Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Paralelní grafové algoritmy

2017

Michael Chalupa

Vedoucí práce: Mgr. Petr Osička, Ph.D.

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Michael Chalupa
Název práce:	Paralelní grafové algoritmy
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2017
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	34
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Michael Chalupa
Title:	Parallel graph algorithms
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2017
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	34
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Tato práce se zabývá implementací paralelních grafových algoritmů pomocí více vláken. Jsou zde představeny základní pojmy z teorie grafu a základní poznatky z paralelního programování. Algoritmy jsou implementované v programovacím jazyce C# s příslušným uživatelským rozhraním. Na konci celého textu jsou algoritmy experimentálně porovnány.

Synopsis

This thesis deals with the implementation of parallel graph algorithms using multiple threads. There are introduced basic concepts of graph theory and basic knowledge from parallel programming. The algorithms are implemented in C# with the appropriate user interface. At the end of the text, the algorithms are experimentally compared.

Klíčová slova: graf; neorientovaný graf; orientovaný graf; paralelní programování; nejkratší cesta; kostra grafu; barvení grafu

Keywords: graph; directed graph; undirected graph; parallel programming; shortest path; spanning tree; graph coloring

Děkuji Mgr. Petru Osičkovi, Ph.D. za cenné rady a všem svým blízkým za podporu.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Cílem práce bylo nastudovat a implementovat paralelní grafové algoritmy a následně je experimentálně porovnat. Na začátku textu je uvedeno krátké představení grafu a jejich využití v praxi. Dále jsou uvedeny základní pojmy z teorie grafu, které jsou potřebné pro snazší pochopení grafových algoritmů. Následuje kapitola, která pojednává o paralelním programování, představí se zde základní problémy spojené s paralelizaci algoritmů a základní paradigmata paralelního programování. V posledních dvou kapitolách uvedeme algoritmy pro výpočet nejkratší cesty grafu, kostry grafu a barvení grafu. Dále uvedeme jak lze tyto algoritmy zparalelizovat a provedeme experimentální porovnání výkonnosti.

2 Grafy

Tato kapitola čerpá informace z [1], [2], [3].

Grafy jsou abstraktní matematické objekty, které se využívají v situacích, kdy je dána nějaká množina objektů a kde tyto objekty mají mezi sebou určité vztahy. V teorií grafu, jsou tyto objekty označovány jako vrcholy a vztahy mezi nimi jsou reprezentovány pomocí hran.

Uveďme si zde pár příkladu, kde grafové algoritmy hrají důležitou roli.

Mapy. Osoba využívající mapy, může chtít najít nejkratší cestu z města A do města B. V této úloze existují vztahy mezi městy, které jsou dané silnicemi vedoucími mezi městy.

Webový obsah. Při prohlížení webu, se v jednotlivých stránkách vyskytují odkazy na další stránky. Celý web je graf, kde stránky jsou vrcholy a odkazy jsou reprezentovány hranami. Grafové algoritmy jsou základní komponenty vyhledávačů, které pomáhají získávat informace na webu.

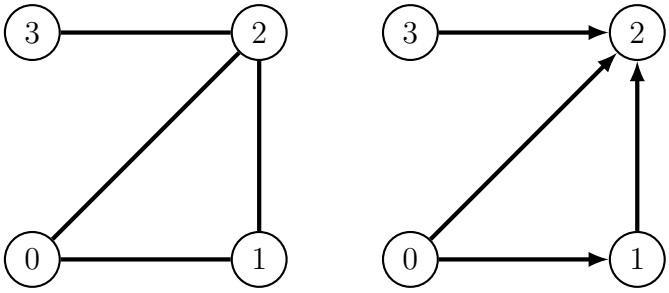
Elektrické obvody. Elektrické obvody obsahují zařízení jako tranzistory, rezistory a kondenzátory, které jsou mezi sebou složitě propojeny. K zjištěni zda-li obvod vykonává potřebné funkce, používáme grafové algoritmy.

Plánování. Výrobní proces zahrnuje posloupnost prací, kde některé práce mají omezení, že nemůžou začít dokud určitá práce nebyla dokončena. Grafové algoritmy nám můžou naplánovat celý výrobní proces, tak aby byl dokončen co v nejmenším čase.

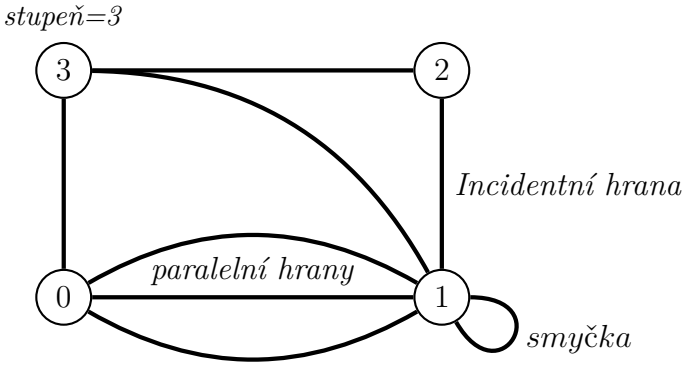
2.1 Základní pojmy

Definice 1 (Neorientovaný graf)
Neorientovaný graf je dvojice $G = (V, E)$ tvořená konečnou neprázdnou množinou V, jejíž prvky nazýváme vrcholy a konečnou množinou E obsahující dvouprvkové množiny vrcholů, kde tyto prvky nazýváme neorientované hrany.

Neorientovaný graf je používán v situacích kdy je orientace hran nepodstatná, jinými slovy nerozlišujeme mezi koncovým a počátečním vrcholem. Jména hran nejsou pro definici nějak podstatná ale aby bylo možné odkazovat na konkrétní vrcholy, tak se budeme držet konvence, takové že budeme vrcholům přiřazovat čísla od 0 do $|V| - 1$ viz. Obr. 1.



Obrázek 1: Neorientovaný graf (nalevo), orientovaný graf (napravo).



Obrázek 2: Multi graf.

O hraně e spojující dva vrcholy v, w říkáme, že je *incidentní* s oběma vrcholy v, w a o těchto vrcholech říkáme, že jsou *sousední*. *Stupeň* vrcholu v je číslo, určeno počtem incidentních hran s tímto vrcholem. Je-li hrana e incidentní pouze s jedním vrcholem, pak tuto hranu e nazýváme *smyčkou*. Dvě hrany e_1, e_2, které obsahují stejné vrcholy nazýváme *paralelní*. Graf bez paralelních hran se občas nazývá *prostý graf*, zatímco graf s paralelními hrany se nazývá *multigraf* viz. Obr. 2.

Máme-li dán graf G, pak množinu všech jeho vrcholu budeme značit $V(G)$ a množinu jeho hran $E(G)$.

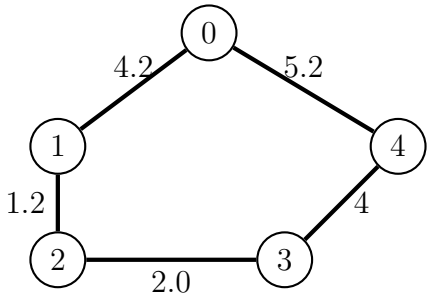
Definice 2 (Orientovaný graf)
Orientovaný graf je dvojice $G = (V, E)$ tvořená konečnou neprázdnou množinou V, jejíž prvky nazýváme vrcholy a konečnou množinou E obsahující uspořádané dvojce vrcholů, kde tyto prvky nazýváme orientované hrany.

O orientované hraně e říkáme, že vede z vrcholu v do vrcholu w a také, že spojuje vrchol v s w. Dále stupeň vrcholu v je dán součtem vstupního a výstupního stupně vrcholu, kde vstupní stupeň je počet hran vedoucích do v a výstupní stupeň je počet hran vedoucích z v.

V mnoha situacích je potřeba hranám nebo vrcholům přiřadit určité hodnoty, většinou to jsou reálná čísla, které můžou např. představovat vzdálenost mezi městy, pravděpodobnost jevu, elektrický odpor apod.

Definice 3 (Ohodnocení)

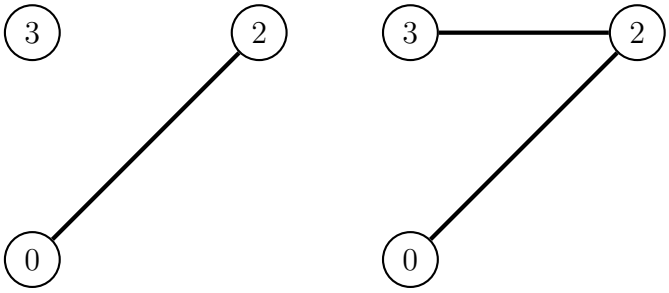
Hranové ohodnocení grafu $G = (V, E)$ s množinou D je funkce $w : E \to D$. Vrcholové ohodnocení grafu $G = (V, E)$ s množinou D je funkce $w : V \to D$.



Obrázek 3: Ohodnocený graf.

Definice 4 (Podgraf)

Graf G_1 je podgrafem grafu G_2, právě když platí $V(G_1) \subseteq V(G_2)$, $E(G_1) \subseteq E(G_2)$ a G_1 tvoří graf. Graf G_1, který je podgrafem grafu G_2, se nazývá *indukovaný*, jestliže pro libovolné vrcholy $v, w \in V(G_1)$ platí $(v, w) \in E(G_2) \implies (v, w) \in E(G_1)$.



Obrázek 4: Podgrafy.

Na Obr. 4 můžeme vidět dva grafy, které jsou podgrafy grafu z Obr. 1, z nichž druhý graf je indukovaný.

Definice 5 (Izomorfismus)

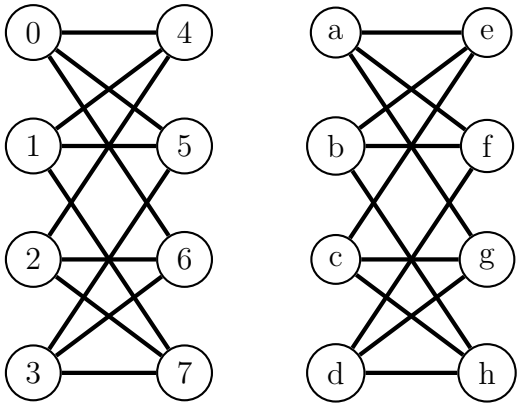
Neorientované grafy G_1, G_2 nazýváme *izomorfní*, pravě když existuje bijektivní zobrazení $f : V(G_1) \longrightarrow V(G_2)$, kde platí

$$\{v, w\} \in E(G_1) \iff \{f(v), f(w)\} \in E(G_2)$$

Orientované grafy G_1, G_2 nazýváme izomorfní, pravě když existuje bijektivní zobrazení $f : V(G_1) \longrightarrow V(G_2)$, kde platí

$$\langle v, w\rangle \in E(G_1) \iff \langle f(v)f(w)\rangle \in E(G_2)$$

Vztah izomorfismu značíme $G_1 \cong G_2$.



Obrázek 5: Izomorfní grafy.

Oba grafy z obr. 5 jsou navzájem izomorfní, existuje k ním bijektivní zobrazení takové, že $f(0) = a, f(1) = b, f(2) = c, f(3) = d, f(4) = e, f(5) = f, f(6) = g, f(7) = h$.

Definice 6 (Cestování)
Sled grafu v G je posloupnost vrcholů a hran

$$v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n,$$

kde $v_i \in V(G)$ a $e_i \in E(G)$ a platí, že

- $e_i = \{v_i - 1, v_i\}$ pro $i = 1, \ldots, n$ je-li G neorientovaný,
- $e_i = \langle v_i - 1, v_i \rangle$ pro $i = 1, \ldots, n$ je-li G neorientovaný.

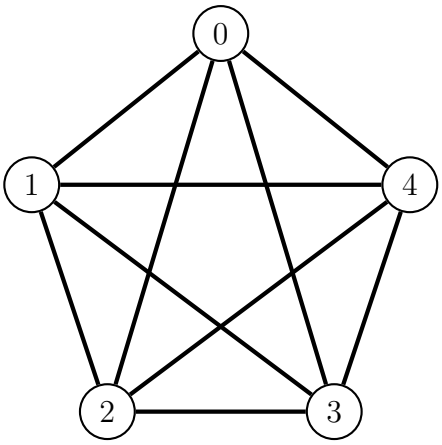
Délka sledu je rovna, počtu hran ve sledu. Sled $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$, se nazývá

- *tah*, neopakuje-li se v něm žádná hrana,
- *cesta*, neopakuje-li se v něm žádný vrchol,
- *kružnice*, je-li sled neorientovaný a platí $v_0 = v_n$ a kromě v_0 a v_n jsou vrcholy různé.
- *cyklus*, je-li sled orientovaný a platí $v_0 = v_n$ a kromě v_0 a v_n jsou vrcholy různé.

Můžeme si všimnout, že z podmínky pro cestu nemůže nastat aby se opakovali hrany, proto je každá cesta zároveň tahem, zatímco tah vždy cestou být nemusí.

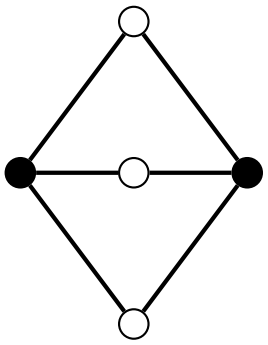
Definice 7 (Dostupnost, souvislost, úplnost grafu)
Řekneme, že vrchol w je *orientovaně (neorientovaně) dostupný* z vrcholu v, jestliže existuje orientovaný (neorientovaný) sled vedoucí z vrcholu v do vrcholu w. Graf G je *souvislý*, pokud libovolné dva vrcholy $v \in V(G), w \in V(G), v \neq w$ jsou spojeny sledem. Graf, který není souvislý se skládá z množiny souvislých grafu, které tvoří maximální souvislé podgrafy grafu G. *Úplný graf* je prostý graf bez smyček, takový že v něm jsou každé dva vrcholy spojené hranou.

Zřejmě každý úplný graf je souvislým grafem viz. Obr 6.



Obrázek 6: Úplný graf.

Definice 8 (Bipartitní graf)
Bipartitní graf je graf G, jehož vrcholy můžeme rozdělit do dvou disjunktních množin S, T, tak že všechny hrany mají jeden vrchol v množině S a druhý v T. Množiny S, T nazýváme *stranami* bipartitního grafu.



Obrázek 7: Bipartitní graf.

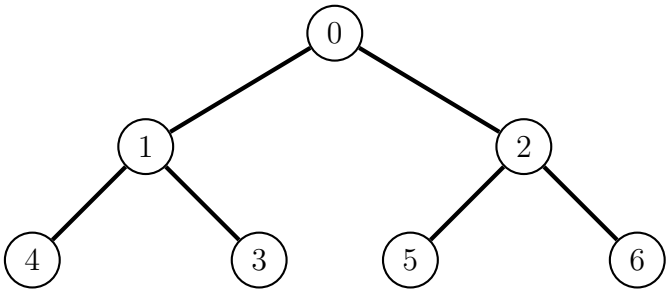
Řekneme, že graf je acyklický pokud neobsahuje žádný cyklus.

Definice 9 (Strom, kostra)
Strom je *acyklický* souvislý graf. Disjunktní množina stromu se nazývá *lesem*. Kostra souvislého grafu G_1 je podgraf G_2, který obsahuje všechny hrany grafu G_1 a je stromem.

Graf G je stromem, právě když splňuje některou z následujících podmínek:

- G má $|V| - 1$ hran a neobsahuje kružnici,
- G má $|V| - 1$ hran a je souvislý,
- G je souvislý a odebráním libovolné hrany se rozdělí na dva souvislé grafy,

- G je acyklický a přidáním jakékoliv hrany vznikne kružnice,
- právě jedna cesta vede mezi libovolnými dvěma vrcholy v grafu G.



Obrázek 8: Strom.

3 Paralelní programování

Konkurentní programování je jeden ze způsobu jak zparalelizovat sekvenční program. V tomto textu se budeme zabývat pouze paralelizaci algoritmu konkurentním programováním.

Konkurentní programy obsahují dva a více procesů, které pracují společně na dané úloze. Každý proces je sekvenční program, konkrétně sekvence příkazu, které jsou vykonávané jedna za druhou.

Procesy v konkuretním programu pracují spolu tak, že spolu komunikují. Komunikace může probíhat pomocí *sdílených proměnných* nebo *posílání zpráv*. Při sdílených proměnných, procesy zapisují do sdílených proměnných, které jsou čtené jinými procesy. Při posílání zpráv, procesy posílají jeden druhému zprávy.

Při použití jakékoliv komunikace, je třeba procesy synchronizovat jak uvidíme dále. Jsou dva základní typy jak synchronizovat procesy *vzájemné vyloučení* a *podmíněná synchronizace*. Vzájemné vyloučení je problém takový, který zajišťuje aby *kritická sekce* (více viz. [4]) nebyla vykonávaná více vlákny ve stejný čas. Podmíněná synchronizace je problém, kdy je proces pozdržen do doby, než je splněna určitá podmínka.

3.1 Paradigmata paralelního programování

Máme čtyři základní paradigmata pro psaní paralelního programu: *iterativní paralelismus*, *rekurzivní paralelismus*, *producent a konzument*, *klient a server*. V tomto textu využijeme iterativní paralelismus a producent a konzument, které dále popíšeme.

Iterativní paralelismus je používán v situacích, kdy má program několik identických procesů se stejným kódem ale rozlišnými daty, které obsahují jeden nebo více cyklů. Procesy tedy pracují spolu, tak aby vyřešily jednu úlohu. Komunikace může probíhat pomocí sdílených proměnných nebo posíláním zpráv.

U producenta a konzumenta, producent periodicky vytváří data a ukládá je do společné paměti (bufferu), konzument tyto data vyzvedne a zpracuje je.

3.2 Multivláknové programování

V tomto textu se budeme zabývat paralelizaci grafových algoritmů, pomocí více vláken. Při spuštění programu vzniká nový proces, který obsahuje jedno nebo více vláken. Samotné vlákna vykonávají kód programu a sdílejí spolu v rámci jednoho procesu operační paměť. Z toho vyplývá, že při konkuretním programování s více vlákny, je komunikace mezi vlákny, realizovaná pomocí sdílených proměnných, což sebou nese určité komplikace. Komunikace pomocí zasílaní zpráv je také možná ale v praxi se příliš nevyskytuje.

Uvažujme následující algoritmus 1 s globální proměnnou *zustatek*. Mějme *zustatek = 100* a předpokládejme, že obě vlákna zavolají funkci *VYBER(60)* současně. Nastane-li situace kdy obě vlákna vyhodnotí podmínku na řádku 2 před tím, než jakékoliv vlákno odebere částku ze zůstatku, tak nastane situace kdy je zůstatek v záporných číslech. Tuto část kódu, kde je čteno a zapisováno do sdílené proměnné více vlákny, nazýváme problémem *kritické sekce* (část kódu, která nemůže být vykonávaná více vlákny současně). Proto je potřeba vlákna synchronizovat aby nenastala situace, kdy více vláken současně vykonává kód v kritické sekci. Dále požadujeme aby nedocházelo k *uváznutí* (deadlock) vláken, tzn. že pokud se vlákna snaží vstoupit do kritických sekcí, tak jedno z nich musí někdy uspět. Poslední podmínkou pro splnění korektnosti synchronizace je absence *vyhladovění*, tzn. pokud se nějaké vlákno snaží vstoupit do kritické sekce, tak musí někdy uspět.

Algoritmus 1 Problém kritické sekce

```
1: procedure VYBER(castka)
2:     if castka <= zustatek then
3:         zustatek = zustatek − castka
```

Synchronizaci realizujeme pomocí *synchronizačních primitiv*. Mezi základní synchronizační primitiva patří *zámky*, *bariéry*, *monitory*, *semafory* více viz. kapitola 4.2.

4 Implementace

Algoritmy, které budou představeny v následující kapitole jsou implementované v programovacím jazyce C#. Dále v této sekci uvedeme možnosti reprezentace grafů, implementované API, a synchronizační primitiva v jazyce C#.

4.1 Reprezentace grafu

Grafy lze v počítači reprezentovat mnoha způsoby, kde žádná reprezentace grafu, není natolik univerzální aby se dala výhodně použít v jakékoliv situaci. Každá reprezentace má své výhody a nevýhody viz. tabulka 1.

Matice sousednosti, reprezentována dvourozměrným booleovským polem o velikosti $|V| \times |V|$, která zachycuje sousednost vrcholů, tj. pokud vede hrana mezi vrcholy v a w, pak je na pozici (v, w) true. Jedná-li se o neorientovaný graf pak je matice symetrická. Jestliže pracujeme s ohodnoceným grafem, tak místo logických hodnot můžeme volit celočíselné hodnoty. Pokud mezi některými vrcholy nevede ohodnocená hrana, zapíšeme na konkrétní pozici vhodnou konstantu, která nemůže být ohodnocením žádné hrany,

datová struktura	**paměťová složitost**	**přidání hrany**	**jsou v a w sousední**	**projít sousedy** v
matice sousednosti	$\|V\|^2$	1	1	$\|V\|$
seznam hran	$\|E\|$	1	$\|E\|$	$\|E\|$
seznam sousedů	$\|E\|$	1	*stupeň*(v)	*stupeň*(v)

Tabulka 1: Srovnání složitosti datových struktur

záleží to hlavně na vlastnostech dané úlohy, například při hledání nejkratší cesty, můžou být vhodné konstanty int.MAX nebo ∞. Výhodou této struktury je možnost zjištěni sousednosti dvou hran v konstantním čase, naopak nevýhodou je paměťová složitost, která je $\Theta(|V|^2)$ a nemožnost reprezentovatelnosti paralelních hran. Je-li je graf řídký (obsahuje málo hran) tak je tato reprezentace nevhodná, vzhledem k tomu, že iterace všech sousedů zabere $\Theta(|V|)$ času.

Seznam sousedů, kde udržujeme vrcholově indexované pole odkazujících na seznam sousedů. Je-li graf ohodnoceny, tak použijeme místo seznamu sousedů, seznam incidentních hran, které budou obsahovat příslušné ohodnocení. Časová složitost zjištěni sousednosti vrcholů v a w je $O($*stupeň*$(v))$, proto máme-li hustý graf, tak tato datová struktura nemusí být příliš vhodná.

Seznam hran, jednoduše udržujeme pole, obsahující objekty (hrany), které nesou informace o sousednosti vrcholů. Pokud je graf ohodnocený, tak k objektům přidáme informaci o ohodnocení. Problémem této datové struktury je časová složitost zjištěni sousednosti vrcholů v a w, která je $O(|E|)$, to samé platí k získání všech sousedních vrcholů.

4.2 Synchronizace vláken

Základní prvek pro synchronizaci, který umožňuje vzájemné vyloučení je *lock*. Ukázka použití *lock*

```
1   static readonly object locker = new object();
2   static int zustatek = 100;
3
4   static void Vyber(int castka)
5   {
6     lock (locker)
7     {
8       if (castka <= zustatek)
9         zustatek -= castka
10    }
11  }
```

Zdrojový kód 1: C#

Pouze jednomu vláknu je umožněno v daný čas zamknout synchronizační objekt (v

tomto případě *locker*) a jakékoliv další vlákna, která se pokusí zamknout již zamknutý objekt, jsou uspána dokud není zámek uvolněn. Po uvolnění zámku je vybráno jedno vlákno, které bude probuzeno (více viz. [5]).

Další možnosti jak synchronizovat vlákna je bariéra. Pomocí bariéry můžeme vytvořit bod v programu, kde musí všechna zaregistrovaná vlákna v bariéře dorazit, než jim bude dovoleno pokračovat. *Synchronizace bariérou* se používá hlavně v situacích, kdy máme n vláken, které vykonávají stejný kód s různými daty (iterativní paralelismus). Základním atributem iterativních algoritmů je, že každá iterace závisí na výsledcích předchozí iterace.

4.3 Uživatelské rozhraní

K reprezentací neorientovaného grafu slouží následující API (tabulka 2), které definuje základní operace, jaké můžeme s grafem provádět. Tato API obsahuje dva konstruktory, které vytvoří prázdny graf nebo graf podle souboru. Dále obsahuje vlastnosti třídy pro získání počtu vrcholů a hran v grafu a metody pro přidaní hrany do grafu a získání všech sousedních vrcholů s daným vrcholem.

public class **Graph**		
	Graph(int V)	*vytvoří prázdný graf s V vrcholy*
	Graph(string fileName)	*vytvoří graf podle souboru*
int	NumberOfVertices	*počet vrcholů*
int	NumberOfEdges	*počet hran*
void	AddEdge(int v, int w)	*přidá hranu*
IEnumerable<int>	Adjacent(int v)	*sousední vrcholy s v*

Tabulka 2: API pro neorientovaný graf.

Nastává otázka jak reprezentovat ohodnocení hran v grafu. Pokud graf reprezentujeme maticí sousednosti tak je situace jednoduchá, jak již bylo zmíněno, tak místo booleovských hodnot bude obsahovat délky hran. Při použití seznamu sousedů zavedeme novou třídu *Edge* viz. tabulka 3.

public class **Edge** : IComparable<Edge>		
	Edge(int v, int w, double weight)	*konstruktor*
double	Weight	*délka hrany*
int	Either()	*„první“ vrchol*
int	Other(int v)	*„druhý“ vrchol*
int	CompareTo(Edge other)	*porovnání*

Tabulka 3: API pro ohodnocenou hranu.

Taktéž je potřeba reprezentovat ohodnocený graf, který reprezentován následujícím API (tabulka 4).

public class **EdgeWeightedGraph**		
	EdgeWeightedGraph(int V)	*vytvoří prázdný graf s V vrcholy*
	EdgeWeightedGraph(string fileName)	*vytvoří graf podle souboru*
int	NumberOfVertices	*počet vrcholů*
int	NumberOfEdges	*počet hran*
void	AddEdge(Edge e)	*přidá hranu*
IEnumerable<Edge>	Adjacent(int v)	*incidentní hrany s v*
IEnumerable<Edge>	Edges(int v)	*všechny hrany*

Tabulka 4: API pro ohodnocený graf.

5 Paralelní algoritmy

V této sekci si představíme algoritmy pro výpočet koster, nejkratších cest a barevnosti grafu. Dále u těchto algoritmu uvedeme způsoby, jak lze dosáhnout paralelizace a v další kapitole provedeme experimentální porovnání.

5.1 Greedy algoritmy

Před přestavením samotných algoritmů, se krátce podíváme na techniku návrhů algoritmu, kterou využijeme u všech představených algoritmu. *Greedy algoritmus* je založen na tom, že v každém svém kroku vybere možnost, která se zdá v daném momentě nejlepší. To znamená, že vybere lokální optimální řešení bez ohledu na to, zda vede k nejlepšímu řešení. Tato technika je podrobně popsána v [10].

5.2 Nejkratší cesty

Mějme orientovaný nebo neorientovaný graf $G = (V, E)$ a hranové ohodnocení $w : E \to \mathbb{R}$. Dvojici (G, w) říkáme *síť* a číslo $w(e)$ se nazývá *délka hrany* e. Ohodnocení hrany budeme brát jako délku v eukleidovském prostoru a vrcholy jako body v prostoru, což je výhodné pro intuitivní chápaní. Ale ohodnocení hrany může obecně reprezentovat jakoukoliv veličinu např. čas, pravděpodobnost, atd. Pro každou cestu $W = (e_1, \ldots, e_n)$ definujeme *délku cesty* jako

$$w(W) = w(e_1) + \ldots + w(e_n) \tag{1}$$

Dále zavedeme *vzdálenost* $d(v, w)$ mezi dvěma vrcholy v, w v (G, w) takto:

$$d(v,w) = \begin{cases} \infty, & \text{pokud mezi } v \text{ a } w \text{ nevede cesta} \\ min\{w(W) : W \text{ je cesta z } v \text{ do } w\} & \text{jinak} \end{cases} \tag{2}$$

Jakákoliv cesta W, která je dána funkcí (2) se nazývá *nejkratší cesta* mezi vrcholy v a w. Všimněme si, že vždy platí $d(v, v) = 0$, jelikož prázdna suma je braná jako hodnota 0.

V této sekci se tedy budeme zabývat problémem hledání nejkratší cesty, tak že bude dán ohodnoceny graf G a počáteční vrchol v, kde budeme chtít implementovat odpovědi na následující dotazy. *Vede mezi dvěma vrcholy v a w cesta?* Pokud ano, tak najdi *nejkratší* cestu.

O grafech budeme předpokládat, že mají následující vlastnosti:

- Délky hran jsou pozitivní.
- Paralelní hrany a smyčky můžou být přítomné.
- Nejkratší cesty nejsou nezbytně unikátní.
- Cesty jsou neorientované.
- Všechny vrcholy nemusí být dostupné.

Výsledkem výpočtu bude strom nazývaný jako *strom nejkratších cest*, který obsahuje nejkratší cesty z počátečního vrcholu v ke všem dostupným vrcholům z v.

Algoritmy pro nejkratší cesty implementují následující API (tabulka 5).

public class **SP**

	SP(EdgeWeightedGraph G, int s, int numberOfThreads)	*konstruktor*
double	DistTo(int v)	*délka z vrcholu s do v*
int	HasPathTo(int v)	*existuje cesta mezi s v*
IEnumerable<Edge>	PathTo(int v)	*cesta od s do v*

Tabulka 5: API pro ohodnocený graf.

Konstruktor vytvoří strom nejkratších cest. Metoda *DistTo* umožňuje vrátit délku k danému vrcholu v konstantním čase, *PathTo* vrací seznam hran tvořících nejkratší cestu.

5.2.1 Dijkstrův algoritmus

Dijkstrův algoritmus 2 pracuje následovně. Vstupem je neorientovaný graf G, jeho hranové ohodnocení w a počáteční vrchol s. Výstupem algoritmu je strom nejkratších cest. Algoritmus si udržujeme množinu vrcholů T a pole l. Množina T obsahuje vrcholy ke kterým již byla nalezena nejkratší cesta z počátečního vrcholu, na začátku je tato množina nastavena na $T = \{s\}$. Pole l obsahuje reálná čísla, která reprezentují prozatímní nejkratší cestu nalezenou k vrcholu v. Na začátku se nastaví $l[v] = w(s, v)$ pro všechny vrcholy $v \in V(G)$ ($l[s] = 0$). Algoritmus je založený na greedy přístupu tak, že v každé iteraci dochází k nalezení nejkratší cesty mezi počátečním vrcholem a vrcholem u, který není obsažen v T. Tento vrchol u se přidá do množiny T. Dále může být upravena hodnota pro vrchol v v poli l, to se stane když přes nově přidaný vrchol u vede cesta, která je kratší než doposud nalezená cesta k v. Konkrétně

$$l[v] = min(l[v], l[u] + w(u, v)).$$

Algoritmus je ukončen po přidání všech vrcholů do T.

Algoritmus 2 Dijkstrův algoritmus

```
1: procedure DIJKSTRA(G, w, s, p)
2:     T ← {s};
3:     for all v ∈ (V \ T) do
4:         l[v] ← w(s, v);          ▷ Pokud hrana mezi vrcholy neexistuje tak se nastaví ∞
5:     while T ≠ V do
6:         najdi vrchol u takový, že l[u] = min{l[v]|v ∈ (V \ T)};
7:         T ← T ∪ {u};
8:         for all v ∈ (V \ T) do l(v) ← min(l(v), l[u] + w(u, v));
```

Paralelizace. Paralelní formulace Dijkstrova algoritmu vychází z [7]. Mějme p vláken a ohodnocený graf G s n vrcholy. Dále mějme množiny $V_0 \dots V_{p-1}$, které tvoří rozklad na množině $V(G)$ a každá z těchto množin obsahuje n/p po sobě jdoucích vrcholů.

Ke každému í-tému vláknu T_i přiřadíme množinu V_i, se kterou bude vlákno pracovat. Každé z těchto vláken bude pracovat s části pole l, která koresponduje s jeho množinou V_i (vlákno T_i pracuje s částí $l[u]$, $u \in V_i$). V každé iteraci vlákna T_i uloží do globálního pole $localMinsVertex$ vrchol u, který má lokální minimální délku v poli l. Po vyhledání lokálního minima, je vlákna potřeba synchronizovat bariérou. Poté co všechny vlákna dorazí k bariéře, tak dojde k výpočtu globálního minimálního vrcholu u přes pole $localMinsVertex$ a tento vrchol je přidán do množiny T. Na konci iterace je ještě potřeba aktualizovat pole l

$$\forall v \in (V_i \setminus T), l[u] = min(l[u], l[v] + w(u, v)).$$

Podmínka ukončení je stejná jako u sekvenčního algoritmu viz. 3.

Algoritmus 3 Paralelizace Primova algoritmu

```
1: procedure PRIM(G, w, s)
2:     nastav délky hran;        ▷ Pokud hrana mezi vrcholy neexistuje tak se nastaví ∞
3:     for i ← 0 ... p − 1 do in parallel
4:         while T ≠ V do
5:             najdi lokální minimum;
6:             synchronizace bariérou;        ▷ jedno z vláken najde globální minimum a
   nastaví vrchol u
7:             aktualizuj délky hran;
```

5.3 Minimální kostra

Mějme síť (G, w). Pro každou množinu T, která je podmnožinou $G(E)$, definujeme délku hran množiny T takto:

$$w(T) = \sum_{e \in T} w(e)$$

Kostra grafu G se nazývá *minimální kostra* (MK), pokud délka množiny hran je minimální, vzhledem k ostatním kostrám.

O grafech budeme předpokládat, že mají následující vlastnosti:

- Délky hran můžou být nulové a negativní.
- Paralelní hrany a smyčky můžou být přítomné.
- Graf je souvislý.
- Cesty jsou neorientované.
- Délky všech hran jsou různé. Pokud hrany můžou stejné délky, minimální kostra nemusí být unikátní.

Připomeňme si dvě ekvivalentní vlastnosti stromu:

- G je souvislý a odebráním libovolné hrany se rozdělí na dva souvislé grafy,

- G je acyklický a přidáním jakékoliv hrany vznikne kružnice,

Tyto vlastnosti tvoří základ pro dokazování základní vlastnosti o MK, která vede k algoritmům pro MK.

Mějme graf G. *Řez grafu* G je rozdělení $S = \{X, X'\}$ množiny $V(G)$ do dvou neprázdných podmnožin. *Přechodová hrana* řezu je hrana, která spojuje dva vrcholy, které leží v rozdílných množinách řezu.

Věta 10 (Vlastnost řezu)
Mějme daný jakýkoliv řez ohodnoceného grafu, přechodová hrana řezu s minimální délkou patří do MK grafu.

Důkaz
Mějme minimální přechodovou hranu e a předpokládejme, že T tvoří minimální kostru. Předpokládejme, že T neobsahuje e. Dále uvažujme graf, který vznikne přidáním hrany e do T. V tomto grafu vznikne kružnice, která obsahuje e a kružnice musí obsahovat alespoň jednu další přechodovou hranu, např. f, která má větší délku než e. Můžeme obdržet kostru s ostře menší délkou odstraníme-li f a přidáme e. Tím jsme došli ke sporu. Proto MK musí obsahovat přechodové hrany s minimální délkou každého řezu. □

Algoritmy pro MK implementují následující API (tabulka 6).

public class **MST**		
	MST(EdgeWeightedGraph G, int s, int numberOfThreads)	*konstruktor*
double	Weight	*délka MK*
IEnumerable<Edge>	Edges()	*hrany patřící do MK*

Tabulka 6: API pro MK.

5.3.1 Primův algoritmus

Primův algoritmus je založený na greedy přístupu. Algoritmus začne vybráním libovolného startovního vrcholu. Dále přidává vrcholy a hrany u kterých je jisté, že patří do MK. Algoritmus pracuje do té doby, dokud nebyly přidány všechny vrcholy.

Algoritmus 4 používá množinu vrcholů T, pomocí které bude průběžně získávat řezy grafu. Dále používá pole $d[n]$, které obsahuje na pozici $d[v]$ pro každý vrchol $v \in (G(V) \setminus T)$, délku jeho nejkratší přechodové hrany. Na začátku T obsahuje zvolený počáteční vrchol s a dále nastaví $d[v] = 0$, $d[v] = w(s, v)$ pro všechny $v \in (G(V) \setminus T)$, pokud přechodová hrana existuje, jinak $d[v] = \infty$. Během každé iterace, se vybere vrchol v z množiny $T \setminus V$, jehož délka nastavena v poli d je minimální vzhledem k ostatním vrcholům

a následně se přidá do T. Po přidání vrcholu do množiny vznikne nový řez, proto se musí provést aktualizace vzdálenosti přechodových hran $d[w] = min(d[w], w(v, w))$ pro všechny $w \in (G(V) \setminus T)$. Algoritmus se ukončí po přidání všech hran do množiny T.

Algoritmus 4 Primův algoritmus

```
1: procedure PRIM(G, w, s)
2:     T ← {s};
3:     d[s] ← 0;
4:     for all v ∈ (V \ T) do
5:         d[v] ← w(s, v);        ▷ Pokud hrana mezi vrcholy neexistuje tak se nastaví ∞
6:     while T ≠ V do
7:         najdi vrchol u takový, že d[u] = min{d[v]|v ∈ (V \ T)};
8:         T ← T ∪ {u};
9:         for all v ∈ (V \ T) do d(v) ← min(d(v), w(u, v));
```

Paralélnost. Paralelní formulace Primova algoritmu vychází z [7]. Algoritmus funguje na stejném principu jako Dijsktruv algoritmus. Mějme p vláken a ohodnocený graf s n vrcholy. Dále mějme množiny $V_0 \dots V_{p-1}$, které tvoří rozklad na množině $V(G)$ a každá z těchto množin obsahuje n/p po sobě jdoucích vrcholů. Ke každému í-tému vláknu T_i přiřadíme množinu V_i, se kterou bude vlákno pracovat. Každé z těchto vláken bude pracovat s části pole d, která koresponduje s jeho množinou V_i (vlákno T_i pracuje s částí $d[v]$, $v \in V_i$). Každé vlákno T_i během svoji iterace spočítá svojí minimální přechodovou hranu $l[i] = min\{d[v]|v \in V_i\}$ (l je globální pole, kde jsou uloženy minimální přechodové hrany všech vláken). Po vyhledání lokálního minima, je vlákna potřeba synchronizovat bariérou. Poté co všechny vlákna dorazí k bariéře, tak dojde k výpočtu globální minimální hrany e pomocí pole l. Dále do množiny T přidáme ten vrchol g, který leží v hraně e a doposud není v množině T obsažen (jeden vrchol z hrany e musí vždy ležet v množině T, protože pracujeme pouze s přechodovými hranami). Nakonec po nalezení globálního minima, si všechny vlákna T_i aktualizuji hodnoty v poli $d[v] = min(d[v], w(g, v))$ pro všechna $v \in V_i$.

Algoritmus 5 Paralelizace Primova algoritmu

```
1: procedure PRIM(G, w, s, p)
2:     for i ← 0 . . . p − 1 do in parallel
3:         nastav délky hran;   ▷ Pokud hrana mezi vrcholy neexistuje tak se nastaví ∞
4:         najdi lokální minimum;
5:         while T ≠ V do
6:             synchronizace bariérou;        ▷ jedno z vláken najde globální minimum a
   nastaví vrchol g
7:             aktualizuj délky hran;
8:             najdi lokální minimum;
```

5.4 Barvení grafu

V této sekci představíme některé paralelní algoritmy pro barvení grafu, které jsou založené na heuristických sekvenčních algoritmech. O praktických aplikacích barvení grafu a jejich paralelizaci pojednávají publikace [8] a [9].

Barvení grafu G, přiřazuje barvu každému vrcholu $v \in V(G)$ tak, že libovolné sousední vrcholy mají rozdílnou barvu. Přesněji máme zobrazení $c : V \to C$, kde množinu C interpretujeme jako množinu barev, a požadujeme aby platilo $c(v) \neq c(w)$ pro všechny $(v, w) \in E(G)$. *Chromatické číslo* $\chi(G)$ je minimální počet barev, které jsou potřeba k obarvení grafu G.

Nalezení barevnosti grafu s minimálním počtem barev je NP-Těžký problém. Jednodušší problém je obarvit graf s malým počtem barev, kde počet barev nemusí být nezbytně nejmenší. Existují algoritmy s polynomiální složitosti, které řeší tento problém s relativně malým počtem barev.

Ukážeme zde jak lze paralelizovat známe sekvenční algoritmy Largest-Degree-First, Smallest-Degree-First a porovnáme je s Jones-Plassmanovim algoritmem.

Algoritmy pro barvení grafu implementují následující API 7.

public class **GC**

	GC(Graph G, int numberOfThreads)	*množiny barev*
IList<ColorSet>	ColorsSets()	*množiny barev*
int	ColorsCount()	*počet použitých barev*
int	ColorOfVertex(int v)	*barva vrcholu v*

Tabulka 7: API barvení grafu.

5.4.1 Sekvenční greedy algoritmus

Před tím než si představíme paralelní algoritmy, tak se podíváme na jednoduchý sekvenční algoritmus 6. Barvy hran budeme reprezentovat pomocí čísel $1, 2 \ldots$. Mějme danou libovolnou uspořádanou sekvenci vrcholů $v_0, \ldots, v_{n-1}$ grafu G, budeme postupně přiřazovat vrcholům barvy, kde vždy použijeme nejmenší možnou barvu.

Algoritmus 6 Sekvenční algoritmus pro barvení

```
1: procedure COLOR(G, v)
2:     c(v_0) ← 0
3:     for i = 1 ... n − 1 do
4:         S ← {Barvy všech sousedních vrcholů v_i}
5:         c(v_i) ← Nejmenší barva která není v S
```

Věta 11 (Horní hranice algoritmu)
Mějme graf G. Pak $\chi(G) \leq \Delta(G)+1$, kde $\Delta(G)$ představuje maximální stupeň vrcholu v G.

Důkaz
Mějme libovolně uspořádanou sekvenci vrcholů $v_0 \dots v_{n-1}$ grafu G. Vrchol v_i má nejvíce $\Delta(G)$ sousedních předchůdců při obarvování v kroku 5, proto maximálně $\Delta(G)$ barev nelze přiřadit v_i, z toho plyne $c(v_i) \leq \Delta(G) + 1$ □

5.4.2 Paralelní barvení grafu

Paralelní barvení grafu je založeno na jednoduchém pozorování, každá množina nezávislých vrcholů může být obarvená paralelně, kde nezávislou množinou se myslí množina nesousedních vrcholů. Obecný algoritmus je ukázán v 7.

Algoritmus 7 Obecný algoritmus pro paralelní obarveni vrcholů

```
1: procedure PARALLEL-COLOR(G)
2:     U ← G(V)
3:     G' ← G
4:     while U ≠ ∅ do in parallel
5:         I ← vyber nezávislou množinu z G'
6:         Obarvi všechny vrcholy v I
7:         U ← U \ I
8:         G' ← graf indukovaný množinou U
```

Algoritmus 7 má řadu variant, které se liší tím, jak je vybrána nezávislá množina vrcholů a jak jsou tyto vrcholy obarveny. Některé z těchto algoritmu představíme v této sekci.

5.4.3 Jones-Plassmann algoritmus

Algoritmus v prvním kroku přiřadí všem vrcholům $v \in V(G)$ náhodnou váhu (náhodné číslo z množiny přirozených čísel), označovanou jako $w(v)$. Poté v každé iteraci paralelně provede vyhledání nezávislé množiny vrcholů, metodou která přidá vrchol do množiny jehož váha je lokální maximum. Při kolizi vah se použije unikátní identifikační číslo každého vrcholu. Dále dojde k paralelnímu obarvení vrcholů, kde bude vrcholu přiřazena nejmenší dostupná barva vzhledem k jeho okolí. Tato procedura se opakuje dokud graf nebude úspěšně obarven viz. algoritmus 8.

5.4.4 Largest-Degree-First

Largest-Degree-First algoritmus může být paralelizován použitím podobné metody, která je použitá u Jones-Plassmann algoritmu. Jediný rozdíl je vtom jak jsou přiřazeny váhy vrcholům, místo přiřazení náhodného čísla má vrchol přiřazenou váhu, která je určena jeho stupněm. Při konfliktu se použije unikátní identifikační číslo každého vrcholu. V této metodě vrcholy nejsou obarveny v náhodném pořadí, ale v sestupném pořadí vzhledem ke stupni vrcholů.

Algoritmus 8 Joness-Plassmann algoritmus

```
1: procedure JONES-PLASSMANN(G)
2:     U ← G(V)
3:     while U ≠ ∅ do in parallel
4:         for all v ∈ U do in parallel
5:             I ← I ∪ {v| tak, že pro všechny sousední vrcholy u w(v) > w(u)}
6:             for all v ∈ I do in parallel
7:                 nastav nejmenší přípustnou barvu v
8:         U ← U \ I
```

Tento přístup má za cíl použít menší počet barev než Jones-Plassmann algoritmus.

5.4.5 Block partition algoritmus

Algoritmus 9 je rozdělen do tří fázi. V první fázi je množina vrcholů $V(G)$ rozdělena do p bloků $V_1, \ldots, V_p$, které tvoří rozklad množiny. Vrcholy v každém bloků jsou paralelně obarveny s použitím p vláken. Paralelní barvení se skládá z n/p paralelních kroků, které jsou synchronizované bariérou na konci každého kroku. Při barvení vrcholu, je možné pracovat s lokálními vrcholy a vrcholy z jiného bloku, proto může nastat situace, kdy dvě vlákna ve stejný čas budou přiřazovat barvy sousedním vrcholům. Pokud těmto vrcholům je přiřazená stejná barva, tak výsledné barvení se stane nevalidní a proto získané barvení nazýváme *pseudo barvení*. Ve druhé fázi každé vlákno T_i paralelně zkontroluje všechny vrcholy v V_i, jestli mají přiřazenou správnou barvu. Pokud nastane konflikt, tak hrana s menším indexem je vložena množiny A. V poslední fázi jsou vrcholy uloženy A obarveny sekvenčně.

Algoritmus 9 Block partition algoritmus

```
1: procedure BLOCK-PARTITION-COLORING(G)
2:     1.
3:     rozděl V do p ekvivalentních bloků V_1, ..., V_p
4:     for i = 1...p do in parallel
5:         for all v_j ∈ V_i do
6:             přiřaď nejmenší přípustnou barvu vrcholu v_j
7:             synchronizace bariérou
8:     2.
9:     for i = 1...p do in parallel
10:        for all v_j ∈ V_i do
11:            for all každý soused u vrcholu v_j do
12:                if color(v_j) = color(u) then
13:                    A ← A ∪ min(u, v_j)
14:    3.
15:    sekvenční obarvení vrcholů v A
```

5.4.6 Vylepšený block partition algoritmus

V této sekci ukážeme jak může být algoritmus 9 modifikován, tak aby byl počet použitých barev menší nebo při nejmenším stejný.

Tento vylepšený algoritmus se skládá ze čtyř fázi. První fáze je stejná jako v algoritmu 10. V tento moment máme k barev které jsou rozděleny množin $C_1, \ldots, C_k$. Druhá fáze se skládá z k kroků. V každém kroku i, jsou vrcholy množiny C_{k-i-1} přebarveny podobným způsobem jako ve fázi 1. Zbývající dvě fáze jsou stejné jako fáze dvě a tři algoritmu 9.

Algoritmus 10 Improved block partition algoritmus

```
1:  procedure IMPROVED-BLOCK-PARTITION-COLORING(G)
2:      1.
3:      stejné jako fáze 1 algoritmu 9          ▷ máme nezávislé množiny C_1, ..., C_k
4:      2.
5:      for j = k ... 1 do
6:          rozděl C_j do p ekvivalentních bloků V_1, ..., V_p
7:          for i = 1 ... p do in parallel
8:              for all v_j ∈ V_i do
9:                  přiřaď nejmenší přípustnou barvu vrcholu v_j
10:     3.
11:     Stejné jako fáze 2 algoritmu 9
12:     4.
13:     Stejné jako fáze 3 algoritmu 9
```

6 Experimentální výsledky

V této sekci experimentálně demonstrujeme výkon paralelních algoritmů představených v této práci. Experimenty jsou provedeny na procesoru Intel Core i7-2600 v programovacím jazyce C#.

6.1 Testovací grafy

Testovací grafy jsou rozděleny do dvou kategorií, z nichž první kategorie je použitá pro testování Primova a Dijkstrova algoritmu viz. tabulka 8 a druhá kategorie pro barvení grafu viz. tabulka 9. Sloupec n představuje kolik se nachází vrcholů v grafu a sloupec m představuje kolik se nachází hran v grafu.

Problém	n	m
dense	10000	4000000
medium	10000	61731

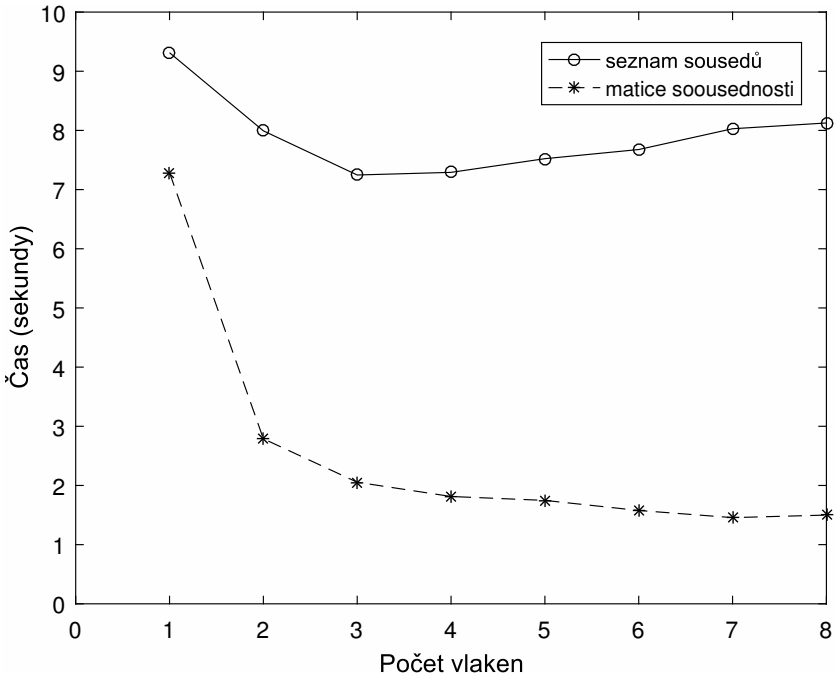
Tabulka 8: Testovací grafy pro Primův a Dijkstrův algoritmus.

Problém	*n*	*m*
large	1000000	7586063
dense	200000	4000000

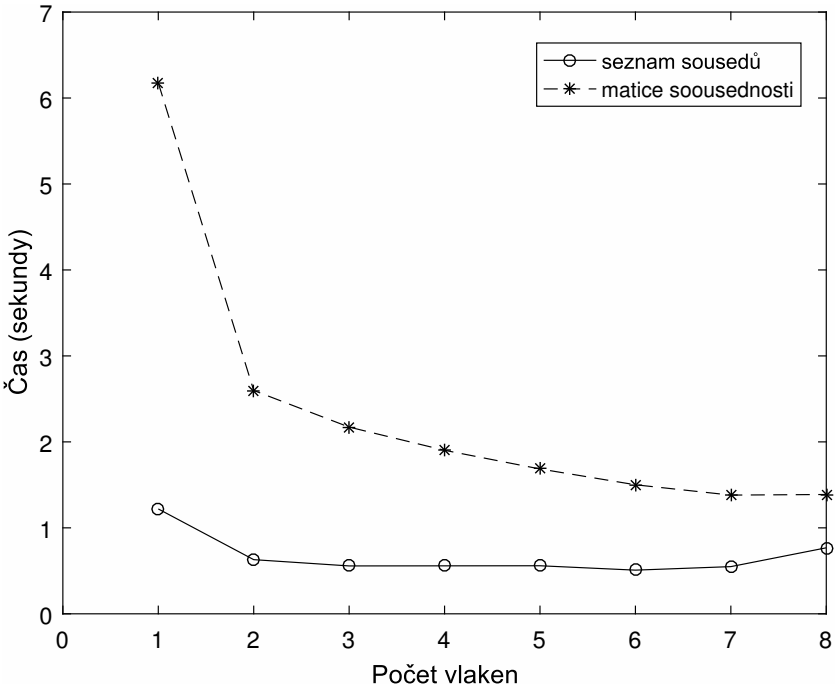
Tabulka 9: Testovací grafy pro barvení grafu.

6.2 Prim

Na následujících dvou obrázcích můžeme získat informace o efektivitě algoritmu, při zvyšujícím se počtu vláken. Lze zpozorovat, že reprezentace grafu má zřetelný vliv na výkon algoritmu. Zatímco na obr. 9, který představuje problém s hustým graf, má graf reprezentován maticí sousednosti výkon až šestkrát lepší než graf reprezentován seznamem sousedu. Ve druhém případě, má navrch graf reprezentován seznamem sousedů, avšak rozdíl už není tolik zřetelný. Taktéž lze vidět, že výkon matice sousednosti s přibývajícími vlákny roste, zatímco u seznamu sousedů se výkon dokonce s přibývajícími vlákny blíží zpět k výkonnosti s jedním vláknem.



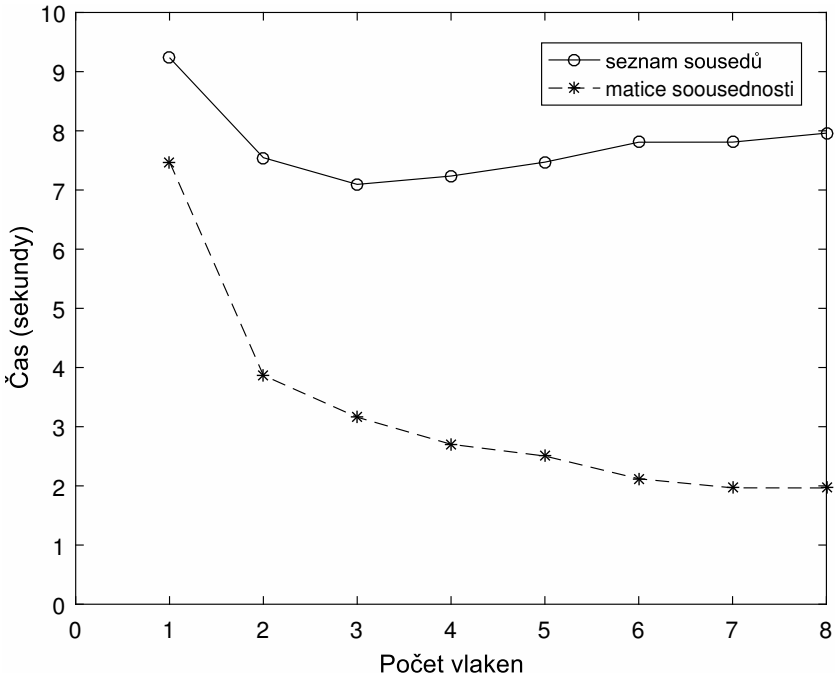
Obrázek 9: Porovnání Primova algoritmu na *medium* grafu.



Obrázek 10: Porovnání paralelního Primova algoritmu na *sparse* grafu.

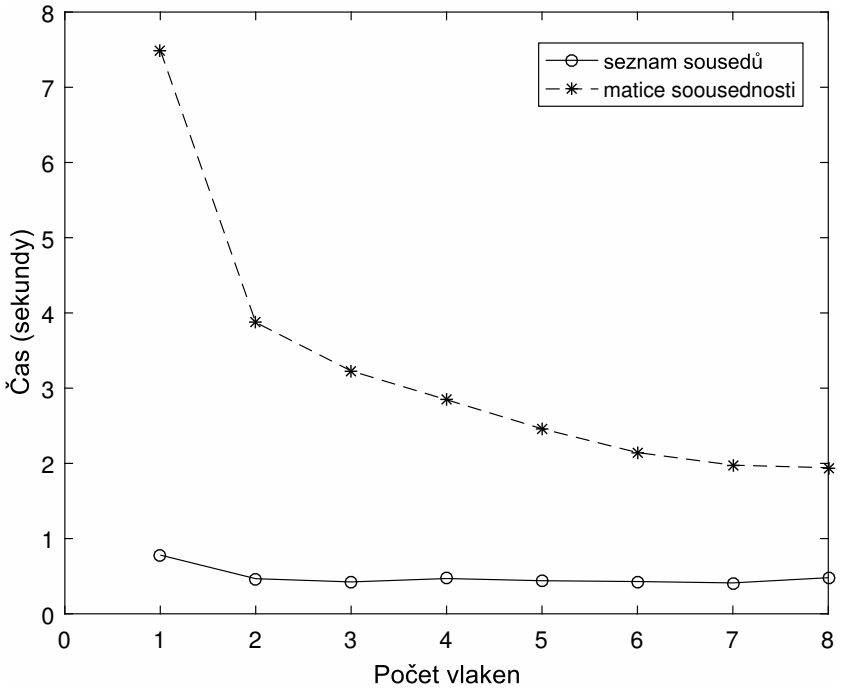
6.3 Dijkstra

Platí podobné poznatky jako v předchozí podkapitole.



Obrázek 11: Porovnání paralelního Dijkstrova algoritmu na *medium* grafu.

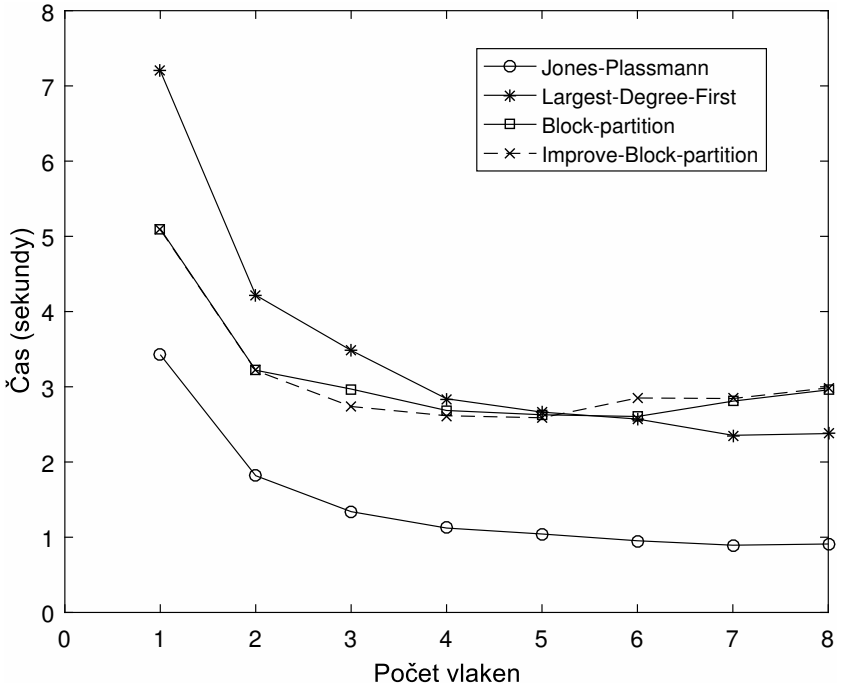
s



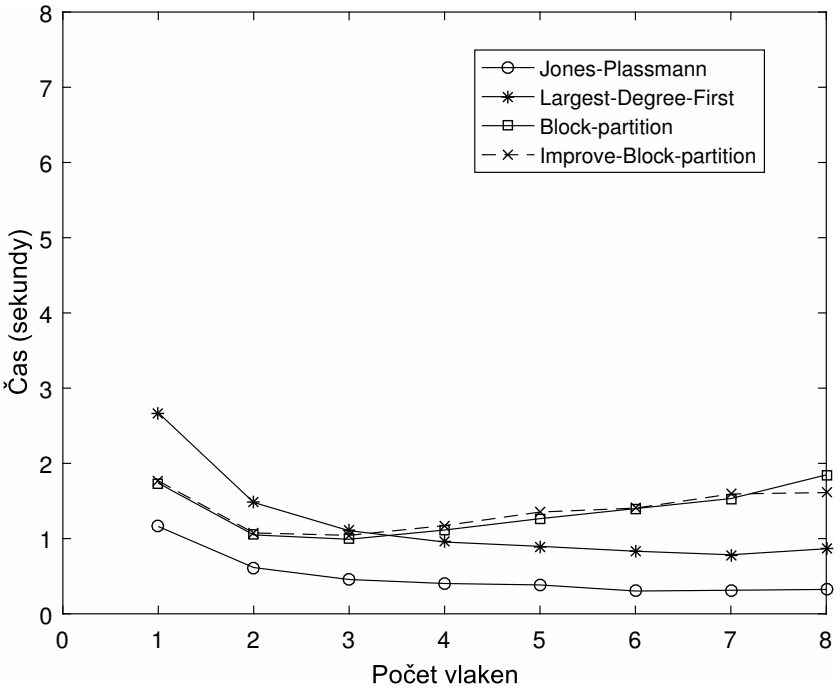
Obrázek 12: Porovnání paralelního Dijkstrova algoritmu na *sparse* grafu.

6.4 Barvení grafu

V této podkapitole můžeme navzájem srovnat jednotlivé algoritmy pro barvení grafu. Jednoznačně nejvýkonnější je Jones-Plassmannův algoritmus, který je přibližně dvakrát výkonnější než všechny ostatní algoritmy.



Obrázek 13: Porovnání paralelních algoritmů pro barvení grafu na *large* grafu



Obrázek 14: Porovnání paralelních algoritmů pro barvení grafu na *dense* grafu

Závěr

Výsledkem této práce je sada paralelních grafových algoritmu s příslušným uživatelským rozhraním. U každého algoritmu je možnost volby s jakou reprezentací grafu chceme pracovat a kolik vláken chceme použít pro výpočet.

Conclusions

The result of this work is a set of parallel graph algorithms with the corresponding user interface. For each algorithm, it is possible to choose with which representation we want to work with, and how many threads we want to use for the calculation.

Literatura

[1] SEDGEWICK, Robert a Kevin Daniel WAYNE. *Algorithms*. 4th ed. Upper Saddle River, NJ: Addison-Wesley, c2011. ISBN 978-0321573513.

[2] DEMEL, Jiří. *Grafy a jejich aplikace*. Praha: Academia, 2002. ISBN 80-200-0990-6.

[3] BĚLOHLÁVEK, Radim a Vilém VYCHODIL. *Diskrétní matematika pro informatiky II*. Olomouc.

[4] ANDREWS, Gregory R. *Foundations of multithreaded, parallel, and distributed programming*. Reading, Mass.: Addison-Wesley, c2000. ISBN 978-0201357523.

[5] ALBAHARI, Joseph a Ben ALBAHARI. *C# 6.0 in a nutshell: the definitive reference*. Sixth edition. In a nutshell (O'Reilly & Associates). ISBN 978-1491927069.

[6] JUNGNICKEL, D. *Graphs, networks, and algorithms*. Fourth edition. ISBN 978-3-642-32277-8.

[7] GRAMA, Ananth. *Introduction to parallel computing*. 2nd ed. New York: Addison-Wesley, 2003. ISBN 0-201-64865-2.

[8] ALLWRIGHT, J. R., et al. *A comparison of parallel graph coloring algorithms*. Technical report, SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1995.

[9] GEBREMEDHIN, Assefaw Hadish; MANNE, Fredrik. *Scalable parallel graph coloring algorithms*. Concurrency - Practice and Experience, 2000, 12.12: 1131-1146.

[10] CORMEN, Thomas H. *Introduction to algorithms*. 3rd ed. Cambridge, Mass.: MIT Press, c2009. ISBN 978-0-262-03384-8.

A Obsah přiloženého DVD

Na přiloženém DVD, jsou k nalezení všechny součástí této práce.

doc/
: Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace, tj. zdrojový text dokumentace, vložené obrázky, apod.

src/
: Kompletní zdrojové kódy všech algoritmů.

readme.txt
: Obsahuje popis jak spustit algoritmy.

Navíc CD/DVD obsahuje:

data/
: Obsahuje data, na kterých byly algoritmy testovány.