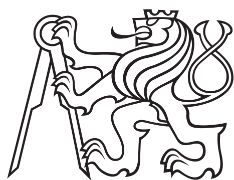
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Porovnání výkonu server-side JavaScript prostředí

Dejan Ribarovski

Školitel: RNDr. Ondřej Žára

Květen 2024

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Ribarovski** Jméno: **Dejan** Osobní číslo: **507603**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Porovnání výkonu server-side JS prostředí

Název bakalářské práce anglicky:

Server-side JS performance comparison

Pokyny pro vypracování:

Seznamte se s JS runtime Node.js, Deno a Bun. Nastudujte jejich společné rysy i hlavní rozdíly, sepište je a ověřte, jakými technologiemi jsou implementovány.
Navrhněte následně sadu výkonových testů (benchmarků) pro porovnání těchto prostředí. Testy by měly probíhat v těchto kontextech:
- zátěž CPU
- zpracování síťových požadavků (HTTP)
- disk I/O
- JavaScript vs. TypeScript, tj. čas strávený transpilací v rámci startu skriptu
- asynchronní vs. synchronní API (diskové IO)
- Windows vs. Linux (na identickém hardware)

Proveďte měření všech JS runtimů těmito testy. Vyzkoušejte jejich stabilní (LTS) verze. Výsledky popište jak slovně, tak pomocí tabulek a grafů.
Sepište doporučení, na základě kterého je možné zvolit využití konkretního server-side JS prostředí v závislosti na požadovaném výkonu.

Seznam doporučené literatury:

https://github.com/tsliwowicz/go-wrk
https://nodejs.org/docs/latest/api/
https://docs.deno.com/runtime/manual
https://bun.sh/docs

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Ondřej Žára Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **08.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

RNDr. Ondřej Žára podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání	Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Poděkování

Chtěl bych poděkovat mému vedoucímu práce, RNDr. Ondřeji Žárovi, za pomoc při zpracování této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 17.5.2024

Abstrakt

Tato práce se zabývá porovnáním výkonu tří vybraných JavaScript serverových prostředí - Node.js, Deno a Bun v několika různých oblastech - zpracování síťových požadavků, zátěž procesoru, synchronní a asynchronní diskové operace a transpilace TypeScriptu. Součástí práce je i porovnání běhu v rámci různých operačních systémů na stejném hardware. Na konci práce je sepsáno doporučení pro volbu ideálního serverového prostředí.

Klíčová slova: JavaScript, Node.js, Deno, Bun, web, server, výkonový test

Školitel: RNDr. Ondřej Žára

Abstract

The aim of this thesis is to compare and benchmark the performance of three server-side JavaScript runtimes - Node.js, Deno and Bun in different areas of measurement - processing network requests, CPU utilization, synchronous and asynchronous disc operations and TypeScript transpilation. Part of the work is also a comparison of the performance within different operating systems on the same hardware. At the end of the thesis, a recommendation for choosing an ideal server environment is written.

Keywords: JavaScript, Node.js, Deno, Bun, web, server, benchmark

Title translation: Performance testing of server-side JavaScript runtimes

Obsah

1 Úvod 1

2 Analýza 3

2.1 JavaScript 3
2.1.1 Charakteristika 3
2.2 Běhové prostředí 4
2.2.1 Jádro 4
2.2.2 Web APIs 5
2.2.3 Smyčka událostí 5
2.3 Serverové prostředí 6
2.3.1 Node.js 6
2.3.2 Deno 7
2.3.3 Bun 8
2.3.4 Srovnání 9

3 Implementace 11

3.1 Zpracování HTTP požadavků 12
3.1.1 Node.js server 12
3.1.2 Deno server 13
3.1.3 Bun server 13
3.1.4 Výkonové testy 14
3.1.5 Měření 15
3.2 Zátěž CPU 16
3.2.1 Rekurzivní výpočet Fibonacciho posloupnosti 16
3.2.2 Násobení matic 17
3.2.3 Výkonové testy 18
3.2.4 Měření 19
3.3 I/O operace 19
3.3.1 Node.js 20
3.3.2 Deno varianty 21
3.3.3 Bun varianty 22
3.3.4 Výkonové testy 23
3.3.5 Měřící program 23
3.3.6 Měření 24
3.4 Transpilace TypeScriptu 24
3.4.1 Hledání nejkratší cesty v grafu 24
3.4.2 Výkonové testy 27
3.4.3 Měřící program 27
3.4.4 Měření 28

4 Výsledky 29

4.1 Prostředí 29
4.2 Zpracování HTTP požadavků 30
4.2.1 Go-wrk 30
4.2.2 HTTPBenchmarkTool 32
4.2.3 Srovnání měřících nástrojů 35
4.3 Zátěž CPU 38
4.3.1 Výpočet Fibonacciho posloupnosti 38
4.3.2 Násobení Matic 39
4.4 I/O operace 40
4.4.1 zápis 10 souborů o velikosti 1GB 40
4.4.2 zápis 100 souborů o velikosti 100MB 41
4.4.3 zápis 1000 souborů o velikosti 10MB 42
4.4.4 Porovnání synchronních a asynchronních variant 43
4.5 Transpilace TypeScriptu 44
4.5.1 Hledání cesty v malém grafu 44
4.5.2 Hledání cesty ve velkém grafu 44
4.6 Windows vs. Linux 46
4.6.1 Výpočet Fibonacciho posloupnosti 46
4.6.2 Násobení Matic 46
4.7 Doporučení 48

5 Závěr 49

Literatura 51

Obrázky

2.1 Způsob vykonání JavaScriptu v prohlížeči 4
2.2 Běhové prostředí Node.js 7
2.3 Běhové prostředí Deno 8
2.4 Běhové porstředí Bun 9

3.1 Fáze jednoho požadavku 14

4.1 Počet zpracovaných požadavků za vteřinu v závislosti na počtu klientů 31
4.2 Průměrný čas odezvy v závislosti na počtu klientů 32
4.3 Marginální histogram dat z měření prostředí Node.js 33
4.4 Marginální histogram dat z měření prostředí Deno 34
4.5 Marginální histogram dat z měření prostředí Bun 34
4.6 Celkový histogram dat ze všech tří dílčích měření 35
4.7 Doby výpočtu 43. prvku Fibonacciho posloupnosti 38
4.8 Počet iterací násobení 10x10 matic po dobu 1s 39
4.9 Doba zápisu 10 souborů o velikosti 1 GB 40
4.10 Doba zápisu 100 souborů o velikosti 100 MB 41
4.11 Doba zápisu 1000 souborů o velikosti 10 MB 42
4.12 Relativní zpomalení asynchronního zápisu vzhledem k synchronnímu zápisu pro dané prostředí a test 43
4.13 Doby hledaní nejkratší cesty v malém grafu 45
4.14 Doby hledaní nejkratší cesty ve velkém grafu 45
4.15 Doby výpočtu 43. prvku Fibonacciho posloupnosti na OS Windows a Ubuntu 47
4.16 Počet iterací násobení 10x10 matic po dobu 1s na OS Windows a Ubuntu (zaokrouhleno na tisíce) 47

Tabulky

2.1 Porovnání běhových prostředí Node.js, Deno a Bun 9

4.1 výkonový test s 10 klienty po dobu 10s 30
4.2 výkonový test s 50 klienty po dobu 10s 30
4.3 výkonový test se 100 klienty po dobu 10s 30
4.4 výkonový test pomocí balíku HTTPBenchmarkTool 33
4.5 srovnání průměrného času odezvy v milisekundách pro daný nástroj a prostředí v závislosti na počtu klientů 36
4.6 srovnání počtu vyřízených požadavků za vteřinu pro daný nástroj a prostředí v závislosti na počtu klientů 36
4.7 srovnání reálné míry paralelizace pro daný nástroj a prostředí v závislosti na počtu klientů 36
4.8 Doby výpočtu 43. prvku Fibonacciho posloupnosti 38
4.9 Počet iterací násobení 10x10 matic po dobu 1s 39
4.10 Doba zápisu 10 souborů o velikosti 1 GB 40
4.11 Doba zápisu 100 souborů o velikosti 100 MB 41
4.12 Doba zápisu 1000 souborů o velikosti 10 MB 42
4.13 Doba hledání nejkratší cesty v malém grafu 44
4.14 Doba hledání nejkratší cesty ve velkém grafu 44
4.15 Doby výpočtu 43. prvku Fibonacciho posloupnosti na OS Windows a Ubuntu 46
4.16 Počet iterací násobení 10x10 matic po dobu 1s na OS Windows a Ubuntu 46
4.17 Celkové srovnání výkonu testovaných běhových prostředí 48

Úvod

Programovací jazyk JavaScript se od svého vzniku vyvinul v daleko mocnější nástroj, než se původně očekávalo. Dnes jej lze nalézt na mnoha místech i mimo klientské webové prostředí, pro které byl původně vytvořen. S rychlým vývojem webu v novém tisíciletí rostla i popularita jazyka a objevily se nové možnosti pro jeho využití, mimo jiné i možnost využití na serveru. Nejznámější příklad JavaScript serverového prostředí je Node.js představený v roce 2009 Ryanem Dahlem. Další rozšířená varianta je prostředí Deno, které bylo uvedeno v roce 2018 taktéž Ryanem Dahlem. Nejnovější populární JavaScript serverové prostředí je Bun, jehož první stabilní verze byla představena v září 2023. Autorem je Jarred Sumner.

Cílem této práce je porovnat výše zmíněná běhová prostředí, popsat jejich společné rysy i rozdíly, navrhnout a provést několik výkonových testů v různých kontextech – zpracování síťových požadavků, zátěž CPU, diskové I/O operace asynchronně i synchronně, transpilace TypeScriptu a porovnání zátěže CPU v závislosti na operačním systému (Windows vs. Linux). Posléze analyzovat naměřená data společně s jejich vizualizací a sepsat doporučení pro volbu optimálního běhového prostředí na základě požadovaných vlastností.

Kapitola 2
Analýza

2.1 JavaScript

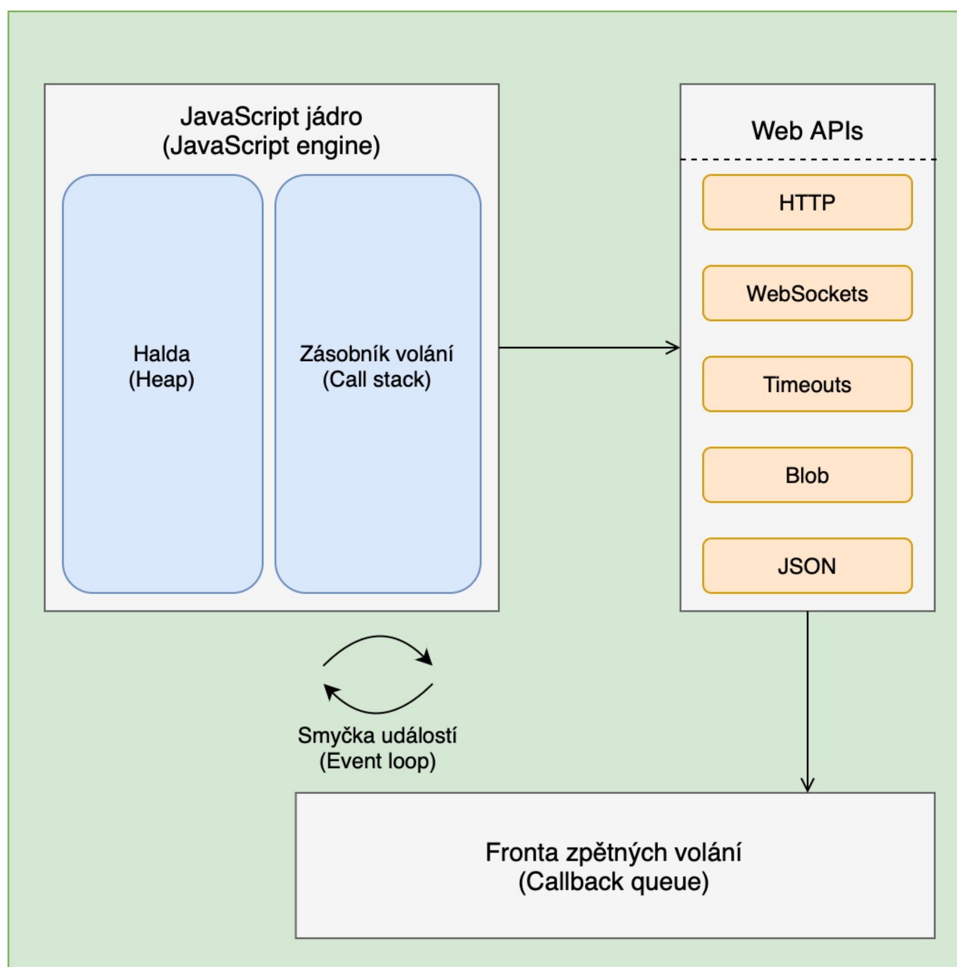
Ačkoli většina lidí používá název JavaScript, formálně se jedná o standardizovaný ECMAScript. Společnost Netscape vyvinula skriptovací jazyk pro jejich vlastní webový prohlížeč již v roce 1995. Název tohoto jazyka se několikrát změnil – nejdříve Mocha, poté LiveScript a nakonec JavaScript. [1]

Za autora je označován Brandan Eich, na jeho vývoji se ale podílelo více osob. ECMA, Evropská asociace výrobců počítačů, standardizovala podobu skriptovacího jazyka pro webové stránky na žádost společnosti Netscape v roce 1997. Důvodem podání žádosti o standardizaci byly obavy z nekompatibility jazyka a možné neochoty Microsoftu spolupracovat na vývoji jednotného skriptovacího jazyka pro web. [2] Microsoft v roce 1996 vydal první verzi skriptovacího jazyka pro vlastní prohlížeč Internet Explorer nazvaný JScript.

S rostoucí popularitou internetu na začátku 21. století se i jazyk JavaScript stával stále více populární. Brzy se tak rozšířil i za hranice prohlížečů. Nejdříve na serverovou část webových aplikací, později bylo možné v tomto jazyce psát i desktopové aplikace. Příkladem JavaScriptové desktopové aplikace je Adobe Acrobat. Dnes je již možné JavaScript použít i v oblastech velice vzdálených od klientského webu, například v něm lze programovat microcontrollery. [3] Díky této universalitě se jazyk JavaScript stal nejpopulárnějším programovacím jazykem současnosti. [4]

2.1.1 Charakteristika

JavaScript je vysokoúrovňový slabě typovaný interpretovaný programovací jazyk, což znamená, že k jeho vykonání je třeba interpretor, který daný kód řádku po řádce překládá do strojového kódu, který je posléze vykonán procesorem počítače. Původní interpretor JavaScriptu, který napsal již zmiňovaný Brendan Eich spolu s původní implementací JavaScriptu, se jmenuje SpiderMonkey a je dodnes používán, samozřejmě v upravené formě, prohlížečem Mozilla Firefox. [5]



Obrázek 2.1: Způsob vykonání JavaScriptu v prohlížeči

Interpretace JavaScriptu byla v porovnání s kompilovanými jazyky jako například C/C++ velmi pomalá, ale na druhou stranu nabízela flexibilitu ve způsobu vykonávání kódu. Tato vlastnost byla klíčová pro webové stránky, které jsou ze své podstaty řízeny událostmi, neboli event-driven. [6][7]

2.2 Běhové prostředí

Způsob vykonávání JavaScriptu se liší v závislosti na prostředí, ve kterém běží. Na obrázku 2.1 je znázorněno běhové prostředí JavaScriptu v prohlížeči.

2.2.1 Jádro

V jádru běhového prostředí se odehrává exekuce kódu. Obsahuje haldu a zásobník volání. Halda je prostor paměti, ve kterém jsou uloženy hodnoty proměnných. Zásobník volání je tzv. LIFO datová struktura, neboli Last-In-First-Out, což znamená, že volání funkce, které je na zásobník vloženo jako

poslední, bude ze zásobníku odebráno jako první.

Jádro JavaScriptu má pouze jedno vlákno, proto je jazyk jako takový často nazýván jednovláknový(single threaded). Paralelní vykonávání kódu umožňuje volání Web APIs spolu se smyčkou událostí. Pokud se na zásobníku volání nachází volání některého z Web APIs, jádro odebere dané volání ze zásobníku, zavolá odkazované API, které je vykonáváno v jiném vlákně a pokračuje ve vykonávání dalších volání ze zásobníku. Tato Funkcionalita umožňuje asynchronní běh vykonávaného programu. Příklady jádra JavaScriptu jsou například V8(Google Chrome, Microsoft Edge) nebo JavaScriptCore(Safari). [8]

2.2.2 Web APIs

Web APIs je set několika rozhraní specifických pro webové prostředí, konkrétně jeho klientskou část. Tato rozhraní jsou standardizována organizací the World Wide Web Consortium (W3C). Jejich součástí jsou například rozhraní DOM, Historie, Časovače nebo WebSockets. Funkcionality, které jsou obsažené v těchto rozhraních jsou pro funkcionalitu prohlížeče nezbytné, avšak nejsou přímou součástí JavaScriptu samotného. Volání těchto standardizovaných funkcí má na starosti běhové prostředí. Po zpracování požadavku převzatého od jádra vloží připravené volání do fronty zpětných volání. Tato datová struktura je na rozdíl od zásobníku volání FIFO – First-In-First-Out. Volání, které bylo do fronty vloženo jako první, bude odebráno také jako první. [8]

2.2.3 Smyčka událostí

Poslední část běhového prostředí, smyčka událostí, neboli event-loop, je jeho klíčovou součástí. Běží cyklicky po celou dobu vykonávání programu a při každé iteraci zkontroluje, zda není zásobník volání prázdný. Pokud ano a fronta zpětných volání není prázdná, tak první volání z fronty přesune do zásobníku volání, kde se začne vykonávat. [9]

Musí být ovšem splněna podmínka, že zásobník volání je prázdný. Vykonávání volání ze zásobníku má totiž přednost před vykonáváním zpětných volání. Tuto vlastnost lze lehce ověřit, a to přímo v jakémkoli prohlížeči, který obsahuje JavaScript konsoli zavoláním jednoduché funkce 2.1

```
> function test_event_loop(){
    console.log(1)
    setTimeout(() => {console.log(2)}, 0)
    console.log(3)
  }
<
> test_event_loop()
< 1
< 3
< 2
```

Listing 2.1: Funkce k ověření funkcionality event loopu

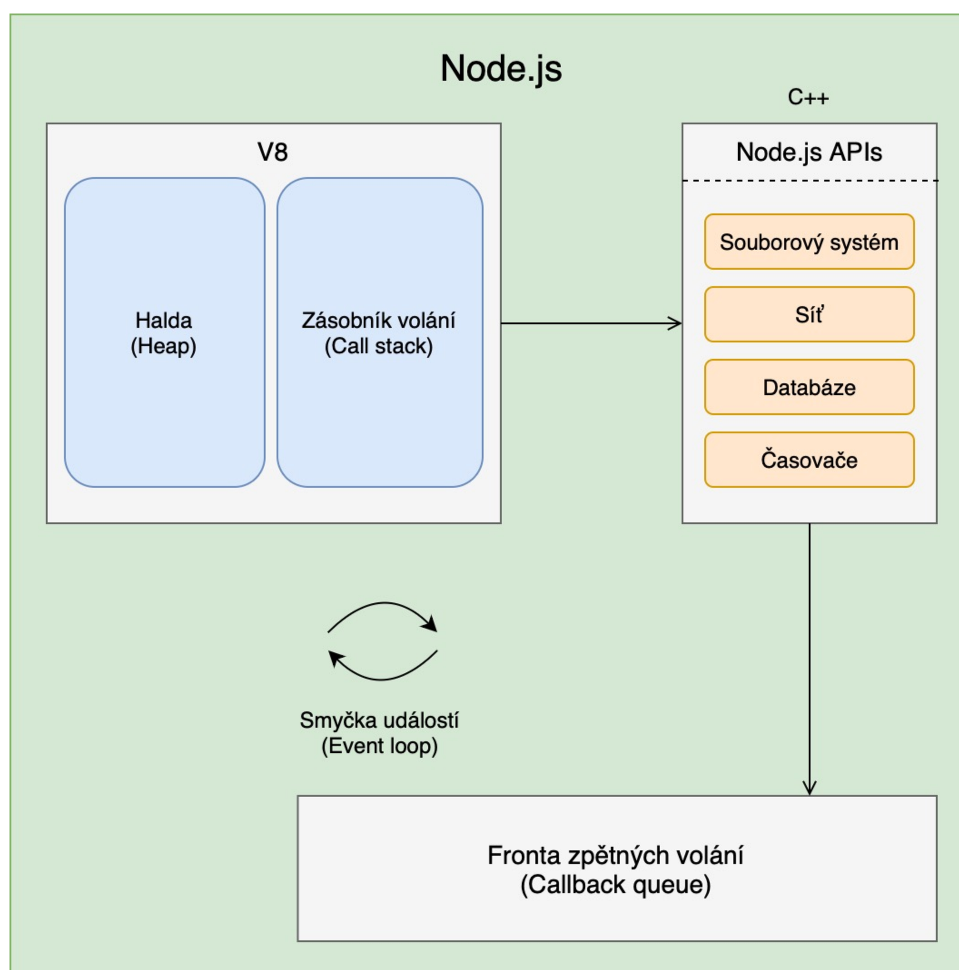
Funkce nejdříve vypíše do konzole `"1"`, poté zavolá funkci `setTimeout()`, jejímž druhým argumentem je čas vyjádřen v milisekundách, po kterém se má vykonat funkce, která je předána jakožto argument první. V tomto konkrétním případě se jedná o anonymní funkci, která pouze zavolá `console.log(2)`. Druhý argument je 0, což značí, že by se mělo vypsání `"2"` provést po 0 milisekundách, čili ihned. Ovšem není tomu tak. Důvodem, proč se nejdříve vypíše `"3"` a až poté `"2"`, je přesměrování volání funkce `setTimeout()` mimo jádro běhového prostředí. Funkce `setTimeout()` je součástí Web APIs, konkrétně rozhraní časovačů. Při zpracování tohoto volání je sice ihned vloženo volání `console.log(2)` do fronty zpětných volání, avšak na zásobníku se již nachází volání `console.log(3)` a volání na zásobníku mají přednost před frontou zpětných volání. S jistotou tedy můžeme říci, že zpětné volání funkcí pomocí funkce `setTimeout()` nebo jiných funkcí ze setu Web APIs se provede nejdříve po požadované době, nikoli přesně v požadovanou dobu.

2.3 Serverové prostředí

2.3.1 Node.js

Node.js byl vytvořen v roce 2009 Ryanem Dahlem. Jedná se o nejpopulárnější JavaScript serverové prostředí [10]. Na obrázku 2.2 je znázorněna jeho vnitřní struktura. Jádro běhového prostředí Node.js je V8 od firmy Google. Stejné jádro využívají i prohlížeče Google Chrome a Microsoft Edge. Jedná se o open source projekt, psaný v jazyce C++. Podobně jako v prohlížečích i Node.js má smyčku událostí a frontu zpětných volání. Na rozdíl od prohlížečů ale nedisponuje implementací Web APIs, jelikož se Ryanu Dahlovi zprvu zdálo, že v serverovém světě nejsou třeba. Nutno podotknout, že v roce 2009 ještě v JavaScriptu neexistovaly některé standarty jako například `Promise` nebo `async/await`. Implementoval tedy vlastní Node.js APIs, které umožňují například přístup k síti, souborovému systému či databázi. Implementace těchto rozhraní je napsaná v C++.

Součástí Node.js je i správce balíčků npm. Jedná se o možnost, jak použít cizí kód z veřejného repositáře. K použití těchto balíčků je třeba i konfiguračních souborů – *Package.json* a *node_modules*. [11]

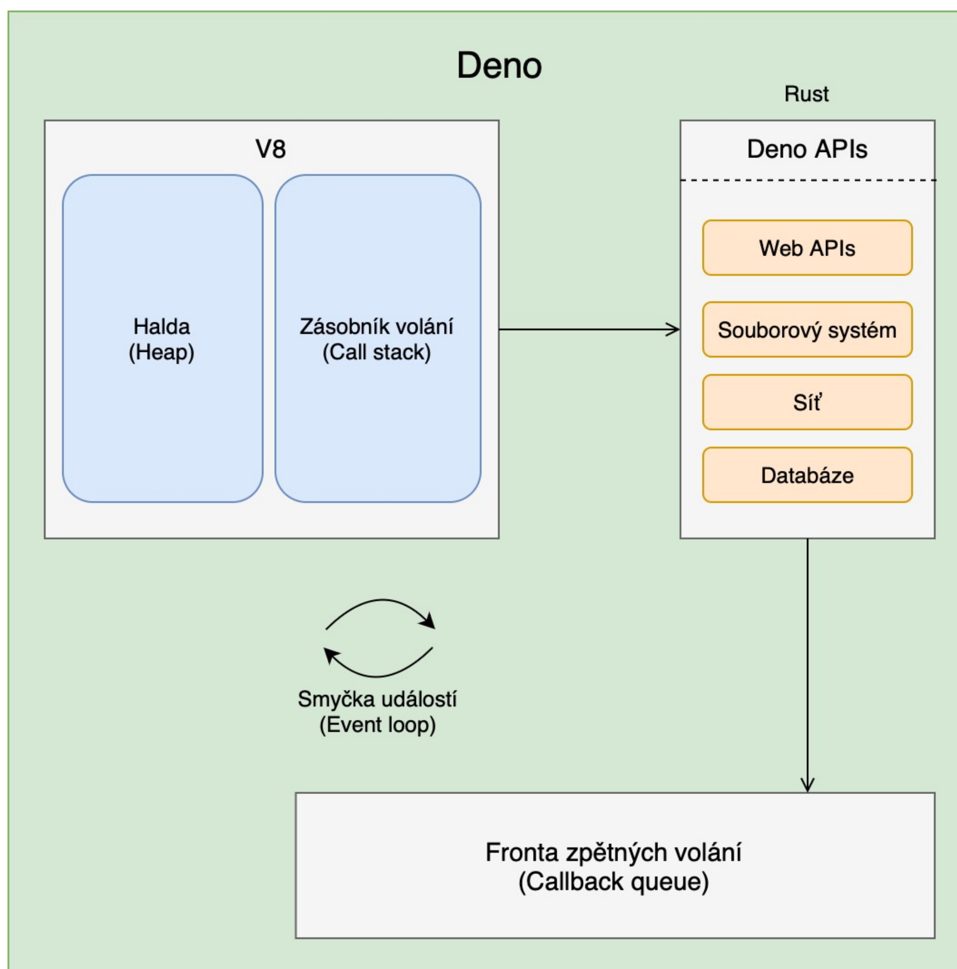


Obrázek 2.2: Běhové prostředí Node.js

2.3.2 Deno

Serverové prostředí Deno bylo vytvořeno v roce 2018 Ryanem Dahlem, opět. Bylo představeno na konferenci o JavaScriptu v rámci jeho přednášky - *"10 věcí ohledně Node.js, kterých lituji"*. [12] Na obrázku 2.3 je znázorněna jeho vnitřní struktura. Některé rysy mají Node.js i Deno společné - oba používají jádro V8 a podobně jako prohlížeče mají smyčku událostí i frontu zpětných volání. Na rozdíl od Node.js, součástí Dena je i vestavěná podpora Web APIs, kromě některých, které v serverovém světě nedávají smysl - například DOM nebo History.

Běhové prostředí Deno bylo implementováno v jazyce Rust. Při spouštění programu musí uživatel z důvodu bezpečnosti explicitně udělit přístupová práva. Další z rozdílů oproti prostředí Node.js je způsob správy balíčků. V prostředí Deno se balíčky třetích stran importují přímo ve zdrojovém kódu pomocí URL na odkazovaný repositář. Není tedy třeba nástroje pro správu jako je npm, ani konfiguračních souborů. [13]



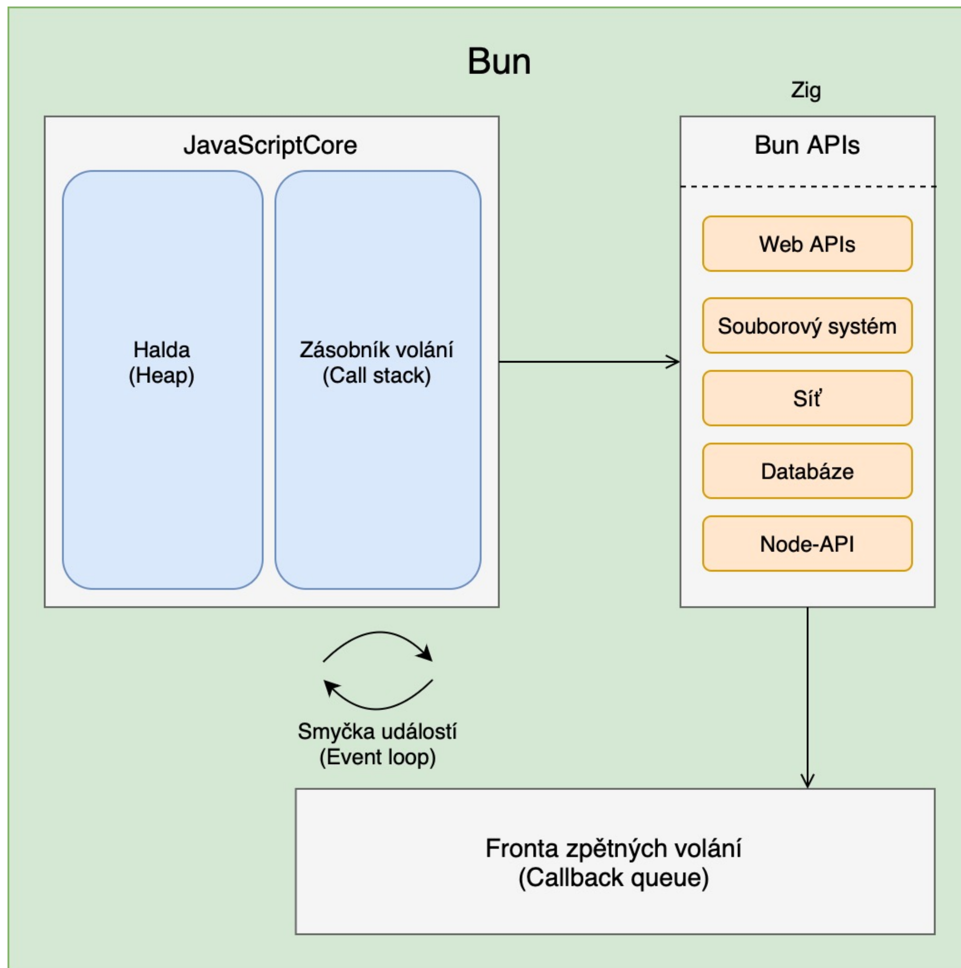
Obrázek 2.3: Běhové prostředí Deno

Součástí Dena je i nativní podpora pro TypeScript, neboli silně typovaný programovací jazyk kompilovaný do JavaScriptu, který vznikl v roce 2012. Node.js je starší, podpora pro TypeScript není jeho nativní součástí, ale díky kompilaci do JavaScriptu je podporován. [14]

2.3.3 Bun

Nejnovější z uvedených JavaScript serverových prostředí je Bun. První stabilní verze byla vydána v září 2023 a jeho autorem již není Ryan Dahl, jako v předchozích dvou případech, ale Jarred Sumner. Jeho struktura je znázorněna na obrázku 2.4.

Zásadní změna oproti prostředím Node.js a Deno je použití jádra JavaScript-Core od firmy Apple. Bun APIs jsou implementovány převážně v jazyce Zig a jejich součástí je podobně jako v Denu nativní podpora většiny Web APIs. Bun navíc klade důraz na zpětnou kompatibilitu se stále velmi populárním prostředím Node.js, proto je jeho součástí i sada Node APIs, avšak zatím



Obrázek 2.4: Běhové porstředí Bun

neúplná. [15]
Bun není pouze běhové prostředí, ale také správce balíčků, podobně jako npm pro Node.js. Podpora pro použití balíčků z repositářů npm je v prostředí Bun vestavěná, podobně jako podpora jazyka TypeScript. [15]

2.3.4 Srovnání

Tabulka 2.1: Porovnání běhových prostředí Node.js, Deno a Bun

Serverové prostředí	Jádro	Hlavní jazyk	Správa a instalace závislostí	Podpora TypeScriptu
Node.js	V8	C++	pomocí npm	ano, po instalaci
Deno	V8	Rust	pomocí URL	nativní
Bun	JavaScriptCore	Zig	pomocí bun install	nativní

Kapitola 3
Implementace

Tato kapitola se zaměřuje na implementaci měřených programů v každém ze tří porovnávaných JavaScript serverových prostředí pro všechny kontexty měření:

- zpracování HTTP požadavků
- zátěž CPU
- diskové I/O operace synchronně i asynchronně
- transpilace TypeScriptu
- zátěž CPU na OS Windows vs. OS Linux na stejném hardware

Pro každý kontext měření jsou uvedeny konkrétní výkonové testy společně se zdůvodněním vhodnosti jejich použití.

3.1 Zpracování HTTP požadavků

3.1.1 Node.js server

Implementace Jednoduchého HTTP serveru v prostředí Node.js je uvedena v kódu 3.1
Komentáře u každého řádku kódu popisují jeho funkcionalitu.

```
1  //promenna http reprezentuje rozhrani node:http z Node APIs
2  const http = require('http');
3
4  //specifikace portu, na kterem bude server vystaven
5  const PORT = 3000;
6
7  //vytvoreni serveru zavolanim funkce z rozhrani node:http.
8  //server na vsechny pozadavky odpovi zpravou "Hello world\n"
9  //se statusem 200(OK)
10 const server = http.createServer((req, res) => {
11   res.end('Hello, World!\n');
12 });
13
14 //spusteni serveru na danem portu a vypsani informativni
15 //hlasky do konsole
16 server.listen(PORT, () => {
17   console.log(`Server is listening on port ${PORT}`);
18 });
```

Listing 3.1: HTTP server v Node.js

Spuštění tohoto programu uloženého v souboru *nodeServer.js* se provádí příkazem 3.2

```
1 node nodeServer.js
```

Listing 3.2: Příkaz ke spuštění serveru

3.1.2 Deno server

Implementace Jednoduchého HTTP serveru v prostředí Deno je uvedena v kódu 3.3. V prostředí Deno je více možností, jak implementovat HTTP server, zvolil jsem preferovanou variantu dle oficiální dokumentace. [16]

```
//specifikace portu, na kterem bude server vystaven
const PORT = 3000;

//specifikace zpravy, kterou server odpovi na pozadavky
const response  = "Hello, world\n";

//spusteni serveru na danem portu pomoci funkce Deno.serve()
//Druhy parametr je anonymni funkce, ktera na kazdy
//pozadavek odpovi zpravou response a statusem 200(OK)
Deno.serve({port:PORT}, req => {return new Response(response)});
```

Listing 3.3: HTTP server v prostředí Deno

Spuštění tohoto programu uloženého v souboru *denoServer.js* se provádí následujícím příkazem 3.4.

```
deno run --allow-net denoServer.js
```

Listing 3.4: Příkaz ke spuštění serveru

Část příkazu `--allow-net` uděluje přístup k síťovým ovladačům počítače, na kterém program běží. Jedná se o jeden ze zmiňovaných bezpečnostních prvků serverového prostředí Deno.

3.1.3 Bun server

Implementace Jednoduchého HTTP serveru v prostředí Bun 3.5

```
//specifikace portu, na kterem bude server vystaven
const PORT = 3000;

//specifikace zpravy, kterou server odpovi na pozadavky
const response  = "Hello, world\n";

//spusteni serveru na danem portu pomoci funkce Bun.serve()
//Druhy parametr je anonymni funkce, ktera na kazdy
//pozadavek odpovi zpravou response a statusem 200(OK)
const server = Bun.serve({
    port: PORT,
    fetch(request) {
        return new Response(response);
    },
});
```

Listing 3.5: HTTP server v prostředí Bun

Spuštění tohoto programu uloženého v souboru *bunServer.js* se provádí následujícím příkazem 3.6.

```
bun bunServer.js
```

Listing 3.6: Příkaz ke spuštění serveru

3.1.4 Výkonové testy

Měřené vlastnosti

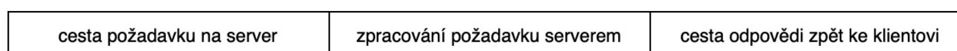
Měřené vlastnosti serverů jsou primárně dvě veličiny, a to:

- Počet zpracovaných síťových požadavků serverem za jednotku času (čím vyšší, tím lepší)
- Průměrný čas odezvy serveru (čím nižší, tím lepší)

Tyto vlastnosti jsou měřeny při různé míře paralelizace, jinými slovy při různých počtech současně připojených klientů.

Volba sítě

Při výběru výkonových testů je třeba dbát zejména na vhodnost použití daného testu v konkrétních podmínkách měření. Na obrázku 3.1 je znázorněn průběh jednoho požadavku.



Obrázek 3.1: Fáze jednoho požadavku

Jeden požadavek zahrnuje tři fáze – nejdříve se musí fyzicky dostat od klienta na server, poté následuje fáze zpracování požadavku serverem a poslední fází požadavku je cesta od serveru zpět ke klientovi. Součet časů strávených v první a třetí fázi požadavku se nazývá latence sítě. [17] V ideálním prostředí pro měření výkonu serveru by latence byla nulová. Toho však z fyzikální podstaty věci není možné docílit. Jedním kritériem vhodného výkonového testu je tedy minimalizace latence sítě u každého z požadavků.

Latenci sítě lze změřit například v příkazové řádce pomocí nástroje `ping`. V mé lokální síti se latence mezi dvěma počítači pohybuje obvykle v rozmezí 100-300 ms a latence localhostu, tedy mezi různými porty stejného počítače, se pohybuje v rozmezí 10-50 μs. Vzhledem k tomuto markantnímu rozdílu jsem se rozhodl pro variantu lokálního testování, kdy běží server i program reprezentující klienty na jednom zařízení.

Volba měřících programů

Měřící program má za úkol simulovat klienty odesílající požadavky na server a zároveň měřit čas zpracování jednotlivých požadavků. Podobně jako u výběru sítě i volba nevhodného měřícího programu může vést k neprůkazným výsledkům. Požadavky pro vhodný měřící program jsou především:

- rychlá příprava požadavků
- schopnost paralelizace odesílání požadavků

Pro své měření jsem se rozhodl použít dva různé měřící programy. Oba splňují výše uvedené požadavky.

1. Nástroj pro příkazovou řádku Go-wrk napsaný v jazyce Go [18]
2. Mnou vytvořený balíček HTTPBenchmarkTool napsaný v jazyce Julia [19]

Nástroj Go-wrk je vhodný díky jeho vysoké efektivitě, plynoucí z implementace v jazyce Go, ovšem absentuje možnost hlubší analýzy naměřených dat, jelikož nástroj data o jednotlivých požadavcích neukládá.
Na druhou stranu balíček HTTPBenchmarkTool jsem implementoval primárně za účelem hlubší analýzy naměřených dat s následnou možností jejich vizualizace. Použití obou dvou měřících nástrojů se tedy jevilo jako vhodný kompromis.

3.1.5 Měření

Každé z běhových prostředí jsem otestoval pomocí stejné sady výkonových testů. Tato sada obsahovala 3 přímočaré testy pomocí nástroje Go-wrk:

- simulace provozu s 10 klienty po dobu 10 sekund (příkaz 3.7)
- simulace provozu s 50 klienty po dobu 10 sekund (příkaz 3.8)
- simulace provozu se 100 klienty po dobu 10 sekund (příkaz 3.9)

a jeden komplexnější test sestávající se z 10 dílčích testů pomocí nástroje HTTPBenchmarkTool:

- simulace vyřízení 100.000 požadavků při různém počtu klientů v rozmezí od 10 do 100 s navýšením počtu klientů o 10 při každém dalším měření (celkem 1.000.000 požadavků) 3.10

```
go-wrk -c 10 -d 10 http://localhost:3000
```

Listing 3.7: Příkaz ke spuštění Go-wrk s 10 klienty po dobu 10s

```
go-wrk -c 50 -d 10 http://localhost:3000
```

Listing 3.8: Příkaz ke spuštění Go-wrk s 50 klienty po dobu 10s

```
go-wrk -c 100 -d 10 http://localhost:3000
```

Listing 3.9: Příkaz ke spuštění Go-wrk se 100 klienty po dobu 10s

```
benchmark("http://localhost:3000";
          req_each=100000,
          client_ns=10:10:100)
```

Listing 3.10: Volání funkce benchmark z balíčku HTTPBenchmarkTool

3.2 Zátěž CPU

V kontextu porovnání zátěže CPU jsou implementace měřených programů záměrně stejné pro všechny tři běhová prostředí. Při volbě vhodného programu pro měření zátěže procesoru je třeba dbát na minimalizaci jeho vedlejších efektů. Například není vhodné v rámci měřeného programu volat některé API směřující mimo jádro prostředí. V takovém případě je převážně měřena efektivita konkrétní implementace volaného API namísto požadované zátěže CPU. Oba zvolené programy jsou tudíž napsány v čistém JavaScriptu a bez vedlejších efektů (s výjimkou vypsání výsledku do konsole). Použité programy se od sebe navzájem liší primárně způsobem exekuce kódu a paměťovou náročností. Zatímco výpočet Fibonacciho posloupnosti je rekurzivní a tudíž i paměťově náročný vzhledem k zásobníku volání, násobení matic je z hlediska způsobu exekuce čistě sekvenční výpočet s nízkou (konstantní) paměťovou náročností.

Na místě je vhodné poznamenat, že rekurzivní výpočet Fibonacciho posloupnosti je velmi neefektivní, s asymptotickou složitostí $O(2^n)$. Pomocí metody dynamického programování je možné výpočet dramaticky urychlit, konkrétně až na úroveň asymptotické složitosti $O(n)$. Cílem programu v kontextu zátěžového testu ovšem není optimalizovat výpočet, nýbrž simulovat zátěž CPU. Je tedy paradoxně vhodnější zvolit neefektivní variantu, která procesor zatěžuje více a jiným způsobem – pomocí rekurze.

3.2.1 Rekurzivní výpočet Fibonacciho posloupnosti

První měřený program 3.11, uložený v souboru *fibonacci.js*, je rekurzivní implementace výpočtu 43. prvku Fibonacciho posloupnosti. Výpočet 43. prvku jsem určil jakožto optimální experimentálně, kdy výpočet na mém konkrétním použitém hardware trval řádově jednotky sekund.

```
1 //Deklarace funkce pro vypocet n-teho prvku fibonacciho
      posloupnosti
2 function fibonacci(n) {
3     if (n <= 1) return 1;
4     return fibonnaci(n - 1) + fibonnaci(n - 2);
5 }
6
7 //Zavolani funkce pro vypocet 43. prvku
8 fibonacci(43);
```

Listing 3.11: Program pro rekurzivní výpočet 43. prvku fibonacciho posloupnosti

3.2.2 Násobení matic

Druhý měřený program 3.12, uložený v souboru *matrixMult.js*, vykonává sekvenční násobení 10x10 matice samu sebou po dobu 1 s. Matice použitá v programu je netriviální, má každý řádek i sloupec jiný a neobsahuje nuly.

```
//funkce pro klasicke nasobeni matic
function matrixMultiplication(a, b) {
    const result = [];
    for (let i = 0; i < a.length; i++) {
        result[i] = [];
        for (let j = 0; j < b[0].length; j++) {
            let sum = 0;
            for (let k = 0; k < a[0].length; k++) {
                sum += a[i][k] * b[k][j];
            }
            result[i][j] = sum;
        }
    }
    return result;
}

//deklarace matice A pouzite pro vypocet
const matrixA = [
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [10, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    [9, 10, 1, 2, 3, 4, 5, 6, 7, 8],
    [8, 9, 10, 1, 2, 3, 4, 5, 6, 7],
    [7, 8, 9, 10, 1, 2, 3, 4, 5, 6],
    [6, 7, 8, 9, 10, 1, 2, 3, 4, 5],
    [5, 6, 7, 8, 9, 10, 1, 2, 3, 4],
    [4, 5, 6, 7, 8, 9, 10, 1, 2, 3],
    [3, 4, 5, 6, 7, 8, 9, 10, 1, 2],
    [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
];

//funkce pro sekvencni nasobeni matice A samu sebou po dobu 1s
function cpuBenchmark() {
    const start = Date.now();
    let count = 0;
    while(Date.now() - start < 1000) {
        matrixMultiplication(matrixA, matrixA);
        count++;
    }
    console.log(count);
}

//zavolani funkce cpuBenchmark()
cpuBenchmark();
```

Listing 3.12: Program pro iterativní násobení 10x10 matic

3.2.3 Výkonové testy

Měřené vlastnosti

Měřené vlastnosti jsou:

- Doba trvání rekurzivního výpočtu (čím nižší, tím lepší)
- Počet iterací sekvenčního výpočtu (čím vyšší, tím lepší)

Měřící program

Použitý měřící program je jednoduchý bashový skript 3.13, uložený v souboru *cpuBenchmark.sh*.

```
#!/bin/bash
//lokalni fuknce pro spusteni prvniho programu a vypsani vysledku
run() {
    local runtimeName=$1
    local runCommand=$2

    local elapsedTime=$( { time $runCommand; } 2>&1 | grep real |
    awk '{print $2}')
    echo "$runtimeName: $elapsedTime"
}

//zavolani funkce run() pro kazde behove prostredi
echo -e "\n1) Recursive computation of 43rd fibonacci number"
run "Node" "node fibonacci.js"
run "Deno" "deno run fibonacci.js"
run "Bun" "bun fibonacci.js"

//zavolani programu matrixMult.js pro kazde behove prostredi
echo -e "\n2) 10x10 matrix multiplication for 1s"
echo "Node: $(node matrixMult.js) iterations made"
echo "Deno: $(deno run matrixMult.js) iterations made"
echo "Bun: $(bun matrixMult.js) iterations made"
```

Listing 3.13: Výkonový test CPU v podobě bashového skriptu

Výkonový test nejprve zavolá lokální funkci `run()`, která spustí program *fibonacci.js* (3.11), postupně pro každé běhové prostředí a pomocí vestavěného nástroje pro příkazovou řádku `time` změří dobu běhu programu. Výstup nástroje `time` je následně přesměrován ze standartního chybového výstupu (deskriptor souboru číslo 2) na standartní výstup (deskriptor souboru číslo 1) pomocí příkazu `2>&1`. Příkaz `grep real` vypíše ze standartního výstupu pouze řádky obsahující slovo *real*, jelikož příkaz `time` vrací tři hodnoty – *real*, *user* a *sys*. Celkový běh programu je udán právě hodnotou *real*. Poté se segreguje časový údaj od slova *real*, což zajišťuje příkaz `awk '{print $2}'`. Nakonec je vypsána naměřená hodnota spolu s názvem měřeného prostředí pomocí nástroje `echo`.
Poté postupně pro každé běhové prostředí spustí program *matrixMult.js* (3.12). Tento program vypíše výsledný počet iterací násobení matic na standartní výstup, tudíž není třeba lokální funkce k získání výsledku měření.

3.2.4 Měření

Každé z běhových prostředí jsem otestoval pomocí stejné sady výkonových testů. Tato sada obsahovala 2 testy za pomoci uvedeného programu:

- Doba výpočtu 43. prvku Fibonacciho posloupnosti 3.14
- Počet iterací násobení 10x10 matic za dobu 1s 3.14

```
bash cpuBenchmark.sh
```

Listing 3.14: Příkaz pro spuštění výkonového testu CPU

3.3 I/O operace

V kontextu vstupně-výstupních operací jsem rozlišoval jejich synchronní a asynchronní varianty. Pro každé běhové prostředí jsem tedy otestoval obě tyto varianty a porovnal je s odpovídajícími variantami ostatních běhových prostředí. Rozdíl mezi synchronní variantou oproti asynchronní spočívá v blokaci jiných operací v průběhu vykonávání. Synchronní varianta je blokující, což znamená, že běhové prostředí nevykoná žádný jiný kód, dokud není dokončena vstupně-výstupní operace. Tato varianta byla dříve poměrně rozšířená z důvodu vysoké neefektivity asynchronních variant, které naopak umožňují vykonávání dalšího kódu během zpracovávání operace. Postupem času se však asynchronní operace výrazně zefektivnily a proto jsou dnes v drtivé většině případů preferované. Dokládá to například i fakt, že nejnovější z testovaných prostředí, Bun, neobsahuje nativní implementaci synchronního zapisování do souboru.

Pro každé běhové prostředí jsou uvedeny dvě varianty měřeného programu – jedna synchronní a druhá asynchronní. Ve všech případech program provede zápis do daného počtu souborů o zadané velikosti. Oba parametry jsou programu předány jako argumenty při spuštění, první argument udává požadovaný počet souborů a druhý argument udává velikost jednoho souboru v MB.

3.3.1 Node.js

Synchronní varianta

```
//promenna reprezentujici rozhrani pro praci se soubory
const fs = require('fs');
const fileCount = process.argv[2] || 500;
const fileSize = process.argv[3] || 10
const DATA_SIZE = 1024 * 1024 * fileSize;
const data = Buffer.alloc(DATA_SIZE, "1");

//deklarace funkce pro synchronni zapis
function syncWrite() {
    for (let i = 0; i < fileCount; i++) {
        const FILE_PATH = `testSync${i}.txt`;
        fs.writeFileSync(FILE_PATH, data);
    }
}

syncWrite();
```

Listing 3.15: Program pro synchronní zápis souborů v prostředí Node.js

Asynchronní varianta

```
//promenna reprezentujici rozhrani pro praci se soubory
const fs = require('fs');
const fileCount = process.argv[2] || 500;
const fileSize = process.argv[3] || 10
const DATA_SIZE = 1024 * 1024 * fileSize;
const data = Buffer.alloc(DATA_SIZE, "1");

//deklarace funkce pro asynchronni zapis
async function asyncWrite() {
    for (let i = 0; i < fileCount; i++) {
        const FILE_PATH = `testAsync${i}.txt`;
        fs.writeFile(FILE_PATH, data, (err) => {
            if (err) throw err;
        });
    }
}

asyncWrite();
```

Listing 3.16: Program pro asynchronní zápis souborů v prostředí Node.js

3.3.2 Deno varianty

Synchronní varianta

```
const fileCount = Deno.args[0] || 500;
const fileSize = Deno.args[1] || 10
const DATA_SIZE = 1024 * 1024 * fileSize;
const data = new Uint8Array(DATA_SIZE).fill(1);

//deklarace funkce pro synchronni zapis
function syncWrite(){
    for(let i = 0; i < fileCount; i++) {
        const FILE_PATH = `testSync${i}.txt`;
        Deno.writeFileSync(FILE_PATH, data);
    }
}

syncWrite();
```

Listing 3.17: Program pro synchronní zápis souborů v prostředí Deno

Asynchronní varianta

```
const fileCount = Deno.args[0] || 500;
const fileSize = Deno.args[1] || 10
const DATA_SIZE = 1024 * 1024 * fileSize;
const data = new Uint8Array(DATA_SIZE).fill(1);

//deklarace funkce pro asynchronni zapis
async function asyncWrite() {
    for (let i = 0; i < fileCount; i++) {
        const FILE_PATH = `testAsync${i}.txt`;
        Deno.writeFile(FILE_PATH, data, (err) => {
            if (err) throw err;
        });
    }
}

asyncWrite();
```

Listing 3.18: Program pro asynchronní zápis souborů v prostředí Deno

3.3.3 Bun varianty

Synchronní varianta

```
//promenna reprezentujici rozhrani pro praci se soubory
const fs = require('fs');
const fileCount = process.argv[2] || 500;
const fileSize = process.argv[3] || 10
const DATA_SIZE = 1024 * 1024 * fileSize;
const data = Buffer.alloc(DATA_SIZE, "1");

//deklarace funkce pro synchronni zapis
function syncWrite(){
    for(let i = 0; i < fileCount; i++) {
        const FILE_PATH = `testSync${i}.txt`;
        fs.writeFileSync(FILE_PATH, data);
    }
}

syncWrite();
```

Listing 3.19: Program pro synchronní zápis souborů v prostředí Bun

V prostředí Bun neexistuje nativní implementace rozhraní pro synchronní zápis, lze však použít modul *fs* z Node APIs, který je v prostředí Bun implementován kvůli zpětné kompatibilitě s prostředím Node.js.

Asynchronní varianta

```
const fileCount = process.argv[2] || 500;
const fileSize = process.argv[3] || 10
const DATA_SIZE = 1024 * 1024 * fileSize;
const data = Buffer.alloc(DATA_SIZE, "1");

//deklarace funkce pro asynchronni zapis
async function asyncWrite(){
    for(let i = 0; i < fileCount; i++) {
        const FILE_PATH = `testAsync${i}.txt`;
        Bun.write(FILE_PATH, data, (err) => {
            if (err) throw err;
        });
    }
}

asyncWrite();
```

Listing 3.20: Program pro asynchronní zápis souborů v prostředí Bun

3.3.4 Výkonové testy

Měřené vlastnosti

Měřené vlastnosti jsou:

- Doba synchronního zápisu souborů (čím nižší, tím lepší)
- Doba asynchronního zápisu souborů (čím nižší, tím lepší)

3.3.5 Měřící program

Použitý měřící program je stejně jako v předchozím případě jednoduchý bashový skript 3.21. Stejně jako v případě jednotlivých měřených programů, i měřícímu programu je třeba zadat dva vstupní parametry, udávající počet souborů k zapsání a jejich dílčí velikost v MB.
Bashový skript je svým způsobem vykonávání jednotlivých měření velmi podobný měření času výpočtu 43. prvku Fibanacciho poslounosti (skript 3.13). I v tomto případě je použita lokální funkce, která spustí měřený program, změří dobu jeho běhu pomocí nástroje `time` a následně získá požadovanou hodnotu pomocí příkazu `2>&1 | grep real | awk '{print $2}'`. V tomto případě se po každém dílčím testu odstraní veškeré vytvořené testovací soubory pomocí příkazu `rm test*`.

```
#!/bin/bash
//vstupni parametry pro spusteni jednotlivych merenych programu
fileCount=$1
fileSize=$2
echo -e "Benchmarking write of" $fileCount "files of size"
    $fileSize "MB\n"

function run() {
    local runtimeName=$1
    local cmd=$2

    local runtime=$( { time $cmd $fileCount $fileSize;} 2>&1 | grep
     real | awk '{print $2}' )
    echo -e "$runtimeName: $runtime"
    rm test*
}

//zavolani funkce run() pro obe varianty zapisu a kazde behove
    prostredi
run "Node.js synchronously" "node nodeSync.js"
run "Node.js asynchronously" "node nodeAsync.js"

run "Deno synchronously" "deno run --allow-write denoSync.js"
run "Deno asynchronously" "deno run --allow-write denoAsync.js"

run "Bun synchronously" "bun bunSync.js"
run "Bun asynchronously" "bun bunAsync.js"
```

Listing 3.21: Program pro výkonové testování I/O operace v podobě bashového skriptu

Program je uložený v souboru *benchmarkIO.sh*.

3.3.6 Měření

Každé z běhových prostředí jsem otestoval pomocí stejné sady výkonových testů. Tato sada obsahovala 3 testy za pomoci uvedeného programu:

- zápis 10 souborů o velikosti 1GB (příkaz 3.22)
- zápis 100 souborů o velikosti 100MB (příkaz 3.23)
- zápis 1000 souborů o velikosti 10MB (příkaz 3.24)

```
1 bash benchmarkIO.sh 10 1000
```

Listing 3.22: Příkaz ke spuštění skriptu pro zápis 10 souborů o velikosti 1GB

```
1 bash benchmarkIO.sh 100 100
```

Listing 3.23: Příkaz ke spuštění skriptu pro zápis 100 souborů o velikosti 100MB

```
1 bash benchmarkIO.sh 1000 10
```

Listing 3.24: Příkaz ke spuštění skriptu pro zápis 1000 souborů o velikosti 10MB

3.4 Transpilace TypeScriptu

Při výběru programu pro testování efektivity použití TypeScriptu je vhodné použít takový program, který využívá hlavní vlastnosti jazyka – typování. Zvolil jsem proto program pro hledání nejkratší cesty v orientovaném grafu, který kromě použití primitivních typů definuje dva své vlastní typy – vrchol a hranu. Definice vlastních typů sice není nutná podmínka pro měřený program, na druhou stranu se jedná o typický příklad použití TypeScriptu v praxi.

Použité grafy jsem representoval pomocí textových souborů s definovaným formátem. Na prvním řádku souboru jsou tři čísla – první udává celkový počet hran grafu (n), druhý a třetí udávají startovní a cílový vrchol. Poté následuje n řádků, které representují hranu grafu opět pomocí tří čísel – id počátečního vrcholu, id koncového vrcholu a cena hrany. [20]
K nalezení nejkratší cesty v orientovaném grafu jsem implementoval známý Dijkstrův algoritmus. [21]

3.4.1 Hledání nejkratší cesty v grafu

Měřený program 3.25, uložený v souboru *shortestPath.ts*, je stejný pro všechna běhová prostředí. Za účelem možného použití programu ve všech prostředích, je nutné deklarovat namespace Deno, a to z důvodu rozdílného rozhraní pro práci se soubory. Namespace neposkytuje implementace, ale pouze deklarace použitých atributů a funkcí jako prostředek k oklamání transpilátoru. Pokud program běží mimo prostředí Deno, deklarované atributy a funkce se nikdy nepoužijí. Prostředí Bun díky snaze o zpětnou kompatibilitu s prostředím Node.js v rámci Node APIs obsahuje implementaci použitého modulu *fs*, tudíž není nutné deklarovat další namespace.

```
//Deklarace namespace Deno pro universalitu programu
declare namespace Deno {
    function readTextFileSync(filePath: string): string;
    export const args: string[];
}

//Pokud program nebezi v prostredi Deno je pouzito rozhrani fs
const denoEnv = typeof Deno !== 'undefined';
const fs = denoEnv ? undefined : require('fs');

//Typ Edge representujici hranu grafu
type Edge = {
    goalVertexId: number;
    price: number;
};

//Typ Vertex representujici vrchol grafu
type Vertex = {
    id: number;
    edges: Edge[];
};

//Funkce pro nacteni grafu z textového souboru
function readGraphFromFile(filePath: string): { vertices: Vertex[];
     startVertexId: number; goalVertexId: number } {
    //Implementace funkce
}

//Funkce pro nalezeni nejkratsi cesty mezi dvema vrcholy
function calculateMinimalCost(graph: { vertices: Vertex[];
    startVertexId: number; goalVertexId: number }): number {
    //Implementace funkce
}

//Nacteni grafu a spusteni programu s naslednym vypsanim vysledku
const filename: string = denoEnv ? Deno.args[0] : process.argv[2];
const graph = readGraphFromFile(filename);
console.log(calculateMinimalCost(graph));
```

Listing 3.25: Program pro nalezení nejkratší cestu v orientovaném grafu

Program je rozsáhlejší, konkrétní implementace použitých funkcí `readGraphFromFile()` (kód 3.26) a `calculateMinimalCost()` (kód 3.27) jsou pro přehlednost uvedeny samostatně.

```
function readGraphFromFile(filePath: string):
{vertices: Vertex[]; startVertexId: number; goalVertexId: number} {
    const data: string = denoEnv ? Deno.readTextFileSync(filePath)
                                 : fs.readFileSync(filePath,'utf8')

    const lines: string[] = data.split('\n');
    const [totalEdges, startVertexId, goalVertexId] =
          lines[0].split(' ').map(Number);
    const vertices: Vertex[] = [];

    for (let i = 1; i <= totalEdges; i++) {
        const [start,goal,price] = lines[i].split(' ').map(Number);
        if (!vertices[start])
           { vertices[start] = { id: start, edges: [] }; }
        vertices[start].edges.push({ goalVertexId: goal, price });
    }

    return { vertices, startVertexId, goalVertexId };
}
```

Listing 3.26: Použitá funkce readGraphFromFile()

```
function calculateMinimalCost(graph: { vertices: Vertex[];
startVertexId: number; finishVertexId: number }): number {
    const { vertices, startVertexId, finishVertexId } = graph;
    const size = vertices.length;
    const visited: boolean[] = Array(size).fill(false);
    const discovered: boolean[] = Array(size).fill(false);
    const distances: number[] = Array(size).fill(Infinity);

    distances[startVertexId] = 0;
    const queue: number[] = [startVertexId];
    while (queue.length > 0) {
        queue.sort((a, b) => distances[a] - distances[b]);
        const currentVertexId = queue.shift()!;
        if (currentVertexId === finishVertexId) break;
        if (distances[currentVertexId] === Infinity) break;
        visited[currentVertexId] = true;

        for (const edge of vertices[currentVertexId].edges) {
            const goalId = edge.goalVertexId;
            if (!visited[goalId] && distances[goalId] >
                distances[currentVertexId] + edge.price) {
                distances[goalId] =
                    distances[currentVertexId] + edge.price;
                if (!discovered[goalId]) {
                    queue.push(goalId);
                    discovered[goalId] = true;
                }
            }
        }
    }
    return distances[finishVertexId];
}
```

Listing 3.27: Použitá funkce calculateMinimalCost()

3.4.2 Výkonové testy

Měřené vlastnosti

Měřená vlastnost je:

- Doba strávená transpilací TypeScriptu (čím nižší, tím lepší)

Doba strávená transpilací TypeScriptu je vypočítána odečtením doby běhu JavaScript programu od doby běhu TypeScript programu.

3.4.3 Měřící program

Použitý měřící program je stejně jako v předchozích případech jednoduchý bashový skript 3.28. V tomto konkrétním případě se v rámci lokální funkce `run()` navíc provede kontrola výsledku programu – cena nejkratší cesty ze startovacího do cílového vrcholu.

```
#!/bin/bash
filename=$1
expectedValue=$2

run() {
    local benchmarkName=$1
    local runCommand=$2
    local output=$( { time $runCommand $filename; } 2>&1 )
    local elapsedTime=$(echo "$output" | grep real
                        | awk '{print $2}')
    local solution=$(echo "$output" | head -n 1)
    if [[ $solution == $expectedValue ]]; then
        echo -e "$benchmarkName solution is correct.
                 Elapsed time: $elapsedTime"
    else
        echo -e "$benchmarkName solution is incorrect.
                 Expected: $expectedValue, Actual: $solution"
        exit 1
    fi
}

echo -e "\nTypeScript:"
run "Node" "bash node.sh"
run "Deno" "deno run --allow-read shortestPath.ts"
run "Bun" "bun shortestPath.ts"

echo -e "\nJavaScript:"
run "Node" "node shortestPath.js"
run "Deno" "deno run --allow-read shortestPath.js"
run "Bun" "bun shortestPath.js"
```

Listing 3.28: Program pro výkonové testování TypeScript transpilace v podobě bashového skriptu

Program je uložený v souboru *benchmarkTS.sh*.

V prostředí Node.js je nutné transpilaci TypeScriptu provést explicitně (neprovede se automaticky) pomocí nástroje *tsc*. Tento proces provádí pomocný bashový skript uložený v souboru *nodeTS.sh* (3.29):

```
#!/bin/bash

program=shortestPath
filename=$1

tsc $program.ts
node $program.js $filename
```

Listing 3.29: Pomocný bashový skript pro spuštění programu v prostředí Node.js

3.4.4 Měření

Každé z běhových prostředí jsem otestoval pomocí stejné sady výkonových testů. Tato sada obsahovala 2 testy za pomoci uvedeného programu:

- nalezení nejkratší cesty v grafu o 6 vrcholech a 12 hranách 3.30
- nalezení nejkratší cesty v grafu o 10 000 vrcholech a 100 000 hranách 3.31

```
bash benchmarkTS.sh graph.txt 7
```

Listing 3.30: Příkaz ke spuštění skriptu pro nalezení nejkratší cesty v malém grafu

```
bash benchmarkTS.sh graphLarge.txt 48
```

Listing 3.31: Příkaz ke spuštění skriptu pro nalezení nejkratší cesty ve velkém grafu

Kapitola 4
Výsledky

Tato kapitola se věnuje analýze naměřených hodnot dílčích výkonových testů v rámci porovnání běhových prostředí. Výsledky testů jsou analyzovány jak slovně, tak pomocí tabulek a grafů. V závěru kapitoly je sepsáno doporučení pro volbu optimálního běhového prostředí na základě preferovaných vlastností.

4.1 Prostředí

Měřená prostředí byla použita v následujících verzích:
Node.js v21.7.1
Deno v1.39.1
Bun v1.0.11

Všechny výkonové testy s výjimkou posledního byly provedeny na následujícím hardware:
CPU Apple M1; 8 jader; 3.2 GHz
GPU Apple M1; 7 jader; 3.2 GHz
RAM 16 GB
Disk SSD 256 GB

Poslední výkonový test – porovnání zátěže CPU na OS Windows vs. OS Linux byl proveden na následujícím hardware:
CPU Intel Core i7-6700HQ; 2.6 GHz
GPU Nvidia GeForce GTX 960M; 1.1 GHz
RAM 16 GB

S následujícími verzemi operačních systémů:
Windows Windows 10 v22H2
Linux Ubuntu v22.04.3 LTS

4.2 Zpracování HTTP požadavků

4.2.1 Go-wrk

V tabulce 4.1 jsou uvedeny naměřené hodnoty prvního výkonového testu s 10 paralelními připojeními (klienty) po dobu 10 sekund. (příkaz 3.7)

Tabulka 4.1: výkonový test s 10 klienty po dobu 10s

Serverové prostředí	Počet požadavků/s	Průměrný čas odezvy(μs)
Node.js	78 748.12	126.987
Deno	98 055.23	101.983
Bun	119 425.36	83.734

V tabulce 4.2 jsou uvedeny naměřené hodnoty druhého výkonového testu s 50 paralelními připojeními po dobu 10 sekund. (příkaz 3.8)

Tabulka 4.2: výkonový test s 50 klienty po dobu 10s

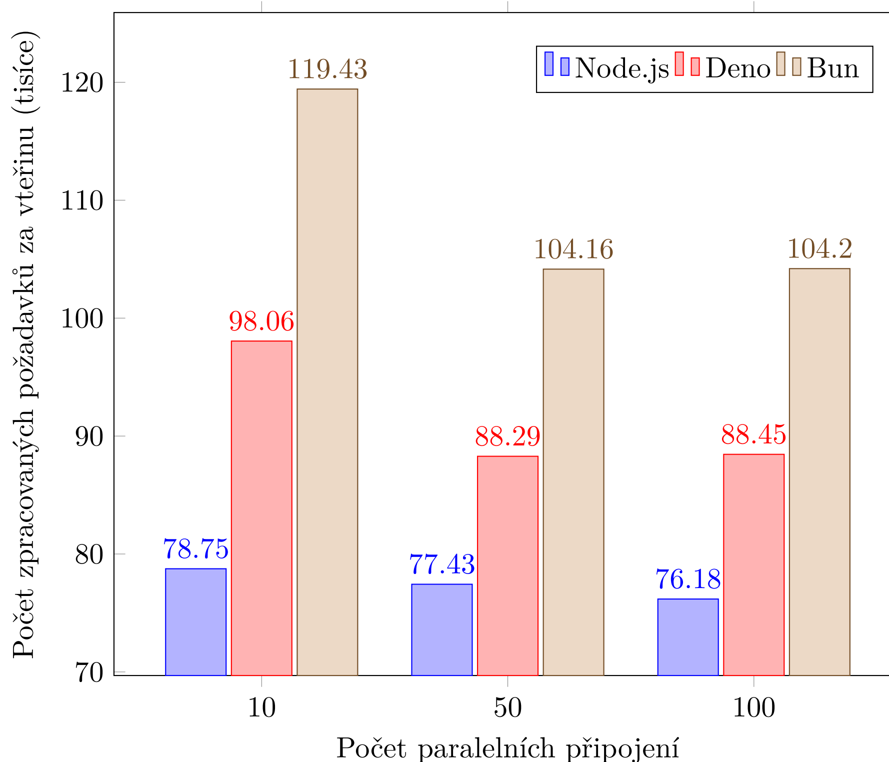
Serverové prostředí	Počet požadavků/s	Průměrný čas odezvy(μs)
Node.js	77 434.15	645.709
Deno	88 285.47	566.344
Bun	104 154.80	480.054

V tabulce 4.3 jsou uvedeny naměřené hodnoty třetího výkonového testu se 100 paralelními připojeními po dobu 10 sekund. (příkaz 3.9)

Tabulka 4.3: výkonový test se 100 klienty po dobu 10s

Serverové prostředí	Počet požadavků/s	Průměrný čas odezvy (ms)
Node.js	76 180.83	1.313
Deno	88 449.78	1.131
Bun	104 197.68	0.960

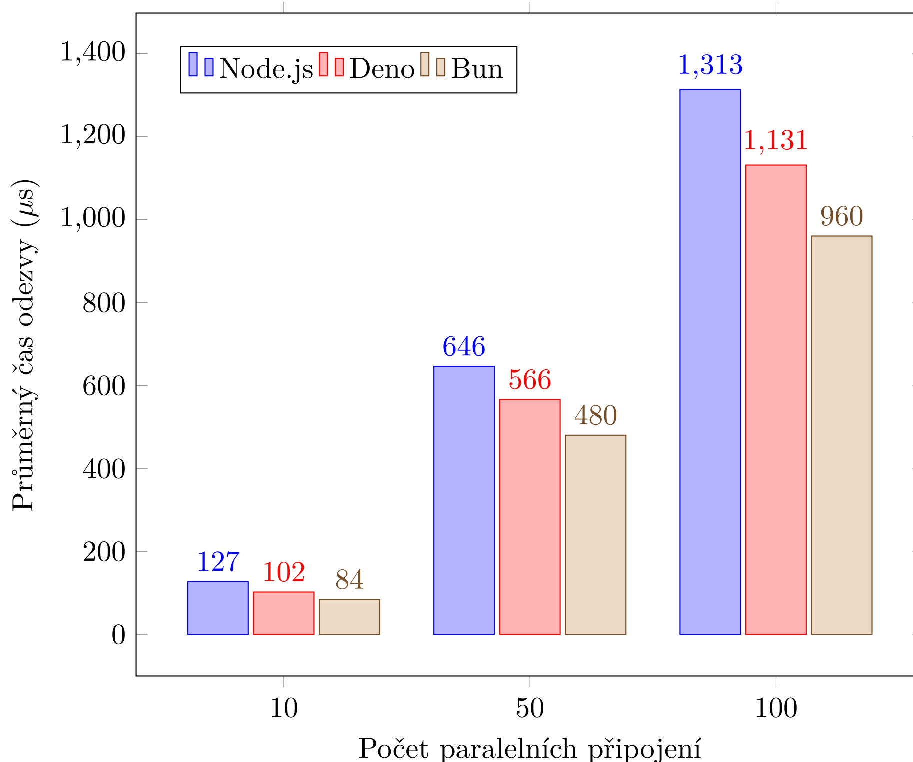
Na obrázku 4.1 je znázorněn počet zpracovaných požadavků za vteřinu v závislosti na počtu klientů(paralelních připojení).



Obrázek 4.1: Počet zpracovaných požadavků za vteřinu v závislosti na počtu klientů

Při měření s 10 paralelními klienty dosáhlo běhové prostředí Node.js hodnot *78 748* zpracovaných požadavků za vteřinu a průměrný čas odezvy *127.0* µs. Běhové prostředí Deno dosáhlo hodnot *98 055* zpracovaných požadavků za vteřinu, což je *1,25*x více než Node.js a průměrný čas odezvy *102.0* µs, což je *0.8* násobek průměrného času odezvy prostředí Node.js. Nejlépe si vedlo běhové prostředí Bun, které dosáhlo hodnot *119 425* zpracovaných požadavků za vteřinu – *1.22*x více než Deno a dokonce *1.52*x více než Node.js a průměrný čas odezvy *83.7* µs, což je *0.82* násobek průměrného času odezvy prostředí Deno a *0.66* násobek průměrného času odezvy prostředí Node.js.

Při zvýšení počtu paralelně připojených klientů na hodnotu 50 výrazně vzrostl průměrný čas odezvy u všech tří běhových prostředí, přičemž počet vyřízených požadavků za vteřinu se u všech běhových prostředí snížil jen mírně. Konkrétně se jedná o následující hodnoty, Bun s průměrným časem odezvy *480.1* µs (*5.732*x) a počtem zpracovaných požadavků za vteřinu *104 155* (*0.87*x), Deno s průměrným časem odezvy *566.3* µs (*5.55*x) a počtem zpracovaných požadavků za vteřinu *88 285* (*0.9*x) a Node.js s průměrným časem odezvy *645.7* µs (*5.085*x) a počtem zpracovaných požadavků za vteřinu *77 434* (*0.98*x). V závorkách jsou uvedeny relativní násobky hodnot vzhledem



Obrázek 4.2: Průměrný čas odezvy v závislosti na počtu klientů

k předchozímu výkonovému testu s 10 klienty.

Poslední výkonový test proběhl se 100 paralelně připojenými klienty. Počet zpracovaných požadavků za sekundu dosáhl srovnatelných hodnot jako v předchozím testu s 50 klienty. Průměrný čas odezvy se zvýšil téměř přímo úměrně dvojnásobnému nárůstu v počtu klientů. Konkrétně se jedná o následující hodnoty, Bun s průměrným časem odezvy 0.96 ms (*2.0*x) a počtem zpracovaných požadavků za vteřinu *104 198* (*1.0*x), Deno s průměrným časem odezvy 1.131 ms (*1.997*x) a počtem zpracovaných požadavků za vteřinu *88 450* (*1.002*x) a Node.js s průměrným časem odezvy 1.313 ms (*2.03*x) a počtem zpracovaných požadavků za vteřinu *76 180* (*0.983*x). V závorkách jsou uvedeny relativní násobky hodnot vzhledem k předchozímu výkonovému testu s 50 klienty.

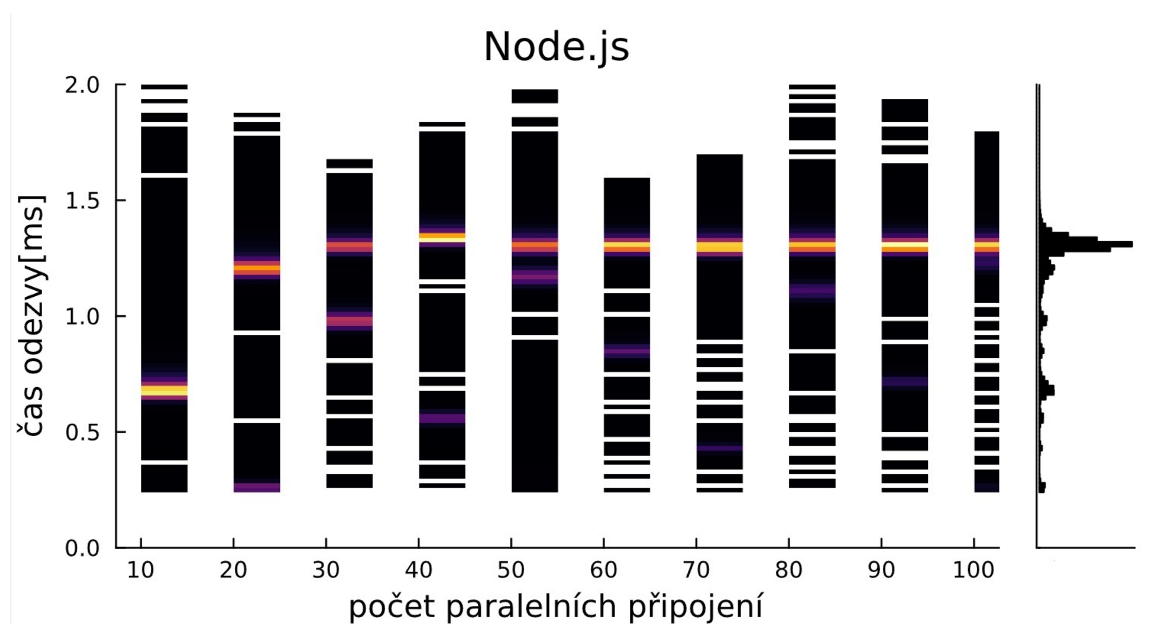
4.2.2 HTTPBenchmarkTool

Použitá funkce `benchmark()` z balíčku HTTPBenchmark vykoná 10 na sobě nezávislých měření požadovaného serveru, všechny výsledky společně s celkovým časem exekuce vrátí jako návratovou hodnotu. K vykreslení následujících grafů byly použity další funkce balíčku ze souboru *vizualisations.jl*.
V tabulce 4.4 jsou uvedeny naměřené hodnoty.
Na obrázcích 4.3, 4.4 a 4.5 jsou vykresleny marginální histogramy časů odezvy

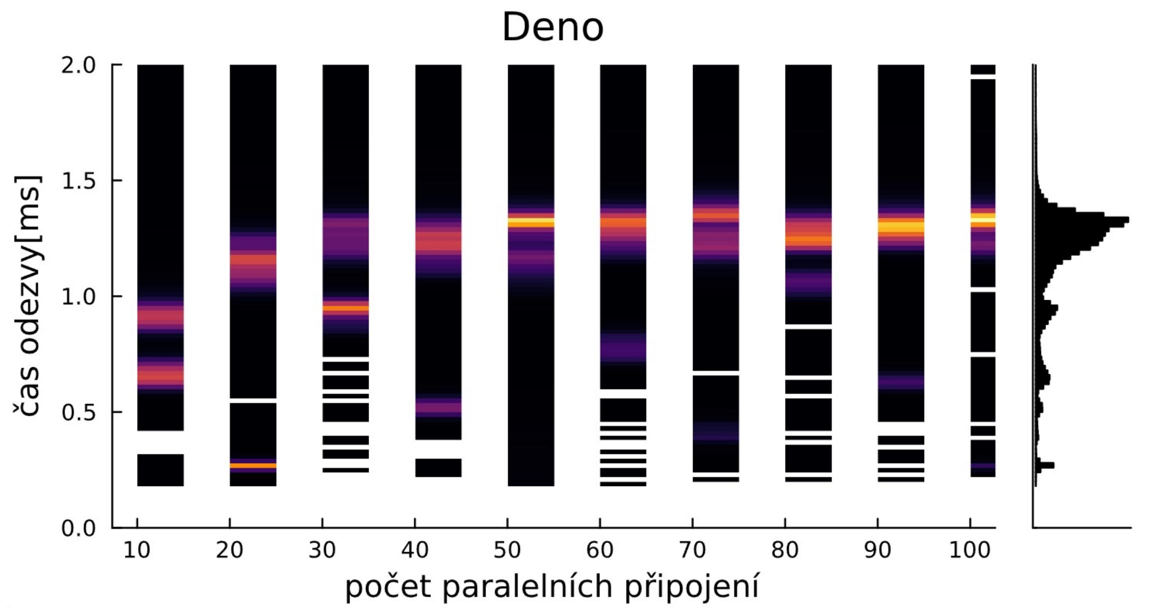
Tabulka 4.4: výkonový test pomocí balíku HTTPBenchmarkTool

Serverové prostředí	Celkový počet požadavků/s	Průměrný čas odezvy (ms)
Node.js	12 453	1.160
Deno	13 071	1.140
Bun	14 085	1.036

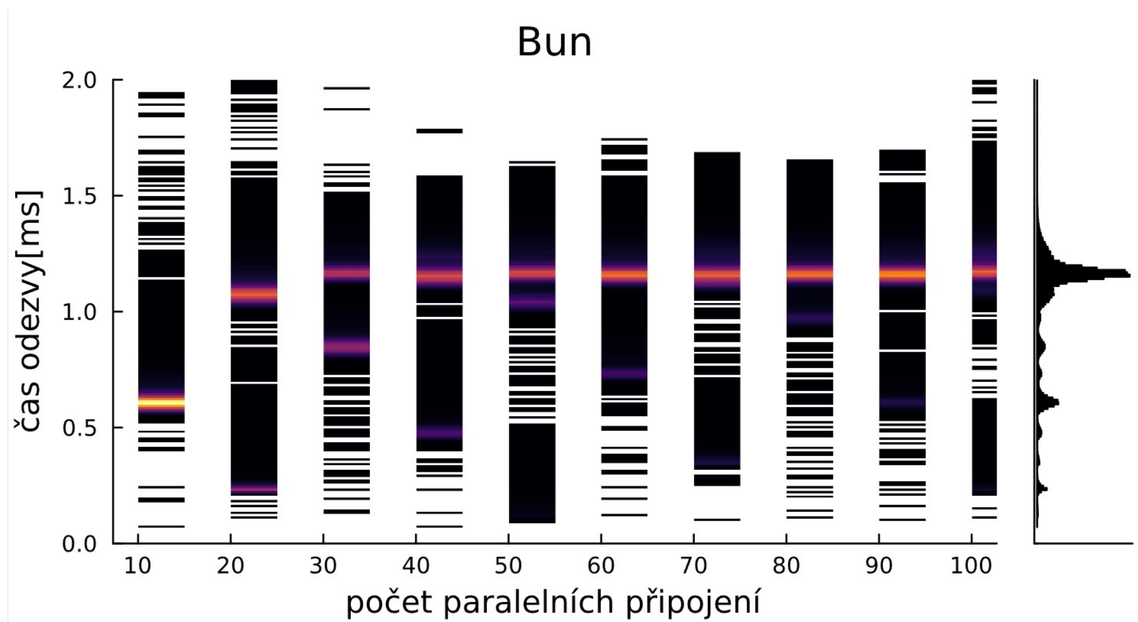
z jednotlivých měření pro dané běhové prostředí. Jeden sloupec reprezentuje vždy jeden dílčí histogram časů odezvy při konkrétním počtu klientů(paralelních připojení). Pro tyto dílčí histogramy platí, že čím je barva tmavší, tím je ve vzorku nižší četnost požadavků s příslušným časem odezvy. Naopak čím světlejší, tím vyšší četnost požadavků s příslušným časem odezvy. Výjimkou je pouze barva bílá, která tvoří pozadí diagramu a v kontextu dílčích histogramů značí žádný výskyt požadavku s příslušnou hodnotou času odezvy. Vpravo je zobrazen histogram všech požadavků v závislosti na času odezvy.



Obrázek 4.3: Marginální histogram dat z měření prostředí Node.js

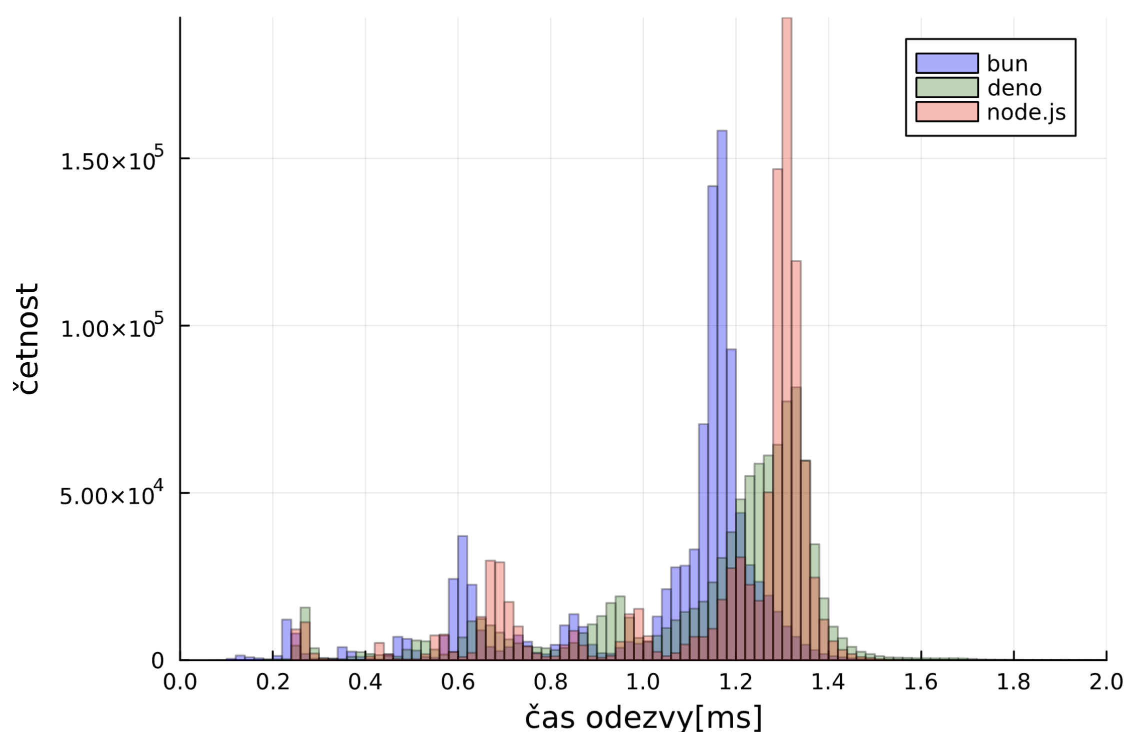


Obrázek 4.4: Marginální histogram dat z měření prostředí Deno



Obrázek 4.5: Marginální histogram dat z měření prostředí Bun

Na obrázku 4.6 je zobrazen histogram časů odezvy všech požadavků s barevným rozlišením měřených běhových prostředí.



Obrázek 4.6: Celkový histogram dat ze všech tří dílčích měření

I v tomto komplexnějším výkonovém testu dosáhlo nejlepších výsledků běhové prostředí Bun. S průměrným časem odezvy 1,036 ms a průměrným počtem zpracovaných požadavků za vteřinu *14 095*. Běhové prostředí Deno dosáhlo hodnot *13 071* zpracovaných požadavků za vteřinu s průměrným časem odezvy 1,140 ms. Naměřené hodnoty pro prostředí Node.js činily *12 453* zpracovaných požadavků za vteřinu a průměrný čas odezvy 1,160 ms. Lepší vizualizace dat v podobě marginálních histogramů umožněna nástrojem HTTPBenchmarkTool odhaluje zajímavý rozdíl v rozptylu časů odezvy jednotlivých běhových prostředí. Zatímco u prostředí Node.js a Bun je rozptyl hodnot času odezvy relativně malý, rozptyl hodnot času odezvy u prostředí Deno je výraznější. Tato skutečnost je lépe patrná z pohledu na společný histogram 4.6, na kterém jsou data prostředí Deno znázorněna zelenou barvou.

4.2.3 Srovnání měřících nástrojů

Naměřená data pomocí obou použitých nástrojů jsou uvedena v tabulkách 4.5 a 4.6.

Tabulka 4.5: srovnání průměrného času odezvy v milisekundách pro daný nástroj a prostředí v závislosti na počtu klientů

Nástroj & prostředí	10 klientů	50 klientů	100 klientů
Go-wrk & Bun	0.08	0.48	0.96
HTTPBenchmarkTool & Bun	0.63	1.09	1.23
Go-wrk & Deno	0.10	0.57	1.13
HTTPBenchmarkTool & Deno	0.68	1.21	1.34
Go-wrk & Node.js	0.13	0.65	1.31
HTTPBenchmarkTool & Node.js	0.72	1.27	1.38

Tabulka 4.6: srovnání počtu vyřízených požadavků za vteřinu pro daný nástroj a prostředí v závislosti na počtu klientů

Nástroj & prostředí	10 klientů	50 klientů	100 klientů
Go-wrk & Bun	119 425	104 154	104 197
HTTPBenchmarkTool & Bun	15 387	13 238	12 953
Go-wrk & Deno	98 055	88 285	88 449
HTTPBenchmarkTool & Deno	14 243	11 963	11 860
Go-wrk & Node.js	78 748	77 434	76 180
HTTPBenchmarkTool & Node.js	13 391	11 539	11 708

Z uvedených údajů lze dopočítat i reálnou míru paralelizace pro jednotlivá měření, a to vynásobením hodnoty průměrného času odezvy (v sekundách) hodnotou počtu vyřízených požadavků za sekundu. Hodnota času odezvy je totiž měřena každým simulovaným klientem izolovaně, zatímco celkový čas měření, ze kterého se posléze dopočítá počet vyřízených požadavků za sekundu, je měřen nástrojem samotným, reprezentujícím všechny klienty současně.

V tabulce 4.7 jsou uvedeny hodnoty reálné míry paralelizace pro všechna dílčí měření.

Tabulka 4.7: srovnání reálné míry paralelizace pro daný nástroj a prostředí v závislosti na počtu klientů

Nástroj & prostředí	10 klientů	50 klientů	100 klientů
Go-wrk & Bun	9.99	49.99	99.99
HTTPBenchmarkTool & Bun	9.69	14.43	15.93
Go-wrk & Deno	9.99	49.99	99.99
HTTPBenchmarkTool & Deno	9.69	14.48	15.89
Go-wrk & Node.js	9.99	49.99	99.99
HTTPBenchmarkTool & Node.js	9.64	14.65	16.16

Všechna prostředí dosahovala horších výsledků při měření pomocí nástroje HTTPBenchmarkTool, v porovnání s nástrojem Go-wrk.
V sekci 3.1.4 byly zmíněny dva požadavky pro vhodný měřící program – rychlá příprava požadavků a schopnost paralelizace odesílání požadavků. Nástroj HTTPBenchmarkTool zaostává oproti nástroji Go-wrk v obou zmiňovaných oblastech.

Pomalejší příprava požadavků je patrná zejména při měřeních s 10 klienty, při kterých nástroj sice dosahuje téměř maximální možné míry reálné paralelizace, přesto jsou naměřené hodnoty horší, než v případě meření pomocí nástroje Go-wrk. Zaostávání nástroje HTTPBenchmarkTool ve schopnosti paralelizace oproti nástroji Go-wrk je patrné při měřeních s 50 a 100 klienty, kdy reálná míra paralelizace dosahuje nižších hodnot než je počet klientů.

Hlavní příčinou zaostávání nástroje HTTPBenchmarkTool je implementace v programovacím jazyce Julia, který je optimalizován primárně pro vědecké výpočty a podobně jako v případě JavaScriptu se jedná o interpretovaný jazyk. Na druhou stranu nástroj Go-wrk, který poskytoval lepší výsledky pro všechna prostředí ve všech měřeních, je napsaný v jazyce Go, což je kompilovaný jazyk s vestavěnou podporou paralelizace pomocí tzv. Goroutines.

Čistý výkon měřených serverů tedy lépe zachycují data naměřená pomocí nástroje Go-wrk. Výsledky získané pomocí nástroje HTTPBenchmarkTool je třeba brát s rezervou z důvodu jeho výše popsaných nedostatků. Jeho výhodou však zůstává ukládání dat z dílčích měření každého klienta, což umožňuje lepší vizualizaci naměřených dat, která je v některých případech žádoucí.

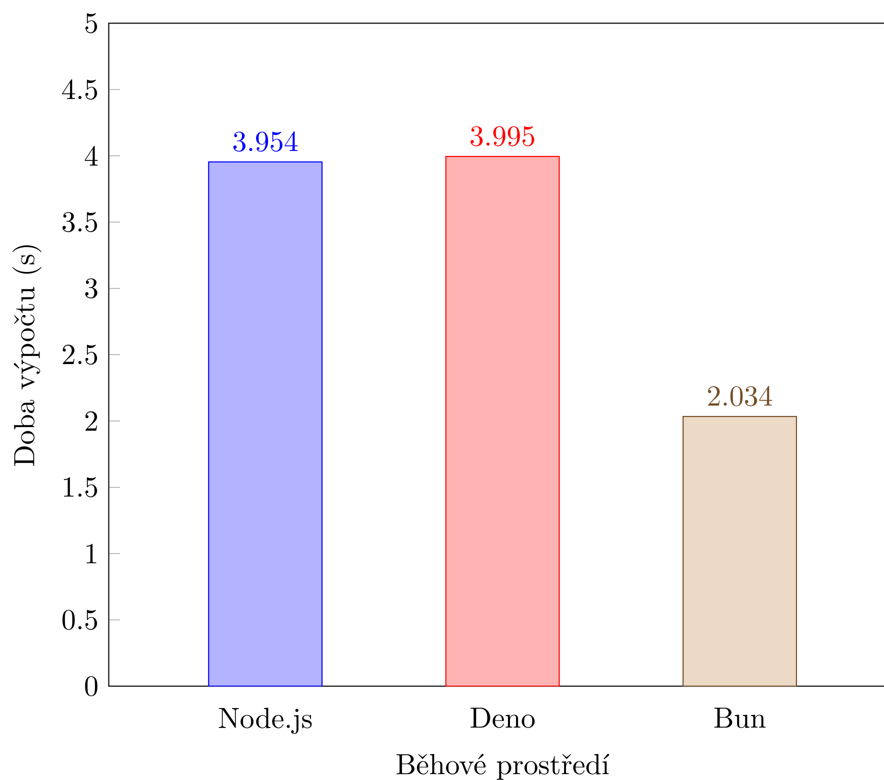
4.3 Zátěž CPU

4.3.1 Výpočet Fibonacciho posloupnosti

V tabulce 4.8 a na obrázku 4.7 jsou uvedeny naměřené hodnoty prvního výkonového testu – rekurzivního výpočtu 43. prvku Fibonacciho posloupnosti. (příkaz 3.14)

Tabulka 4.8: Doby výpočtu 43. prvku Fibonacciho posloupnosti

Serverové prostředí	Doba výpočtu (s)
Node.js	3.954
Deno	3.995
Bun	2.034



Obrázek 4.7: Doby výpočtu 43. prvku Fibonacciho posloupnosti

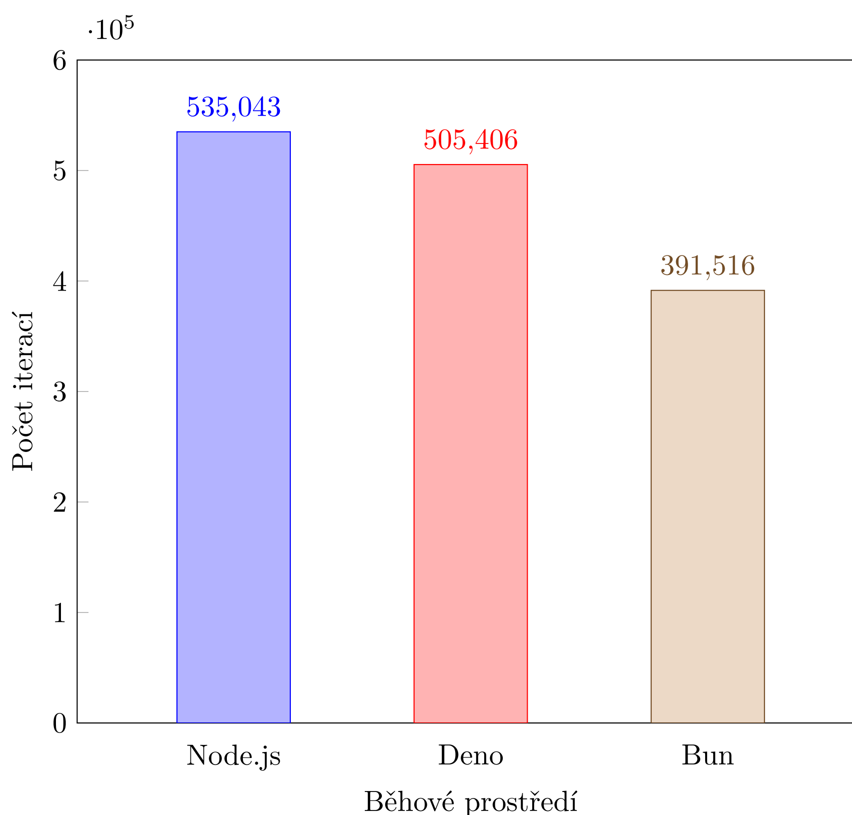
Prostředí Node.js a Deno si v testu vedla podobně - obě dosáhla hodnoty přibližně 4 s. Konkrétně se jedná o hodnoty 3.954 s pro prostředí Node.js a 3.995 s pro prostředí Deno. Nejlepší výsledek zaznamenalo běhové prostředí Bun, a to 2.034 s, což je *1.94*x lepší výsledek než první dvě testovaná prostředí.

4.3.2 Násobení Matic

V tabulce 4.9 a na obrázku 4.8 jsou uvedeny naměřené hodnoty druhého výkonového testu – sekvenčního násobení 10x10 matic. (příkaz 3.14)

Tabulka 4.9: Počet iterací násobení 10x10 matic po dobu 1s

Serverové prostředí	Počet iterací za 1s
Node.js	535 043
Deno	505 406
Bun	391 516



Obrázek 4.8: Počet iterací násobení 10x10 matic po dobu 1s

Zatímco v prvním testu si prostředí Bun vedlo jednoznačně nejlépe, v druhém testu zaznamenalo naopak nejhorší výsledek. Za 1 vteřinu provedlo násobení 10x10 matic pouze *391 516* krát, zatímco prostředí Deno *505 406* krát (*1.29*x více) a prostředí Node.js *535 043* krát (*1.37*x více). Nelze tedy s jistotou říci, že by některé z testovaných prostředí efektivněji zatěžovalo procesor.

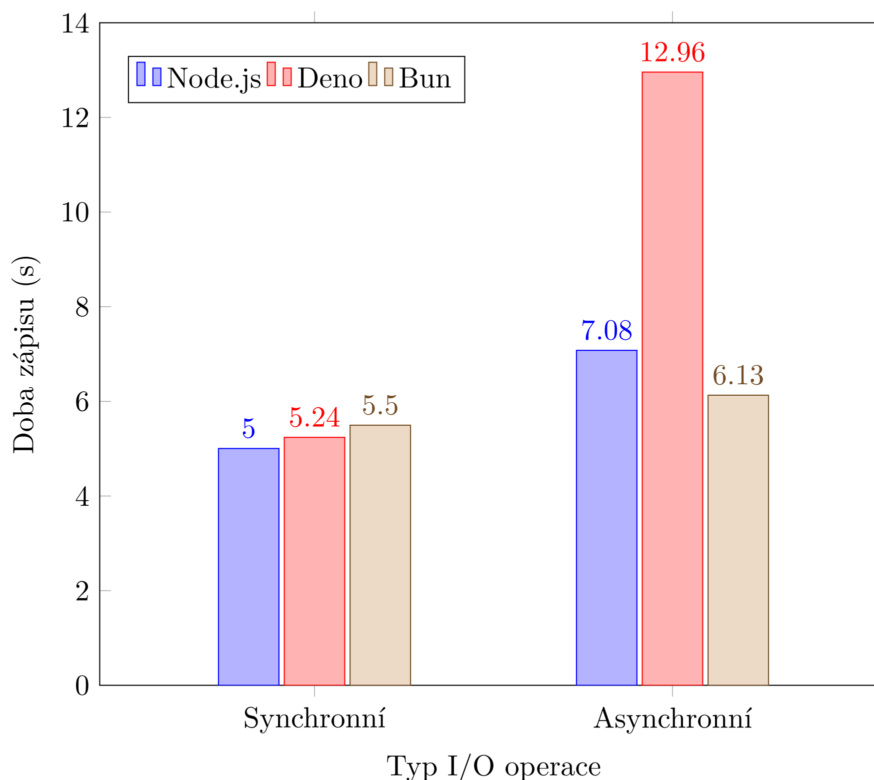
4.4 I/O operace

4.4.1 zápis 10 souborů o velikosti 1GB

V tabulce 4.10 a na obrázku 4.9 jsou uvedeny naměřené hodnoty prvního výkonového testu – zápisu 10 souborů o velikosti 1 GB. (příkaz 3.22)

Tabulka 4.10: Doba zápisu 10 souborů o velikosti 1 GB

Serverové prostředí	Doba synchronního zápisu (s)	Doba asynchronního zápisu (s)
Node.js	5.004	7.078
Deno	5.239	12.957
Bun	5.496	6.130



Obrázek 4.9: Doba zápisu 10 souborů o velikosti 1 GB

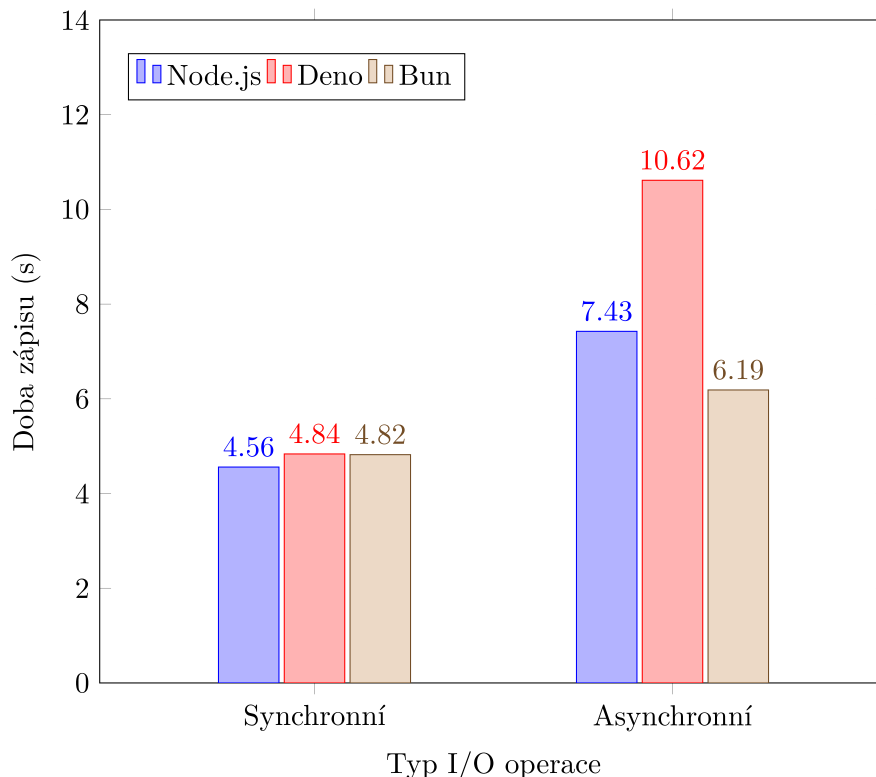
V případě synchronního zápisu si všechna běhová prostředí vedla velmi podobně. Nejlepší výsledek zaznamenalo prostředí Node.js a to *5.004*s, zatímco prostředí Deno *5.239*s (*1.05*x více) a prostředí prostředí Bun *5.496*s (*1.1*x více). Při asynchronním zápisu se ale pořadí obrátilo, nejlepšího výsledku dosáhlo prostředí Bun – *6.130*s, prostředí Node.js asynchronní zápis dokončilo v čase *7.078*s (*1.15*x více) a zdaleka nejhoršího výsledku dosáhlo prostředí Deno – *12.957*, což je *2.11*x více než v případě prostředí Bun.

4.4.2 zápis 100 souborů o velikosti 100MB

V tabulce 4.11 a na obrázku 4.10 jsou uvedeny naměřené hodnoty prvního výkonového testu – zápisu 100 souborů o velikosti 100 MB. (příkaz 3.23)

Tabulka 4.11: Doba zápisu 100 souborů o velikosti 100 MB

Serverové prostředí	Doba synchronního zápisu (s)	Doba asynchronního zápisu (s)
Node.js	4.559	7.425
Deno	4.836	10.615
Bun	4.821	6.187



Obrázek 4.10: Doba zápisu 100 souborů o velikosti 100 MB

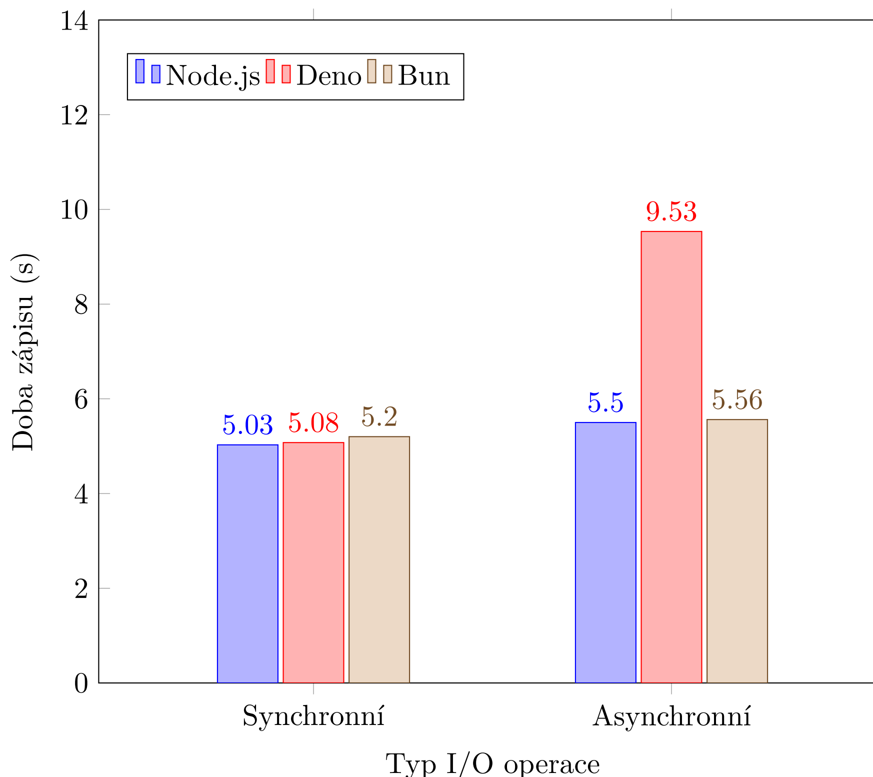
V případě synchronního zápisu si opět všechna běhová prostředí vedla velmi podobně. Nejlepší výsledek zaznamenalo prostředí Node.js a to *4.559*s, zatímco prostředí Bun *4.821*s (*1.06*x více) a prostředí prostředí Deno *4.836*s (*1.06*x více). Stejně jako v případě prvního testu, i v druhém testu dosáhlo nejlepšího výsledku při použití asynchronní metody zápisu prostředí Bun – *6.187*s, prostředí Node.js asynchronní zápis dokončilo v čase *7.425*s (*1.2*x více) a nejhoršího výsledku, ovšem již s menším odstupem, dosáhlo prostředí Deno – *10.62*s, což je *1.72*x více než v případě prostředí Bun.

4.4.3 zápis 1000 souborů o velikosti 10MB

V tabulce 4.12 a na obrázku 4.11 jsou uvedeny naměřené hodnoty třetího výkonového testu – zápisu 1000 souborů o velikosti 10 MB. (příkaz 3.24)

Tabulka 4.12: Doba zápisu 1000 souborů o velikosti 10 MB

Serverové prostředí	Doba synchronního zápisu (s)	Doba asynchronního zápisu (s)
Node.js	5.028	5.500
Deno	5.076	9.534
Bun	5.201	5.562

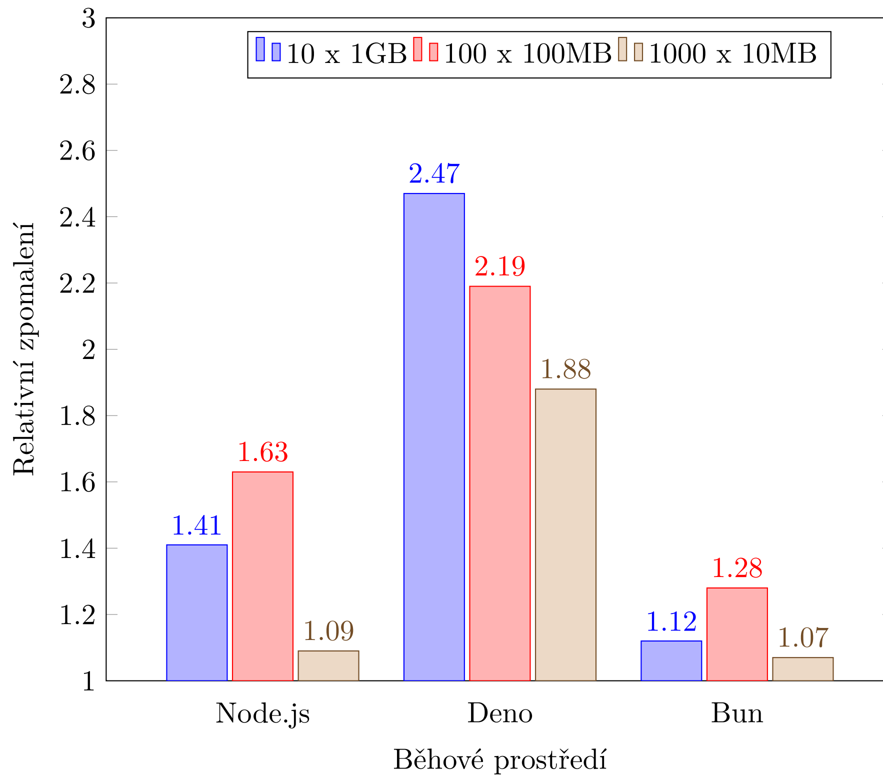


Obrázek 4.11: Doba zápisu 1000 souborů o velikosti 10 MB

I v posledním testu si v případě synchronního zápisu všechna běhová prostředí vedla velmi podobně. Nejlepší výsledek zaznamenalo prostředí Node.js a to *5.028* s, zatímco prostředí Deno *5.076* s (*1.01*x více) a prostředí Deno *5.201* s (*1.03*x více). V případě asynchronního zápisu si tentokrát překvapivě vedlo nejlépe prostředí Node.js s výsledkem *5.500* s, prostředí Bun zaznamenalo podobný výsledek – *5.562* s (*1.01*x více) a nejhoršího výsledku dosáhlo opět prostředí Deno – *9.534* s, což je *1.73*x více než v případě prostředí Node.js.

4.4.4 Porovnání synchronních a asynchronních variant

Na obrázku 4.12 je znázorněno relativní zpomalení asynchronního zápisu vzhledem k době synchronního zápisu. Čím menší hodnota, tím lepší efektivita asynchronní varianty v konkrétním běhovém prostředí.



Obrázek 4.12: Relativní zpomalení asynchronního zápisu vzhledem k synchronnímu zápisu pro dané prostředí a test

Ve všech výkonových testech byla nejefektivnější asynchronní varianta v prostředí Bun s průměrným relativním zpomalením *1.16*, následovalo prostředí Node.js s průměrnou hodnotou relativního zpomalení *1.38*. Nejhorších výsledků v každém z dílčích testů dosahovala asynchronní varianta v prostředí Deno. V průměru byla *2.18*x pomalejší než synchronní varianta.

4.5 Transpilace TypeScriptu

Předmětem výkonových testů bylo porovnat časy transpilace TypeScriptu v rámci startu programu. Pro každý dílčí test jsem změřil dvě hodnoty – dobu běhu TypeScript programu a dobu běhu JavaScript programu. TypeScript program je vykonáván následujícím způsobem: v první fázi se provede transpilace do JavaScriptu a ve druhé fázi samotné vykonání JavaScriptu. Doba strávená transpilací TypeScriptu se tedy získá odečtením naměřené doby běhu JavaScript programu od naměřené hodnoty doby běhu TypeScript programu. Při analýze výsledků jsem nehleděl na samotnou rychlost vykonávání JavaScript programu, nejednalo se totiž o porovnávanou vlastnost.

4.5.1 Hledání cesty v malém grafu

V tabulce 4.13 jsou uvedeny naměřené hodnoty prvního výkonového testu – hledaní nejkratší cesty v malém grafu.
Na obrázku 4.13 je znázorněn čas strávený transpilací v rámci startu programu. (příkaz 3.30)

Tabulka 4.13: Doba hledání nejkratší cesty v malém grafu

Serverové prostředí	Doba běhu TypeScript programu (ms)	Doba běhu JavaScript programu (ms)	Doba strávená transpilací (ms)
Node.js	1 158	34	1 124
Deno	99	19	80
Bun	76	16	60

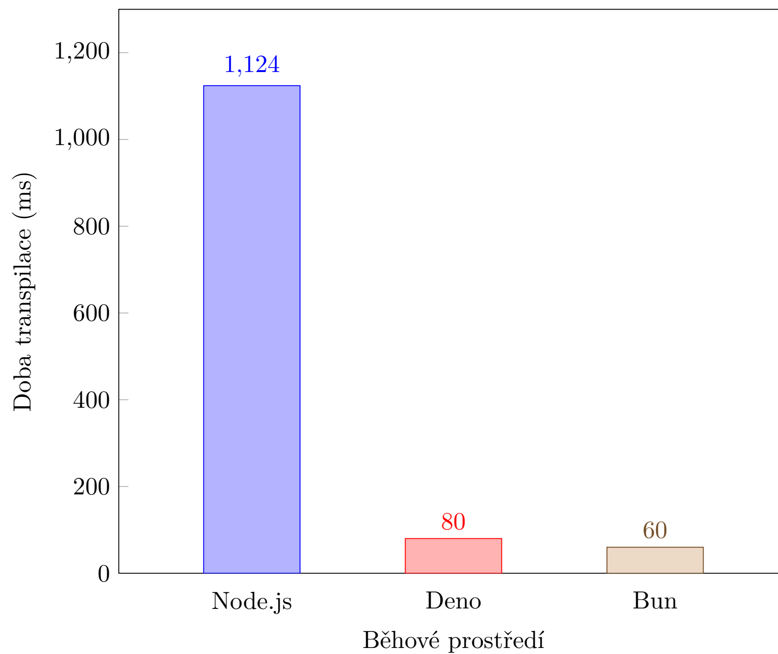
V prvním výkonovém testu si vedlo nejlépe prostředí Bun, ve kterém trvala transpilace TypeScriptu pouze *60* ms, zatímco v prostředí Deno hodnota činila *80* ms (*1.33*x více) a jednoznačně nejhoršího výsledku dosáhlo prostředí Node.js – *1158* ms, což je až *19.3*x více než v prostředí Bun.

4.5.2 Hledání cesty ve velkém grafu

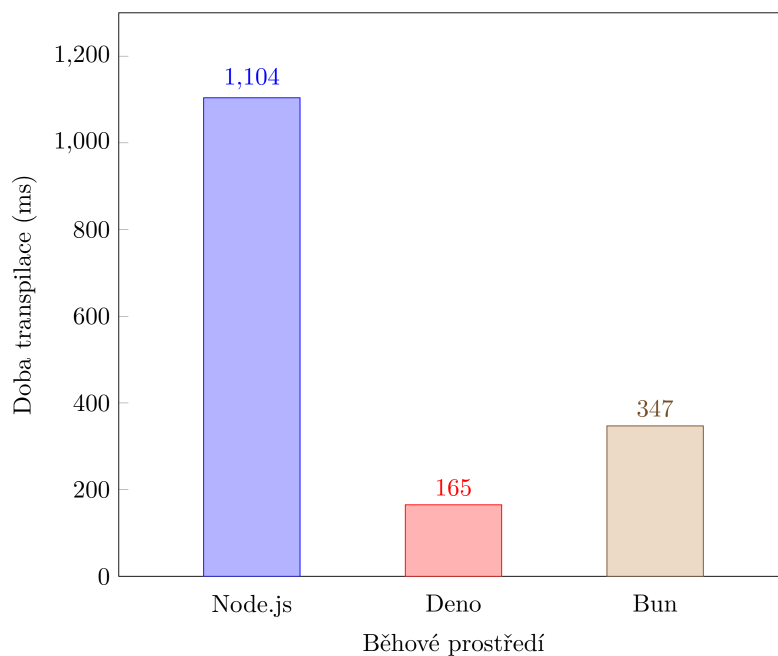
V tabulce 4.14 jsou uvedeny naměřené hodnoty druhého výkonového testu – hledaní nejkratší cesty ve velkém grafu. (příkaz 3.31)
Na obrázku 4.14 je znázorněn čas strávený transpilací v rámci startu programu.

Tabulka 4.14: Doba hledání nejkratší cesty ve velkém grafu

Serverové prostředí	Doba běhu TypeScript programu (ms)	Doba běhu JavaScript programu (ms)	Doba strávená transpilací (ms)
Node.js	1 666	562	1 104
Deno	685	520	165
Bun	1 622	1 275	347



Obrázek 4.13: Doby hledaní nejkratší cesty v malém grafu



Obrázek 4.14: Doby hledaní nejkratší cesty ve velkém grafu

Ve druhém výkonovém testu si vedlo nejlépe prostředí Deno, ve kterém trvala transpilace TypeScriptu pouze *165* ms. V prostředí Bun hodnota činila *347* ms (*2.1*x více) a jednoznačně nejhoršího výsledku opět dosáhlo prostředí Node.js – *1104* ms, což je *6.69*x více než v prostředí Deno.

4.6 Windows vs. Linux

Poslední zkoumanou oblastí v rámci výkonového testovaní běhových prostředí bylo porovnání výkonu distribucí pro různé operační systémy na stejném hardware. Konkrétně se jednalo o operační systémy Windows a Ubuntu (OS na bázi Linuxu). Při volbě vhodného výkonového testu jsem dbal především na rovné podmínky pro testování, jinými slovy, aby testované běhové prostředí při běhu na obou operačních systémech používalo skutečně identický hardware. Tato vlastnost není zaručena zejména při testování I/O operací, které jsem proto z nabízených možností vyloučil. Naopak jako nejvíce přímočará varianta se nabízí testování CPU, kterou jsem nakonec také zvolil.

4.6.1 Výpočet Fibonacciho posloupnosti

V tabulce 4.15 a na obrázku 4.15 jsou uvedeny naměřené hodnoty prvního výkonového testu – rekurzivního výpočtu 43. prvku Fibonacciho posloupnosti. (příkaz 3.14)

Tabulka 4.15: Doby výpočtu 43. prvku Fibonacciho posloupnosti na OS Windows a Ubuntu

Serverové prostředí	Windows (s)	Ubuntu (s)
Node.js	5.599	5.684
Deno	5.944	5.848
Bun	5.454	3.504

Zatímco prostředí Node.js a Deno v prvním výkonovém testu dosáhla srovnatelných výsledků na obou porovnávaných operačních systémech, prostředí Bun dosáhlo *1.56*x lepšího výsledků při běhu na operačním systému Ubuntu.

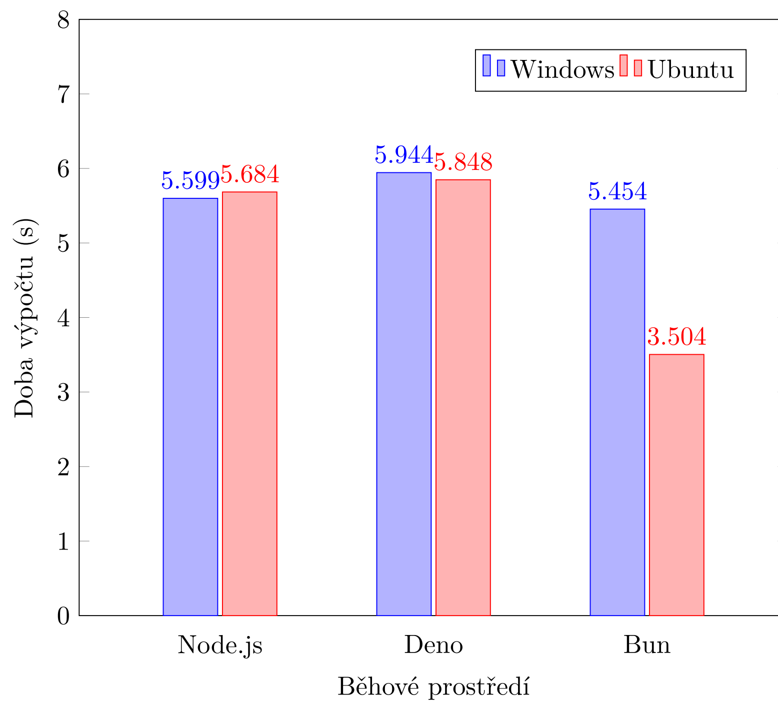
4.6.2 Násobení Matic

V tabulce 4.16 a na obrázku 4.16 jsou uvedeny naměřené hodnoty druhého výkonového testu – sekvenčního násobení 10x10 matic. (příkaz 3.14)

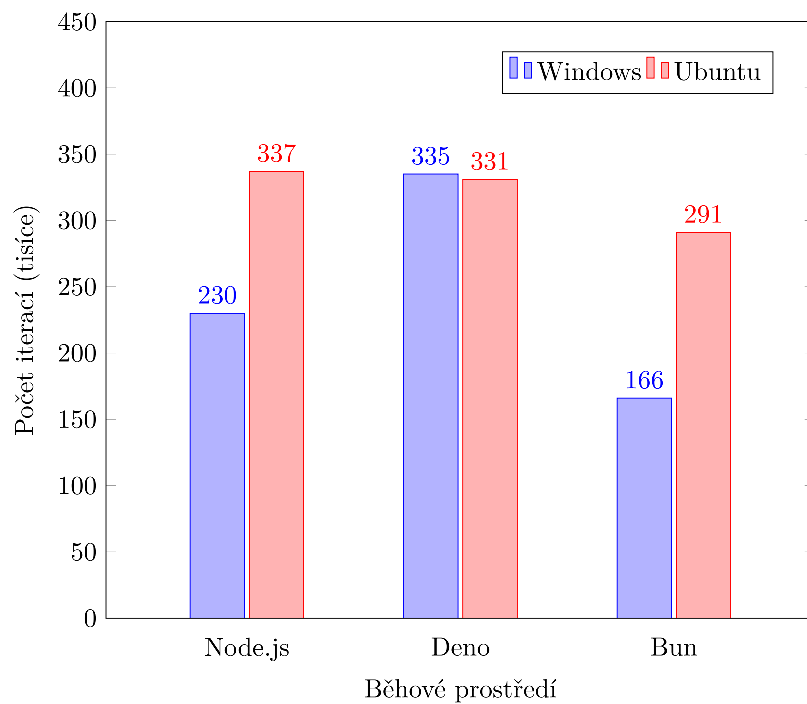
Tabulka 4.16: Počet iterací násobení 10x10 matic po dobu 1s na OS Windows a Ubuntu

Serverové prostředí	Windows	Ubuntu
Node.js	229 889	336 659
Deno	334 616	330 953
Bun	165 938	290 631

Ve druhém výkonovém testu dosáhlo srovnatelných výsledků na obou porovnávaných operačních systémech pouze prostředí Deno. Prostředí Node.js dosáhlo *1.46*x lepšího výsledků při běhu na operačním systému Ubuntu. Podobně si vedlo prostředí Bun, které dosáhlo dokonce *1.75*x lepšího výsledku na operačním systému Ubuntu.



Obrázek 4.15: Doby výpočtu 43. prvku Fibonacciho posloupnosti na OS Windows a Ubuntu



Obrázek 4.16: Počet iterací násobení 10x10 matic po dobu 1s na OS Windows a Ubuntu (zaokrouhleno na tisíce)

4.7 Doporučení

Za účelem volby ideálního běhového prostředí je nutné nejprve zvážit, které vlastnosti mají pro daný případ užití nejvyšší prioritu. Pro někoho, komu záleží pouze na zpracování síťových požadavků, je s velkou pravděpodobností ideálním běhovým prostředím Bun. Naopak, pokud někdo jiný hodlá násobit matice nebo vykonávat jiný sekvenční výpočet, prostředí Bun pro něho nebude nejlepší volbou.

Vytvořil jsem proto bodovou tabulku 4.17, ve které jsou shrnuty výsledky v každé oblasti pro každé testované prostředí. V každé oblasti je celkový nejlepší výsledek ohodnocen 10 body, ostatní běhová prostředí jsou ohodnocena poměrově k nejlepšímu. Pokud tedy například v jedné oblasti (pro kterou platí čím rychlejší, tím lepší) byla nejlepší naměřená hodnota *1* s, prostředí s naměřenou hodnotou *2* s získá 5 bodů z 10 možných, jelikož se jedná o 2x horší výsledek. V oblastech s více testy jsem hodnotu získal zprůměrováním bodů za dílčí testy.
Hodnoty v tabulce je třeba brát s rezervou, jelikož výsledky testů jsou ovlivněny testovacím prostředím (použitým hardware). Zároveň by ale měly poskytnout alespoň hrubou představu o rozdílech testovaných prostředí.

Tabulka 4.17: Celkové srovnání výkonu testovaných běhových prostředí

Serverové prostředí	Síť	CPU -rekurze	CPU -iterace	I/O - sync	I/O - async	Type-Script	Celkem
Node.js	7.1	5.2	10	10	9.0	1.0	42.3
Deno	8.4	5.2	9.4	9.6	5.4	10	48.0
Bun	10	10	7.3	9.4	10	7.0	53.7

V tabulce nejsou zohledněny rozdíly mezi běhovými prostředími na různých operačních systémech. Uvedené hodnoty reflektují situaci při použití operačního systému MacOS. Pro konkrétní server si člověk vybírá s běhovým prostředím i operační systém, na kterém server poběží. V posledním výkonovém testu jsem porovnal operační systémy Windows a Ubuntu (distribuce Linuxu). Z výsledků lze odvodit, že lepší volba je operační systém Ubuntu. Všechna běhová prostředí zaznamenala lepší nebo srovnatelný (rozdíl menší než *2* %) výsledek při běhu na OS Ubuntu. Při použití běhového prostředí Deno je rozdíl mezi operačnímy systémy Windows a Ubuntu minimální, pro prostředí Node.js je rozdíl výraznější(ve prospěch OS Ubuntu) a nejmarkantnější rozdíl je patrný při použití prostředí Bun, pro které se výrazně vyplatí použití OS Ubuntu.

Kapitola 5

Závěr

Cílem této práce bylo vzájemné porovnání JavaScript serverových běhových prostředí Node.js, Deno a Bun a to v rovině teoretické i praktické.

V teoretické části této práce byly analyzovány technologie, kterými jsou daná běhová prostředí implementována spolu s výčtem odlišností a společných rysů daných běhových prostředí.

V praktické části byl porovnán výkon běhových prostředí v různých oblastech. Konkrétně se jednalo o čtyři hlavní oblasti – zpracování síťových požadavků, zátěž procesoru, vstupně-výstupní operace a transpilace TypeScriptu. U vstupně-výstupních operací byly navíc rozlišovány synchronní a asynchronní způsoby zápisu. Součástí praktické části bylo i porovnání zátěže procesoru na různých operačních systémech, Windows a Linux, při použití stejného hardware.

Výsledky byly analyzovány jak slovně, tak pomocí grafů a tabulek. Nelze konstatovat, že by některé běhové prostředí dosáhlo nejlepších výsledků v každé ze zkoumaných oblastí. Pomocí relativizace výsledků lze ovšem dojít k závěru, že nejlepších výsledků dosáhlo prostředí Bun, následované prostředím Deno a nejhorších výsledků dosáhlo prostředí Node.js. Použití operačního systému Linux se prokázalo jako vhodnější oproti volbě operačního systému Windows.

Po přečtení práce by čtenář měl být srozuměn se způsobem vykonávání JavaScriptu na serveru a měl by získat povědomí o rozdílech mezi prostředími Node.js, Deno a Bun.

Hlavní přínos práce pro mě osobně spočíval v podrobném seznámení se s těmito prostředími a jejich specifickými vlastnostmi.

Literatura

[1] *PEYROTT, Sebastian. A Brief History of JavaScript* [online]. [cit. 2023-12-03]. Dostupné z: https://auth0.com/blog/a-brief-history-of-javascript/

[2] *"Industry leaders to advance standardization of Netscape's JavaScript at standards body meeting"* Press release 1996 [cit. 2023-12-03]. Dostupné z: https://web.archive.org/web/19981203070212/http:/cgi.netscape.com/newsref/pr/newsrelease289.html

[3] *Espruino.com* [online]. [cit. 2023-12-03]. Dostupné z: https://www.espruino.com

[4] Most used programming languages among developers worldwide as of 2023. *Statista.com* [online]. 2023 [cit. 2023-12-03]. Dostupné z: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[5] *SpiderMonkey (JavaScript Engine)* [online]. 2022 [cit. 2023-12-09]. Dostupné z: https://handwiki.org/wiki/Software:SpiderMonkey_(JavaScript_Engine)

[6] JavaScript. *Mozilla developer network web docs* [online]. [cit. 2023-12-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[7] Introduction to JavaScript. *Stanford University* [online]. 1998 [cit. 2023-12-03]. Dostupné z: https://web.stanford.edu/class/cs98si/slides/overview.html

[8] JavaScript technologies overview. *Mozilla developer network web docs* [online]. [cit. 2023-12-14]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview

[9] ROBERTS, Philip. *What the heck is the event loop anyway?* [online]. 2014 [cit. 2023-12-09]. Dostupné z: https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html

[10] *2023 Developer Survey* [online]. 2023 [cit. 2023-12-14]. Dostupné z: `https://survey.stackoverflow.co/2023/#web-frameworks-and-technologies`

[11] *Node.js v21.5.0 documentation* [online]. [cit. 2023-12-28]. Dostupné z: `https://nodejs.org/docs/latest/api/`

[12] DAHL, Ryan. *10 Things I Regret About Node.js - Ryan Dahl - JSConf EU* [online]. [cit. 2023-12-28]. Dostupné z: `https://www.youtube.com/watch?v=M3BM9TB-8yA`

[13] *Deno Manual - Permissions* [online]. [cit. 2023-12-28]. Dostupné z: `https://docs.deno.com/runtime/manual/basics/permissions`

[14] *Node.js with TypeScript* [online]. [cit. 2023-12-28]. Dostupné z: `https://nodejs.org/en/learn/getting-started/nodejs-with-typescript`

[15] *What is Bun* [online]. 2023 [cit. 2023-12-28]. Dostupné z: `https://bun.sh/docs`

[16] *HTTP Server APIs* [online]. [cit. 2024-01-04]. Dostupné z: `https://docs.deno.com/runtime/manual/runtime/http_server_apis`

[17] *What is network latency?* [online]. [cit. 2024-01-07]. Dostupné z: `https://aws.amazon.com/what-is/latency/`

[18] *Go-wrk* [online]. [cit. 2024-01-08]. Dostupné z: `https://github.com/tsliwowicz/go-wrk`

[19] *HTTPBenchmarkTool.jl* [online]. [cit. 2024-01-08]. Dostupné z: `https://github.com/ribardej/HTTPBenchmarkTool.jl`

[20] *Textová reprezentace použitých grafů* [online]. [cit. 2024-05-01]. Dostupné z: `https://github.com/ribardej/BachelorThesis/tree/main/typescript`

[21] Dijkstra, E. W. [cit. 2024-05-01] *A note on two problems in connexion with graphs.* Numerische Mathematik 1, 1959, s. 269–271.