Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Porovnání paralelní implementace B algoritmu v Javě a C/C++

Plzeň 2024 Petr Penička

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Petr PERNIČKA**
Osobní číslo:	**A20B0203P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Porovnání paralelní implementace B algoritmu v Javě a C/C++**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s Traffic Assignment Problem (TAP) pro určování zaplněnosti silnic v dopravní síti.
2. Seznamte se s B algoritmem a jeho vlastnostmi.
3. Navrhněte paralelní implementaci B algoritmu pro jazyk Java a C/C++.
4. Navržené řešení implementujte v obou jazycích.
5. Obě implementované verze otestujte pomocí netriviální sady testů a vzájemně porovnejte.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Tomáš Potužák, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **2. října 2023**
Termín odevzdání bakalářské práce: **2. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2023

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 22. dubna 2024

Petr Penička

Abstract

One of the techniques used for traffic forecasting is the four-step model, part of which is traffic assignment. Traffic assignment comes after determining the traffic flow from each origin to each destination in the traffic network taking the form of OD matrix and it entails assignment of this flow to each link in the network so that trips would use cheapest paths. Algorithm B solves static variant of traffic assignment in which elements of the OD matrix are constants and the result represents an average over considered time frame. In this thesis, Algorithm B is parallelized and implemented in Java and C++. Then, the assigned flows, the computation time and memory consumption of both versions are compared. The result is that C++ is ~8% faster and needs ~4% less memory. The parallelization alone is able to speed up the algoritm by ~80% on used computers.

Abstrakt

Jedna z používaných technik pro předpověď dopravy je čtyřkrokový model, jehož součástí je i přiřazení dopravy. Přiřazení dopravy přichází poté, co je již určen dopravní tok z každého počátku do každého cíle v dopravní síti ve formě OD matice a obnáší přiřazení tohoto toku jednotlivým silnicím v síti tak, aby jízdy používaly nejlevnější trasy. Algoritmus B řeší statickou variantu přiřazení dopravy, kdy prvky OD matice jsou konstanty a řešení pak reprezentuje průměr přes modelovaný časový úsek. V této práci je Algoritmus B paralelizován a implementován v Javě a C++ a následně jsou porovnány přiřazené toky, čas a paměť. Výsledkem práce je, že C++ je o ~8% rychlejší a potřebuje o ~4% méně paměti. Samotná paralelizace pak dokáže na použitých počítačích urychlit algoritmus až o ~80%.

Poděkování

Tímto bych rád poděkoval panu Ing. Tomášovi Potužákovi, Ph.D. za odborné vedení práce, cenné rady a čas, který mi ochotně věnoval. Poděkování patří i panu Doc. Ing. Tomášovi Koutnému, Ph.D. za cenné připomínky ohledně C++.

Obsah

1 Úvod

Předpověď dopravy je pokus o odhadnutí počtu vozidel či cestujících, kteří chtějí použít dopravní síť k přepravě mezi různými místy zájmu. Tato oblast dopravního plánování je čím dál tím důležitější pro budoucí vylepšování dopravních sítí, zvláště se stále rostoucím počtem účastníků provozu. Jedním z modelů určující metodiku pro předpověď dopravy je tzv. čtyřkrokový model, jehož posledním krokem je problém přiřazení dopravy. První a nejznámější algoritmus řešící statickou variantu tohoto problému je znám již od padesátých let 20. století, avšak ten použitelné řešení nachází jen pro malé a nezahlcené sítě. S většími a stále více zahlcenými dopravními sítěmi, se kterými se dnes lidé zabývající se dopravním plánováním setkávají, přichází i potřeba pro rychlejší a lepší alternativy řešící tento problém. Algoritmus B, kterým se tato práce zabývá, je jednou z takových alternativ.

Cílem této práce je představit teoretické základy problému přiřazení dopravy jakožto prerekvizity k Algoritmu B a následné popsání samotného algoritmu. Dále je cílem implementování algoritmu v jazycích Java a C++ a navržení a implementování jeho paralelizace pro dnešní vícejádrové procesory. Hlavním bodem zadání je pak porovnat vlastnosti obou implementací algoritmu.

2 Problém přiřazení dopravy

Tato kapitola se zabývá teoretickými prerekvizitami Algoritmu B, který je jedním z algoritmů řešících *problém přiřazení dopravy* (angl. *traffic assignment problem*, TAP). Je zde ve stručnosti popsán model pro předpověď dopravy, jehož součástí je právě přiřazení dopravy. Dále jsou zde uvedeny principy popisující chování cestujících a statická i dynamická verze přiřazení dopravy.

2.1 Čtyřkrokový model

V roce 1954 vydali R. B. Mitchell a C. Rapkin první studii [1] pokládající základy dopravní analýzy. V této práci byl představen tzv. *čtyřkrokový model* (angl. *four step model*, 4SM), jenž určuje metodiku pro identifikaci a rozložení dopravních požadavků generovaných jednotlivci, kteří jsou motivováni se přesouvat mezi body zájmů na základě sociálních, ekonomických či kulturních aktivit. Přiřazení dopravy je posledním krokem tohoto modelu [2].

2.1.1 Základní myšlenka

Základní jednotkou dopravy v 4SM je *jízda* (angl. *trip*), která je uvažována na úrovni domácností a jejíž účelem je agregovat jednotlivé cestující do svazku putujícího z jednoho bodu dopravní sítě do jiného. Důležitým předpokladem 4SM je, že jízdy jsou uvažovány v době, pro kterou je uvažováno plánování (např. ranní dopravní špička). Dále je předpokládáno, že se model aplikuje ve velkém měřítku, nepočítá tedy např. s křižovatkou mezi městskými bloky. Ta by totiž byla společně s okolím zahrnuta do *zóny*. Zóny jsou prostorová data, agregovaná právě kvůli tomuto předpokladu (a také kvůli výpočetní složitosti). Zóna reprezentuje např. menší obec, část města či sousedství [2].

2.1.2 Popis kroků

Prvním krok čtyřkrokového modelu je *generování jízd*, kdy jsou identifikovány počty jízd, které začínají a končí v každé ze zón. Na jeho konci známe rozsah provozu, který bude (v uvažované době) využívat dopravní síť. Druhý krok, *distribuce jízd*, určuje kolik jízd začínajících v určité zóně končí v jiné

určité zóně. Výstupem je *matice počátků a cílů* (angl. *origin-destination matrix*, dále jen *OD matice* či ODM), kde prvek v řádku i a sloupci j je počet jízd začínajících v zóně Z_i a končících v zóně Z_j. Třetí krok, *volba způsobu* (angl. *mode choice/modal split*), se zabývá různými způsoby dopravy, mezi kterými cestující volí. OD matice je v tomto kroku rozdělena podle těchto způsobů. Jak již bylo zmíněno, čtvrtý krok je *přiřazení dopravy* [2].

2.2 Přiřazení dopravy a Wardropova kritéria

V tomto bodě již známe počty jízd z, resp. do jednotlivých zón pro určitý způsob dopravy. Přiřazení dopravy se zabývá zjišťováním, jaké *trasy* (angl. *routes*) cestující zvolí. Jízdy jsou tedy v tomto kroku přiřazeny trasám (čili posloupnostem *silnic*, angl. *link*) v dopravní síti. Je logické, že si cestující zvolí trasu, která je v nějakém smyslu *nejlevnější* a cena by zde mohla reprezentovat např. čas, který trvá trasu překonat [2].

2.2.1 Wardropova kritéria

Základní principy, popisující chování a volbu trasy jednotlivých cestujících, byly formulovány Johnem Wardropem v roce 1952. V jeho práci Wardrop uvedl dvě kritéria pro distribuci dopravy na dopravní síť s alternativními trasami [3]:

- *Uživatelská rovnováha*: Doby jízdy[1] na všech využívaných trasách jsou rovny a/nebo menší než doby, které by byly prožité jediným vozidlem na jakékoli nevyužívané trase.
- *Systémové optimum*: Průměrná doba jízdy je minimální.

2.2.2 Popis kritérií

Uživatelská rovnováha (angl. *user equilibrium*, UE) nastává tehdy, když žádný cestující nemůže snížit dobu své jízdy volbou alternativní trasy. V tomto kritériu neexistuje kooperace mezi jednotlivci a všichni jednotlivci volí trasy sobecky a racionálně. V reálné dopravě se očekává, že budou všichni následovat toto kritérium [2].

[1] Dobou jízdy se rozumí časový úsek mezi výjezdem ze startovní zóny a příjezdem do cílové zóny. Různé trasy pro danou jízdu vyústí v různé doby.

Naopak je nepravděpodobné, že by samovolně nastalo systémové optimum (angl. *system optimum*, SO). V tomto scénáři někteří jednotlivci neputují po nejlevnějších trasách, ale kooperují a volí trasy tak, že je celá dopravní síť využívána co nejefektivněji. SO by se dalo realizovat např. pomocí centrálního navigačního systému [2].

Obě zmíněná kritéria jsou poněkud idealistická. Předpokládají, že cestující mají perfektní informace o topologii sítě a stavu dopravy na ní. Cestující se tak ale zcela nechovají. Nemají kompletní přehled o topologii a hlavně aktuálním stavu sítě a mají tendenci volit své obvyklé trasy, které znají nejlépe a umějí odhadnout dobu jízdy. Tyto případy jsou však potlačovány inteligentními plánovači tras, které jsou dnes nezřídka používány [2].

2.3 Statické přiřazení dopravy

Standardní přístupy k přiřazování dopravy předpokládají, že se doprava chová podle jednoho z Wardropových kritérií a dělí se na statické a dynamické. *Statické přiřazení dopravy* (angl. *static traffic assignment*, STA) se od dynamického liší tím, že pracuje s konstantními dopravními požadavky i dopravní infrastrukturou. To znamená, že OD matice i parametry sítě jsou časově nezávislé. Typicky se tak síti přiřazuje tok z OD matice vytvořené během krátké doby (např. ranní špičky) a přiřazené toky jsou pak průměrem pro danou periodu [2].

2.3.1 Přiřazení *všechno-nebo-nic*

Nejprimitivnější metoda STA se nazývá *přiřazení všechno-nebo-nic* (angl. *all-or-nothing assignment*, AON). AON přiřadí každou jízdu ze zóny Z_i do zóny Z_j těm silnicím, které tvoří nejkratší trasu mezi těmito dvěma zónami. Ačkoliv je výsledek takového přiřazení v praxi nepoužitelný, používá se AON k inicializaci pokročilejších metod [2].

AON nepočítá s kapacitami silnic. To znamená, že není brán v potaz fakt, že zvýšení toku dopravy na silnici vyústí ve zvýšení doby jízdy po této silnici. Metody, které s kapacitami počítají, využívají předpokladu, že se cestující řídí podle Wardropovy uživatelské rovnováhy a nepřiřazují všechny jízdy ze Z_i do Z_j té nejkratší trase, ale rozdělí dopravu mezi více tras, protože jakmile je kapacita nejkratší trasy do určité míry naplněna, není už tato trasa ta nejlevnější [2].

2.3.2 Cenové funkce

Důležitým aspektem těchto metod jsou tzv. *cenové funkce* (v angličtině *travel time functions, volume delay functions, link impedance functions, link performance functions*). Tyto funkce určují cenu trasy nebo silnice s ohledem na kapacitu a aktuální tok. Nejpoužívanější takovou funkcí je BPR (Bureau of Public Roads) funkce, která v obecném tvaru vypadá [4]:

$$c(x) = f \cdot \left(1 + \alpha \left(\frac{x}{K}\right)^{\beta}\right), \tag{2.1}$$

kde α a β jsou koeficienty určující tvar funkce a $c(x)$ je čas potřebný k překonání silnice (neboli cena silnice) s aktuálním dopravním tokem x. Konstanty f a K jsou pak parametry této silnice. Parametr f je *čas volného toku*, tedy čas překonání silnice s nulovým aktuálním tokem. Parametr K je *praktická kapacita* silnice. Je to zlomový bod, kolem kterého se začne cena silnice značně zvyšovat [2].

S využitím cenových funkcí STA algoritmy iterativně přiřazují dopravu — s každou iterací se toky víc a víc blíží uživatelské rovnováze, čímž se minimalizuje tzv. *objektivní funkce*. V roce 1955 totiž Beckmann a další [5] formulovali uživatelskou rovnováhu jakožto konvexní problém skládající se z této objektivní funkce a nerovností/podmínek [2]. Pro řešení tohoto problému existuje několik algoritmů.

2.3.3 Rozdělení STA algoritmů

STA algoritmy se dají rozdělit do tří kategorií. Algoritmy *založené na silnicích* (angl. *link-based*) využívají velmi málo paměti, ale také velmi špatně konvergují. Nejznámější takový algoritmus (a také první STA algoritmus vůbec) je Frankové-Wolfeův (FW) algoritmus, který obecně slouží pro optimalizaci omezených konvexních problémů. Další algoritmy tohoto typu jsou modifikace FW [6].

V algoritmech *založených na trasách* (angl. *path-based*) je problém rozdělen na páry počátků a cílů, kdy se pro každý takový pár zón vytvoří množina tras vedoucích mezi nimi. V této množině je tok je následně přesouván z dražších tras na ty levnější. Trasové algoritmy potřebují hodně paměti, ale také jsou rychlejší než algoritmy založené na silnicích [6].

Nejmladší kategorií jsou algoritmy *založené na keřích* (angl. *bush-based* či *origin-based*). Ty pro každou zónu vytvoří z dopravní sítě tzv. keř. Keř je acyklický podgraf obsahující jen toky, začínající v dané zóně. Keř se následně vyváží tak, že pro každou zónu v keři nalezneme nejlevnější a nejdražší trasu

k ní a následně přesuneme tok z dražší trasy do té levnější, aby si byly rovny [6].

2.4 Dynamické přiřazení dopravy

STA se spoléhá na cenové funkce, s čímž přichází i problém s konceptem kapacity silnice. Zahlcení dopravní sítě je však dynamický jev, jehož časová závislost není zanedbatelná. Proto existují metody *dynamického přiřazení dopravy* (angl. *dynamic traffic assignment*, DTA). Ty byly vytvořeny s myšlenkou, že nahromadění (*buildup*) a rozpuštění (*dissolution*) dopravy v síti hrají důležitou roli a že by se mělo počítat s historií dopravy na síti [2].

Dynamické přiřazení dopravy je časově závislým rozšířením STA, kdy prvky OD matice už nejsou konstanty, ale funkce času/časové posloupnosti. Stejně tak mohou být časově závislé i parametry dopravní sítě. Na rozdíl od STA pro tento problém neexistuje obecná matematická definice. DTA se dá rozdělit na dva podmodely — *dynamické plnění sítě* (angl. *dynamic network loading*, DNL) a *dynamickou uživatelskou rovnováhu* (angl. *dynamic user equilibrium*, DUE) [6].

2.4.1 Dynamické plnění sítě

Model DNL se zabývá načítáním dopravy na síť. Vstupem je množina všech tras v síti a pro každou trasu časově závislá funkce reprezentující cestující, kteří na tuto trasu v daný čas vyjíždějí. Výstupem je funkce času pro každou trasu reprezentující cenu trasy v daný čas. Modely dopravního toku se v DNL dělí na mikroskopické, kdy se modeluje každé vozidlo, méně detailní mezoskopické a nejobecnější makroskopické [6].

2.4.2 Dynamická uživatelská rovnováha

Dynamická uživatelská rovnováha je časovým rozšířením statického Wardropova prvního kritéria. UE může být takto rozšířena více způsoby. Prvním a nejjednodušším rozšířením je *trasově volená* DUE (angl. *route-choice*, RC). Ta vyžaduje aby Wardropova UE byla splněna pro každý časový okamžik. Dalším rozšířením je např. SRDT DUE (angl. *simultaneous route-and-departure-time DUE*), která navíc bere v potaz variabilitu časů odjezdů [6].

3 Algoritmus B

Algoritmus B je iterativní grafový algoritmus pro statické přiřazení dopravy a dosažení uživatelské rovnováhy. Spadá do kategorie STA algoritmů založených na keřích. Stejně jako u ostatních STA algoritmů, je jeho vstupem dopravní síť a OD matice. Výstupem jsou rovnovážné dopravní toky na jednotlivých silnicích.

Nejznámější STA algoritmus je již zmíněný FW algoritmus. Ten se jednoduše implementuje a vyžaduje jen velmi málo paměti, jak vyžadovaly počítače v době jeho představení (rok 1973). Problém FW je, že se po několika iteracích rychlost konvergence značně zpomalí a algoritmus nikdy nedosáhne UE. Až na velmi malé či téměř nezahlcené sítě je výsledkem FW jen hrubá aproximace. B byl vytvořen jakožto jedna z alternativ, která je rychlejší a informativnější pro velké a zahlcené dopravní sítě, která zároveň využije rychlé procesory a velké paměti moderních počítačů [7].

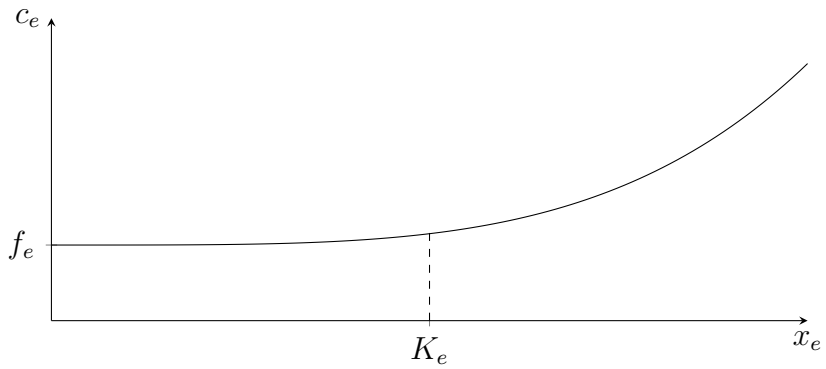
3.1 Matematické pozadí

V této části jsou nejprve definované matematické ekvivalenty pro termíny uvedené v Kapitole 2. Dále je zde do větší hloubky rozvedena BPR funkce, matematický model pro uživatelskou rovnováhu a další matematické pojmy důležité pro Algoritmus B a STA.

- Dopravní síť je reprezentována souvislým orientovaným grafem $G = (V, E)$, kde V je množina vrcholů a $E \subseteq V \times V$ je množina orientovaných hran.
- Hrana reprezentuje silnici v dopravní síti.
- Proměnná x_e je tok na hraně $e \in E$, neboli počet jízd na příslušné silnici.
- Množina $Z \subseteq V$ je množina zón. Ne každý vrchol musí být zónou, některé vrcholy slouží např. jako křižovatky.
- Množina P_{ij} všech cest na grafu G ze zóny $z_i \in Z$ do zóny $z_j \in Z$ pak reprezentuje trasy mezi danými zónami.
- Proměnná x_p je tok na cestě $p \in P_{ij}$ a reprezentuje počet jízd ze z_i do z_j na příslušné trase. Zvýší-li se tok na cestě p o x_0, pak se zvýší toky x_e na všech hranách této cesty ($e \in p$) o x_0.

- OD matice, značená O, je čtvercová matice řádu $|Z|$. Prvek O_{ij} je tok ze zóny z_i do zóny z_j.
- Cena hrany, resp. cesty je synonymem k době potřebné k překonání silnice, resp. trasy.

3.1.1 BPR funkce



Obrázek 3.1: Graf BPR funkce

Obecný předpis pro BPR funkci je (2.1). Za koeficienty α a β jsou zpravidla dosazována čísla 0.15 a 4:

$$c_e(x_e) = f_e \left(1 + 0.15 \left(\frac{x_e}{K_e}\right)^4\right), \tag{3.1}$$

kde c_e je cena hrany e, na které teče tok x_e. Analogicky, f_e a K_e jsou již zmíněné parametry hrany e (čas volného toku a praktická kapacita). Na Obrázku 3.1 je příklad BPR funkce. Je patrné, že po překročení praktické kapacity se cena začne značně zvyšovat [4].

Cena cesty p je definována jako:

$$c_p = \sum_{e \in p} c_e(x_e). \tag{3.2}$$

Doba potřebná pro překonání trasy je součtem dob potřebných k překonání jednotlivých silnic, které trasu tvoří.

Derivace BPR funkce podle toku je:

$$c'_e(x_e) = 0.6 \cdot f_e \cdot \frac{x_e^3}{K_e^4}. \tag{3.3}$$

Součet derivací cen hran cesty p:

$$c'_p = \sum_{e \in p} c'_e(x_e). \tag{3.4}$$

V další kapitole bude ještě užitečný vzorec:

$$\int_0^{x_e} c_e(t)\,dt = f_e \cdot x_e \cdot \left(1 + 0.03\left(\frac{x_e}{K_e}\right)^4\right). \tag{3.5}$$

3.1.2 Matematický model UE

V Kapitole 2.3.2 bylo zmíněno, že Beckmann a další formulovali uživatelskou rovnováhu jakožto řešení optimalizačního problému. Konkrétně jako

$$\min \sum_{e \in E} \int_0^{x_e} c_e(t)dt \tag{3.6}$$

s podmínkami

$$x_e = \sum_{z_i \in Z} \sum_{z_j \in Z} \sum_{p \in P_{ij}} \alpha_p^e\, x_p, \tag{3.7}$$

$$\forall z_i, z_j \in Z : \sum_{p \in P_{ij}} x_p = O_{ij}, \tag{3.8}$$

$$\forall z_i, z_j \in Z\, \forall p \in P_{ij} : x_p \geq 0, \tag{3.9}$$

$$\forall e \in E : x_e \geq 0. \tag{3.10}$$

V podmínce (3.7)

$$\alpha_p^e = \begin{cases} 1 & e \in p \\ 0 & e \notin p \end{cases}$$

[5].

Podmínka (3.7) vyjadřuje, že tok na hraně je roven součtu toků všech cest, jež tuto hranu obsahují. Podmínka (3.8) znamená, že tok ze zóny z_i do zóny z_j je rozdělen mezi všechny cesty začínající v z_i a končící v z_j. Podmínky (3.9) a (3.10) pak zajišťují nezáporné toky.

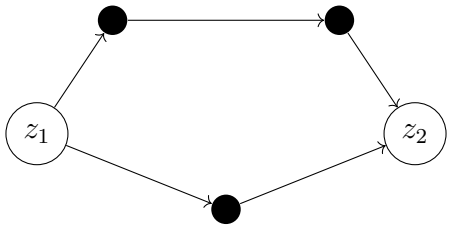
Samotný problém je tedy nalezení minima objektivní funkce

$$of = \sum_{e \in E} \int_0^{x_e} c_e(t)dt, \tag{3.11}$$

kde c_e je cenová funkce. Objektivní funkce je konvexní, pokud je do ní dosazená rostoucí cenová funkce. Uživatelská rovnováha má potom jednoznačné řešení, což je vlastnost, kterou požadujeme pro správné přiřazení dopravy a dále v textu bude tato vlastnost implicitně předpokládána [5].

Po dosazení integrálu BPR funkce (3.5) má objektivní funkce tvar:

$$of = \sum_{e \in E} f_e \cdot x_e \cdot \left(1 + 0.03\left(\frac{x_e}{K_e}\right)^4\right). \tag{3.12}$$



Obrázek 3.2: Ukázka případu se dvěma cestami.

3.1.3 Případ se dvěma cestami

Jednoduchý, avšak jak se ukáže, tak pro Algoritmus B kritický, je případ, kdy se snažíme dosáhnout uživatelské rovnováhy mezi dvěma cestami, které začínají a končí ve stejných zónách a nesdílejí žádné hrany ani vrcholy mezi počátkem a koncem. Cesty si označíme $\underline{p}$ a $\overline{p}$. Tok mezi zónami bude značen O_{12} (platí $x_{\underline{p}} + x_{\overline{p}} = O_{12}$). Objektivní funkce (3.11) pro dvě cesty bude

$$\begin{aligned} of(x_{\underline{p}}) &= \sum_{e \in E} \int_0^{x_e} c_e(t)dt \\ &= \sum_{e \in \underline{p}} \int_0^{x_{\underline{p}}} c_e(t)dt + \sum_{e \in \overline{p}} \int_0^{x_{\overline{p}}} c_e(t)dt \\ &= \int_0^{x_{\underline{p}}} c_{\underline{p}}(t)dt + \int_0^{O_{12}-x_{\underline{p}}} c_{\overline{p}}(t)dt. \end{aligned} \tag{3.13}$$

Víme, že tato funkce je konvexní, má tedy jediný stacionární bod — minimum. Ten najdeme s pomocí samotné definice stacionárních bodů:

$$\frac{d}{dx_{\underline{p}}} of(x^*_{\underline{p}}) = 0, \tag{3.14}$$

kde $x^*_{\underline{p}}$ je argument minima of [7].

Derivujeme objektivní funkci:

$$\begin{aligned} \frac{d}{dx_{\underline{p}}} of(x_{\underline{p}}) &= \frac{d}{dx_{\underline{p}}} \int_0^{x_{\underline{p}}} c_{\underline{p}}(t)dt + \frac{d}{dx_{\underline{p}}} \int_0^{O_{12}-x_{\underline{p}}} c_{\overline{p}}(t)dt \\ &= c_{\underline{p}}(x_{\underline{p}}) - c_{\overline{p}}(O_{12} - x_{\underline{p}}). \end{aligned} \tag{3.15}$$

Uživatelská rovnováha v případu se dvěma cestami nastane právě tehdy, když:

$$\begin{aligned} \arg\min of(x_{\underline{p}}) = x^*_{\underline{p}} \Leftrightarrow c_{\underline{p}}(x^*_{\underline{p}}) - c_{\overline{p}}(O_{12} - x^*_{\underline{p}}) &= 0 \\ c_{\underline{p}}(x^*_{\underline{p}}) - c_{\overline{p}}(x^*_{\overline{p}}) &= 0 \\ c_{\underline{p}}(x^*_{\underline{p}}) &= c_{\overline{p}}(x^*_{\overline{p}}). \end{aligned} \tag{3.16}$$

Výsledkem je závěr, že UE nastane právě tehdy, když je tok O_{12} rozdělen mezi cesty $\underline{p}$ a $\overline{p}$ tak, že se ceny těchto cest rovnají.

Následně lze přikročit k nalezení rovnováhy. Uvažujme počáteční stav případu se dvěma cestami, kdy na cestě $\underline{p}$ teče tok $x^0_{\underline{p}}$ a na cestě $\overline{p}$ teče tok $x^0_{\overline{p}}$. Nejsou-li tyto cesty v rovnováze, můžeme bez újmy na obecnosti říct, že cena jedné cesty bude nižší než cena té druhé:

$$c_{\underline{p}}(x^0_{\underline{p}}) < c_{\overline{p}}(x^0_{\overline{p}}). \tag{3.17}$$

Chceme najít takový tok Δx, který přesuneme z dražší cesty do té levnější tak, aby nastala UE (3.16):

$$c_{\underline{p}}(x^0_{\underline{p}} + \Delta x) = c_{\overline{p}}(x^0_{\overline{p}} - \Delta x), \tag{3.18}$$

$$0 \leq \Delta x \leq \mu. \tag{3.19}$$

Aby se předešlo negativním tokům na hranách, musí být přesouvaný tok nezáporný a menší nebo roven nejmenšímu (počátečnímu) toku na hranách dražší cesty:

$$\mu = \min\{\, x^0_e \,|\, e \in \overline{p}\,\} \tag{3.20}$$

[7].

Alternativně se nalezení Δx dá formulovat jako nalezení kořenu funkce

$$f(\Delta x) = c_{\overline{p}}(x^0_{\overline{p}} - \Delta x) - c_{\underline{p}}(x^0_{\underline{p}} + \Delta x). \tag{3.21}$$

V Algoritmu B se toto realizuje pomocí Newtonovy metody:

$$\begin{aligned} \Delta x_{i+1} &= \Delta x_i - \frac{f(\Delta x)}{f'(\Delta x)} \\ &= \Delta x_i - \frac{c_{\overline{p}}(x^0_{\overline{p}} - \Delta x_i) - c_{\underline{p}}(x^0_{\underline{p}} + \Delta x_i)}{-c'_{\overline{p}}(x^0_{\overline{p}} - \Delta x_i) - c'_{\underline{p}}(x^0_{\underline{p}} + \Delta x_i)} \\ &= \Delta x_i + \frac{c_{\overline{p}}(x^0_{\overline{p}} - \Delta x_i) - c_{\underline{p}}(x^0_{\underline{p}} + \Delta x_i)}{c'_{\overline{p}}(x^0_{\overline{p}} - \Delta x_i) + c'_{\underline{p}}(x^0_{\underline{p}} + \Delta x_i)}, \end{aligned} \tag{3.22}$$

kde c'_p je derivace ceny cesty (3.4).

3.1.4 Relativní mezera

Ke zjištění, zda je řešení poskytnuté STA algoritmem dostatečně blízko optimálnímu řešení, či zda je potřeba další iterace, se používá *relativní mezera* (angl. *relative gap*). *Mezera* v k-té iteraci je nekladná hodnota definována jako

$$g_k = \sum_{e \in E} c_e(x^k_e) \cdot \left(y^k_e - x^k_e\right) \leq 0, \tag{3.23}$$

kde $c_e(x_e^k)$ je cena hrany e s tokem x_e^k a x_e^k je tok, jež byl algoritmem hraně e přiřazen v k-té iteraci. y_e^k je tok na hraně e přiřazený AON algoritmem (Kapitola 2.3), který přiřazuje nejkratším cestám s ohledem na ceny $\{c_e(x_e^k)\}$ v k-té iteraci. *Spodní hranice* (angl. *lower bound*) je součet mezery s hodnotou objektivní funkce v k-té iteraci

$$\begin{aligned} lb_k &= g_k + of_k \\ &= \sum_{e \in E} c_e(x_e^k) \cdot \left(y_e^k - x_e^k\right) + \sum_{e \in E} \int_0^{x_e^k} c_e(t)\, dt. \end{aligned} \tag{3.24}$$

Maximální spodní hranice za všechny proběhlé iterace:

$$blb_k = \max\{\, lb_i \,|\, i = 1, \ldots, k \,\}. \tag{3.25}$$

Relativní mezera v k-té iteraci je nezáporná hodnota rovná negativnímu poměru mezery a absolutní hodnoty maximální spodní hranice

$$rg_k = -\frac{g_k}{|blb_k|} \geq 0 \tag{3.26}$$

[8]. V praxi se považuje řešení s relativní mezerou 0.01 jako přijatelné a 0.0001 jako výborné [7].

3.2 Popis algoritmu

Efektivita B algoritmu vychází z rozložení cyklické sítě na acyklické podgrafy, na kterých je hledání minimálních a maximálních cest výpočetně jednodušší. Tyto podgrafy se nazývají keře. Konkrétněji, keř je faktor[1] souvislého orientovaného grafu, který je acyklický a který má právě jeden kořen (tedy vrchol, do kterého nevstupují žádné hrany) [7].

Pro správné fungování B algoritmu musí být splněno pět předpokladů:

1. Dopravní požadavky[2] musí být konstantní a nezáporné.
2. Cena hrany musí být neklesající kladná spojitá funkce toku na této hraně.[3]
3. Cena cesty je součet cen hran obsažených v této cestě.[4]
4. Toky na hranách jsou nezáporná reálná čísla.

[1] Faktor grafu je takový podgraf, který obsahuje všechny vrcholy originálního grafu.
[2] Tedy prvky OD matice.
[3] BPR funkce tato kritéria splňuje.
[4] Vzorec (3.2).

5. Pro každou hranu v síti z vrcholu i do vrcholu j existuje v síti hrana z j do i.[5]

Zde popsaný B algoritmus se mírně liší od jeho originální verze z [7] v iteračním kroku. Tato verze je popsána v [9] a probíhá takto:

1. Inicializace
2. Iterace
   - Pro každý keř:
     i. Zlepšení topologie keře
     ii. Nalezení minimálního a maximálního stromu
     iii. Vyvážení keře
     iv. Odstranění hran s nulovými toky
   - Terminace

3.2.1 Inicializace

Ze všeho nejdříve se vytvoří keř K_z pro každou zónu $z \in Z$. V dopravní síti nalezneme minimální vzdálenosti z vrcholu z do každého dalšího vrcholu, kdy vzdálenost reprezentuje volný tok. Keř pak bude takový faktor dopravní sítě, který obsahuje jen ty hrany, jejichž počáteční vrchol je blíž vrcholu z než jejich konečný vrchol [7].

Následně se metodou AON přiřadí toky ze z do každé jiné zóny. Toky se přiřadí hranám na keři i hranám na dopravní síti. Alternativně, pokud máme k dispozici jiné přiřazení, které je blíže UE (např. pokud chceme zpřesnit výstup FW algoritmu), můžeme použít to [7].

Po inicializaci následuje iterační část Algoritmu B. V iteraci se pro každý keř provedou čtyři operace mezi které patří i přesunutí toků. Stejně jak se přesunou toky na keři, tak se přesunou i na dopravní síti. V jakémkoliv okamžiku je tok na hraně z u do v v síti roven součtu toků hran z u do v v jednotlivých keřích (pokud ovšem keř tuto hranu obsahuje). Cena hrany je vždy počítána s ohledem na tok v síti, ne v keři.

3.2.2 Zlepšení topologie keře

Zlepšením topologie se rozumí přidání hran do keře tak, aby byla zachována acyklicita. Toky zůstanou invariantní vůči této operaci a tudíž zlepšení topologie neoddálí síť od UE. Naopak, přidáním hran vyvstanou nové cesty,

[5] I když pro každou hranu musí existovat hrana k ní opačná, neexistují žádné omezení pro tuto opačnou hranu. Jednosměrná silnice tedy může reprezentována tak, že příslušné opačné hraně nastavíme velmi velkou cenu a tím jí ve výsledku odstraníme [7].

které mohou být levnější. To by v dalších krocích vedlo k většímu přiblížení k UE. Zároveň, pokud by tato operace nebyla provedena, nikdy by nebyly tyto levnější cesty objeveny, nebyla by jim přiřazena doprava a algoritmus by tak nekonvergoval k UE [7].

K přidání hran do keře musíme nejprve znát maximální vzdálenost každého vrcholu od kořenu. Maximální vzdáleností je myšlena cena nejdelší cesty v keři. Do keře jsou přidány všechny hrany, které končí ve vzdálenějším vrcholu, než ve kterém začínají [9].

3.2.3 Nalezení minimálního a maximálního stromu

Algoritmus k nalezení stromu minimálních a maximálních cest v orientovaném acyklickém grafu běží v lineárním čase na rozdíl od obecných grafů, kde nalezení minimální cesty vyžaduje linearitmický čas a nalezení maximální cesty je dokonce NP-těžký problém. Nicméně, tento algoritmus prochází hrany v topologickém pořadí jejich konečných vrcholů. Stačí, aby každá hrana byla navštívena jednou. Navíc počet hran v keři je oproti síti zhruba poloviční. Důležitým detailem je, že maximální strom musí obsahovat jen hrany s nenulovým tokem na keři [7].

3.2.4 Vyvážení keře

V této části se pro každý vrchol přemístí tok mezi jeho minimální a maximální cestou tak, aby mezi těmito cestami nastala UE. Toky se ale nepřemisťují na celých cestách, ale jen na segmentech těchto dvou cest, které:

1. začínají ve společném vrcholu m,
2. končí v daném vrcholu n, pro který se přemisťují toky a
3. nemají žádné společné vrcholy mezi m a n.

Přesouvání toku mezi těmito segmenty je ekvivalentní případu se dvěma cestami popsanému v Kapitole 3.1.3 a jak bylo v této kapitole zmíněno, B nachází přesouvaný tok Δx pomocí Newtonovy metody. Tok je přesunut ze segmentu maximální cesty do segmentu minimální cesty. Oba segmenty pak mají stejnou cenu nebo má jeden ze segmentů cenu nižší, pokud byla zapotřebí podmínka (3.19). V každém případě pro tuto část keře nastává UE a to posune k UE celou síť [7].

3.2.5 Odstranění hran s nulovými toky

Nakonec se z keře odstraní všechny hrany, které mají nulový tok na keři. Výjimkou jsou hrany, které tvoří minimální strom, ty odstraněny nejsou [9].

3.2.6 Terminace

Algoritmus končí, pokud nastane nějaká ze zastavovacích podmínek. Může to být např. maximální počet iterací, rozdíl mezi dvěma posledními hodnotami objektivní funkce či již zmíněná relativní mezera.

4 Implementace

Algoritmus B byl implementován v jazycích Java a C++. Obě implementace byly vytvořeny, aby si byly tak podobné, jak jen to jde v rámci vlastností jazyků a nejlepších praktik. V této kapitole je popsána sekvenční implementace Algoritmu B a následně je zde vysvětleno, jak byly některé kroky této implementace paralelizovány. Dále jsou zde uvedeny poznámky k použitým datovým strukturám a metodám. Na konci je stručně zmíněno testování.

4.1 Sekvenční implementace

Sekvenční implementace Algoritmu B je popsána po krocích zobrazených na Obrázku 4.1:

1. Vstupem programu je dopravní síť G, OD matice O, maximální počet iterací k_{max} a minimální relativní mezera rg_{min}.

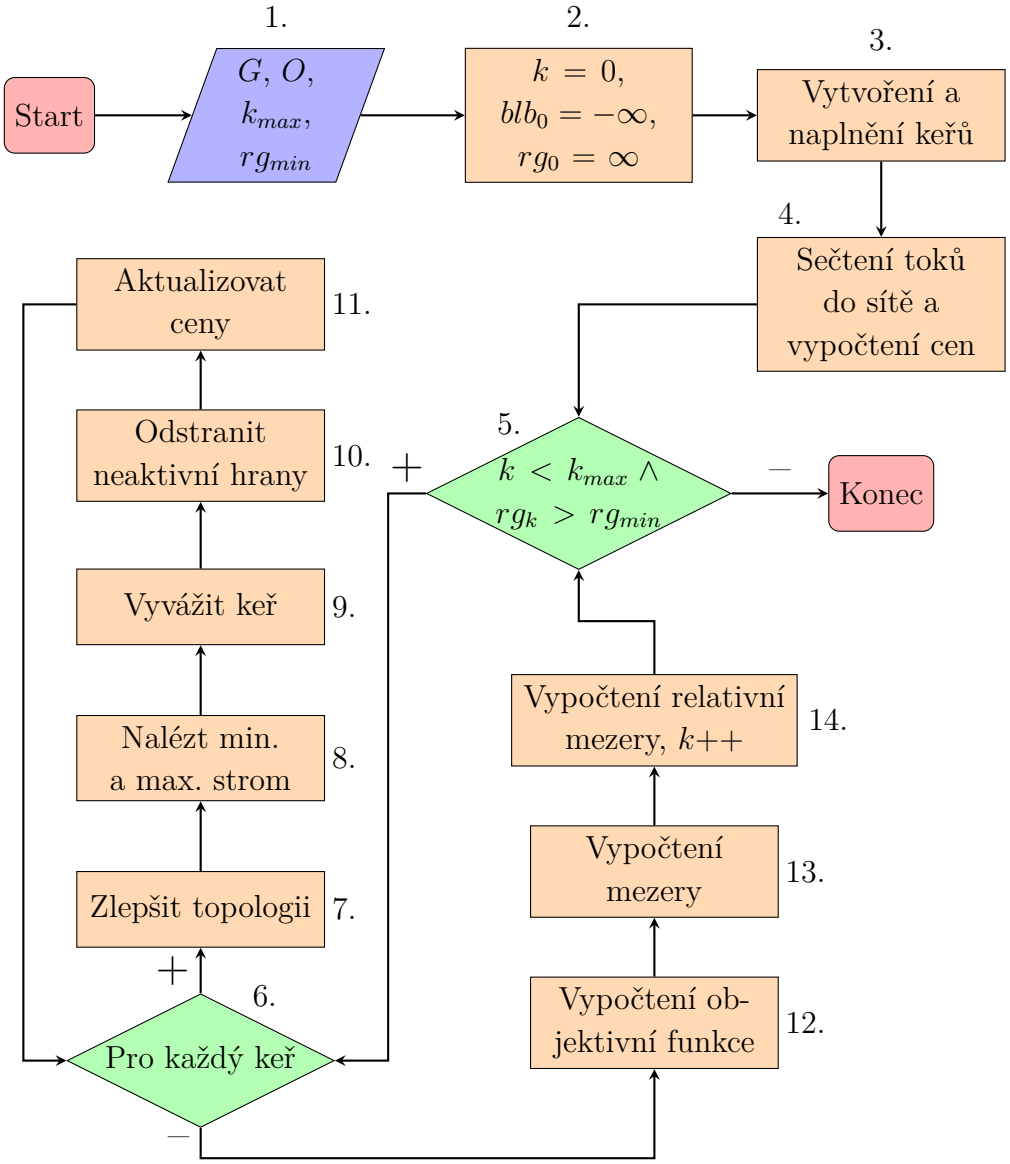
2. Inicializujeme počet proběhlých iterací k na 0, maximální spodní hranici blb_0 na $-\infty$ a relativní mezeru rg_0 na ∞.

3. Pro každou zónu $z \in Z$ se vytvoří keř K_z. Z vrcholu z se na síti spustí Dijkstrův algoritmus a naleznou se nejkratší cesty a jejich vzdálenosti ke každému vrcholu s ohledem na volné toky. Následně se iteruje přes všechny hrany sítě a ty hrany, které začínají v bližším vrcholu než ve kterém končí, jsou přidány do keře. Nakonec se pro každý vrchol v naplní nejkratší cesta k němu tokem O_{zv} — tok se přičte ke každé hraně této cesty.

4. Následně se toky v keřích sečtou do toků v síti. Iterujeme přes všechny keře. Pro každou hranu z u do v se tok v daném keři na této hraně přičte toku na hraně z u do v v síti. Z toků na síti je BPR funkcí vypočtena cena hran.

5. Iterační část běží dokud nenastane zastavovací podmínka — iterace nepřekročí maximální počet nebo relativní mezera nebude nižší než rg_{min}.

6. V jedné iteraci se nad každým keřem provedou operace přibližující síť k UE. Na konci iterace se aktualizují proměnné k a rg_k.



Obrázek 4.1: Průběh Algoritmu B.

7. Zlepšení topologie keře K_z probíhá následovně. Nejdříve nalezneme maximální vzdálenost d_{max} (s ohledem na ceny hran) ke každému vrcholu v keři. To je provedeno standardním algoritmem pro nalezení maximální vzdálenosti v orientovaném acyklickém grafu:

   i. Vzdálenost ke každému vrcholu kromě kořenu z je inicializována na $-\infty$. $d_{max}(z) = 0$.

   ii. Vrcholy jsou topologicky seřazeny — pro každou hranu z u do v je vrchol u v tomto řazení před vrcholem v.

   iii. Pro vrchol u v topologickém pořadí, pro sousední vrchol v vrcholu

u, pokud je $d_{max}(v)$ menší než $d_{max}(u) + c_e$, kde c_e je cena hrany z u do v, nastav $d_{max}(v) = d_{max}(u) + c_e$.

Po vypočtení maximálních vzdáleností se iteruje přes všechny hrany dopravní sítě a pokud hrana začíná ve vrcholu, který je blíž kořenu než vrchol, ve kterém hrana končí, je přidána do keře. Hrany, které začínají nebo končí ve vrcholu, který nebyl dosažen (jeho d_{max} zůstala $-\infty$), do keře nejsou přidány.

8. V procesu analogickém tomu, jenž byl popsán v předešlém kroku, jsou nalezeny stromy nejkratších a nejdelších cest. Musí se však dbát na to, že při přidávání hran do maximálního stromu musí být ignorovány hrany, které mají nulový tok v keři.

9. Následuje vyvážení keře přemístěním toků z dražších cest na levnější. Pro každý vrchol v:

   i. Nalezneme segmenty minimální a maximální cesty, mezi kterými budou přesouvány toky. To provedeme tak, že budeme couvat od vrcholu v po minimální i maximální cestě, dokud se cesty nepotkají ve společném vrcholu u. Pokud je u bezprostředním předkem v, je vrchol v přeskočen, protože oba segmenty jsou tvořeny jednou a tou samou hranou.

   ii. Vypočítáme přesouvaný tok Δx pomocí Newtonovy metody. Ve vzorci (3.22) $\overline{p}$ značí segment maximální cesty a $x^0_{\overline{p}}$ jeho stávající tok na tomto segmentu v síti. Analogicky, $\underline{p}$ je segment minimální cesty a $x^0_{\underline{p}}$ je jeho stávající tok. Newtonova metoda terminuje pokud $|\Delta x_{k-1} - \Delta x_k| \leq 10^{-10}$ nebo pokud počet iterací přesáhne 100. Originální verze Algoritmu B provádí pouze jednu iteraci [7]. Pro ověření podmínky (3.19) vypočteme μ tak, že iterujeme přes hrany segmentu maximální cesty a nalezneme minimální hodnotu toku v keři na těchto hranách. Nakonec ověříme tuto podmínku a pokud Δx nespadá do intervalu $\langle 0, \mu \rangle$, nastavíme Δx na nejbližší hranici tohoto intervalu.

   iii. Iterujeme přes hrany segmentu minimální cesty a ke každé hraně přičteme přesouvaný tok Δx z předešlého kroku (přičítá se k tokům na keři). Δx je přičteno i k příslušným hranám v síti (přičítá se k tokům v síti). Analogicky je Δx odečteno od hran segmentu maximální cesty.

10. Po vyvážení keře iterujeme přes všechny hrany keře a odstraníme ty,

které mají nulový tok na keři. Hrany z minimálního stromu vytvořeného v kroku 8. se však v keři ponechají.

11. Z toků na síti je pomocí BPR funkce přepočítána cena hran.

12. Poté, co jsou všechny keře vyvážené, se aktualizuje relativní mezera. Prvním krokem je výpočet objektivní funkce pro tuto iteraci of_k podle vzorce (3.12).

13. Následně je vypočtena mezera g_k. Nejdříve přiřadíme síti toky AON metodou podobně jako v kroku 3. a 4., tentokrát však nevytváříme keře, nejkratší cesty nacházíme s ohledem na aktuální ceny hran a toky jsou sčítány rovnou během jejich přiřazení. Teď už máme k dispozici všechny hodnoty pro vypočtení mezery v aktuální iteraci: toky v síti x_e^k, ceny hran $c_e(x_e^k)$ a AON toky y_e^k. Použijeme vzorec (3.23).

14. Nakonec dopočteme relativní mezeru. Spodní hranici lb_k dostaneme sečtením hodnoty objektivní funkce a mezery (vzorec (3.24)), nalezneme maximální spodní hranici blb_k za všechny proběhlé iterace (vzorec (3.25)) a podle vzorce (3.26) vypočteme relativní mezeru rg_k pro aktuální iteraci. Inkrementujeme počet iterací k.

4.2 Paralelizace

V paralelní implementaci B algoritmu je paralelizováno vytváření a naplnění keřů (krok 3.), vypočtení mezery (krok 13.) a iterace přes keře (krok 6.). Paralelizováno by mohlo být i vypočtení cen (krok 11. a část kroku 4.) a vypočtení objektivní funkce (krok 12.), ale tyto operace tvoří zlomek času celého algoritmu a jejich urychlení paralelizací by bylo přinejmenším nevýznamné. Počet vláken, na kterých budou paralelizované části běžet, označíme T.

Vytváření a naplnění keřů je paralelizováno přímočaře. Jelikož se z existujících datových struktur v tomto kroku jen čte a toky se zapisují do právě vytvořeného keře, stačí množinu zón Z rovnoměrně rozdělit mezi T vláken a pokračovat v souladu s 3. krokem.

Paralelizace výpočtu mezery už je komplikovanější, protože se na rozdíl od kroku 3. toky nepřičítají do toků na keři, který je exkluzivní pro dané vlákno, ale do toků y společných pro celou tuto operaci. Pokud by pak dvě vlákna přičítala toky nejkratším cestám, které obě obsahují hranu e, mohl by nastat souběh nad y_e. Toto je vhodně vyřešeno atomickou operací přičítání.

Tato část paralelního B je jediná, která do algoritmu vnáší nedeterminismus. Jelikož floating-point aritmetika není asociativní, záleží v jakém pořadí atomické přičítání toků proběhne. Pozorovaný rozdíl ve vypočtené relativní mezeře při dvou různých spuštění je však naprosto miniaturní — řádu 10^{-15}. Proto tento nedeterminismus není ošetřován.

Paralelizace kroku 6., není nutně náročná, ale je trochu záludná. Iterace přes $|Z|$ keřů je nahrazena iterací přes $|Z|/T$ T-tic keřů[1], kdy v jedné iteraci každé vlákno zpracovává jeden keř. Zlepšení topologie a nalezení stromů mohou probíhat paralelně, protože dochází ke čtení a zapisování dat jen v rámci daného keře.

Z kroku 9. (vyvážení keřů) však může probíhat paralelně pouze nacházení segmentů, jelikož je to opět operace neovlivňující ostatní keře. Sekvenčně pak musí probíhat nalezení Δx a přelévání toku, protože v části iii. jsou modifikovány toky v síti, které jsou čteny v části ii. při výpočtu Δx. Nepomohlo by ani dát bariéru mezi tyto části. Jsou tedy vykonávány jen jedním vláknem. Tato část kódu ovšem tvoří přibližně 4% celkového času algoritmu — není časově kritická.

Odstranění neaktivních hran opět může probíhat paralelně. Aktualizování cen proběhne na konci pouze jednou, po paralelním zpracování všech T keřů. To má za následek to, že paralelní B může potřebovat více iterací ke konvergenci než jeho sekvenční verze.

4.3 Implementační detaily

Java a C++ jsou jazyky, které jsou si syntakticky i sémanticky velice podobné — oba spadají do rodiny jazyků podobných C. Proto, pokud není řečeno jinak, jsou popisované implementace stejné až na podrobnosti specifické pro daný jazyk (např. implementace v C++ je rozdělena na zdrojové a hlavičkové soubory, v Javě je použito standardní pole, zatímco v C++ je použit `std::vector`, apod.).

Jak v Javě, tak v C++ byl pro paralelní části použit fond vláken aby se předešlo režijním nákladům při vytváření vláken. V Javě byl použit fond vracený standardní metodou `Executors.newFixedThreadPool`, zatímco v C++ byla použita knihovna `BS::thread_pool`. Fondy jsou inicializovány s pevným počtem T vláken. Při vytváření keřů a výpočtu mezery se výpočty fondu předloží všechny najednou a fond si práci rozdělí rovnoměrně sám. Při iteraci přes keře se výpočty fondu předkládají po T-ticích.

Java verze B algoritmu je implementována pro SDK 17, zatímco C++

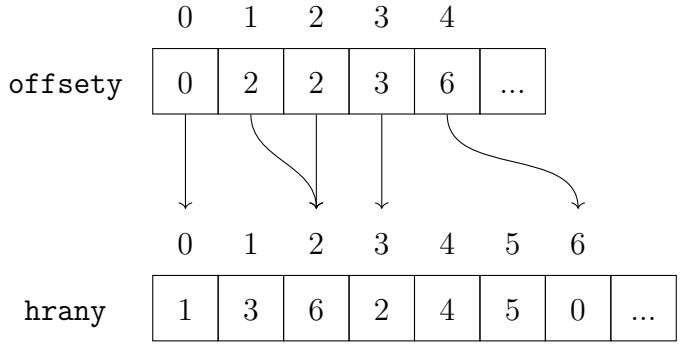
[1]Samozřejmě se musí ošetřit případ kdy $|Z|$ není dělitelné T.

verze pro standard C++ 20. V obou verzích byl pro reálná čísla použit primitivní typ `double`. V obou verzích je nastaven výstup reálných čísel na přesnost 15 desetinných míst. V Javě a C++ však existují rozdíly v konverzi reálných čísel na textové řetězce a proto se mohou výstupy Javy a C++ v několika posledních desetinných místech lišit. Například číslo typu `double`, jehož bitová reprezentace v hexadecimální formě je `4162b7ea0f3727bb`, je vypsáno jako

- `9813840,475482812000000` (Java),
- `9813840.475482812151313` (C++).

V Javě je k vypsání použito `System.out.printf("%.15f", x)`, zatímco v C++ `std::cout « std::fixed « std::setprecision(15) « x`.

4.3.1 Graf a keř



Obrázek 4.2: Compressed Sparse Row implementace grafu.

Dopravní síť je zcela jistě řídký graf. Dále se topologie dopravní sítě během algoritmu nemění. Pro statické řídké grafy je ideální *compressed sparse row* (CSR) implementace. Tato implementace téměř eliminuje paměťový overhead seznamu sousednosti, má lepší paměťovou lokalitu a díky tomu jsou operace nad ní rychlejší [10].

CSR implementace se skládá ze dvou polí – offsetů a hran. Pole offsetů má velikost $|V| + 1$ a pole hran pak $|E|$. Na indexu v pole `offsety` je číslo ukazující na hranu v poli hran, se kterou začínají hrany vycházející z vrcholu v. Na Obrázku 4.2 jsou hrany reprezentovány číslem — vrcholem, ve kterém končí. Všechny hrany začínající ve vrcholu v se dají reprezentovat podpolem pole `hrany` v rozsahu $\langle$`offsety[v]`, `offsety[v + 1]`$)$. Proto má pole `offsety` na posledním prvku délku pole `hrany` [10].

V samotné implementaci jsou vrcholy reprezentovány čísly 0 až $|V| - 1$, kdy prvních $|Z|$ vrcholů jsou zóny. Hrana je pak objekt obsahující informace důležité pro průběh B algoritmu. Mezi konstantní atributy patří počáteční vrchol, koncový vrchol, index samotné hrany v poli `hrany`, kapacita a volný tok. Proměnnými atributy jsou aktuální tok a cena hrany.

CSR implementace grafu umožňuje jednoduchou implementaci keře. Jelikož je konkrétní hrana v keři kromě toků stejná jako její příslušná hrana v grafu, stačí nám pro reprezentaci keře dvě pole. Obě pole mají stejnou délku jako pole `hrany` (tedy $|E|$). První pole je pole booleanů a indikuje jaké hrany se v keři vyskytují — je-li na indexu `i` v tomto poli hodnota `true`, v keři se vyskytuje hrana `hrany[i]`. Druhé pole je polem reálných čísel — toků na keři.

4.3.2 Ostatní datové struktury

Největším rozdílem je v Javě implementace tříd `IntQueue` a `IntSet` reprezentující frontu, respektive množinu celých čísel. Důvodem, proč nebyly použity knihovní kolekce `Queue<Integer>` a `Set<Integer>` je, že tyto třídy jsou generické a tudíž místo primitivních typů `int` ukládají obalové objekty `Integer`. To přináší další paměťové nároky i časové nároky ve formě dereferencí a autoboxingu — automatického převodu mezi primitivním typem a jeho obalovou třídou. Autoboxingu by se mělo vyhýbat ve výkonově citlivém kódu [11]. V C++ byla také implementována vlastní třída `Queue` i když ve standardní knihovně fronta existuje. Důvod je ten, že `std::queue` není implementována pomocí standardního pole, ale spojovým seznamem nebo polem polí. To přidává zbytečné dereference a potlačuje paměťovou lokalitu.

Prioritní fronta, používaná v Dijkstrovo algoritmu, byla také implementována ačkoliv oba jazyky tuto strukturu poskytují. Co však neposkytují, je metoda pro snížení priority daného vrcholu. Bez této metody by se do fronty musely přidávat vrcholy, které už v ní jsou, fronta by byla delší a to by vedlo k horším časům ostatních operací (přidání vrcholu a vybrání vrcholu s nejmenší prioritou). V Javě se také vyhneme autoboxingu.

OD matice je implementována jedním polem, ve kterém jsou řádky matice poskládány za sebou. Opět pro lepší paměťovou lokalitu.

Za poslední zmínku stojí v Javě implementovaná třída `ArrayView`, která reprezentuje podpole nějakého pole mezi dvěma indexy bez jakéhokoliv kopírování. Je používaná s polem `hrany`, kdy `ArrayView(hrany, offsety[i], offsety[i + 1])` reprezentuje podpole hran vycházejících z vrcholu `i`. V C++ je pro tyto účely použita knihovní třída `std::span`.

4.3.3 Metody

Celý sekvenční algoritmus je implementován jako třída `Algorithm`, jejíž atributy jsou (kromě jiného) vstupní hodnoty předané v konstruktoru. Metody této třídy přibližně odpovídají procedurám na Obrázku 4.1 a jsou volány metodou `start()`, která spustí Algoritmus B. Třída `ParallelAlgorithm` reprezentující paralelní verzi B dědí od `Algorithm` a obsahuje navíc fond vláken. `ParallelAlgorithm` překrývá metodu `start()`, která v této třídě spouští některé metody paralelně, jak je popsáno v Kapitole 4.2.

Za zmínku stojí paralelizovaná metoda `gap()` ve třídě `Network` vypočítávající mezeru pro aktuální toky na síti. Jak bylo řečeno v Kapitole 4.2, musí se zde přičítat atomicky do společných toků. V standardních knihovnách Javy neexistuje třída `AtomicDouble`, která by (stejně jako `AtomicInteger` apod.) podporovala všechny atomické aritmetické operace. V knihovně se však třída `DoubleAdder`, která podporuje alespoň atomické přičítání. Než však reprezentovat společné toky polem `DoubleAdder`ů, bylo uznáno za vhodnější použít třídu `AtomicDoubleArray` z knihovny Guava, která používá pole `long`ů. V C++ byl použit typ `std::atomic`, který od standardu C++ 20 podporuje atomické přičítání nad typem `double`. Ve výsledku jsou toky reprezentovány typem `std::vector<std::atomic<double>>`.

4.3.4 Vstupy a výstupy

Jak bylo řečeno v Kapitole 4.1, vstupem je dopravní síť, ODM, maximální počet iterací a relativní mezera. Programům se navíc musí specifikovat výstupní soubor a pokud chceme spustit paralelní verzi Algoritmu B musíme ještě specifikovat počet vláken. V Javě i C++ programu se tyto hodnoty předávají jakožto argumenty příkazové řádky. Pro jejich jednoduché parsování byla v Javě použita knihovna JCommander a v C++ knihovna argparse. Argumenty jsou nastaveny tak, že jsou všechny kromě počtu vláken a relativní mezery povinné. Pokud je vynechán počet vláken, spustí se sekvenční verze a pokud je vynechána relativní mezera, pak nebude vypočítávána a jedinou ukončovací podmínkou zůstane maximální počet iterací. Síť a OD matice se předávají jako cesty k příslušným TNTP (*transportation network test problem*) souborům, jejichž formát je popsaný zde [12].

Jedním z výstupů programů je TNTP soubor s přiřazenými toky. Cesta k němu je předána programu při spuštění. Druhým výstupem je vypisování do konzole. Při standardním průběhu programy na počátku vypíšou úspěšné načtení sítě a OD matice, verzi B algoritmu (sekvenční/paralelní), ukončovací podmínky (max. počet iterací a rel. mezeru) a počáteční hodnotu objektivní funkce po naplnění sítě toky. Následně se každou iteraci vypíše

její pořadové číslo, hodnota objektivní funkce na konci této iterace, a pokud byla na vstupu specifikována relativní mezera, pak je vypsána mezera i relativní mezera. Na konci je vypsán čas algoritmu v milisekundách.

4.4 Jednotkové testování

Pro každou z osmi metod volaných hlavní metodou `start()` byl vytvořen testovací případ. Testují se standardní i okrajové případy. Pro Javu byla použita knihovna JUnit, zatímco pro C++ knihovna Catch a testy byly psány tak, aby v obou programech ověřovaly stejné situace. Dohromady bylo napsáno 41 testů.

Testovací případ	Počet testů	Počet *assert*	Řádek kódu (Java)	Řádek kódu (C++)
`createBush`	2	29	102	80
`findFlowDelta`	5	5	112	83
`getMaxDistance`	4	4	108	100
`getTrees`	8	16	229	247
`improveBush`	2	7	71	55
`LCA`	13	13	126	84
`removeUnusedArcs`	4	16	130	113
`shiftFlows`	4	18	124	94

Tabulka 4.1: Statistiky testovacích případů.

Program	Řádek kódu testů	Řádek kódu programu	Velikost spustitelného souboru
Java	1002	1380	3305kB
C++	856	1628	355kB Windows/288kB Linux

Tabulka 4.2: Statistiky programů.

Jednotkové testy se skládají ze tří částí známých jako *arrange*, *act* a *assert*, kdy se (v tomto pořadí) připraví vstup pro testovanou metodu, metoda se zavolá a ověří se výstup metody. Algoritmus B má jakožto grafový algoritmus netriviální stavy i pro triviální případy (graf se 4 vrcholy může mít až 12 orientovaných hran, každá hrana má kapacitu, volný tok, aktuální tok a cenu, z tohoto grafu mohou být vytvořeny 4 keře, atd.). Proto jsou některé jednotkové testy (a zvlášť pak části *assert*) delší, než by zprvu mohlo být uznáno za vhodné.

Testy byly psány tak, aby pokrývaly 100% kódu příslušných metod. Podle IDE IntelliJ IDEA testy dohromady ve třídách používaných v algoritmu pokrývají 54 z 63 metod, 293 z 476 řádek a 130 z 240 podmínek.

5 Porovnání

V této části jsou uvedeny a porovnány výsledky testování času, paměti a výstupu programů implementujících Algoritmus B v Javě a C++. Java projekt byl sestaven pomocí nástroje Gradle a C++ s nástrojem CMake se zapnutými optimalizacemi `-O3`.

5.1 Způsob testování a porovnání

Porovnávány jsou rozdíly jak absolutní, tak relativní. Pro paměť a čas je absolutní rozdíl počítán podle vzorce

$$\text{ABS} = y - x, \tag{5.1}$$

zatímco relativní podle vzorce

$$\text{REL} = \frac{y - x}{y} \cdot 100\%, \tag{5.2}$$

kde x značí jednu hodnotu získanou testováním programu v konfiguraci X a y značí druhou hodnotu, získanou testováním programu v jiné konfiguraci Y. Vzorce záměrně nejsou symetrické — zaměníme-li mezi sebou porovnávané hodnoty, nevyjdou stejné výsledky. Důvodem je fakt, že chceme ve výsledku vidět, která z konfigurací přinese lepší či horší výsledek — informace, která by byla ztracena, pokud by byly vzorce symetrické. Ale protože záleží na pořadí, je dále v kapitole určeno jako porovnání konfigurace X oproti konfiguraci Y (např. rychlost C++ programu oproti Java programu).

Naopak u rozdílů toků je symetrie vzorců žádoucí a jsou počítány jako aritmetické průměry přes všechny hrany:

$$\text{ABS} = \frac{1}{|E|} \cdot \sum_{e \in E} |y_e - x_e|, \tag{5.3}$$

$$\text{REL} = \frac{100\%}{|E|} \cdot \sum_{e \in E} \frac{|y_e - x_e|}{max(x_e, y_e)}. \tag{5.4}$$

Zde x_e značí tok přiřazený hraně e po spuštění v konfiguraci X a y_e značí tok přiřazený stejné hraně po spuštění v konfiguraci Y.

5.1.1 Testovací stroje

Porovnání proběhlo na dvou počítačích. Na prvním počítači (dále jen P1) běží operační systém Windows 10. Disponuje procesorem AMD Ryzen 5 1600X (6 fyzických a 12 logických jader) a 16 GB paměti RAM. Použitá verze Javy je Java HotSpot 17.0.5 a překladač pro C++ pak MinGW 13.2.0.

Druhý počítač (dále jen P2) s operačním systémem Debian 11 disponuje procesorem AMD Ryzen 3500U (4 fyzická a 8 logických jader) a 18 GB paměti RAM. Využívá Javu OpenJDK 21.0.1 a překladač pro C++ GCC 10.2.1.

5.1.2 Testovací sítě

Síť	Počet vrcholů	Počet zón	Počet hran	Celkový tok
Antverpy	16 345	2 671	35 930	272 772
Berlín	12 981	865	28 376	168 222
Birmingham	14 639	898	33 937	633 870
Goldcoast	4 807	1 068	11 140	139 253
Sydney	33 113	3 264	75 379	849 339

Tabulka 5.1: Vlastnosti testujících dopravních sítí.

Porovnání proběhlo na pěti sítích, které reprezentují reálné dopravní sítě. Vlastnosti sítí jsou zobrazeny v Tabulce 5.1. Celkový tok je roven součtu všech prvků OD matice. Síť Antverp byla poskytnuta vedoucím práce a ostatní sítě jsou dostupné z [12].

5.1.3 Konfigurace vstupních parametrů

G, O	T		k_{max}	rg_{min}
	P1	P2		
Antverpy	SQ	SQ	200	10^{-4}
Berlín	2	2		
Birmingham	4	4		
Goldcoast	6	8		
Sydney	12			

Tabulka 5.2: Vstupní parametry programů.

Na P1 byly programy spouštěny v pěti konfiguracích paralelizace: sekvenční a paralelní se 2, 4, 6 a 12 vlákny. Na P2 byly programy spouštěny ve čtyřech konfiguracích paralelizace: sekvenční a paralelní se 2, 4 a 8 vlákny.

V Tabulce 5.2 G a O značí graf a OD matici dopravní sítě, T značí počet pracovních vláken paralelní verze programů (SQ označuje sekvenční verzi). V každé konfiguraci na obou strojích program běžel, dokud relativní mezera nebyla menší než 10^{-4} nebo dokud algoritmus nepřesáhl 200 iterací. Každá kombinace hodnot z každého sloupce v tabulce reprezentuje jednu porovnávanou konfiguraci. Dohromady je tedy porovnáváno 25 konfigurací vstupních parametrů na P1 a 20 konfigurací na P2.

5.2 Výstupy

Výstupem obou programů je soubor s přiřazenými toky na každé hraně. Také se každou iteraci do konzole vypisuje hodnota objektivní funkce, mezery a relativní mezery. Toky přiřazené oběma programy jsou (pro stejnou konfiguraci vstupních parametrů) stejné. Jak bylo řečeno v Kapitole 4.3, jediné rozdíly, které se ve výstupních souborech dají najít, jsou způsobeny zaokrouhlováním Javy. Stejně tak Java zaokrouhluje hodnoty obj. funkce a mezery. Vypsaná relativní mezera je identická pro oba programy[1].

5.2.1 Porovnání mezi sekvenční a paralelní verzí

T	2	4	6	12
ABS [tok]	0.6593	0.9093	0.6375	0.6735
REL [%]	0.3639	0.4043	0.2897	0.3629

Tabulka 5.3: Průměrné absolutní (ABS) a relativní (REL) rozdíly v přiřazených tocích mezi sekvenční a paralelní verzí.

Přiřazené toky sekvenční a paralelní verze programů se mírně liší, což však nutně neznamená, že by jeden z výsledků byl horší. Algoritmus B nalézá minimum objektivní funkce a toto minimum je jednoznačné, protože je objektivní funkce konvexní (Kapitola 3.1.2). Avšak pro hodnoty obj. funkce vyšší než minimum (i ty velice blízké minimu) existuje více variant přiřazených toků. A jakožto numerický algoritmus Algoritmus B minima nikdy

[1]Naprosto identická je jen pro sekvenční verzi. V paralelní verzi se liší (minimálně), ale to je kvůli nedeterminismu metody (Kapitola 4.2) a ne tím, že by se lišily přiřazené toky, ze kterých se rel. mezera počítá.

nedosáhne — může se k němu jen přibližovat a tudíž sekvenční a paralelní verze B algoritmu mohou dávat různé přiřazení toků, které jsou ovšem stejně (nebo velmi podobně) blízko UE.

V Tabulce 5.3 jsou průměrné absolutní a relativní rozdíly v přiřazených tocích mezi sekvenční a paralelní verzí s T vlákny. Rozdíly jsou testovány s Java programem na P1 se vstupní sítí Antverp, $k_{max} = 200$ a $rg_{min} = 10^{-4}$. Průměrný absolutní rozdíl nepřesahuje 1 a průměrný rel. rozdíl nepřesahuje ani 0.5%. Obě verze tak dávají velice podobné výsledky.

5.2.2 Porovnání s referenční implementací

Jelikož byla vedoucím práce poskytnuta implementace B algoritmu popsaná v [9], budou zde uvedeny i rozdíly ve výsledných tocích mezi touto referenční implementací a implementací vytvořenou v rámci této práce. Porovnání proběhlo mezi Java programem a referenčním Java programem na P1 se vstupní sítí Antverp, $k_{max} = 200$ a $rg_{min} = 10^{-4}$. Oba programy byly spuštěny v jejich sekvenční verzi. Průměrný absolutní rozdíl mezi výslednými toky těchto programů je 0.5835 a průměrný relativní rozdíl je 0.5451%. Opět se jedná o velice podobné výsledky.

5.3 Čas

Čas běhu programů je měřen v milisekundách přímo v kódu pomocí metod

- `System.currentTimeMillis()` (Java) a
- `std::chrono::system_clock::now()` (C++).

Měří se čas čistě části programu provádějící B algoritmus, čtení vstupních souborů, parsování dat do datových struktur apod. se do výsledného času nezapočítává. Pro každou konfiguraci byl čas měřen sedmkrát. Nejvyšší a nejnižší hodnoty z těchto sedmi byly vyřazeny a ze zbylých pěti hodnot byl vypočítán aritmetický průměr.

	P1				P2			
Síť	T	k	Java	C++	T	k	Java	C++
Antverpy	SQ	24	277 092	259 027	SQ	24	279 264	268 114
	2	20	128 479	118 613	2	20	156 983	159 276
	4	21	72 949	67 766	4	21	112 545	120 036
	6	35	83 195	79 052	8	23	94 079	98 859

	12	34	58 028	51 690				
Berlín	SQ	5	15 652	14 189	SQ	5	16 520	13 903
	2	5	8 726	7 791	2	5	11 471	9 763
	4	5	4 890	4 161	4	5	8 473	6 996
	6	5	3 646	2 877	8	5	7 705	5 433
	12	5	2 840	1 917				
Birmingham	SQ	6	25 585	23 019	SQ	6	25 939	23 392
	2	6	14 453	12 815	2	6	18 800	17 811
	4	6	8 168	7 141	4	6	13 820	12 468
	6	6	6 044	5 400	8	6	10 943	9 602
	12	6	4 761	3 577				
Goldcoast	SQ	40	58 292	53 126	SQ	40	57 106	53 185
	2	47	41 379	37 994	2	47	49 481	48 738
	4	41	21 695	20 498	4	41	29 561	29 315
	6	54	21 388	20 507	8	48	25 548	29 451
	12	46	14 332	12 495				
Sydney	SQ	11	347 798	326 221	SQ	11	492 547	394 813
	2	11	217 660	189 411	2	11	295 026	273 551
	4	11	109 270	105 111	4	11	181 622	182 378
	6	11	78 778	80 338	8	11	139 226	139 760
	12	12	66 337	64 499				

Tabulka 5.4: Naměřené časy běhu B algoritmu (v ms).

Průměry pro každou konfiguraci vstupních parametrů jsou k vidění v Tabulce 5.4. Sloupec T značí počet vláken, se kterými byla spuštěna paralelní verze programu, SQ v tomto sloupci značí sekvenční verzi programu. Ve sloupci k je počet iterací, po kterých algoritmus dosáhl relativní mezery 10^{-4}. Časy jsou v tabulce uvedeny v milisekundách.

5.3.1 Porovnání mezi Javou a C++

	P1			P2		
Síť	T	ABS [ms]	REL [%]	T	ABS [ms]	REL [%]
Antverpy	SQ	18 065	6.5	SQ	11150	4.0
	2	9 866	7.7	2	-2293	-1.5
	4	5 183	7.1	4	-7491	-6.7
	6	4 143	5.0	8	-4780	-5.1
	12	6 338	10.9			

Berlín	SQ	1 463	9.3	SQ	2617	15.8
	2	935	10.7	2	1708	14.9
	4	729	14.9	4	1477	17.4
	6	769	21.1	8	2272	29.5
	12	923	32.5			
Birmingham	SQ	2 566	10.0	SQ	2547	9.8
	2	1 638	11.3	2	989	5.3
	4	1 027	12.6	4	1352	9.8
	6	644	10.7	8	1341	12.3
	12	1 184	24.9			
Goldcoast	SQ	5 166	8.9	SQ	3921	6.9
	2	3 385	8.2	2	743	1.5
	4	1 197	5.5	4	246	0.8
	6	881	4.1	8	-3903	-15.3
	12	1 837	12.8			
Sydney	SQ	21 577	6.2	SQ	97734	19.8
	2	28 249	13.0	2	21475	7.3
	4	4 159	3.8	4	-756	-0.4
	6	-1 560	-2.0	8	-534	-0.4
	12	1 838	2.8			

Tabulka 5.5: Absolutní (ABS) a relativní (REL) rychlost C++ programu oproti Java programu.

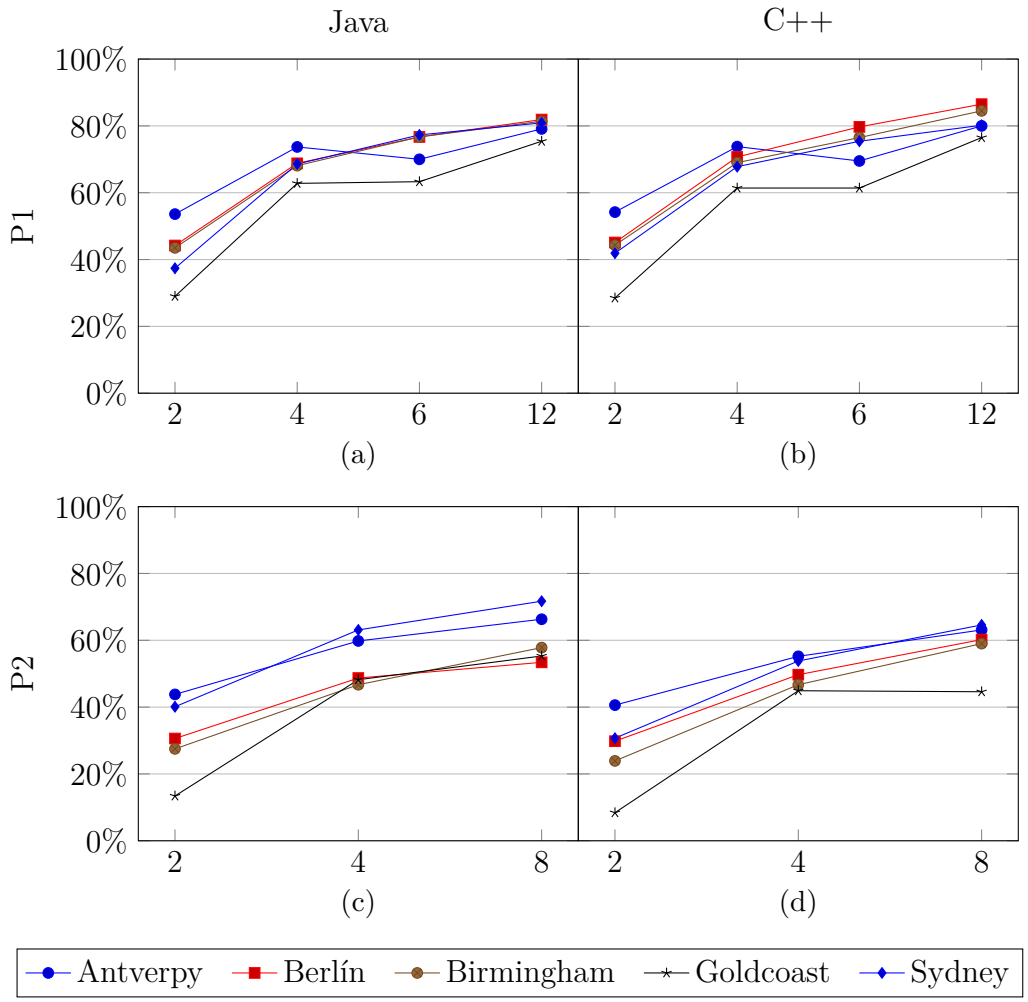
C++ program je na P1 průměrně o 4.9s (10.3%) rychlejší a o 6.5s (6.3%) na P2. Absolutní rozdíly se s počtem vláken snižují z průměrných 9.8s pro sekvenční verzi na 2.4s pro paralelní verzi s 12 vlákny na P1. Na P2 je sekvenční C++ program průměrně rychlejší o 23.6s zatímco jeho paralelní verze s 8 vlákny je na P2 průměrně o 1.1s pomalejší. Je nepochybně zajímavé, že nastaly případy, kdy je Java rychlejší než C++. Tyto rozdíly nejsou zanedbatelné, v jednom případě je C++ dokonce o 7.5s (6.7%) pomalejší. Tyto případy sice nastávaly hlavně na P2, ale jeden se objevil i na P1. Možné vysvětlení tohoto jevu je, že optimalizace JIT překladače Javy v těchto případech fungují lépe než optimalizace AOT překladače GCC/MinGW. Tento jev může být předmětem dalšího zkoumání.

5.3.2 Porovnání mezi sekvenční a paralelní verzí

V Tabulce 5.6 je vidět, že relativní urychlení paralelní verze oproti sekvenční verzi je pro každou síť u Javy i C++ velmi podobné. Průměrně se příslušné

		Síť									
		Antverpy		Berlín		Birmingham		Goldcoast		Sydney	
	T	Java	C++	Java	C++	Java	C++	Java	C++	Java	C++
P1	2	53.6	54.2	44.2	45.1	43.5	44.3	29.0	28.5	37.4	41.9
	4	73.7	73.8	68.8	70.7	68.1	69.0	62.8	61.4	68.6	67.8
	6	70.0	69.5	76.7	79.7	76.4	76.5	63.3	61.4	77.3	75.4
	12	79.1	80.0	81.9	86.5	81.4	84.5	75.4	76.5	80.9	80.2
P2	2	43.8	40.6	30.6	29.8	27.5	23.9	13.4	8.4	40.1	30.1
	4	59.7	55.2	48.7	49.7	46.7	46.7	48.2	44.9	63.1	53.8
	8	66.3	63.1	53.4	60.9	57.8	59.0	55.3	44.6	71.7	64.6

Tabulka 5.6: Relativní urychlení paralelní verze oproti sekvenční verzi (v %).



Obrázek 5.1: Tabulka 5.6 v grafech (horizontální osa značí počet vláken).

hodnoty liší o 1.5 na P1 a o 4.7 na P2.

Pro dvě vlákna je urychlení 28.5%–54.2% (průměrně 42.2%) na P1 a 8.4%–43.8% (průměrně 28.9%) na P2. Pro čtyři vlákna je urychlení 61.4%–73.8% (průměrně 68.5%) na P1 a 44.9%–63.1% (průměrně 51.7%) na P2. Na P1 je urychlení pro šest vláken 61.4%–79.7% (průměrně 72.6%) a pro dvanáct vláken 75.4%–86.5% (průměrně 80.6%). Na P2 je urychlení pro osm vláken 44.6%–71.7% (průměrně 59.7%).

Pokud bychom porovnali průměrné urychlení na P1 pro čtyři vlákna (68.5%) a šest vláken (72.6%), zjistíme, že urychlení na šesti vláknech je jen o málo větší. To je způsobeno sítěmi Antverpy a Goldcoast, což je ostatně vidět i na Obrázku 5.1a a 5.1b. U Goldcoastu je zvýšení urychlení neznatelné a u Antverp se urychlení dokonce sníží. Podíváme-li se zpět do Tabulky 5.4, sloupce k, uvidíme, že sítě Antverpy a Goldcoast jsou s ohledem na počet vláken velmi volatilní, zatímco sítě Berlín, Birmingham a Sydney jsou naopak velmi stabilní — zvyšování T jejich konvergenci téměř neovlivňuje. U Antverp i Goldcoastu nastávají případy, kdy se zvýšením T se sníží počet iterací, což je opak toho, co by se očekávalo. Stejně tak u těchto sítí ale i nastávají případy, kdy se zvýšením T se zvýší počet potřebných iterací tolik, že ve výsledku potřebuje algoritmus víc času ke konvergenci. Vlastnost sítě, která tuto volatilitu způsobuje by mohla být předmětem dalšího výzkumu.

5.3.3 Porovnání s referenční implementací

T	SQ	2	4	6	12
Ref	108 265	77 584	46 397	37 387	34 545
Java	25 585	14 453	8 168	6 044	4 761
ABS [ms]	82 680	63 131	38 229	31 343	29 784
REL [%]	76.4	81.4	82.4	83.8	86.2

Tabulka 5.7: Naměřené časy referenční implementace a Java programu na P1 se vstupní sítí Birmingham (v ms), absolutní (ABS) a relativní (REL) rychlost Java programu oproti referenční implementaci.

V Tabulce 5.7 jsou naměřené časy běhu referenční implementace B algoritmu [9] (Ref) a Java programu vytvořeného v rámci této práce (Java) měřené na P1 se vstupní sítí Birmingham, $rg_{min} = 10^{-4}$ a $k_{max} = 200$. Dále jsou v tabulce absolutní a relativní rychlosti Java programu oproti referenční implementaci. Java program potřebuje průměrně o 82% méně času pro zmíněnou konfiguraci vstupů.

5.4 Paměť

Paměť byla měřena pomocí různých nástrojů. U Javy byla měřena využitá paměť po spuštění správce paměti (garbage collectoru). Pro každou konfiguraci byla paměť měřena pouze jednou, protože je (na rozdíl od času běhu) její spotřeba deterministická. U každého případu se brala v potaz maximální hodnota z grafu využité paměti.

Na P1 i P2 byla paměť Java programu měřena pomocí nástroje JDK Flight Recorder, který je zabudován přímo do virtuálního stroje Javy a zaznamenává nejrůznější informace o běhu programu, mezi které patří i paměť. Nástroj se zapne, pokud je Java program spuštěn s přepínačem `-XX:StartFlightRecording`. Vygenerovaný `.jfr` soubor byl pak nahrán do nástroje JDK Mission Control. Graf využité paměti je vidět v záložce *Garbage Collections*, konkrétně je to graf *Heap Post GC*.

Na P1 byla paměť C++ programu měřena nástrojem MTuner. Tomu se jako argument příkazové řádky předá spustitelný `.exe` soubor, načež je program spuštěn a je vygenerován `.MTuner` soubor, který se dá zobrazit v grafickém rozhraní MTuneru. Použitá hodnota se nalézá v okně *Memory tag tree* a je to hodnota *Peak Usage*.

Pro P2 byl použit nástroj Valgrind Massif. Podobně jako u předchozího nástroje byl i tento spuštěn z příkazové řádky generujíc `.out` soubor. Tento soubor byl otevřen v nástroji massif-visualizer, kde je nad grafem zvýrazněná použitá hodnota.

	P1			P2		
Síť	T	Java	C++	T	Java	C++
Antverpy	SQ	952	924	SQ	915	880
	2	957	924	2	917	881
	4	961	927	4	915	881
	6	964	929	8	916	883
	12	973	935			
Berlín	SQ	231	229	SQ	230	218
	2	232	229	2	231	218
	4	239	231	4	232	219
	6	237	233	8	233	220
	12	245	236			
Birmingham	SQ	301	284	SQ	288	270
	2	302	284	2	289	270
	4	302	286	4	289	271
	6	301	288	8	290	272

	12	305	292			
Goldcoast	SQ	118	117	SQ	120	112
	2	118	117	2	120	112
	4	118	118	4	120	112
	6	118	118	8	121	112
	12	120	120			
Sydney	SQ	2 350	2 307	SQ	2240	2147
	2	2 350	2 308	2	2250	2149
	4	2 340	2 313	4	2250	2148
	6	2 360	2 316	8	2260	2151
	12	2 360	2 332			

Tabulka 5.8: Naměřená paměť během běhu programů (v MB).

5.4.1 Porovnání mezi Javou a C++

	P1			P2		
Síť	T	ABS [MB]	REL [%]	T	ABS [MB]	REL [%]
Antverpy	SQ	28	2.9	SQ	35	3.8
	2	33	3.4	2	36	3.9
	4	34	3.5	4	34	3.7
	6	35	3.6	8	33	3.6
	12	38	3.9			
Berlín	SQ	2	0.9	SQ	12	5.2
	2	3	1.3	2	13	5.6
	4	8	3.3	4	13	5.6
	6	4	1.7	8	13	5.6
	12	9	3.7			
Birmingham	SQ	17	5.6	SQ	18	6.3
	2	18	6.0	2	19	6.6
	4	16	5.3	4	18	6.2
	6	13	4.3	8	18	6.2
	12	13	4.3			
Goldcoast	SQ	1	0.8	SQ	8	6.7
	2	1	0.8	2	8	6.7
	4	0	0	4	8	6.7
	6	0	0	8	9	7.4
	12	0	0			
Sydney	SQ	43	1.8	SQ	93	4.2

	2	42	1.8	2	101	4.5
	4	27	1.2	4	102	4.5
	6	44	1.9	8	109	4.8
	12	28	1.2			

Tabulka 5.9: Absolutní (ABS) a relativní (REL) rozdíly v potřebné paměti C++ programu oproti Java programu.

C++ program na testujících sítích potřeboval o 0–44 MB méně paměti na P1 a o 8–109 MB méně na P2 s tím, že minima tvoří nejmenší síť (Goldcoast) a maxima tvoří pak síť největší (Sydney) — dle naměřených dat se rozdíl v potřebné paměti zvyšuje s velikostí sítě. Průměrně C++ potřebuje o 18.3 MB méně paměti na P1 a o 35 MB na P2. V relativních rozdílech potřebuje C++ o 0–6% (průměrně 2.5%) méně paměti na P1 a o 3.6–7.4% (průměrně 5.4%) na P2.

5.4.2 Porovnání mezi sekvenční a paralelní verzí

		Síť									
		Antverpy		Berlín		Birmingham		Goldcoast		Sydney	
	T	Java	C++	Java	C++	Java	C++	Java	C++	Java	C++
P1	2	5	0	1	0	1	0	0	0	0	1
	4	9	3	8	2	1	2	0	1	-10	6
	6	12	5	6	4	0	4	0	1	10	9
	12	21	11	14	7	4	8	2	3	10	25
P2	2	2	1	1	0	1	0	0	0	10	2
	4	0	1	2	1	1	1	0	0	10	1
	8	1	3	3	2	2	2	1	0	20	4

Tabulka 5.10: Paměťový overhead paralelní verze oproti sekvenční verzi (v MB).

Nárůst spotřeby paměti paralelní oproti sekvenční verzi je nepatrný na P1 i P2, pro Javu i C++. Na P1 je u Javy největší overhead u Antverp, zatímco u C++ je to u Sydney — asi 2 MB za každé pracovní vlákno. Na P2 je největší overhead u Sydney pro Javu i C++ — opět přibližně 2 MB na vlákno.

6 Závěr

V této práci byl představen problém přiřazení dopravy a Algoritmus B řešící statickou variantu tohoto problému. Algoritmus byl implementován v jazycích Java a C++. Následně byla navržena paralelizace algoritmu a i ta byla implementována v obou jazycích. Nakonec byly varianty algoritmu porovnány jak mezi jazyky, tak mezi paralelizacemi.

V práci bylo zjištěno, že dopravní toky přiřazené implementovaným sekvenčním B algoritmem se od referenční implementace [9] liší přibližně o 0.5%. Podobně se liší i toky přiřazené implementovanou paralelní verzí od sekvenční verze.

Při porovnávání času mezi Javou a C++ bylo zjištěno, že C++ program běží průměrně o 10% rychleji na počítači s Windows (6 fyzických a 12 logických jader) a o 6% rychleji na počítači s Linuxem (4 fyzická a 8 logických jader). Ovšem nastaly i případy, kdy při určitých konfiguracích vstupních parametrů byla Java nezanedbatelně rychlejší. Porovnání sekvenční verze oproti paralelní verzi přineslo závěr, že oba jazyky paralelizací urychlují algoritmus skoro stejně. Pro Javu i C++ platí, že paralelní verze je přibližně o 80% rychlejší než sekvenční verze na Windows s 12 vlákny a přibližně o 60% rychlejší na Linuxu s 8 vlákny. Také ale bylo zjištěno, že zvýšení počtu vláken někdy může zvýšit počet iterací potřebných ke konvergenci natolik, že ve výsledku algoritmus potřebuje víc času.

Porovnání paměti poskytlo závěr, že C++ program oproti Java programu potřebuje průměrně o 2.5% méně paměti na Windows a o 5.4% na Linuxu. Tyto rozdíly se zvyšují s velikostí sítě. Nárůst spotřeby paměti paralelní oproti sekvenční verzi je téměř neznatelný.

Během porovnávání byl objeven zajímavý jev, kdy se některé dopravní sítě jeví s ohledem na počet vláken jako velmi volatilní, tedy že počet iterací potřebných k jejich konvergenci k uživatelské rovnováze se značně mění s různým počtem vláken. Naopak ostatní sítě se jeví jako stabilní — počet iterací se s počtem vláken nemění skoro vůbec. Zjištění, která vlastnost sítě způsobuje tuto volatilitu a obecně konvergence vzhledem k počtu vláken, by mohla být předmětem budoucí práce. Stejně tak by mohla být pro Algoritmus B v budoucnosti navržena paralelizace např. pro grafické karty či distribuce na více počítačů.

A Adresářová struktura odevzdávaného archivu

- `A20B0203P_prilohy.zip`
  - `Aplikace_a_knihovny`
    - `cpp` ... C++ projekt
      - `docs` ... Doxygen dokumentace
      - `include` ... knihovny
      - `src` ... zdrojové soubory
      - `test` ... zdrojové soubory jednotkových testů
      - `BP_Cpp` ... přeložený C++ program pro Debian
      - `BP_Cpp.exe` ... přeložený C++ program pro Windows
      - `CMakeLists.txt` ... konfigurační soubor CMake
      - `Doxyfile` ... konfigurační soubor Doxygen
    - `java` ... Java projekt
      - `docs` ... Javadoc dokumentace
      - `gradle` ... Gradle soubory
      - `src` ... zdrojové soubory
      - `BP-Java.jar` ... sestavený Java program
      - `build.gradle` ... konfigurační soubor Gradle
      - `gradlew` ... spouštěcí skript Gradle (Linux)
      - `gradlew.bat` ... spouštěcí skript Gradle (Windows)
      - `settings.gradle` ... konfigurační soubor Gradle
  - `Text_prace` ... TEX soubory a PDF textu bakalářské práce
  - `Vstupni_data` ... použité vstupní TNTP soubory dopravních sítí a OD matic
  - `Vysledky` ... adresář s výsledky BP
    - `Antverpy_toky` ... adresář s TNTP soubory toků sítě Antverp
    - `Birmingham_cas_ref` ... adresář s konzolovými výstupy při testování času referenční implementace
    - `P1_cas` ... adresář s konzolovými výstupy při testování času na P1
    - `P1_pamet` ... adresář s výstupními soubory JFR a MTuner při testování paměti na P1
    - `P2_cas` ... adresář s konzolovými výstupy při testování času na P2
    - `P2_pamet` ... adresář s výstupními soubory JFR a Valgrind Massif při testování paměti na P1
    - `skripty` ... adresář s konzolovými skripty použitými při testování
      - `P1_cas.bat`
      - `P1_pamet.bat`
      - `P2_cas.sh`
      - `P2_pamet.sh`
    - `Vysledky.xlsx` ... MS Excel tabulky se souhrnem výsledků
  - `Readme.txt`

B Uživatelská příručka

Oba programy jsou spouštěny stejně s následujícími argumenty:

- `-net <cesta k TNTP souboru se sítí>`, povinný, TNTP soubory se sítí jsou pojmenovány ve formátu `<jméno>_net.tntp`,
- `-odm <cesta k TNTP souboru s OD maticí>`, povinný, TNTP soubory s OD maticí jsou pojmenovány ve formátu `<jméno>_trips.tntp`,
- `-o <cesta výstupního TNTP souboru>`, povinný,
- `-i <max. počet iterací algoritmu>`, povinný,
- `-rg <minimální relativní mezera>`, nepovinný (pokud nespecifikováno, relativní mezera každou iteraci není počítána),
- `-t <počet vláken>`, nepovinný (pokud nespecifikováno, je puštěna sekvenční verze).

B.1 Java program

Java projekt je sestaven pomocí příkazů

- `gradlew.bat build` (Windows),
- `./gradlew build` (Linux).

Po sestavení je výsledný JAR soubor ve složce `build/libs`. Testy jsou spuštěny příkazy

- `gradlew.bat test` (Windows),
- `./gradlew test` (Linux).

B.2 C++ program

Je-li na počítači nainstalováno MinGW (Windows) či GCC (Linux), je C++ projekt je sestaven posloupností příkazů

1. `mkdir build`,
2. `cd build`,

3. `cmake .. -G "MinGW Makefiles"` (Windows) či `cmake ..` (Linux),
4. `make`.

Výsledný EXE soubor je ve složce `build/src`. EXE soubor spouštějící testy je ve složce `build/test`.

Literatura

[1] Mitchell, R. B. – Rapkin, C. *Urban Traffic: A Function of Land Use*. Columbia University Press, 1954. ISBN 978-02-3194-522-6.

[2] Maerivoet, S. – De Moor, B. Transportation Planning and Traffic Flow Models. Technical Report 05-155, Katholieke Universiteit Leuven, July 2005.

[3] Wardrop, J. G. Some Theoretical Aspects of Road Traffic Research. *Proceedings of the Institution of Civil Engineers*. January 1952, 1, 3, s. 325–362. ISSN 0001-0782. doi: 10.1680/ipeds.1952.11259. Dostupné z: https://doi.org/10.1680/ipeds.1952.11259.

[4] *Traffic Assignment Manual: For Application with a Large High Speed Computer*. U.S. Department of Commerce, Bureau of Public Roads, Office of Planning, Urban Planning Division, 1964.

[5] Beckmann, M. – McGuire, B. C. – Winsten, C. B. *Studies in the Economics of Transportation*. Yale University Press, 1955. ISBN 978-05-9867-517-0.

[6] Kolovský, F. *Algorithms in transportation*. PhD thesis, University of West Bohemia, 2019.

[7] Dial, R. B. A path-based user-equilibrium traffic assignment algorithm that obviates path storage and enumeration. *Transportation Research Part B*. December 2006, 40, 10, s. 917–936. ISSN 0191-2615. doi: 10.1016/j.trb.2006.02.008. Dostupné z: https://doi.org/10.1016/j.trb.2006.02.008.

[8] Boyce, D. – Ralevic-Dekic, B. – Bar-Gera, H. Convergence of Traffic Assignments: How Much is Enough? *Journal of Transportation Engineering*. January 2004, 130, 1, s. 49–55. ISSN 0733-947X. doi: 10.1061/(ASCE)0733-947X(2004)130:1(49). Dostupné z: https://doi.org/10.1061/(ASCE)0733-947X(2004)130:1(49).

[9] Potužák, T. – Kolovský, F. Parallelization of the B static traffic assignment algorithm. *Ain Shams Engineering Journal*. March 2022, 13, 2, s. 101576–101594. ISSN 2090-4479. doi: 10.1016/j.asej.2021.09.003. Dostupné z: https://doi.org/10.1016/j.asej.2021.09.003.

[10] Kelly, T. Compressed Sparse Row Format for Representing Graphs. *;login:*. 2020, 45, 4, s. 76–82. Dostupné z: https://www.usenix.org/system/files/login/articles/login_winter20_16_kelly.pdf.

[11] *Autoboxing* [online]. Oracle. [cit. 2024/02/25]. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/language/autoboxing.html.

[12] Transportation Networks for Research Core Team. Transportation Networks for Research, 2016. Dostupné z: https://github.com/bstabler/TransportationNetworks. Citováno 2024/03/23.

Seznam zkratek

Seznam obrázků

Seznam tabulek