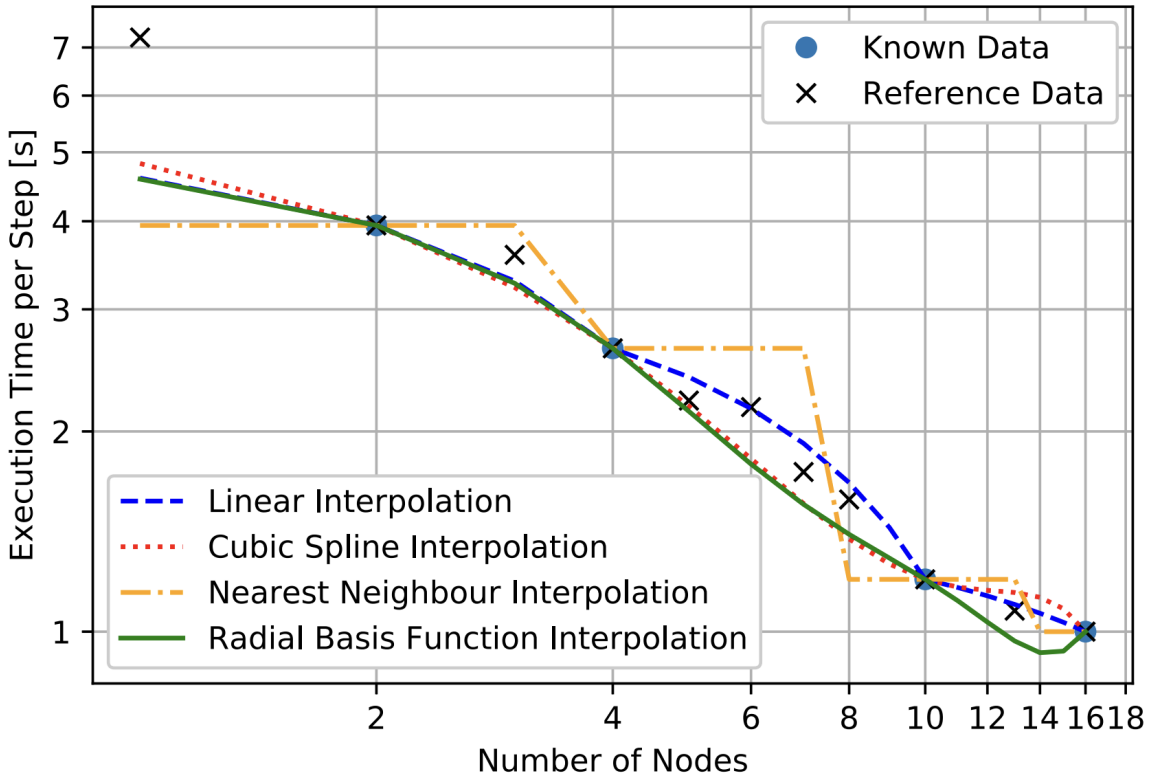
Existující řešení a návrh vlastního řešení

Tato kapitola nejdříve v podkapitole 6.1 přináší krátké shrnutí toho, jak problém odhadu doby běhu algoritmu řeší jiní autoři. Představí řešení používané v k-Dispatch [15] pro odhad doby běhu programu k-Wave MPI. Následně uvede další 2 práce, které se odhadem doby běhu algoritmů zabývají. V podkapitole 6.2 navrhne řešení pro odhad doby běhu programu k-Wave, který se inspiruje současným vědním poznáním, zároveň je ale inovativním co se týče predikce času odhadu pro program k-Wave. Následně navrhne práci se získanými daty a navrhne způsoby pro otestování funkčnosti a přesnosti daných metod a určí cíle, které od nových metod požadujeme. Tato podkapitola je spíše úvahou autora, která popisuje, jak nad daným problémem přemýšlel a proč se rozhodl zvolit právě toto řešení. Taky je zde zvolené řešení popsáno s vyšší mírou abstrakce. Detailněji je pak popsáno v kapitole 7.

6.1 Současný stav a existující řešení

Problematika odhadu doby běhu k-Wave simulace je pro koncového uživatele důležitá. Marta Jaroš et al. se mimo jiné odhadem doby běhu simulace k-Wave zabývají v práci Estimation of Execution Parameters for k-Wave Simulations. [14] V práci se porovnávají průměrné chyby odhadů pomocí lineární interpolace, kubických splajnů, metody nejbližších sousedů a sítí RBF (radial bases functions). Pro známé úlohy nejlépe performovala metoda odhady pomocí kubických splajnů, která měla průměrnou chybu pod 2,29 %. Obrázek 6.1 znázorňuje výpočetní čas potřebný k simulování prostoru o rozměrech $768 \times 900 \times 600$. Na neznámých úlohách, kde algoritmus odhadu nezná ani jeden čas běhu simulace dané úlohy, byla průměrná chyba odhadu až 15 % podle dané domény.

Jiní autoři přišli s metodou, která zkoumá zdrojové kódy distribuovaného software, na jejich základě konstruuje konečné automaty, a ty jsou spolu s datech o provedených bězích vstupem pro neuronovou síť, která provádí odhady doby běhu programu. [6] Frank Hutter et al. ve své práci *Algorithm runtime prediction: Methods evaluation* [11] nabízí shrnutí používaných přístupů k modelování doby běhů konkrétních algoritmů a přináší srovnání daných metod na vybraných 11 algoritmech. Zabývá se jak tradičními interpolačními technikami, tak i neuronovými sítěmi, náhodnými lesy (random forest), regresními stromy a dalšími metodami.



Obrázek 6.1: Graf znázorňující predikci známé k-Wave úlohy o velikosti $768 \times 900 \times 600$ pomocí lineární interpolace, kubických splajnů, metody nejbližších sousedů a RBF (radial bases functions). Modrým kolečkem jsou znázorněny známé body a černým křížkem jsou znázorněny neznámé body. Jedná se o MPI implementaci k-Wave, data jsou naměřená na superpočítači Anselm. Převzato z [14].

6.2 Návrh řešení

Tato práce si klade za cíl vytvořit model pro predikci doby běhu programu k-Wave na superpočítači Barbora. Jako model strojového učení použijeme symbolickou regresi a neuronové sítě. Budeme tedy zkoumat, které z těchto metod mají větší přesnost na daných datech a z nich vybereme, nebo poskládáme model výsledný.

6.2.1 Práce s daty

V rámci řešení této práce jsem nasbíral velké množství dat o délce běhu programu k-Wave v závislosti na velikosti vstupní úlohy (viz kapitola 5). Data jsem se snažil sbírat tak, aby byla jednak hustě zastoupená oblast, která se v praxi často simuluje (krychle $512^3 - 1024^3$ s vhodným rozkladem na prvočísla), zároveň ale, aby byly k dispozici data o „exotických“ úlohách, úlohách spuštěných s nevhodným nastavením počtu ranků, simulacích se špatným rozkladem velikosti osy na prvočísla (pomalejší běh FFTW, viz 3.3) nebo simulací s různorodou velikostí os (hranoly a kvádry).

Protože tato různorodá data mohou zanést šum do natrénovaných modelů, zkusím data filtrovat podle několika kritérií a vytvořit tím několik datových sad. Zkusím data filtrovat podle 3 filtrů:

- Krychle
- Vhodný *factor*
- Vhodný počet ranků

Celkem tak budu mít 8 datových sad, které se budou lišit nastavením výše uvedených filtrů. Vzniknou tak tyto datové sady:

- Bez filtru
- Pouze krychle
- Pouze úlohy s vhodným příznakem *factor*
- Pouze úlohy s vhodným počtem ranků
- Krychle s vhodným příznakem *factor*
- Krychle s vhodným počtem ranků
- Krychle s vhodným příznakem *factor* a vhodným počtem ranků
- Úlohy s vhodným příznakem *factor* a vhodným počtem ranků

Pro každou z těchto datových sad vytvořím model pomocí neuronové sítě a pomocí symbolické regrese.

Každou z těchto datových sad rozdělím do trénovací, validační a testovací sady podle úlohy (velikosti simulačního prostoru). Všechna data z validační a testovací sady budou mít své úlohy, které nejsou zastoupeny v žádné jiné sadě. Pro porovnání s již existujícím prediktorem [14] tak budou relevantní pouze výsledky na neznámých úlohách. Poměr rozdělení bude 70 % úloh v trénovací sadě, 15 % ve validační sadě a 15 % v testovací sadě.

6.2.2 Symbolická regrese

Pro trénování modelů symbolické regrese pomocí genetického programování použiji nástroj HeuristicLab[1]. K tomuto nástroji vytvořím plugin, který do symbolické regrese přidá nové operace, konkrétně tyto:

- Modulo
- Zaokrouhlení nahoru
- Zaokrouhlení dolů
- Zaokrouhlení na celé číslo

Rozhodnutí pro výběr HeuristicLabu padlo po vyzkoušení několika různých knihoven pro Python, které umí dělat symbolickou regresi. Vyzkoušel jsem alespoň dále uvedené: DEAP[2], PySR[3] a gplearn[4]. Žádná z těchto knihoven ale podle subjektivního názoru autora

[1] https://dev.heuristiclab.com
[2] https://deap.readthedocs.io/en/master/
[3] https://github.com/MilesCranmer/PySR
[4] https://gplearn.readthedocs.io/

nepředčí Heuristiclab v řešení problému této práce. Buď do nich nelze přidávat vlastní matematické operace (modulo, zaokrouhlení), nebo neumožňují podmínky typu *if-then-else*, nebo nemají implementovanou takovou paletu algoritmů pro mutaci, křížení a ostatní parametry genetického programování...

Pro každou datovou sadu (různé nastavení filtrů dat) natrénuju model pro symbolickou regresi. Hyperparametry trénování budu experimentálně upravovat tak, abych minimalizoval průměrnou chybu odhadu.

6.2.3 Neuronová síť

Vytváření a trénování neuronových sítí v jazyce Python budu dělat pomocí knihovny Tensorflow[5]. Budu vytvářet běžnou neuronovou síť pro regresi. Typická architektura je taková, která má několik neuronů v první vrstvě (podle počtu příznaků) s aktivační funkcí ReLU, dále obsahuje několik skrytých vrstev také s touto aktivační funkcí a v poslední vrstvě (výstupní) je jeden neuron, jehož hodnotu můžeme brát jako predikovanou hodnotu pro daný vstup. Data před vstupem do první vrstvy budu normalizovat. Pro zamezení přetrénování budu využívat *dropout* vrstvy, které budou umístěné mezi různými vrstvami v neuronové síti. Počet epoch pro trénování určím za běhu trénovacího algoritmu pomocí metody brzkého zastavení *(early stopping)*.

Trénování a vyhodnocování modelu neuronové sítě budu dělat v Jupyter notebook souboru. Jako běhové prostředí použiju službu Google Colab[6], která umožňuje snadno spouštět Jupyter notebooky a nabízí hardwarové akcelerátory pro trénování neuronových sítí spolu s předem připraveným prostředím pro využití populárních Python knihoven, včetně knihovny Tensorflow. Hyperparametry neuronové sítě budu experimentálně upravovat na základě dosažených výsledků tak, abych minimalizoval průměrnou chybu odhadu.

6.2.4 Výsledný model

Po vyhodnocení přesnosti odhadů na různých datových sadách se rozhodnu pro využijí jednoho (nejlepšího) modelu, který prohlásím za finální model pro predikci nových dat v produkčním prostředí, nebo udělám model kombinovaný, pokud experimenty ukáží, že pro specifická data funguje lépe jiný model, než pro data obecná. Např. pro úlohy, které mají vhodný rozklad velikosti osy, může mít lepší přesnost jeden model, zatímco pro všeobecná data (bez filtru) může fungovat nejlépe model jiný. Taky může pro některou datovou sadu být nejlepší model vytvořený symbolickou regresí a pro jiná data to může být neuronová síť. Při skládání výsledného modelu bude kladen důraz na co největší přesnost odhadů.

6.2.5 Nástroj příkazového řádku

Abych výsledek mé práce udělal co nejsnadněji využitelný v praxi, rozhodl jsem se vytvořit CLI nástroj pro správu dat a vytváření predikcí. Pro vytvoření běhového prostředí použiju Docker Compose[7], který bude obsahovat 2 kontejnery:

- PostgreSQL databázi
- Linuxový container s Pythonem a potřebnými knihovnami

[5] https://www.tensorflow.org
[6] https://colab.research.google.com
[7] https://docs.docker.com/compose

Databáze bude po spuštění iniciována s nasbíranými daty a bude propojena s druhým kontejnerem, ze které se bude spouštět Python script. Tento script bude umět hlavně tyto dvě funkce:

- Exportovat data do formátů, který se vhodný pro trénování modelů v HeuristicLabu a v Tensorflow. Data budou před exportováním filtrovaná na základě požadovaných vlastností (viz kapitolu 6.2.1).
- Vytvářet predikce pro simulaci k-Wave se zadanými vstupy na základě výsledného modelu

Kapitola 7

Implementace

V této kapitole stručně popíšu, jak jsem implementoval řešení problému, kterým se tato práce zabývá. Nejdříve v této kapitole popíšu, jak jsem sbíral data. Poté popíšu, jak daná data uchovávám v databázi, abych s nimi později mohl snadno pracovat. Poté popíšu nástroj příkazového řádku, který jsem udělal pro snadnou práci s daty a pro vytváření nových predikcí. Následuje popis pluginu do HeursiticLabu, který přidává nové funkce do symbolické regrese. Na závěr popíšu parser, který jsem udělal pro vyhodnocování nových odhadů z modelů symbolické regrese a kapitolu zakončím trénováním neuronových sítí.

V rámci řešení této práce jsem tedy implementoval následující funkcionality:

- Skript pro vytváření k-Wave úloh a jejich spouštění na superpočítači
- Uchování nasbíraných dat v PostgreSQL databázi
- Nástroj pro export nasbíraných dat z databáze, včetně možnosti použití filtrů a výpočtů přídavných příznaků pro usnadnění trénování modelů
- Plugin do software HeuristicLab, který rozšiřuje možnosti symbolické regrese
- Prostředí Docker Compose vytvářející standardizované rozhraní pro nástroj příkazového řádku a propojení s vytvořenou databází
- Parser pro vyhodnocování nových odhadů z modelu symbolické regrese, jenž je exportován z HeuristicLabu v prefixové notaci
- Vytváření a trénování neuronových sítí pomocí knihovny Tensorflow v Jupyter notebook souborech

7.1 Sběr dat

7.1.1 Generování úloh

Vytváření simulačních úloh pro k-Wave probíhalo pomocí scriptů dostupných v příloze. Ta obsahuje scripty:

- `generate_data.sh` pro načtení Matlabu, přídání cesty k nainstalovanému balíčku k-Wave a spuštění scriptu pro generování zadání
- `generate_data.m` pro samotné vytváření simulační úlohy a její uložení na disk

Vytváření úloh pro k-Wave není vhodné spouštět na `login` uzlech, ale je potřeba využít výpočetní uzly. Ze zkušeností autora je nutné vytvářet úlohy o velikosti větší než 1024^3 na uzlech se zvýšenou velikostí paměti (využití front `qfat` nebo `qgpu`). Pro uložení úlohy je vhodné využít úložiště `scratch` (viz kapitolu 2.4) kvůli velikosti souboru.

7.1.2 Spuštění k-Wave simulací

Po vygenerování potřebných úloh je nutné iterativně spouštět simulace k-Wave. Každá simulace dané úlohy se liší pouze počtem použitých výpočetních uzlů a počtem MPI ranků na nich spuštěných.

```
#!/bin/bash

domain=1024x1024x1024
nodes=16
ranks=512

qsub -l select=$nodes:ncpus=36:mpiprocs=$ranks -l walltime=08:00:00 \
    -v nodes=$nodes,ranks=$ranks,domain=$domain \
    -q qprod -A OPEN-26-56 run.pbs
```

Výpis 7.1: Spuštění úlohy na superpočítači. Úloha poběží na 16 uzlech a spustí script `run.pbs` z výpisu 7.2.

```
#!/bin/bash

ml FFTW/3.3.9-gompi-2021a HDF5/1.12.2-gompi-2022a OpenMPI/4.1.4-GCC-11.3.0

mpirun -np ${ranks} --map-by node \
    /home/xbucht28/k-Wave-Fluid-MPI/kspaceFirstOrder3D-MPI \
    -i /scratch/project/open-26-56/Data/input_${domain}.h5 \
    -o /scratch/project/open-26-56/Data/tmp/${domain}-${nodes}-${ranks}.h5 \
    --p_final \
    --fftw_planner_flag FFTW_EXHAUSTIVE \
    --verbose 2 \
    --benchmark 100 \
  > /scratch/project/open-26-56/Logs/output-${domain}-${nodes}-${ranks}.log

rm /scratch/project/open-26-56/Data/tmp/${domain}-${nodes}-${ranks}.h5
```

Výpis 7.2: Script `run.pbs`, který spouští simulaci k-Wave. Pokud je spuštěn pomocí scriptu z výpisu 7.1, spustí simulaci k-Wave úlohy o rozměru 1024^3 na 16 uzlech s 512 MPI ranky.

Na výpisu 7.1 je vidět script, který odešle výpočetní úlohu do fronty `qprod`. Tato úloha bude mít po spuštění k dispozici 16 uzlů po dobu 8 hodin. Na výpisu 7.2 je script, který se spustí, jakmile je daná úloha spuštěna. Tento script si nejdříve načte potřebné moduly a pak spustí MPI verzi k-Wave. Velikost úlohy, počet uzlů a počet ranků se propaguje ze scriptu, který úlohu odeslal. Simulace je spuštěna s příznakem `FFTW_EXHAUSTIVE`, který

před samotnou simulací provede iniciaci knihovny FFTW, aby samotný běh této knihovny byl co nejrychlejší na dané architektuře. Samotný program k-Wave je spuštěný s příznakem `p_final`, díky kterému se během simulace neukládají průběžná data na disk. Výsledný čas tedy není ovlivněn vstupně/výstupními operacemi, které by jinak výsledek značně ovlivňovaly. Simulace běží prvních 100 kroků, které se experimentálně ukázaly jako dostatečně dlouhý počet pro zjištění průměrné doby simulace pro jeden krok.

Po dokončení simulace je smazán výstupní soubor obsahující data ze senzorů. Záznam z běhu k-Wave je uložen v `/scratch` úložišti v textové podobě. Po dokončení úloh jsem tyto textové záznamy procházel a zaznamenával z nich dobu běhu simulace, která je očištěná o dobu iniciace knihovny FFTW a o dobu načítání zadání a ukládání výsledku. Tyto hodnoty jsem pak dělil, abych získat počet milisekund potřebných na simulaci jednoho kroku a hodnoty jsem ukládal do databáze.

7.2 Docker Compose prostředí

Kvůli tomu, aby výsledky mé práce byly snadno využitelné v praxi, jsem se rozhodl vytvořit nástroj příkazového řádku, který umí mimo jiné provádět odhad času simulace pro zadané vstupní parametry (rozměr úlohy, počet uzlů a ranků). Mimo jiné tento nástroj umožňuje snadný export dat z databáze do formátu, který je vhodný pro trénování modelů symbolické regrese a neuronových sítí. Tento nástroj také umožňuje snadno, pomocí použití vybraných parametrů, filtrovat data, které se pro export použijí.

Pro zajištění co nejsnazšího použití mých výsledků v praxi jsem se rozhodl vytvořit prostředí Docker Compose, které se skládá ze dvou kontejnerů:

1. PostgreSQL databáze
2. Python container

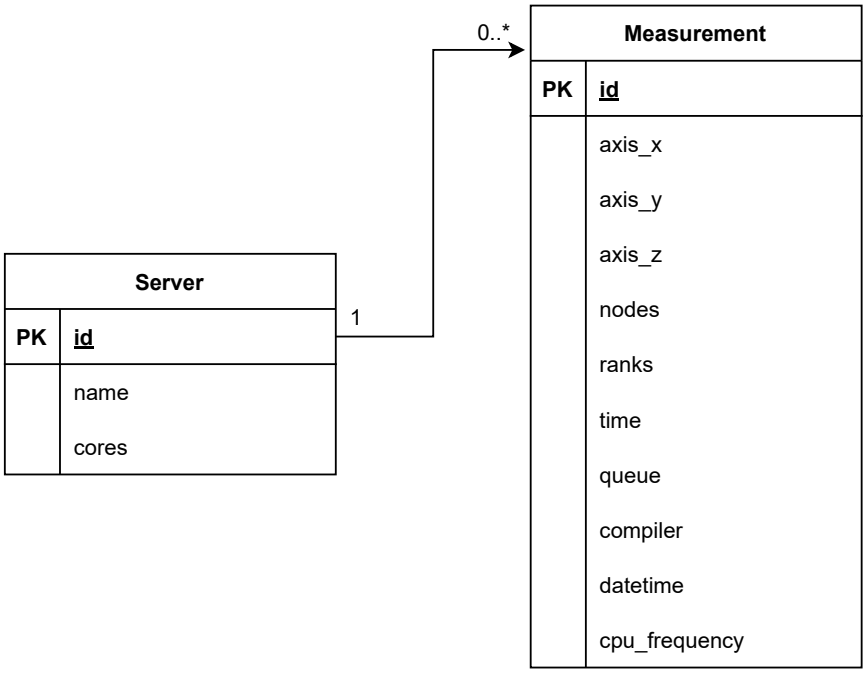
Struktura databáze je automaticky vytvořena a zároveň se do databáze importují nasbíraná data. Pro jednoduché spuštění jsem vytvořil script `run.sh`, který spustí Docker Compose prostředí a automaticky spustí interaktivní terminál v Python containeru. Pro použití praktických výsledků této práce tak stačí jen spustit tento script a je možno používat nástroj v Pythonu s automaticky vytvořenou a připojenou databází.

7.3 Uchování dat a export

7.3.1 Struktura dat

Pro uchování nasbíraných dat jsem se rozhodl vytvořit PostgreSQL databázi, ve které jsou uchovávaná nasbíraná data ve strukturované formě. Pro snadné připojení a čtení dat z databáze v Python scriptech jsem použil objektově relační mapování pomocí knihovny SQLAlchemy[1]. Na obrázku 7.1 je znázorněna datová struktura pomocí ER diagramu. Schéma databáze obsahuje dvě entitní množiny. Jedna je pro servery (superpočítače), druhá pro informace o provedených úlohách.

[1] https://www.sqlalchemy.org



Obrázek 7.1: ER diagram databáze pro uchování nasbíraných dat. Databáze obsahuje dvě entitní množiny. Entitní množina *Server* obsahuje jméno serveru a počet jader, které má každý uzel. Entitní množina *Measurement* zaznamenává údaje o každém běhu simulace, obsahuje velikost simulovaného prostoru v každé ose, počet uzlů, počet použitých MPI ranků a výsledný čas na jeden krok simulace, dále obsahuje detailní informace o konkrétním běhu (frekvenci procesoru, název fronty…) pro úplnost.

7.3.2 Export dat

Pro usnadnění spouštění trénování symbolické regrese a neuronových sítí jsem vytvořil script, který nasbíraná data vyfiltruje, vhodně rozdělí na trénovací, validační a testovací množiny a data uloží do CSV souborů. Po spuštění scriptu `run.sh`, který spustí Docker Compose prostředí a přesměruje do interaktivního terminálu (viz sekci 7.2) se export dat provede příkazem na výpisu 7.3. Ten vyfiltruje data podle zadaných parametrů a uloží data do souborů ve formátu CSV. Každá množina dat (trénovací, validační, testovací) je uložena do vlastního souboru. Zároveň jsou na standardní výstup vypsány souhrnné informace o výsledném rozdělení a počtu dat v každé z množin.

7.4 HeuristicLab plugin

Nástroj HeuristicLab podporuje v základu velkou množinu matematických funkcí, které mohou být použité při konstruování modelů symbolické regrese. Jedná se o jednoduché aritmetické operace (sčítání, odčítání, násobení, dělení), trigoniometrické funkce (sinus, cosinus, tangens, hyperbolický tangens), exponenciální a logaritmické funkce, mocniny, podmínky a další. Pro řešení tohoto problému bylo dohodnuto, že chceme přidat funkce zbytku po dělení a zaokrouhlení, protože tyto funkce by mohly pomoct s modelováním tohoto pro-

```
#!/bin/bash

python app/app.py \
    --export barbora.it4i.cz \
    --output data/nice-cubes \
    --validation 15 \
    --test 15 \
    --cube \
    --factor \
    --distributed
```

Výpis 7.3: Export nasbíraných dat do formátu CSV. Tento příkaz vyfiltruje data pouze ze superpočítače Barbora. Vybere pouze data o měřeních, která simulují prostor o tvaru krychle, jejíž rozklad na prvočísla je vhodný a je vhodně zvolený počet ranků. Data rozdělí tak, aby 15 % z vyfiltrovaných velikostí rozměrů bylo ve validační sadě a dalších 15 % v testovací sadě.

blému.[2] Tyto funkce ale v HeuristicLabu chybí. Jelikož je HeuristicLab software připravený pro rozšíření pomocí pluginů a je šířený pod obecnou veřejnou licencí GNU, rozhodl jsem se vybrané zdrojové kódy převzít a rozšířit tak, abych do funkcionality symbolické regrese přidal funkce modulo, zaokrouhlení na celé číslo, zaokrouhlení nahoru a zaokrouhlení dolů.

Nový plugin se jmenuje *ModuloRoundPlugin* a může být zprovozněn pomocí návodu od HeuristicLabu[3]. Stačí přeložit soubor `ModuloRoundPlugin.sln` ve Visual Studiu a výsledek zkopírovat do složky `/bin` v adresáři, kde je nainstalován HeuristicLab. Plugin je poté k dohledání v okně Plugin Manager (obrázek 7.2) a je správně nainstalován. Po nainstalování pluginu je před spuštěním trénování potřeba změnit gramatiku a stromový interpret na ty, které jsou poskytnuty novým pluginem. V nastavení gramatiky se pak objeví možnost povolení funkcí modulo, zaokrouhlení na celé číslo, zaokrouhlení nahoru a zaokrouhlení dolů.

7.5 Symbolická regrese

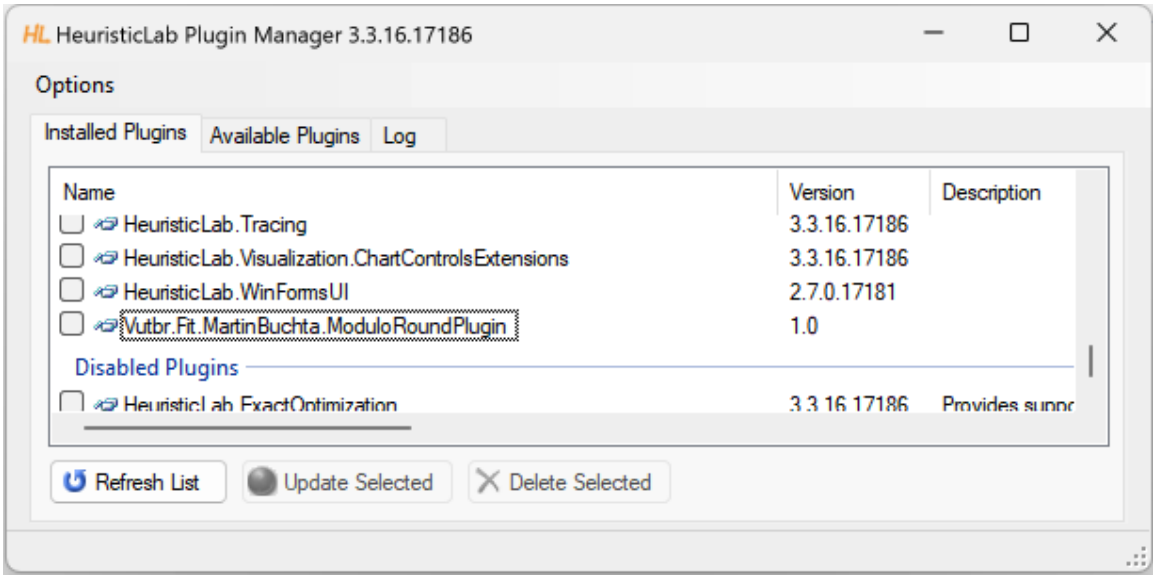
7.5.1 Trénování

Trénování modelů symbolické regrese probíhalo s pomocí pluginu popsaného v kapitole 7.4 a s daty získanými pomocí nástroje popsaného v kapitole 7.3.2, z nich jsem použil pouze trénovací množinu dat. Validační a testovací množiny jsem použil jen pro vyhodnocení na neviděných datech během trénování a pro porovnání s modely neuronových sítí, viz kapitolu 7.6.

Níže uvedené nastavení parametrů jsem nastavil na základě experimentů, ve kterých jsem postupně měnil chybovou funkci (Mean squared error, Mean absolute error, Maximum absolute error a Mean relative error), omezení velikosti stromu do hloubky a velikosti, počet generací a počet jedinců v každé generaci. Nastavení parametrů bylo provedeno výběrem experimentu s nejmenší průměrnou relativní odchylkou.

[2] Funkce zaokrouhlení byla použita i v manuálním výpočtu pomocného příznaku *Podíl procesorů s dostatečným množstvím dat*, viz sekci 5.2.

[3] https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/DevelopmentCenter/CreateNewPluginUsingVS



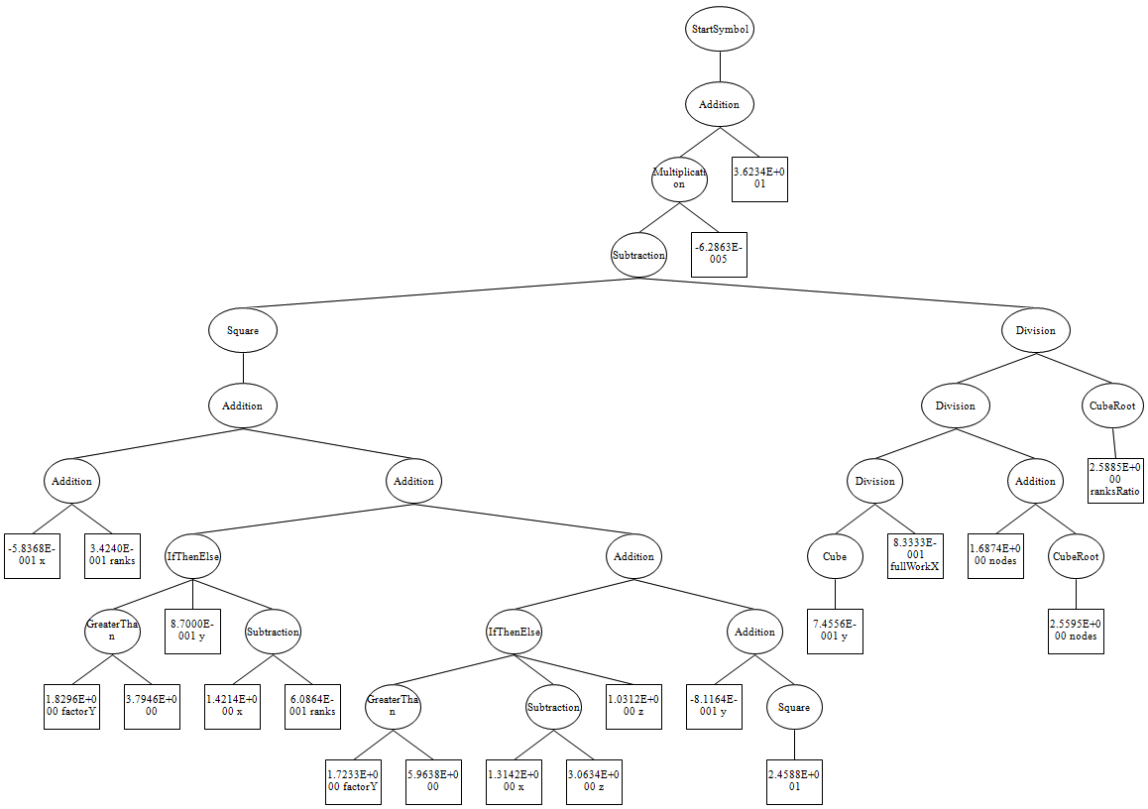
Obrázek 7.2: Funkční HeuristicLab plugin se zobrazí v okně Plugin Manager a je připravený k použití.

Během trénování symbolické regrese bylo vytvořeno a vyhodnoceno 1 000 000 generací pro každý trénovaný model. Každá z generací měla 100 jedinců. Generování stromů (včetně jejich křížení a mutací) bylo omezeno maximální hloubkou stromu o hloubce 16 a maximální velikosti stromu 42 uzlů. Jako chybová funkce byla použita funkce *mean squared error*. Cílem genetického programování bylo hodnotu této funkce minimalizovat. Chybová funkce pro každého jedince se počítala ze všech dat v trénovací množině. Během evoluce byl v populaci zachováván 1 nejlepší jedinec (1 elita). Pravděpodobnost mutace byla nastavena na 10 %. Byla použita gramatika a interpret z ModuloRound pluginu popsaného v této kapitole. Stromy se mohly skládat z těchto typů uzlů:

- Aritmetické operace (sčítání, odčítání, násobení, dělení)
- Modulo
- Zaokrouhlení (na celé číslo, nahoru, dolů)
- Exponenciální funkce
- Logaritmus
- Konstanta
- Vstupní proměnná násobená konstantou
- Mocninné funkce (na druhou, na třetí, druhá odmocnina, třetí odmocnina, obecná mocnina, obecná odmocnina)
- Podmínka *if-then-else*
- Porovnání (je větší než, je menší než)
- Logické operátory (a zároveň, nebo, zápor, XOR)

Zbytek nastavení HeuristicLabu zůstal na výchozím nastavení. Protože běh takového optimalizačního algoritmu je výpočetně náročný, prováděl jsem jej na superpočítači Barbora. Software HeuristicLab potřebuje ke svému běhu knihovny .NET frameworku a Windows Forms. I přes snahu vyhnout se virtualizaci se mi nepodařilo zprovoznit HeuristicLab na Linuxu pomocí Mono ani pomocí Wine. Zprovoznil jsem tedy virtuální stroj s Windows 11 na superpočítači Barbora pomocí nástroje QEMU[4]. Tento virtuální stroj jsem ovládal pomocí VNC. Nainstaloval jsem na něj HeuristicLab i s novým pluginem a spustil trénování. Protože Barbora má hardwarovou podporu pro virtualizaci[5], trénování fungovalo bez problémů. Doba trénování se liší v závislosti na velikosti trénovací množiny. Všechna trénování s výše uvedeným nastavením a použitým paralelním enginem trvala od 10 do 20 hodin čistého času za použití 1 uzlu fronty *qprod* na superpočítači Barbora. Trénování na každé datové sadě proběhlo právě jednou.

Po skončení trénování pomocí genetického programování jsem výsledný model uložil do textového souboru ve formátu prefixové notace.[6] Výsledný strom modelu pro datovou sadu „Krychle s vhodným příznakem factor a vhodným počtem ranků“ je znázorněn na obrázku 7.3.



Obrázek 7.3: Znázornění stromu nejlepšího modelu symbolické regrese pro datovou sadu „Krychle s vhodným příznakem factor a vhodným počtem ranků“ (viz 6.2.1).

[4] https://www.qemu.org/

[5] https://docs.it4i.cz/software/tools/virtualization/

[6] Pro export natrénovaného modelu do textové podoby je nutné zobrazit grafickou reprezentaci výsledného modelu a kliknout pravým tlačítkem na ikonku oken, poté vyberte *Textual Representation*. V této práci jsem používal *Default String Formatter*, který reprezentuje model pomocí prefixové notace.

7.5.2 Vyhodnocení

Pro snadné vyhodnocení natrénovaného modelu ze symbolické regrese na validační a testovací sadě jsem vytvořil funkci `predict` v modulu `parser` v Python nástroji. Tato funkce bere jako parametry název souboru s uloženým modelem, počet diskrétních bodů v osách X, Y a Z, počet uzlů, počet ranků a počet jader na jednom uzlu. Tato funkce si načte požadovaný model ve formě textového souboru, pomocí rekurzivního sestupu vyčíslí hodnotu modelu pro dané parametry a vrátí odhadovanou hodnotu.

7.6 Neuronové sítě

Pro vytváření neuronových sítí a jejich trénování jsem použil knihovnu Tensorflow. Trénování jsem prováděl v prostředí Google Colab, které umožňuje spouštět výpočty na určených hardwarových akcelerátorech a obsahuje mnoho knihoven předinstalovaných k použití. Záznamy z trénování jsou dostupné v jednotlivých Jupyter noteboocích.

Data, která vstupují do neuronové sítě, nejdříve normalizuji pomocí vrstvy `Normalization`[7]. Tato normalizace upraví data tak, aby měla střední hodnotu 0 a standardní odchylku 1. Ve výpisu 7.4 je popsána architektura použitých neuronových sítí. První vrstva neuronové sítě má 12 neuronů (podle struktury dat, viz kapitolu 5.2). Za ní se 6x opakuje plně propojená vrstva s aktivační funkcí *ReLU* a *Dropout* vrstvou, která je aktivní pouze během trénování. Na konci se nachází poslední plně propojená vrstva s jedním neuronem, jehož hodnota má význam predikované hodnoty. K této architektuře a tomuto nastavení hyperparametrů jsem dospěl experimentálním přístupem po vyzkoušení mnoha různých kombinací. Zejména jsem zkoušel měnit počty neuronů, přidávat/ubírat vrstvy a měnit dobu trénování.

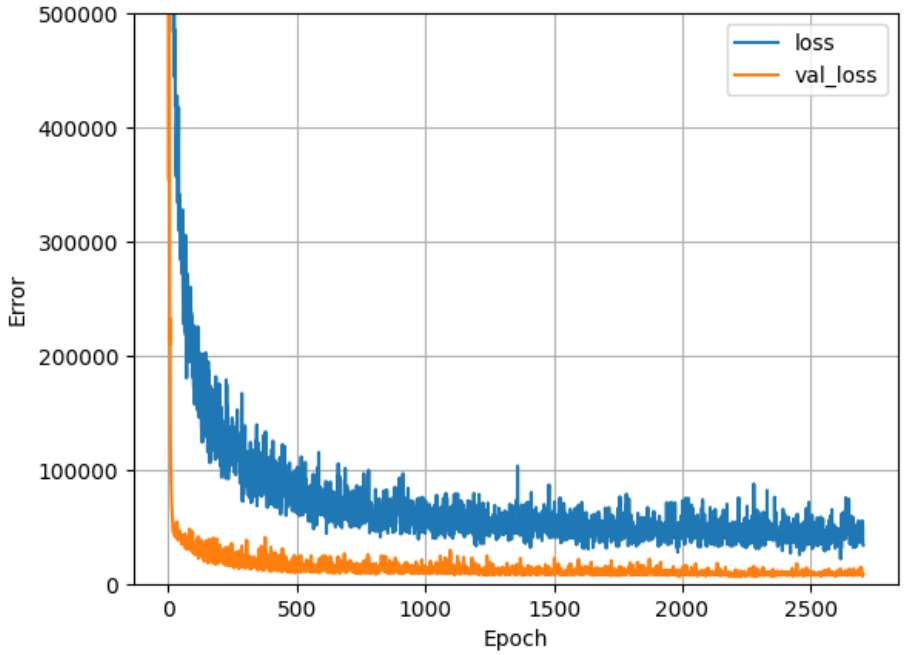
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(1),
])
```

Výpis 7.4: Architektura použité neuronové sítě.

Jako chybovou funkci jsem použil *mean squared error*. Trénování jsem prováděl optimalizátorem Adam s parametrem 10^{-4}. Pro ukončení trénování jsem použil techniku brzkého zastavení, která ukončí trénování modelu, pokud se chybová funkce na validační sadě ne-

[7] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization

zmenšila za posledních 500 iterací. Tato technika zabraňuje tomu, aby se neuronová síť učila příliš dlouho a aby došlo k přetrénování. Zároveň ale s použitím *Dropout* vrstev se zvyšuje počet potřebných epoch k natrénování. Proto se použití této techniky osvědčilo. Neuronovou síť jsem, stejně jako u symbolické regrese, natrénoval pro každou datovou sadu z 8 různých kombinací filtrů zvlášť. Výsledný model jsem uložil a vyhodnotil jeho průměrnou relativní chybu na všech datech. Na obrázku 7.4 je graf průběhu trénovací a validační chyby během trénování na všech datech z Barbory. Trénování je ukončeno, jakmile se validační chyba nesníží po dobu 500 epoch.



Obrázek 7.4: Graf znázorňující průběh trénování neuronové sítě na všech datech z Barbory.

Kapitola 8

Testy a experimenty

Tato kapitola popisuje, jak se ověřila funkčnost řešení vytvořeného v této práci. Protože je tato práce spíše experimentálního charakteru, budu zde převážně porovnávat predikovaná data vůči skutečným datům. Nejdříve uvedu souhrnné statistiky pro všechny modely na všech datových sadách, které vznikly. Poté budu zkoumat a komentovat jednotlivé datové sady a budu porovnávat odhadnuté hodnoty od hodnot reálných z vybraných velikostí simulačního prostoru. Budu porovnávat data jak z validační a testovací množiny, tak z množiny trénovací.

Sekce 8.1 popisuje vliv filtrování dat na přesnost odhadu. Ukazuje se zde, že je vhodné filtrovat data podle toho, zda mají vhodný rozklad velikosti domény na prvočísla. Pokud ano, je vhodnější použít model symbolické regrese. Pokud ne, je lepší použít neuronovou síť. Sekce 8.2 podrobně popisuje výsledky experimentů na datové sadě s filtrem vhodného rozkladu na prvočísla. Jsou zde uvedeny průměrné chyby odhadů na daných doménách a zároveň jsou výsledky vybraných domén prezentovány formou grafu, který porovnává skutečné a odhadované hodnoty. Níže uvedené hodnoty průměrné chyby jsou zprůměrované hodnoty od 1 do 32 uzlů. Sekce 8.3 popisuje experimenty na datové sadě bez použitého filtru. Použitím metod z této práce na jiném superpočítači se zabývá závěrečná sekce 8.4.

8.1 Vliv filtrování dat

Pro to, abych zjistil, jaký vliv mají filtry dat na přesnost natrénovaných modelů, jsem natrénoval modely neuronových sítí a symbolické regrese na všechny datové sady (všechny možné kombinace filtrů, viz kapitolu 6.2.1). Přesnost vytvořených modelů jsem porovnával pomocí průměrné relativní odchylky. Tu jsem počítal zvlášť pro každý model na trénovací, validační a testovací množině. Níže uvádím přehled průměrné relativní odchylky na všech datových sadách:

	Symbolická regrese	Neuronová síť
Trénovací data	13,70 %	9,35 %
Validační data	9,31 %	10,33 %
Testovací data	13,55 %	8,25 %

Tabulka 8.1: Průměrná relativní odchylka na datově sadě *Bez filtru*

Průměrná relativní odchylka na testovací sadě je znázorněna na grafu 8.1. Model symbolické regrese nejlépe performuje, pokud se v datech neobjevují data s nevhodným rozkladem

	Symbolická regrese	Neuronová síť
Trénovací data	7,78 %	7,79 %
Validační data	9,90 %	34,42 %
Testovací data	9,97 %	17,50 %

Tabulka 8.2: Průměrná relativní odchylka na datově sadě *Pouze krychle*

	Symbolická regrese	Neuronová síť
Trénovací data	5,58 %	5,75 %
Validační data	11,26 %	21,71 %
Testovací data	5,64 %	12,46 %

Tabulka 8.3: Průměrná relativní odchylka na datově sadě *Pouze úlohy s vhodným příznakem factor*

	Symbolická regrese	Neuronová síť
Trénovací data	18,62 %	6,32 %
Validační data	22,51 %	6,49 %
Testovací data	15,16 %	11,84 %

Tabulka 8.4: Průměrná relativní odchylka na datově sadě *Pouze úlohy s vhodným počtem ranků*

	Symbolická regrese	Neuronová síť
Trénovací data	10,25 %	5,99 %
Validační data	11,65 %	19,11 %
Testovací data	8,85 %	21,16 %

Tabulka 8.5: Průměrná relativní odchylka na datově sadě *Krychle s vhodným příznakem factor*

	Symbolická regrese	Neuronová síť
Trénovací data	14,53 %	6,83 %
Validační data	16,38 %	13,90 %
Testovací data	17,77 %	7,83 %

Tabulka 8.6: Průměrná relativní odchylka na datově sadě *Krychle s vhodným počtem ranků*

	Symbolická regrese	Neuronová síť
Trénovací data	8,93 %	7,11 %
Validační data	7,94 %	27,72 %
Testovací data	5,68 %	23,25 %

Tabulka 8.7: Průměrná relativní odchylka na datově sadě *Krychle s vhodným příznakem factor a vhodným počtem ranků*

na prvočísla. Data s velkým rozkladem na prvočísla mohou být velmi rozdílná od ostatních dat. Naopak vyfiltrování dat s nevhodným počtem ranků chybovost nesnížilo, to si vysvětluji přítomností přidaného příznaku *Podíl procesorů s dostatečným množstvím dat*, který do dat zavádí informaci o tom, jak dobře je zvolen počet ranků.

	Symbolická regrese	Neuronová síť
Trénovací data	7,27 %	7,50 %
Validační data	14,01 %	15,45 %
Testovací data	7,48 %	19,38 %

Tabulka 8.8: Průměrná relativní odchylka na datově sadě *Úlohy s vhodným příznakem factor a vhodným počtem ranků*

8.1.1 Výsledný složený model

Na základě výsledků těchto experimentů jsem se rozhodl, že jako výsledný model použiju model složený z těchto modelů popsaných výše:

- Model symbolické regrese pro data, která mají vhodný rozklad na prvočísla – testovací chyba 5,64 %.
- Model neuronové sítě pro všechna ostatní data – testovací chyba 8,25 %.

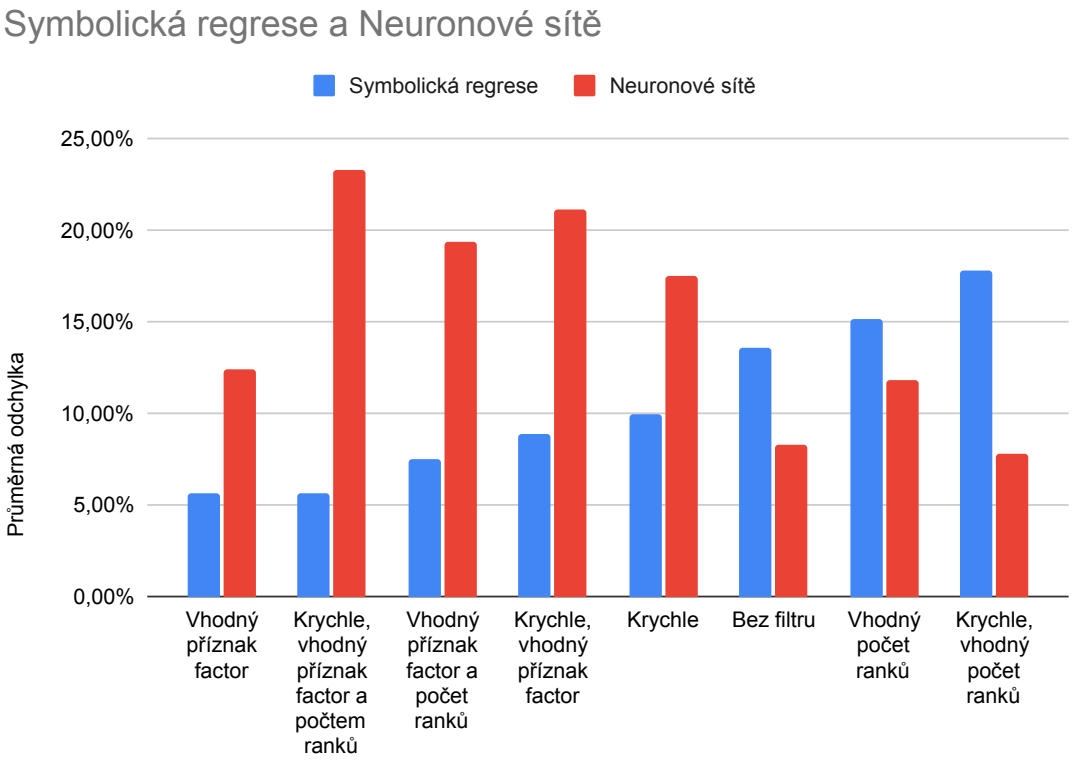
Graf 8.2 obsahuje krabicový graf relativní odchylky přes všechna data v testovací sadě výše zmíněných dvou modelů. Převážně v této práci uvádím aritmetický průměr relativní odchylky na všech měřeních v dané sadě, nicméně tento graf může poskytnout informaci o rozložení chyby a o velikosti maximální chyby.

8.2 S filtrem vhodného rozkladu na prvočísla

Tato sekce nabídne detailní pohled na predikce modelů na data vybraných rozměrů z datové sady *Pouze úlohy s vhodným příznakem factor*.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
$256 \times 256 \times 1536$	5,19 %	12,13 %
$256 \times 256 \times 2048$	8,66 %	12,56 %
$432 \times 432 \times 1728$	4,33 %	6,87 %
512^3	14,88 %	7,29 %
756^3	4,69 %	5,13 %
768^3	3,41 %	3,42 %
810^3	2,51 %	7,16 %
960^3	5,34 %	4,22 %
1280^3	6,21 %	2,48 %
2048^3	3,10 %	4,00 %

Tabulka 8.9: Relativní odchylka na vybraných **trénovacích datech** z datové sady *Pouze úlohy s vhodným příznakem factor* podle velikosti simulačního prostoru. Trénovací sada má celkem 24 různé velikosti prostoru.



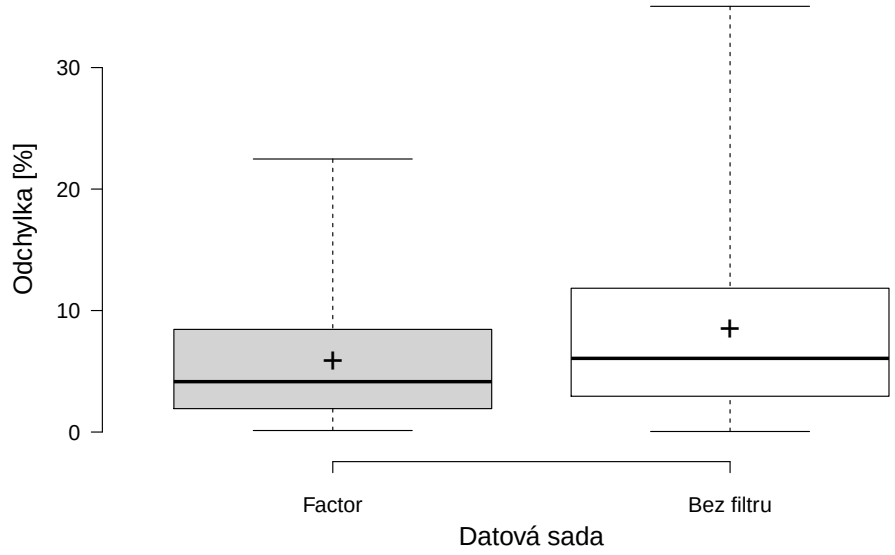
Obrázek 8.1: Graf znázorňuje průměrnou relativní odchylku modelů symbolické regrese a neuronových sítí na různě vyfiltrovaných datech. Data jsou seřazená od nejmenší odchylky s modelem symbolické regrese.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
$256 \times 256 \times 1024$	13,48 %	50,41 %
$384 \times 384 \times 1024$	15,59 %	32,13 %
$512 \times 512 \times 1024$	11,53 %	30,37 %
675^3	10,12 %	13,70 %

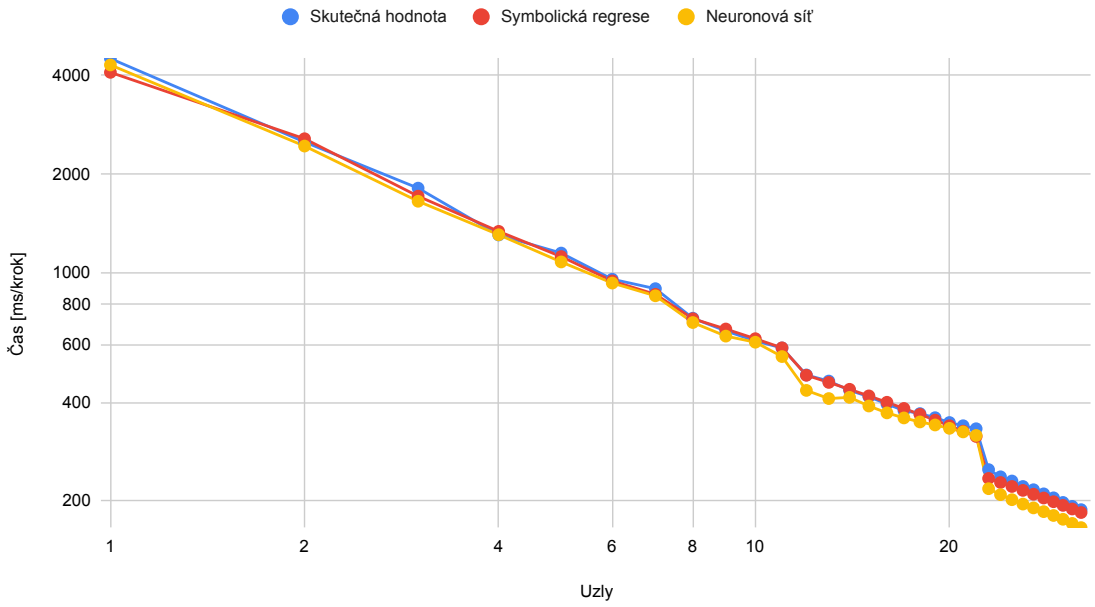
Tabulka 8.10: Relativní odchylka na datově sadě *Pouze úlohy s vhodným příznakem factor* **na validační sadě** podle velikosti simulačního prostoru. Validační sada má celkem 4 různé velikosti prostoru.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
$384 \times 384 \times 1536$	8,14 %	13,05 %
576^3	7,99 %	15,24 %
896^3	3,17 %	6,12 %
1024^3	4,78 %	14,40 %

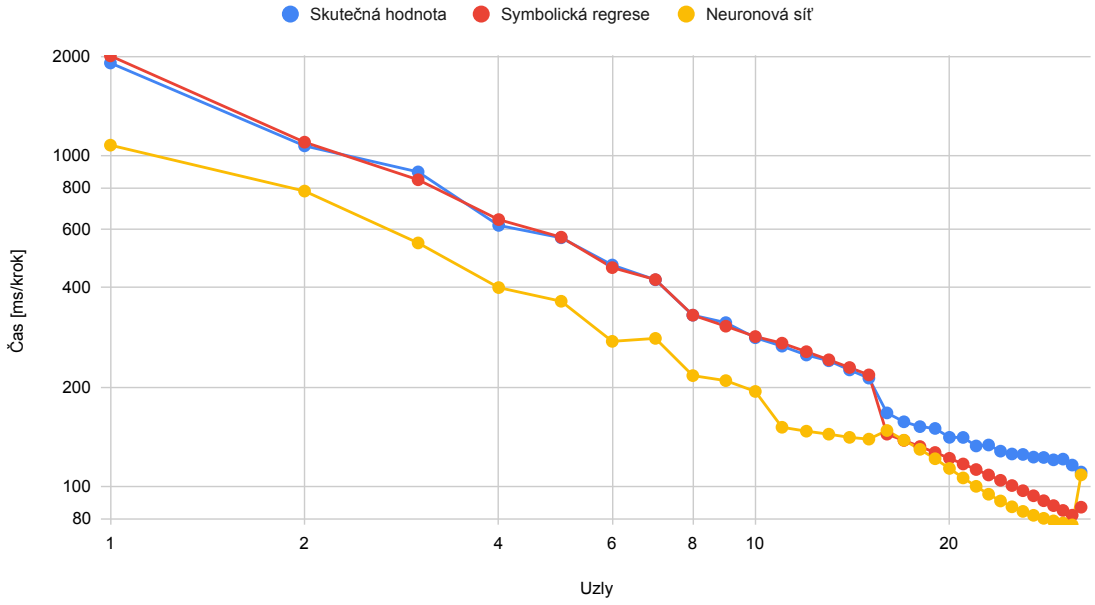
Tabulka 8.11: Relativní odchylka na datově sadě *Pouze úlohy s vhodným příznakem factor* **na testovací sadě** podle velikosti simulačního prostoru. Testovací sada má celkem 4 různé velikosti prostoru.



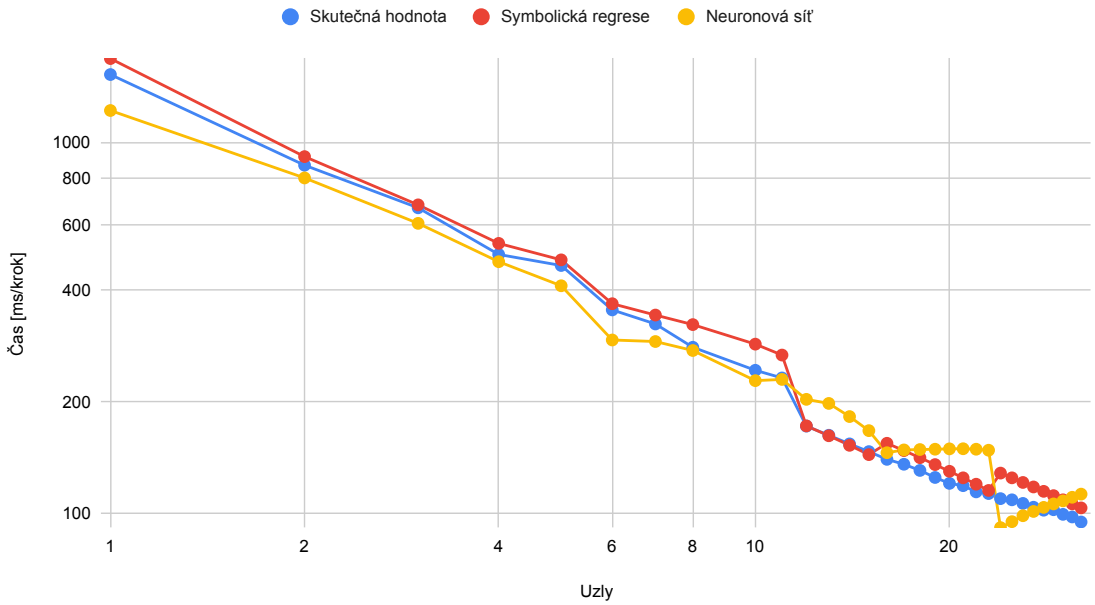
Obrázek 8.2: Krabicový graf relativní odchylky každého měření na testovací sadě dvou modelů, které jsou použity ve výsledném složeném modelu. Vlevo je model symbolické regrese na data s vhodným rozkladem domény na prvočísla. Vpravo je model neuronové sítě bez použití filtru dat. V grafu je znázorněna nejmenší a největší chyba, 1. a 3. kvartil, medián a aritmetický průměr (křížek).



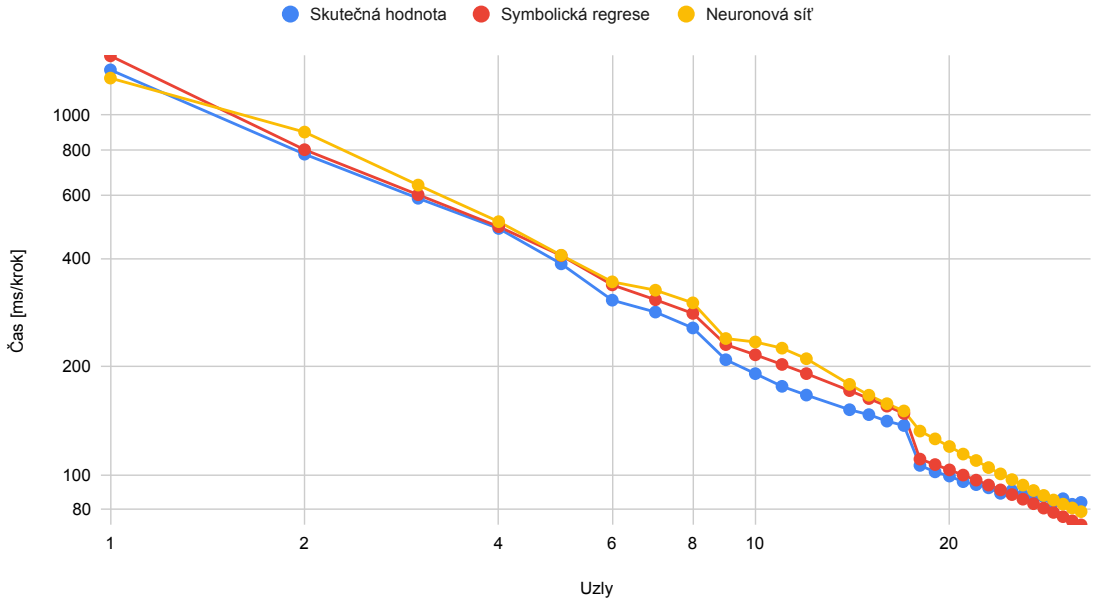
Obrázek 8.3: Graf reálného času na prostoru 810^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Pouze úlohy s vhodným příznakem factor*. Tato velikost prostoru se nachází **v trénovací sadě**.



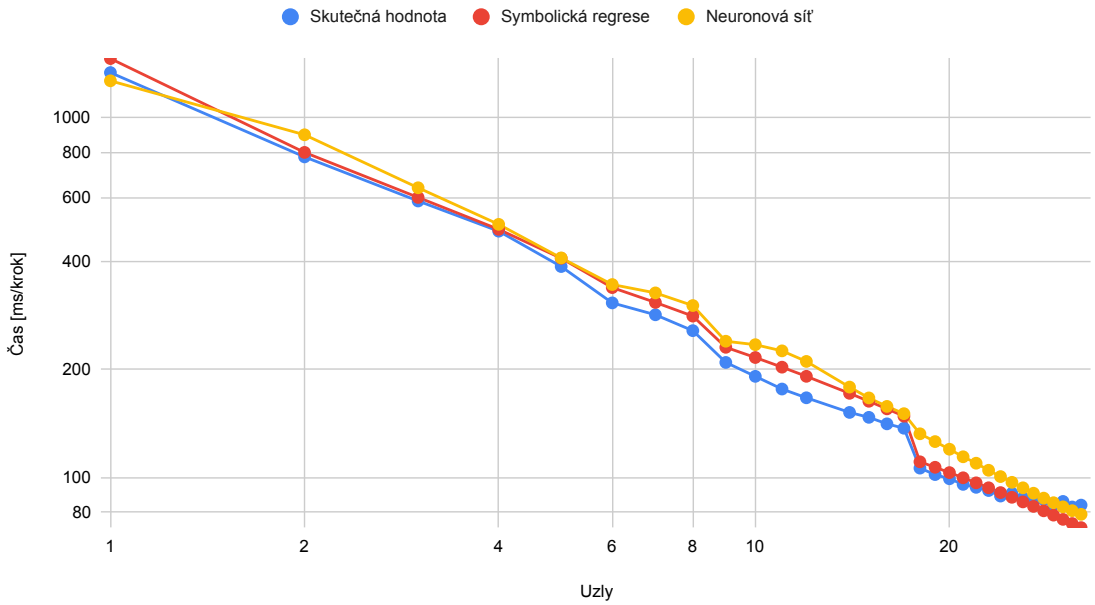
Obrázek 8.4: Graf reálného času na prostoru $512 \times 512 \times 1024$, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Pouze úlohy s vhodným příznakem factor*. Tato velikost prostoru se nachází **ve validační sadě**.



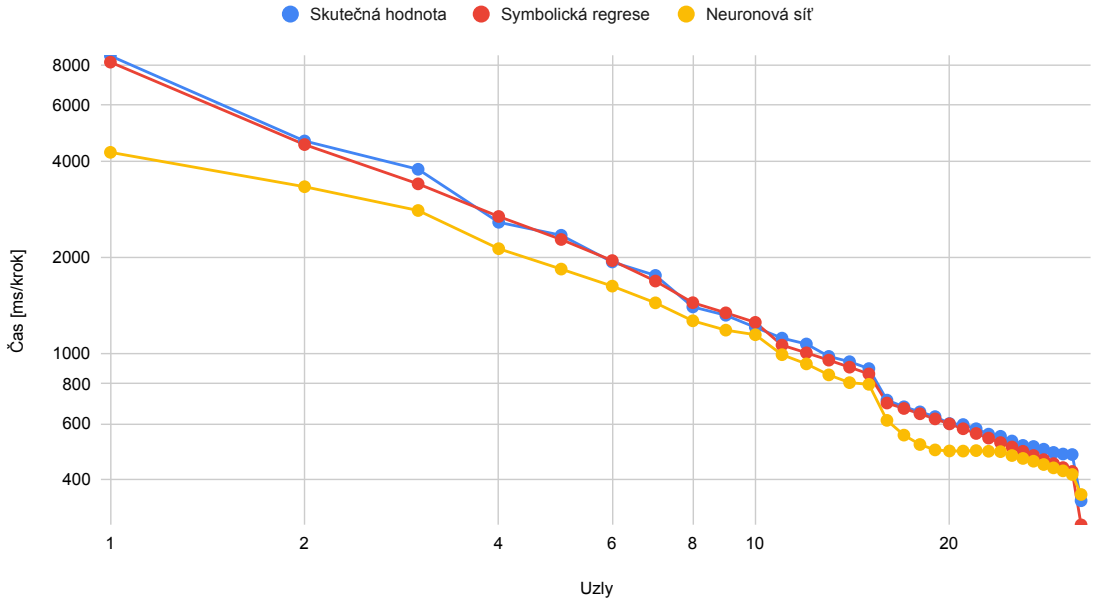
Obrázek 8.5: Graf reálného času na prostoru $384 \times 384 \times 1536$, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datove sadě *Pouze úlohy s vhodným příznakem factor*. Tato velikost prostoru se nachází **v testovací sadě**.



Obrázek 8.6: Graf reálného času na prostoru 576^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Pouze úlohy s vhodným příznakem factor*. Tato velikost prostoru se nachází **v testovací sadě**.



Obrázek 8.7: Graf reálného času na prostoru 896^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Pouze úlohy s vhodným příznakem factor*. Tato velikost prostoru se nachází **v testovací sadě**.



Obrázek 8.8: Graf reálného času na prostoru 1024^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Pouze úlohy s vhodným příznakem factor*. Tato velikost prostoru se nachází **v testovací sadě**.

8.3 Bez filtru

Tato sekce nabídne detailní pohled na predikce modelů na data vybraných rozměrů z datové sady bez použití filtrů dat.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
$256 \times 256 \times 2048$	19,23 %	22,48 %
$384 \times 384 \times 1024$	23,37 %	16,69 %
$384 \times 256 \times 1536$	8,11 %	9,74 %
648^3	16,14 %	7,22 %
650^3	10,46 %	12,70 %
756^3	11,99 %	7,67 %
800^3	5,96 %	7,94 %
832^3	7,33 %	8,73 %
1024^3	5,79 %	2,91 %
2048^3	4,24 %	3,96 %

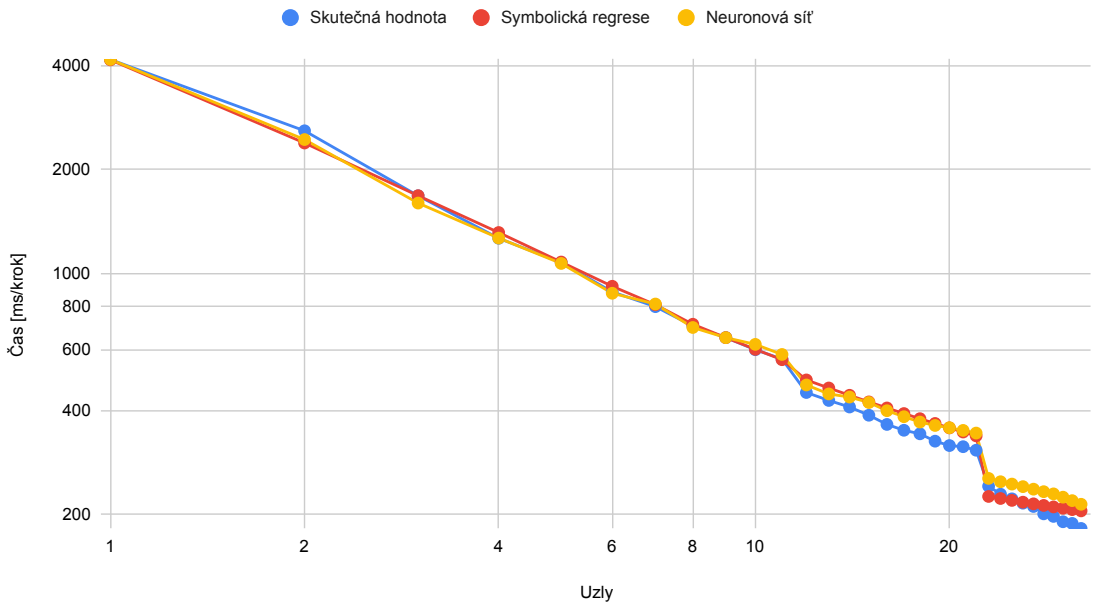
Tabulka 8.12: Relativní odchylka na vybraných **trénovacích datech** z datové sady *Bez filtru* podle velikosti simulačního prostoru. Trénovací sada má celkem 29 různé velikosti prostoru.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
$384 \times 512 \times 1536$	6,70 %	7,08 %
$512 \times 512 \times 1536$	11,91 %	7,21 %
$512 \times 512 \times 2048$	14,91 %	11,96 %
675^3	7,05 %	8,59 %
704^3	11,80 %	15,48 %

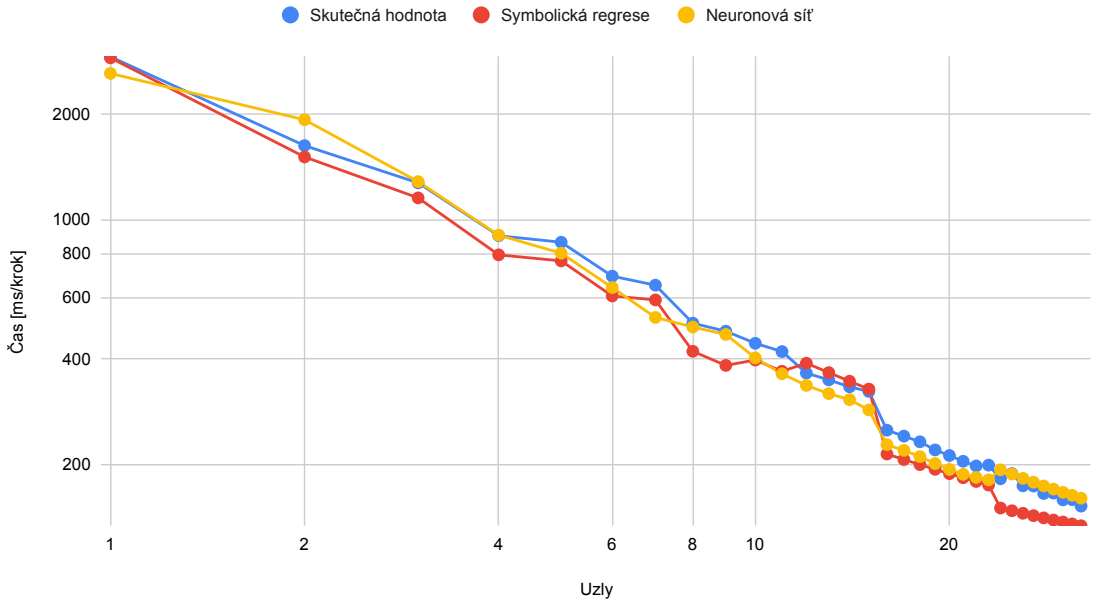
Tabulka 8.13: Relativní odchylka na datově sadě *Bez filtru* **na validační sadě** podle velikosti simulačního prostoru. Validační sada má celkem 5 různých velikostí prostoru.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
640^3	17,92 %	10,82 %
736^3	30,49 %	5,42 %
810^3	4,22 %	6,34 %
840^3	4,89 %	6,09 %
1536^3	8,45 %	8,49 %

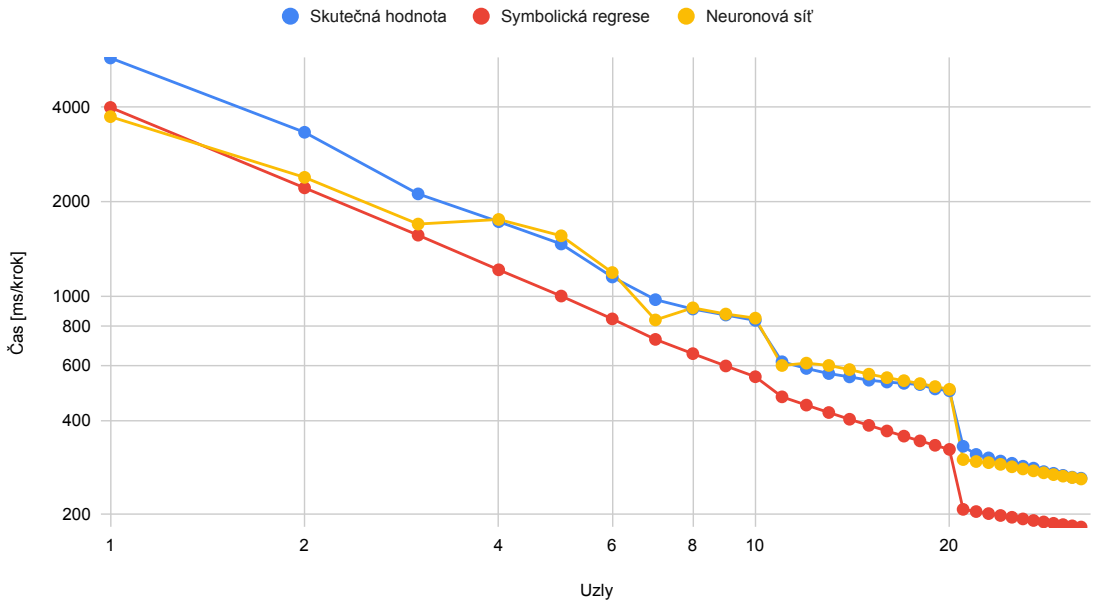
Tabulka 8.14: Relativní odchylka na datově sadě *Bez filtru* **na testovací sadě** podle velikosti simulačního prostoru. Testovací sada má celkem 5 různých velikostí prostoru.



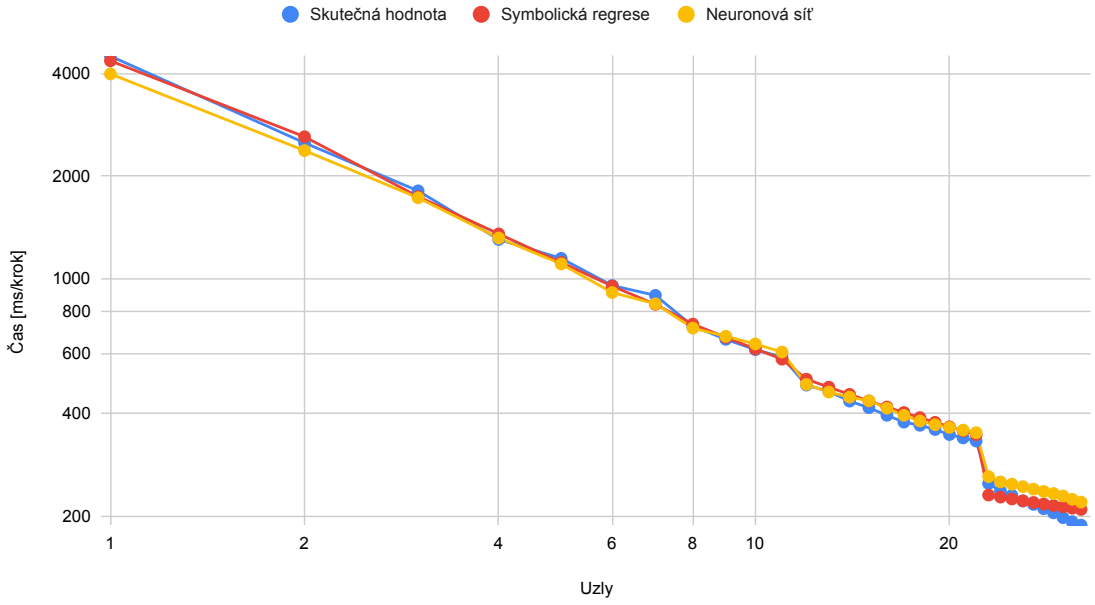
Obrázek 8.9: Graf reálného času na prostoru 800^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Bez filtru*. Tato velikost prostoru se nachází **v trénovací sadě**.



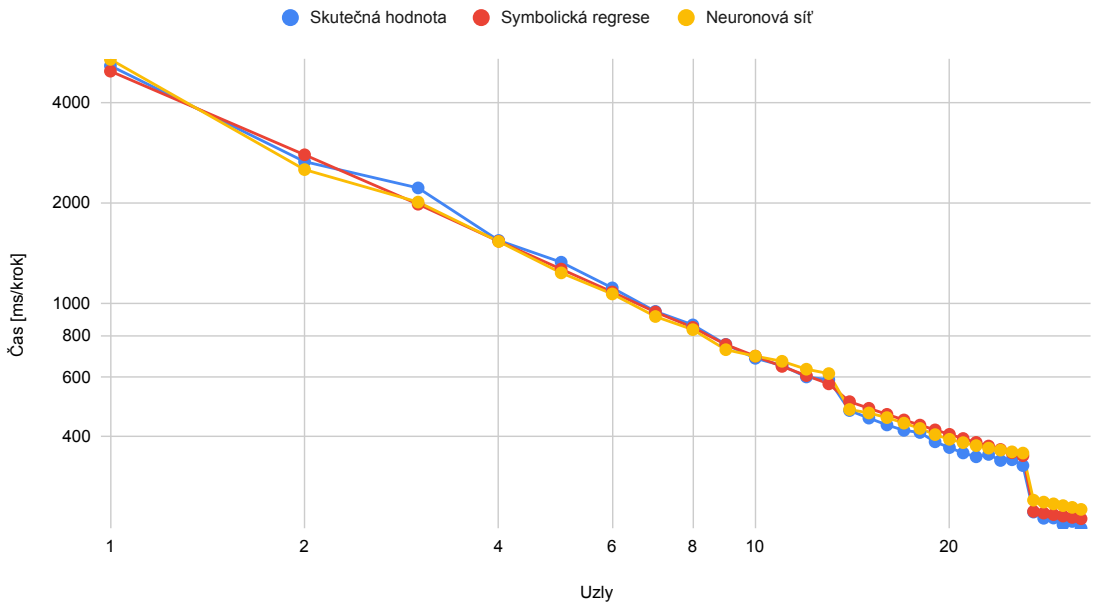
Obrázek 8.10: Graf reálného času na prostoru $512 \times 512 \times 1536$, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Bez filtru*. Tato velikost prostoru se nachází **ve validační sadě**.



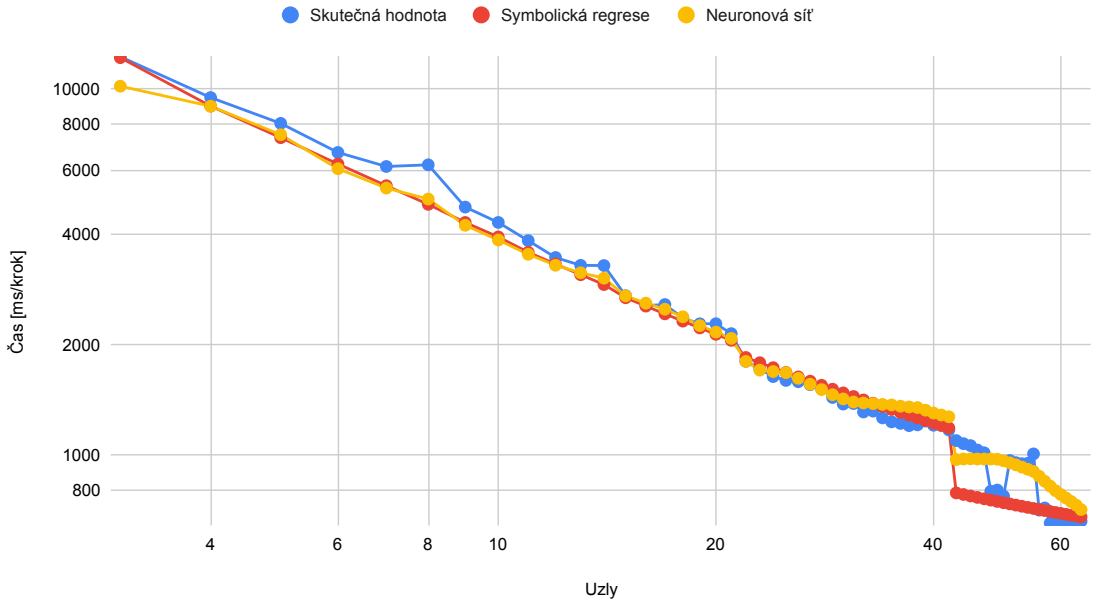
Obrázek 8.11: Graf reálného času na prostoru 736^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Bez filtru*. Tato velikost prostoru se nachází **v testovací sadě**. Tato úloha má **nevhodný rozklad na prvočísla.**



Obrázek 8.12: Graf reálného času na prostoru 810^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Bez filtru*. Tato velikost prostoru se nachází **v testovací sadě**.



Obrázek 8.13: Graf reálného času na prostoru 840^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Bez filtru*. Tato velikost prostoru se nachází **v testovací sadě**.



Obrázek 8.14: Graf reálného času na prostoru 1536^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Trénováno na datové sadě *Bez filtru*. Tato velikost prostoru se nachází **v testovací sadě**.

8.4 Karolina

Tato sekce, na rozdíl od výše uvedených sekcí, popisuje výsledky modelů natrénovaných pro data naměřená na superpočítači *karolina.it4i.cz*. Tato práce se obecně více zajímá o superpočítač *Barbora*, nicméně data z Karoliny mohou sloužit jako ukázka, že techniky popsané v této práci, mohou být použité i na jiných výpočetních zdrojích. Také tato sekce může sloužit jako základ pro další pokračování v tomto tématu.

Na Karolině bylo nasbíráno menší množství dat. Z toho důvodu jsem zde neprováděl pokusy s filtrováním dat, ale vytvořil jednu datovou sadu, která obsahuje všechna data s vhodným rozkladem domény na prvočísla. Trénovací sada obsahuje 6 velikostí domény, validační a testovací mají po 2 velikostech.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
576^3	25,77 %	11,91 %
768^3	11,64 %	7,70 %
840^3	13,69 %	8,22 %
960^3	12,00 %	4,71 %
1280^3	5,36 %	5,47 %
2048^3	4,41 %	3,57 %

Tabulka 8.15: Relativní odchylka na **trénovací sadě** podle velikosti simulačního prostoru. Data jsou ze superpočítače *Karolina*. Trénovací sada má celkem 6 různých velikostí prostoru.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
512^3	58,38 %	100,18 %
896^3	13,56 %	9,27 %

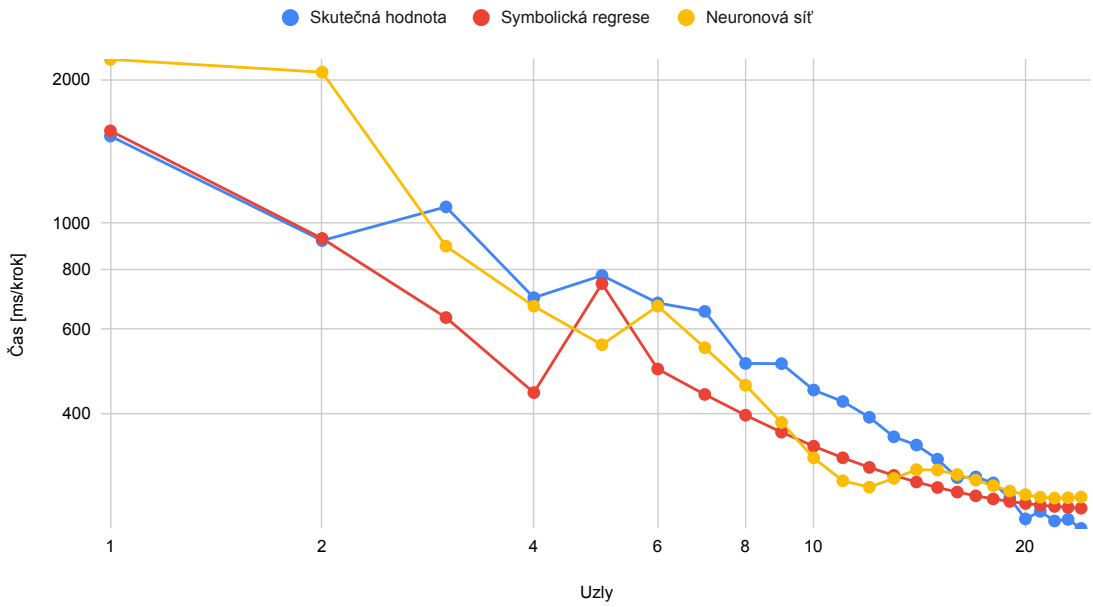
Tabulka 8.16: Relativní odchylka na **validační sadě** podle velikosti simulačního prostoru. Data jsou ze superpočítače *Karolina*. Trénovací sada má celkem 2 různé velikostí prostoru.

Rozměr úlohy	Odchylka – symbolická regrese	Odchylka – neuronová síť
640^3	15,40 %	19,19 %
1024^3	9,47 %	44,62 %

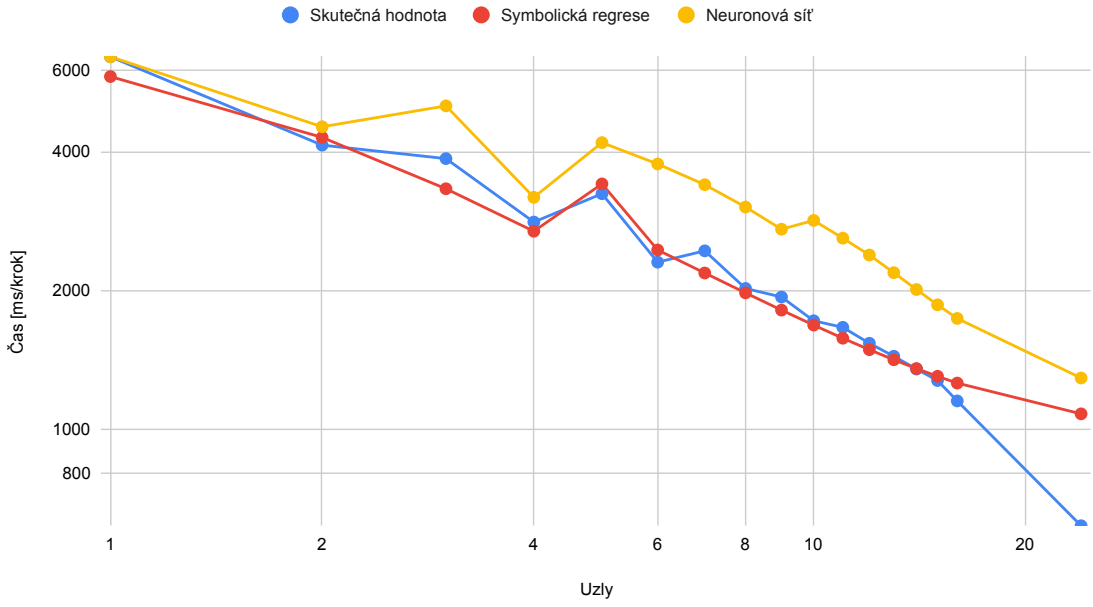
Tabulka 8.17: Relativní odchylka na **testovací sadě** podle velikosti simulačního prostoru. Data jsou ze superpočítače *Karolina*. Testovací sada má celkem 2 různé velikostí prostoru.

Tyto experimenty ukázaly, že řešení vyvinuté v této práci je použitelné i na ostatní superpočítače. Metoda symbolické regrese dosahuje i zde uspokojivých výsledků na neviděných datech (s výjimkou domény 512^3, která je menší, než nejmenší doména v trénovací sadě). Neuronové sítě zde mají znatelně horší přesnost a dochází k přetrénování. To vyplývá z velice omezené datové sady nasbírané na tomto superpočítači.

Na superpočítači Karolina funguje k-Wave obecně hůře kvůli nepoměru výkonu procesoru, pomalé paměti a pomalé propojovací síti. Proto i naměřená data nejsou tak predikovatelná, jako jsou na Barboře.



Obrázek 8.15: Graf reálného času na prostoru 640^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Data pocházejí ze superpočítače *Karolina*. Tato velikost prostoru se nachází **v testovací sadě**.



Obrázek 8.16: Graf reálného času na prostoru 1024^3, odhadu pomocí symbolické regrese a pomocí neuronové sítě. Data pocházejí ze superpočítače *Karolina*. Tato velikost prostoru se nachází **v testovací sadě**.

Kapitola 9

Závěr

Cílem práce bylo vytvořit metodu, pomocí které můžeme odhadnout čas ultrazvukové simulace k-Wave na superpočítači na základě zadané velikosti domény. Tento cíl byl splněn natrénováním neuronové sítě, která dokáže odhadnout čas běhu pro libovolnou doménu s průměrnou relativní odchylkou 8,25 %. Pokud je velikost dané domény optimalizovaná pro knihovnu FFTW, můžeme použít model symbolické regrese, který má průměrnou relativní odchylku 5,64 %.

Během řešení této práce jsem se seznámil technikami strojového učení v oblasti predikce a interpolace, které jsem poté popsal v tomto textu. Následně jsem se seznámil v nástrojem k-Wave a prostudoval jsem chování a škálování distribuované implementace v závislosti na typu a velikosti vstupních dat a množství přidělených výpočetních zdrojů. Vytvořil jsem datovou sadu obsahující 2 080 záznamů o času běhu simulace. Navrhl jsem metodu pro odhad doby běhu programu k-Wave pro danou velikost vstupních dat a množství přidělených výpočetních zdrojů. Tuto metodu jsem implementoval a otestoval jsem její přesnost. Zhodnotil jsem dosažené výsledky a diskutoval jsem přínos pro další směrování vývoje balíku k-Wave. Nad rámec zadání jsem experimentoval s různými typy modelů strojového učení a snažil se dosáhnout co nejlepších výsledků v oblasti predikce.

Kromě samotného trénování modelů jsem vytvořil nástroj pro správu naměřených dat, jejich export, vytvořil jsem prostředí Docker Compose pro snadnou použitelnost vytvořeného nástroje. Software HeuristicLab, který jsem použil pro trénování symbolické regrese, jsem obohatil o nový plugin, který do symbolické regrese přidává nové matematické funkce. Vytvořil jsem parser výsledného modelu v prefixové notaci tak, aby výsledný model byl jednoduše použitelný.

Pro další pokračování v této práci bych doporučil vyzkoušet další modely strojového učení, které jsou vhodné pro řešení regresního problému (např. random forest, support vector regrese a jiné). Za další prozkoumání stojí nastavení hyperparametrů neuronových sítí a genetického programování. Naopak bych neplýtval čas hledáním (vytvářením) dalších nástrojů pro symbolickou regresi (např. knihovny v jazyce Python).

Výsledky této práce překonávají existující řešení, které se spoléhá na manuální odhad nebo fixní odhad 24 hodin. Tyto výsledky také překonávají výsledky predikce pomocí splajnu a interpolace, které byly publikovány pro nástroj k-Wave na starším superpočítači IT4I (ty měly chybu na neviděných doménách až 15 %). Tato práce má potenciál mít reálné použití v toolboxu k-Wave nebo v nástroji k-Dispatch, pro který jsou spolehlivé odhady výpočetního času klíčové. Použití těchto odhadů může pomoct s optimalizováním nákladů spojených s placením za výpočetní čas na superpočítači. Metodu představenou v této práci

jde použít i na jiných superpočítačích s použitím jiné datové sady, nebo po upravení odhadu pro superpočítač Barbora (např. vynásobením vhodnou konstantou).

Práce mi dala příležitost, abych se podílel na něčem, co bude mít (věřím) dopad pro reálné použití. Také jsem si osvěžil teorii i praxi různých metod strojového učení, naučil jsem se pracovat s toolboxem k-Wave, připomenul jsem si Matlab, Python, C#, Docker a další nástroje... Prohloubil jsem své znalosti v oblasti superpočítačů. Seznámil jsem se s nástrojem HeuristicLab a osvěžil jsem si práci s knihovnou TensorFlow.

Literatura

[1] AL Ma'amari, M. *Deep neural networks for regression problems*. Towards Data Science, Oct 2018. Dostupné z: https://towardsdatascience.com/deep-neural-networks-for-regression-problems-81321897ca33.

[2] Amor, E. *Understanding non-linear activation functions in neural networks*. May 2020. Dostupné z: https://medium.com/ml-cheat-sheet/understanding-non-linear-activation-functions-in-neural-networks-152f5e101eeb.

[3] Bleuler, S., Brack, M., Thiele, L. a Zitzler, E. Multiobjective genetic programming: reducing bloat using SPEA2. In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*. 2001, sv. 1, s. 536–543 vol. 1. DOI: 10.1109/CEC.2001.934438.

[4] Bradley Treeby, B. C. a Jaros, J. *K-Wave User Manual* [http://www.k-wave.org/manual/k-wave_user_manual_1.1.pdf]. Srpen 2016. (Accessed on 12/12/2022).

[5] Chahar, V., Katoch, S. a Chauhan, S. A Review on Genetic Algorithm: Past, Present, and Future. *Multimedia Tools and Applications*. Únor 2021, sv. 80. DOI: 10.1007/s11042-020-10139-6.

[6] Cheon, H., Ryu, J., Ryou, J., Park, C. a Han, Y.-S. ARED: automata-based runtime estimation for distributed systems using deep learning. *Cluster Computing*. Duben 2021, s. 1–13. DOI: 10.1007/s10586-021-03272-w.

[7] Chu, P. C. a Beasley, J. E. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*. Springer. 1998, sv. 4, č. 1, s. 63–86.

[8] Dongarra, J. J., Luszczek, P. a Petitet, A. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*. 2003, sv. 15, č. 9, s. 803–820. DOI: https://doi.org/10.1002/cpe.728. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728.

[9] Fajmon, B., Hlavičková, I., Novák, M. a Vítovec, J. *Numerická matematika a pravděpodobnost*. Ústav matematiky FEKT VUT v Brně, 2014.

[10] Frigo, M. a Johnson, S. G. *FFTW for version 3.3.10*. Dec 2020. Dostupné z: https://www.fftw.org/fftw3.pdf.

[11] Hutter, F., Xu, L., Hoos, H. H. a Leyton Brown, K. Algorithm runtime prediction: Methods evaluation. *Artificial Intelligence*. 2014, sv. 206, s. 79–111. DOI: https://doi.org/10.1016/j.artint.2013.10.003. ISSN 0004-3702. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0004370213001082.

[12] Jansík, B. IT4Innovations národní superpočítačové centrum, Jan 2023. Dostupné z: https://docs.it4i.cz/.

[13] Jaros, J., Rendell, A. P. a Treeby, B. E. Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *The International Journal of High Performance Computing Applications*. 2016, sv. 30, č. 2, s. 137–155. DOI: 10.1177/1094342015581024. Dostupné z: https://doi.org/10.1177/1094342015581024.

[14] Jaros, M., Sasak, T., Treeby, B. E. a Jaros, J. Estimation of Execution Parameters for k-Wave Simulations. In: Kozubek, T., Arbenz, P., Jaroš, J., Říha, L., Šístek, J. et al., ed. *High Performance Computing in Science and Engineering*. Cham: Springer International Publishing, 2021, s. 116–134. ISBN 978-3-030-67077-1.

[15] Jaroš, M., Treeby, E. B., Jaroš, J. a Georgiou, P. K-Dispatch: A Workflow Management System for the Automated Execution of Biomedical Ultrasound Simulations on Remote Computing Resources. In: *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2020*. Association for Computing Machinery, 2020, s. 1–10. DOI: 10.1145/3394277.3401854. ISBN 978-1-4503-7993-9. Dostupné z: https://www.fit.vut.cz/research/publication/12125.

[16] Jebari, K. Selection Methods for Genetic Algorithms. *International Journal of Emerging Sciences*. Prosinec 2013, sv. 3, s. 333–344.

[17] Jette, M. a Auble, D. *Slurm Workload manager*. SchedMD LLC, Aug 2021. Dostupné z: https://slurm.schedmd.com/overview.html.

[18] Koza, J. R. Survey of genetic algorithms and genetic programming. In: Western Periodicals Company. *Wescon conference record*. 1995, s. 589–594.

[19] Li, B. a Lu, P. The Evolution of Supercomputer Architecture: A Historical Perspective. In: Xu, W., Xiao, L., Li, J. a Zhang, C., ed. *Computer Engineering and Technology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, s. 145–153. ISBN 978-3-662-49283-3.

[20] Navara, M. a Němeček, A. *Numerické metody*. Skriptum FEL ČVUT, 2005. ISBN ISBN 80-01-02689-2.

[21] Pfister, G. F. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*. 2001, sv. 42, 617-632, s. 102.

[22] Rajan, S. *Genetic algorithms and its use-cases in machine learning*. Jun 2021. Dostupné z: https://www.analyticsvidhya.com/blog/2021/06/genetic-algorithms-and-its-use-cases-in-machine-learning/.

[23] Scapa, J. R. *PBS Professional 2021.1 Big Book*. Altair Engineering, Inc., Mar 2021. Dostupné z: https://2021.help.altair.com/2021.1/PBSProfessional/PBS2021.1.pdf.

[24] Schnur, J. J. a Chawla, N. V. Information fusion via symbolic regression: A tutorial in the context of human health. *Information Fusion*. 2023, sv. 92, s. 326–335. DOI: https://doi.org/10.1016/j.inffus.2022.11.030. ISSN 1566-2535. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1566253522002470.

[25] Stokes, J. *SIMD architectures*. Mar 2000. Dostupné z: https://arstechnica.com/features/2000/03/simd/.

[26] Strohmaier, E., Dongarra, J., Simon, H. a Meuer, M. *November 2022*. TOP500.org, Nov 2022. Dostupné z: https://www.top500.org/lists/top500/2022/11/.

[27] Ter Haar, G. a Coussios, C. High intensity focused ultrasound: Physical principles and devices. *International journal of hyperthermia : the official journal of European Society for Hyperthermic Oncology, North American Hyperthermia Group*. Duben 2007, sv. 23, s. 89–104. DOI: 10.1080/02656730601186138.

[28] Topchy, A. a Punch, W. F. Faster Genetic Programming Based on Local Gradient Search of Numeric Leaf Values. In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, s. 155–162. GECCO'01. ISBN 1558607749.

[29] Treeby, B., Cox, B. a Jaroš, J. *Simulations In Three Dimensions Example*. Dostupné z: http://www.k-wave.org/documentation/example_ivp_3D_simulation.php.

[30] Wilstrup, C. a Kasak, J. Symbolic regression outperforms other models for small data sets. *CoRR*. 2021, abs/2103.15147. Dostupné z: https://arxiv.org/abs/2103.15147.

[31] Yoo, A. B., Jette, M. A. a Grondona, M. SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D., Rudolph, L. a Schwiegelshohn, U., ed. *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 44–60. ISBN 978-3-540-39727-4.

[32] Zhang, G., Eddy Patuwo, B. a Y. Hu, M. Forecasting with artificial neural networks:: The state of the art. *International Journal of Forecasting*. 1998, sv. 14, č. 1, s. 35–62. DOI: https://doi.org/10.1016/S0169-2070(97)00044-7. ISSN 0169-2070. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0169207097000447.

[33] Zhou, B. a Lu, L. An effective 3-D fast fourier transform framework for multi-GPU accelerated distributed-memory systems. *The Journal of Supercomputing*. Květen 2022, sv. 78, s. 1–19. DOI: 10.1007/s11227-022-04491-7.

Příloha A

Dokumentace

Tato příloha se snaží pomoct uživatelům s použitím nástroje vytvořeného v této práci a také se snaží pomoct při navazování na tuto práci.

A.1 Nástroj příkazového řádku

Tento nástroj umí exportovat naměřená data a provádět predikce. Nástroj se nachází ve složce `Sources/estimation`.

A.1.1 Spuštění prostředí Docker Compose

Před samotnou prací s tímto nástrojem je nutné spustit prostředí Docker Compose. To se spustí zavoláním scriptu z výpisu A.1. To stáhne potřebné knihovny, vytvoří databázi a naplní ji daty a spustí interaktivní bash.

```
./run.sh
```

Výpis A.1: Spuštění prostředí Docker Compose.

A.1.2 Vytváření predikcí

Ve spuštěném prostředí Docker Compose můžeme získat predikci doby běhu k-Wave pomocí volání A.2. To automaticky vybere vhodný model (podle zadaného názvu superpočítače a podle velikosti domény). Na superpočítač Karolina se používá model symbolické regrese. Na superpočítač Barbora se používá model symbolické regrese pro domény s vhodným rozkladem na prvočísla, jinak se používá model neuronové sítě. Následující volání provede odhad doby běhu k-Wave na superpočítači Barbora pro doménu 810^3 spuštěnou na 2 uzlech s použitím 72 ranků.

```
python app/app.py --predict barbora.it4i.cz --x 810 --y 810 --z 810 \
                  --nodes 2 --ranks 72
```

Výpis A.2: Získání predikce doby běhu k-Wave pomocí CLI nástroje.

Výsledný odhad doby běhu programu k-Wave se vypíše na standardní výstup zaokrouhlený na 2 desetinná místa, viz A.3. Výsledný čas je počet milisekund pro simulování jednoho kroku.

```
2566.96
```

Výpis A.3: Odhad se vypíše na standardní výstup jako desetinné číslo.

A.1.3 Export dat

Nástroj umožňuje exportovat data o zaznamenaných spuštění k-Wave z vybraného superpočítače, vypočítá příznaky popsané v kapitole 5.2, umožňuje filtrování dat pro vznik filtrovaných datových sad (viz 6.2.1), umí rozdělit domény do trénovací, validační a testovací sady a výsledek exportuje do CSV souborů.

```
python app/app.py --export barbora.it4i.cz --output nice-cubes \
                  --validation 15 --test 15 \
                  --factor --cube --distributed
```

Výpis A.4: Export dat ze superpočítače Barbora. Exportuje pouze krychle s vhodným počtem ranků a vhodným rozkladem na prvočísla. Domény jsou rozděleny na trénovací, validační a testovací sadu.

A.2 Trénování symbolické regrese

Protože trénování symbolické regrese je výpočetně náročné, je vhodné to spustit na superpočítači. HeuristicLab je software, který je implementován v jazyce C. Grafické prostředí je ovšem vytvořeno pomocí Windows Forms, které nejsou plně podporované mimo operační systém Windows. Ačkoliv HeuristicLab nabízí návod na zprovoznění na Linuxu[1], během řešení této práce se nepovedlo plně zprovoznit software HeuristicLab přímo na superpočítači Barbora ani Karolina. Z těchto důvodů jsem se rozhodl spustit virtuální stroj s operačním systémem Windows na superpočítači Barbora.

A.2.1 Spuštění virtuálního stroje na superpočítači

Superpočítač Barbora nabízí hardwarovou podporu pro virtualizaci pomocí nástroje QEMU.[2] Nejdříve je nutné stáhnout instalační soubor typu `.iso` s instalačním software pro operační systém Windows. Poté je potřeba zažádat o přístup na výpočetní uzel, protože nelze spustit virtualizaci na login uzlu. Na výpočetním uzlu se virtuální stroj spustí příkazem A.5. Poté je možné se k virtuálnímu stroji připojit pomocí VNC tunelu[3]. Tento stroj je pak snadno ovladatelný pomocí grafického rozhraní, které je přenášeno ze superpočítače na osobní počítač, ale s výraným navýšením výpočetního výkonu.

[1] https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/DevelopmentCenter/Compile%20HeuristicLab%20under%20Linux

[2] https://docs.it4i.cz/software/tools/virtualization/

[3] https://docs.it4i.cz/general/accessing-the-clusters/graphical-user-interface/vnc/

```
ml QEMU

qemu-system-x86_64 -hda win.img -enable-kvm -cpu host -smp 36 -m 32768 \
                   -vga std -localtime -usb -usbdevice tablet \
                   -cdrom win-install.iso -boot d -vnc :0
```

Výpis A.5: Spuštění virtuálního stroje s Windows na superpočítači Barbora.

A.2.2 Instalace HeuristicLab

Pro instalaci ze zdrojových kódů je nutné stáhnout zdrojové kódy HeuristicLab a poté pomocí Virtual Studio sestavit 2 řešení: `HeuristicLab.ExtLibs.sln` a `HeuristicLab 3.3.sln`. Pro detailní návod viz dokumentaci HeuristicLab[4].

A.2.3 Instalace pluginu

Plugin pro HeuristicLab vytvořený v této práci je možné sestavit také pomocí Virtual Studio ze souboru `ModuloRoundPlugin.sln` dostupného na médiu této práce. Sestavené `.dll` soubory je potřeba nahrát do `bin` složky ve složce HeuristicLab. Po správné instalaci se plugin `ModuloRoundPlugin` zobrazí v seznamu dostupných pluginů, viz obrázek 7.2. Plugin byl vytvořen pomocí návodu[5], který může případně napovědět více.

A.2.4 Trénování

Po spuštění HeursiticLab je potřeba zvolit možnost Optimizer. Dalším krokem je vytvoření algoritmu, kterým se bude trénovat. Klikneme na *File*, *New...* a zvolíme možnost *Genetic Algortihm (GA)*. Ve vytvořené záložce klikneme na tlačítko *New problem* a vybereme *Symbolic Regression Problem (single-objective)*. Dále klikneme na tlačítko *Import a Symbolic Regression Problem (single-objective) from a file in the CSV File format.*, vybereme soubor s trénovacími daty (viz A.1.3). Jako *Target Variable* vybereme *time* a poměr trénovacích dat nastavíme na 100 % a klikneme na tlačítko *Ok*.

Pro využití funkcí poskytovanými novým pluginem je nutné kliknout na tlačítko *Show hidden parameters* v záložce daného problému. Dále je nutné změnit typ gramatiky (*SymbolicExpressionTreeGrammar*) na *SymbolicExpressionTreeGrammar* z daného pluginu (tlačítko *Set Value* s ikonkou tužky) a vybrat požadované operace. Dále je třeba změnit interpret (*SymbolicExpressionTreeInterpreter*) na *SymbolicDataAnalysisExpressionTreeLinearInterpreter* ze stejného pluginu. V záložce problému jsem ještě měnil chybovou funkci (*Evaluator*) na hodnotu *Mean squared error Evaluator* a měnil jsem maximální velikost stromu na hodnotu 42 a maximální hloubku stromu na 16.

V záložce *Algorithm* jsem měnil maximální počet generací a pravděpodobnost mutace. Dále je dobré v záložce *Engine* přepnout na *Parallel Engine* a vše je připravené ke spuštění optimalizace pomocí tlačítka (*Start*).

Po skončení genetického algoritmu je výsledek trénování možné vidět v záložce *Results*. Po kliknutí na *SymbolicRegressionSolution* a poté na *SymbolicRegressionSolution* vidíme strom reprezentující výraz, kterým se data modelují. V pravém horním rohu můžeme po

[4] https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/DevelopmentCenter/DownloadAndBuildSource

[5] https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/DevelopmentCenter/CreateNewPluginUsingVS

kliknutím pravým tlačítkem na ikonku přepnout na textovou reprezentaci modelu, kterou jsem použil jako model a která je na připojeném médiu k této práci. Zároveň tento formát je zpracováván nástrojem příkazové řádky a použitý při dělání predikcí (viz A.1.2).