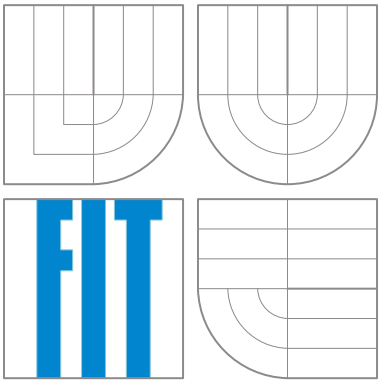
Fakulta informačních technologií
Vysoké učení technické v Brně



Dokumentace projektu do předmětu GAL

Paralelizace Egerváryho algoritmu

Tým 33
Vendula Poncová (xponco00)
Alena Chernikava (xcerni07)

22. srpna 2016

1 Úvod

V této dokumentaci popisujeme návrh a implementaci sekvenční a paralelní varianty Egerváryho algoritmu pro nalezení maximálního párování v bipartitních neorientovaných grafech. Sekvenční variantu algoritmu jsme převzaly z [1], pro paralelní variantu se nám nepodařil najít vhodný zdroj, proto jsme vymyslely vlastní řešení. Experimenty provedené z oběma implementacemi popisujeme a vyhodnocujeme v kapitole 5.

2 Egerváryho algoritmus

Egerváryho algoritmus a uvedené definice jsme převzaly z knihy Graph Theory [1].

2.1 Maximální párování

Definice 1 *Párování M v grafu $G = (V, E)$ je podmnožina množiny hran E taková, že žádné dvě hrany nemají společný vrchol. Párování M je maximální, pokud neexistuje párování s větším počtem hran.*

Obrázek 1: Bipartitní graf a maximální párování v tomto grafu.

Definice 2 *M-alternující cesta P v grafu G s párováním M je taková cesta, na které jsou hrany střídavě v M a v $E - M$. Tedy pro každé dvě sousední hrany platí, že jedna z nich patří do M a druhá nepatří.*

Definice 3 *M-rozšiřující cesta P v grafu G s párováním M je M-alternující cesta, kde počáteční ani koncový uzel cesty nepatří do párování M. Cesta se nazývá rozšiřující, neboť změnou párování na hranách cesty se velikost množiny M zvětší o jedna.*

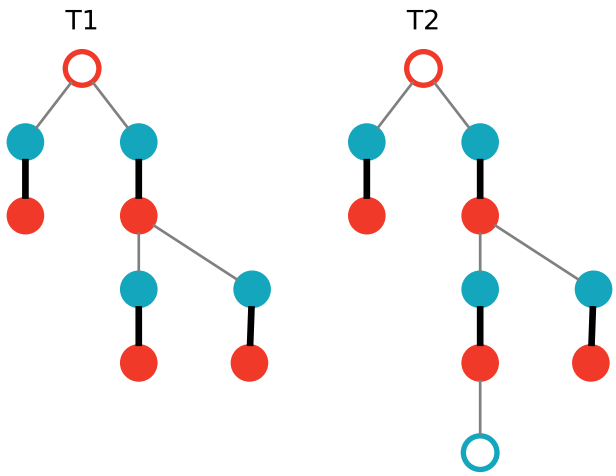
Obrázek 2: M-rozšiřující cesta a nové párování M'.

Věta 1 (Berge) *Párování M v grafu G je maximální párování tehdy a jen tehdy, když G neobsahuje M-rozšiřující cestu.*

Definice 4 *Nechť G je graf, M je párování v G a u je vrchol z G nepokrytý párováním M. Pak strom T s kořenem ve vrcholu u v grafu G je M-alternující u-strom, pokud pro každý uzel v stromu T je uTv M-alternující cesta.*

Definice 5 *Strom T grafu G je M-pokrytý u-strom, jestliže párování $M \cap E_T$, kde E_T je množina všech hran stromu T, pokrývá všechny vrcholy stromu T kromě vrcholu u.*

Definice 6 *APS-strom (z angličtiny Augmenting Path Search) je maximální M-pokrytý u-strom. Tedy strom, který nemůže dál růst.*



Obrázek 3: M-pokrytý u-strom a M-alternující u-strom.

2.2 Popis Egervarýho algoritmu

Egerváryho algoritmus je založený na větě 1. Vstupem algoritmu je bipartitní graf $G = (X, Y, E)$, výstupem je maximální párování M v G. Algoritmus vybere z grafu G vrchol u nepokrytý M a zavolá na něj funkci APS (Augmenting Path Search). Funkce APS vytvoří strom T s kořenový uzlem u a obarví u na červeno. Pokud existuje hrana z nějakého červeného uzlu x stromu T do uzlu y, který nepatří do T a není pokrytý párováním M, lze přidat uzel y do stromu T a obarvit jej na modro. Pokud existuje hrana z nějakého modrého uzlu y do uzlu z, který nepatří do T, ale je pokrytý párováním M, pak lze přidat uzel z do stromu T a obarvit jej na červeno. Pokud nějaký modrý uzel y z T není incidentní s hranou z M, existuje M-rozšiřující cesta z kořene stromu u do vrcholu y. Pokud nelze ke stromu T přidat žádnou hranu, strom T je APS strom. Funkce APS tedy vrátí M-rozšiřující cestu, nebo APS strom. Pokud vrátí cestu, upraví se na ní párování M tak, že hrany, které patřily do M, z M odebereme, a hrany, které nepatřily do M, do M přidáme. Pokud se vrátí APS strom, pokračuje se s grafem $G - T$.

3 Sekvenční varianta algoritmu

V této kapitole popisujeme návrh a implementaci sekvenční varianty Egerváryho algoritmu.

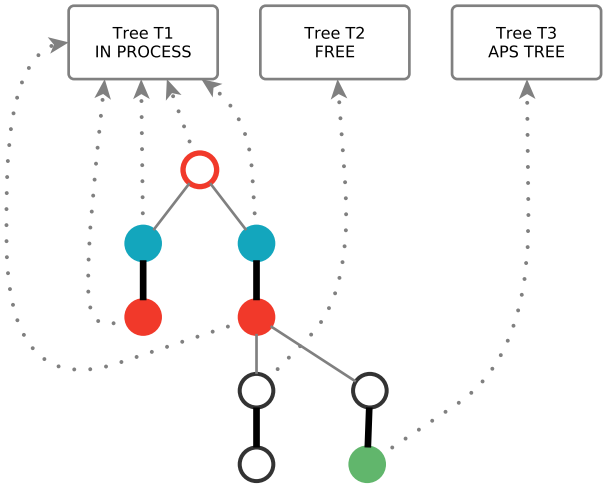
3.1 Návrh řešení

Vzhledem k tomu, že algoritmus sekvenční varianty je popsán v kapitole 2, zbývá vhodně navrhnout datové struktury.

Algoritmus s grafem pracuje tak, že přechází z uzlu na sousední uzly. Navíc, o hustotě vstupních grafů nic nevíme, proto je vhodnou reprezentací seznam sousedů. Potřebujeme reprezentovat bipartitní neorientovaný graf. Každý hrana tedy bude zastoupena dvakrát, jednou v každém směru. Vhledem k tomu, že atributem hrany je v našem případě příslušnost do párování M, je třeba nějak sdružit odpovídající si hrany. To je možné například tak, že každá hrana bude obsahovat ukazatel na hranu opačnou. Pokud se pak v jedné hraně změní příslušnost do M, lze okamžitě upravit příslušnost opačné hrany.

Uzly se po jednom připojují ke stromu, dokud se nenalezne M-rozšiřující cesta, nebo APS tree. Pokud se nalezne cesta, je nutné projít stromem po této cestě a upravit na jejích hranách párování. Proto je výhodné si u každého uzlu pamatovat hranu z jeho předchůdce. Cestu pak lze projít od listového uzlu ke kořeni v lineárním čase k výšce stromu.

Pokud je nalezen APS tree, je třeba uzly stromu nějak vyloučit z grafu. Stejně tak, po zpracování cesty je třeba uzly stromu uvolnit pro další stromy (říkáme, že strom je zahozen). Pokud bychom uzly zpracovávaly po jednom, bude časová složitost každé takové úpravy stromu lineární vůči počtu uzlů ve stromu. Nabízí se vytvořit datovou strukturu stromu, která bude obsahovat položku se statusem stromu. Uzly stromu se pak budou odkazovat na tuto strukturu. Pokud bude status stromu `FREE`, pak jsou uzly bílé a dostupné pro nový strom. Pokud je status `ACTUAL`, uzly jsou modré nebo červené a jsou součástí právě zpracovávaného stromu. Jestliže je status `APSTREE`, pak jsou uzly zelené a algoritmus je ignoruje. Změna statusu je pak možná v konstantním čase. Příklad takové datové struktury je na obrázku 4.



Obrázek 4: Datové struktury uzlu a stromu.

3.2 Popis implementace

Algoritmus jsme implementovaly v jazyce `C`.

Datová struktura `TGraph` reprezentuje bipartitní neorientovaný graf. Obsahuje počet uzlů `n`, počet hran `m`, pole uzlů `nodes` a seznam stromů `trees`. Uzly jsou reprezentované strukturou `TNode` s položkami: pořadové číslo uzlu `id`, seznam hran `edges`, ukazatel na vstupní hranu `entry` a ukazatel na strom `tree`. Pokud uzel patří nějakému stromu, jeho ukazatel na vstupní hranu je nastavený na hranu z předchůdce uzlu do tohoto uzlu. Hranu reprezentuje struktura `TEdge`: `M` značí, zda hrana patří do dosud nalezeného párování grafu, `node` je ukazatel na uzel, do kterého hrana vstupuje, a `reversed` je ukazatel na hranu, která je k hraně opačná. Struktura `TQueue` spolu s `TItem` reprezentují abstraktní datový typ fronta.

Graf lze inicializovat, vytvořit a zrušit funkcemi `initGraph`, `loadGraph` a `freeGraph`. Strom lze vytvořit zavoláním funkce `createTree`. Funkce `processPath` projde strom od listového uzlu ke kořenovému a upraví na cestě párování. Funkce `applyAPS` se snaží nalézt M-rozšiřující cestu. Pokud ji nalezne, zavolá pro ni funkci `processPath` a nastaví status stromu na `FREE`, jinak nastaví status stromu na `APSTREE`. Funkce `findMatching` prochází pole uzlů `nodes` grafu `graph` a pokud některá jeho vstupní hrana není v dosud nalezeném párování, vytvoří se strom s nalezeným uzlem jako kořenem a zavolá se funkce `applyAPS`. Funkce `main` zkontroluje vstupní parametry, ze vstupního souboru vytvoří graf a zavolá pro něj funkci `findMatching`.

3.3 Spuštění aplikace

Naše implementace se nachází ve složce `sequence` v souboru `matching.c`. Program lze přeložit příkazem `make` a spustit příkazem `./matching file`, kde `file` je název vstupního souboru ve formátu popsaném v kapitole 5.1.

4 Paralelní varianta algoritmu

V této kapitole popisujeme analýzu, návrh a řešení paralelní varianty Egerváryho algoritmu. Neboť jsme nenašly vhodný zdroj, podle které bychom mohly paralelní variantu navrhnout a implementovat, věnujeme se velmi podrobně analýze problému paralelizace.

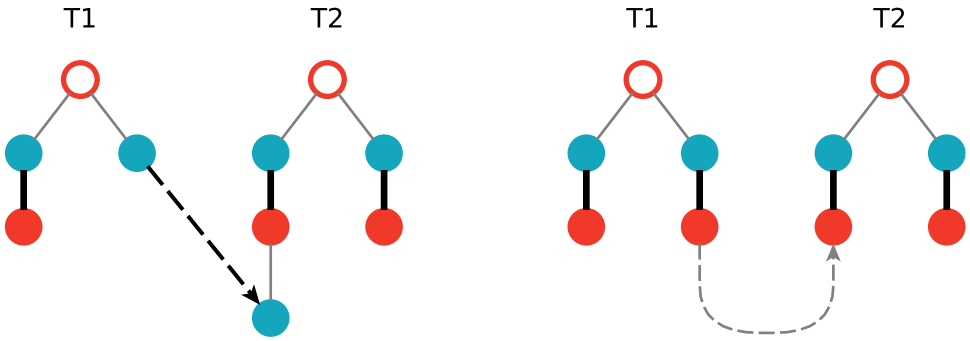
4.1 Analýza problému

V popisu Egerváryho algoritmu si lze všimnout, že se vždy pracuje pouze s vytvářeným stromem a jeho nejbližším okolím. Pokud je nalezena M-rozšiřující cesta, dochází ke změnám na hranách, které patří do tohoto stromu, pokud je nalezen APS strom, přebarví se pouze uzly tohoto stromu.

Jako přirozený způsob paralelizace Egerváryho algoritmu se nabízí zpracovávání grafu více procesy tak, že každý proces bude vytvářet vlastní strom. Pokud bychom zaručily, že takové stromy budou vždy navzájem disjunktní, s ohledem na předcházející paragraf by paralelní zpracování nijak neovlivnilo fungování Egerváryho algoritmu. Na víceprocesorových počítačích by se však výpočet urychlil.

Disjunktnost stromů ale zaručit nemůžeme, proto je třeba najít způsob, jak se vypořádat s konflikty mezi stromy. Obecná situace je taková, že proces 1 buduje strom T_1, proces 2 buduje strom T_2 a proces 1 bude chtít k uzlu v_1 svého stromu T_1 připojit uzel v_2 stromu T_2. Podle barev uzlů v_1 a v_2 rozlišujeme následující typy konfliktů.

Konflikt typu B-B je konflikt mezi modrým uzlem v_1 a modrým uzlem v_2. Příklad takového konfliktu je na obrázku 5. Na obrázku si můžeme všimnout toho, že v takovém případě existuje M-rozšiřující cesta z kořene stromu T_1 do kořene stromu T_2. Nalezení cesty je žádoucí, proto pokud dojde k takovému konfliktu, lze zastavit růst obou stromů, upravit párování na hranách nalezené cesty a uzly ve stromech přebarvit na bílo, aby byly k dispozici dalším stromům. Konflikt typu R-R pro červené uzly v_1, v_2 je analogický konfliktu B-B (viz. 5).

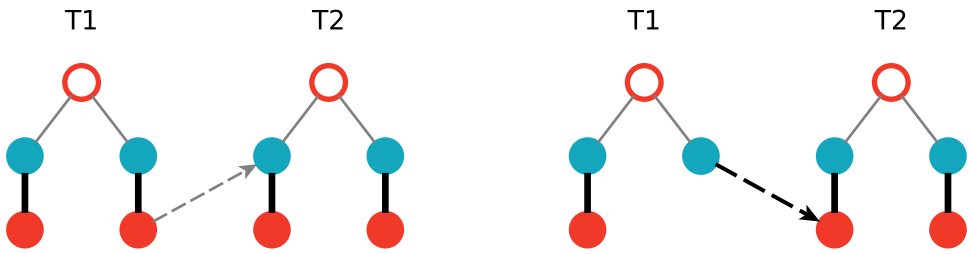


Obrázek 5: Konflikty B-B a R-R.

Konflikt typu R-B je konflikt mezi červených uzlem v_1 a modrým uzlem v_2. Na obrázku 6 vidíme, že v tomto případě se M-rozšiřující cesta nevytvoří. Nabízí se několik možných řešení. Strom T_1 by mohl získat uzel v_2, pak by ale strom $T2$ nemohl pokračovat ve výpočtu a všechny jeho uzly by musely být obarveny na bílo a uvolněny. Stejně tak by mohl T_1 prohrát uzel v_2, T_2 by dál pokračoval ve výpočtu a T_1 by se ukončil. Třetí možností je ignorace konfliktní hrany, kdy T_1 i T_2 dál pokračují ve výpočtu, nijak neovlivněné tímto konfliktem. Ignorace hrany je možná, neboť pokud existuje M-rozšiřující cesta z kořenového uzlu stromu T_1 procházející hranou v_1v_2 končící v nějakém uzlu w, pak též musí existovat cesta z kořenového uzlu stromu T_2 procházející uzlem v_2 a končící v uzlu w. Pokud strom T_1 bude hranu v_1v_2 ignorovat, pak cestu nalezne strom T_2.

Konflikt typu B-R je konflikt mezi modrým uzlem v_1 a červeným uzlem v_2. Z obrázku 6 je patrné, že v takovém případě by z uzlu v_2 vedly dvě hrany náležící párování M. M by potom

nebylo platné párování. Párování může být dočasně neplatné jen v případě, že nějaký strom nalezl M-rozšiřující cestu a upravuje párování na hranách této cesty. Pokud jsou oba stromy T_1, T_2 ve fázi růstu pak párování na hranách vycházejících z uzlů těchto dvou stromů je platné a neměnné. V našem případě předpokládáme, že oba stromy rostou, proto k B-R konfliktu nemůže dojít.



Obrázek 6: Konflikty R-B a B-R.

Dále je třeba uvažovat o způsobu, jakým se budou synchronizovat procesy. Je zřejmé, že je třeba zaručit, aby s každým uzlem pracoval v jeden okamžik nejvýše jeden proces. Potom bude třeba synchronizovat přístup ke stromům uzlů. Pokud bychom se inspirovaly návrhem sekvenčního algoritmu, pak informaci o stavu stromu nese pouze strom. Pak při každém přístupu k uzlu je potřeba nejprve zamknout příslušný strom a pak teprve uzel. Pokud očekáváme konflikt, je třeba zamknout oba stromy a konfliktní uzly. Druhou možností je propagovat informaci o stavu stromu do jeho uzlů. Pro přístup k uzlu by pak stačilo zamknutí uzlu.

4.2 Diskuze k návrhu řešení

Naše výsledné řešení vyplynulo z několika dalších řešení, která jsme implementovaly a testovaly.

Konflikt typu R-B jsme původně řešily tak, že proces 1 vyhrál uzel stromu procesu 2, pokud jeho strom T_1 obsahoval více uzlů než strom T_2. Jinak proces 1 prohrál a byl nucen uvolnit svůj strom. V obou případech došlo k zahození jednoho z konfliktních stromů. Vzhledem k tomu, že další dva typy konfliktů R-R a B-B výpočet urychlují, myslely jsme si, že zpomalení výpočtu konfliktem R-B se ve výsledku neprojeví. Ukázalo se, že to není pravda. K zahazování stromů docházelo neustále a počet vytvořených stromů rostl vzhledem k velikosti vstupního grafu polynomiálně. To značně zpomalovalo výpočet. Efektivnějším řešením konfliktu se ukázala být ignorace konfliktní hrany. Počet vytvořených stromů byl pak lineární a většina M-rozšiřujících cest byla nalezena R-R a B-B konflikty.

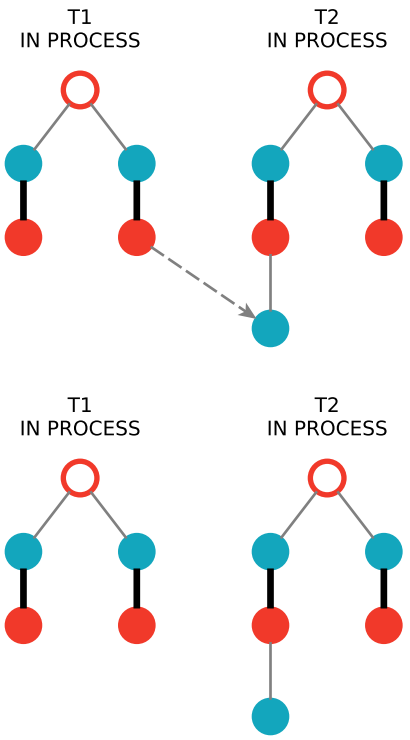
Synchronizaci procesů jsme zkusily řešit zamykáním stromů. Algoritmus byl tedy velmi podobný sekvenčnímu řešení. Uzly se odkazovaly na stromy a stromy obsahovaly informaci o svém stavu. Rostoucí strom byl ve stavu `INPROCESS`. Vytvoření APS stromu bylo možné nastavením stavu na `APSTREE`. Uvolnění stromu proběhlo upravením stavu na `FREE`. Úprava stavu tak sice byla možná v konstantním čase, ale kdykoliv chtěl proces přidat ke svému stromu nový uzel, musel zamknout svůj strom i strom toho uzlu. Stromy se tak staly úzkým hrdlem programu. To se projevilo zejména tak, že s vyšším počtem dostupných procesorů se časová efektivita programu nezlepšovala.

Řešením tohoto problému byla propagace stavu stromu ze stromu do uzlů. Stav stromu pak bylo možné vyčíst z barvy jeho uzlu. Bílý uzel nepatřil žádnému stromu, zelený uzel patřil APS stromu, modrý a červený uzel patřil zpracovávanému stromu. Toto řešení je výhodné v tom, že pokud chce proces přidat ke svému stromu nový uzel, vyčte z barvy tohoto uzlu, jestli má komunikovat i s jeho stromem, nebo ne. Nevýhodou je časová složitost přebarvení všech uzlů stromu. Důležité však je, že se takto zjednodušila synchronizace, což umožnilo více využívat dostupné procesory.

4.3 Návrh řešení

Na začátku jsou všechny uzly obarveny na bílo a vloženy do fronty kořenových uzlů. Pokud je kořenová fronta prázdná, proces i se ukončí, jinak z fronty vyzvedne uzel u. Pokud je uzel u bílý, proces jej přidá do svého stromu T_i jako kořen a obarví jej na červeno. Proces i začne hledat M-rozšiřující cestu. Postupně připojuje k uzlům v_i uzly v_j.

Pokud se pokusí přidat bílý uzel v_j, proces uzel obarví na červenou nebo modrou a přivlastní si jej. Pokud se pokusí přidat zelený uzel v_j, pak jej ignoruje, neboť uzel patří do APS stromu. Pokud se pokusí přidat červený nebo modrý uzel v_j, došlo ke konfliktu.



Obrázek 7: Řešení R-B konflitu.

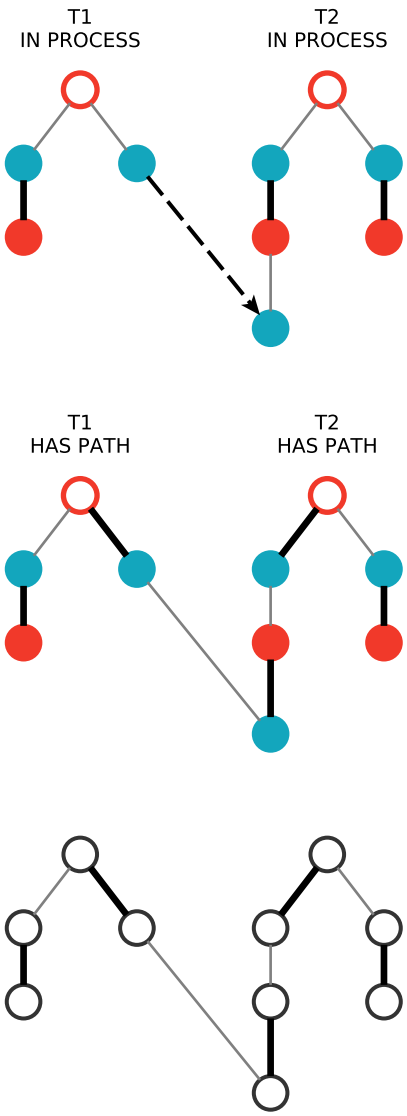
V případě R-B konfliktu proces uzel ignoruje (viz. obrázek 7). V případě R-R a B-B konfliktu proces přistoupí ke stromu uzlu. Jestliže je strom T_j ve stavu `HASPATH`, pak proces uzel ignoruje. Pokud je strom ve stavu `INPROCESS`, je nalezena M-rozšiřující cesta (viz. obrázek 8). Pak proces změní stav stromu T_j na `HASPATH` a nastaví stromu ukazatel na konec cesty. Následně změní stav svého stromu T_i na `HASPATH` a nastaví mu ukazatel na konec cesty. Nakonec upraví párování hrany z v_i do v_j.

Pokud proces i nalezne cestu, nastaví stav stromu T_i na `HASPATH`. Pokud proces i zjistí, že jeho strom je ve stavu `HASPATH`, pak upraví párování na nalezené cestě, změní jeho stav na `FREE` a obarví všechny jeho uzly na bílo. Pokud proces i skončí hledání M-rozšiřující cesty nalezením APS stromu, pak nastaví stav stromu na `APSTREE` a obarví všechny jeho uzly na zeleno. Nakonec si proces i vyzvedne z fronty kořenových uzlů nový uzel a celý proces se opakuje.

4.4 Popis implementace

Algoritmus jsme implementovaly v jazyce `C` s využitím knihovny `pthread`.

Datové struktury `TList` a `TQueue` reprezentují abstraktní datové typy seznam a fronta nad prvky typu ukazatel na `void`. `TGraph`, `TTree`, `TNode` a `TEdge` jsou v podstatě stejné jako u sekvenční varianty. Liší se zejména přítomností mutexů, které zajišťují exkluzivní přístup ke strukturám (z výjimkou `TEdge`). `TTree` si navíc uchovává pořadové číslo stromu `id`, pořadové



Obrázek 8: Řešení R-R a B-B konfliktu.

číslo `owner` procesu, který ho vlastní, ukazatel `pathEnd` na konec cesty a seznam uzlů `nodes`, které jsou ve stromu. Vlákno je reprezentováno datovým typem `TThread`, mutex typem `TMutex`. Datová struktura `TThreadData` zapouzdřuje vstupní data procesů.

Graf lze inicializovat, vytvořit a zrušit funkcemi `initGraph`, `loadGraph` a `freeGraph`. Zamykání uzlů a stromů umožňují funkce `lockNode`, `unlockNode`, `lockTree`, `unlockTree`, `lockNodes` a `lockTrees`, přičemž funkce `lockNodes` a `lockTrees` zamykají dvojice uzlů nebo stromů v pořadí určeném jejich `id`. Strom lze vytvořit a zrušit zavoláním funkce `createTree` a `freeTree`. Funkce `colourNodes` obarví všechny uzly stromu na požadovanou barvu, `processPath` projde stromem od listového uzlu ke kořeni a změní na hranách cesty párování.

Funkce `addNodeToTree` se snaží k uzlu `nodeA` stromu `treeA` přidat uzel `nodeB` přes hranu `AB` s očekávanou příslušností k párování `M`. Funkce vrací hodnotu z výčtu: `OK`, `ABORT`, `IGNORE`, `CONFLICT` nebo `PATH`.

Funkce `applyAPS` se pomocí stromu `tree` snažít nalézt M-rozšiřující cestu. Pokud nalezne cestu, zavolá na ni `processPath`, funkcí `colourNodes` obarví uzly stromu na bílo a vrátí `OK`. Pokud nalezne APS strom, obarví uzly na zeleno a vrátí `OK`.

Funkce `findMatching` zpracovává frontu kořenových uzlů. Pokud fronta není prázdná, vybere se uzel, který je bílý a žádná jeho hrana není v párování. Vytvoří se strom, kde vybraný uzel bude

jeho kořenovým uzlem, a zavolá se funkce `applyAPS`. Funkce `findMatchingParallel` vytváří a spouští vlákna nad funkcí `findMatching` s daty typu `TThreadData`. Funkce `main` kontroluje vstupní parametry, vytváří ze vstupního souboru graf a volá pro něj funkci `findMatchingParallel`.

4.5 Spuštění aplikace

Výsledný program `matching.c` je umístěný v adresáři `parallel`. Lze jej přeložit příkazem `make` a spustit příkazem `./matching file n`, kde `file` je název vstupního souboru a `n` je počet vláken. Očekávaný formát vstupního souboru popisujeme v kapitole 5.1.

5 Experimenty

V této kapitole popisujeme, jaké experimenty jsme s implementovanými algoritmy provedly a jaké výsledky jsme dostaly.

5.1 Generování vstupních souborů

Pro vygenerování vstupních souborů jsme v jazyce `python 2.7` s použitím knihovny `python-igraph 0.7` [2] napsaly skript `generator.py`. Skript generuje bipartitní neorientované grafy $G = (X, Y, E)$ a ukládá je do souborů `graph_x_y_p_m`, kde x je počet uzlů v množině X, y je počet uzlů v množině Y, p je procentuální vyjádření počtu hran množiny E a m je počet hran v maximálním párování. Pak soubor obsahuje počet uzlů množiny $X \cup Y$, počet hran množiny E a výčet všech dvojic uzlů, mezi kterými existuje v E hrana, přičemž každá hrana je reprezentovaná právě jednou.

Skript je umístěný ve složce `graph`. Příkazem `python2.7 generator.py` se podle parametrů nastavených ve scriptu vygeneruje soubor s náhodným grafem. Příkaz `python2.7 generator.py test` vygeneruje všechny testovací soubory pro naše experimenty a uloží je do složky `test/files`. Knihovna `python-igraph` na školním serveru `merlin` není dostupná, proto v archivu odevzdáváme část testovacích souborů.

Testovací soubory jsme vygenerovaly pro grafy $G = (X, Y, E)$, kde $|X| = |Y|$, s velikostmi množiny X 10, 100, 500, 1000 a hustotami hran v E 0.1, 0.25, 0.75, 0.99. Snažily jsme se tak pokrýt různé velikosti grafů, řídké i husté.

Všechny testovací soubory jsou umístěné v archivu `GAL2014_testfiles.zip` na adrese: `https://www.dropbox.com/s/dpazvpcltv3hdi2/GAL2014_testfiles.zip?dl=0`

5.2 Spuštění testů

Testy se spouští příkazem `./test.sh` ve složce `test`. Skript přeloží programy, načte testovací soubory ze složky `test/files` a pro každý soubor zavolá paralelní a sekvenční program pro různé počty vláken a dostupných procesorů. Výsledky se ukládají do souboru `results.out`.

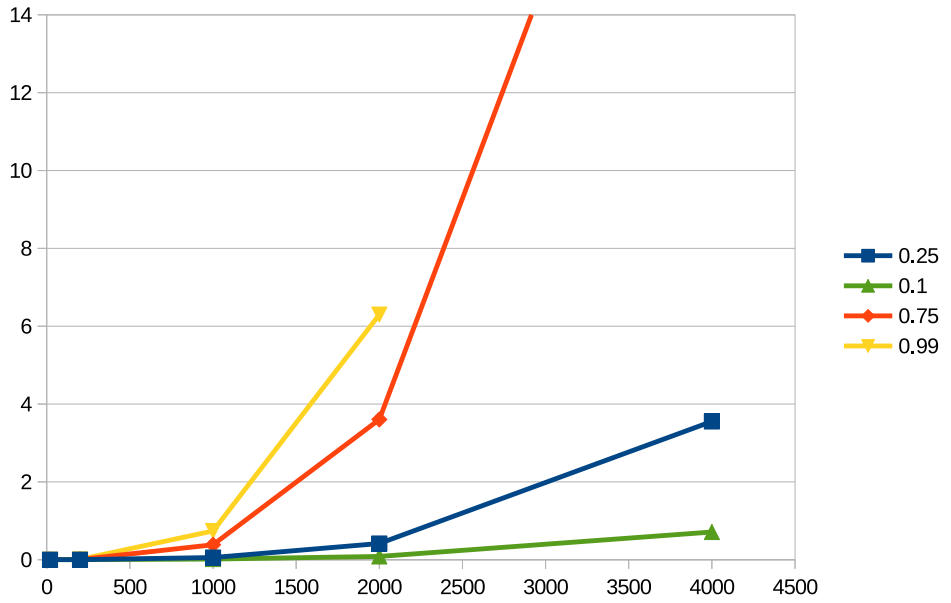
5.3 Vyhodnocení testů

Experimenty jsme provedly na školním serveru `merlin` pro maximální počet procesorů 12 a maximální počet vláken 15. Větší počet vláken nám server pro některé grafy nedovolil vytvořit. Zaměřily jsme se na časovou složitost algoritmů v závislosti na různých veličinách. Měřily jsme celkový čas běhu programu. V grafech je udávaný v sekundách.

Na obrázku 9 je znázorněná závislost času na počtu vrcholů pro hustoty 0.1, 0.25, 0.75 a 0.99 v sekvenčním algoritmu. Na obrázku vidíme, že pro větší a hustější grafy trvá výpočet déle než pro menší a řidší. Sekvenční algoritmus se tedy chová tak, jak jsme očekávaly.

Obrázek 10 znázorňuje závislost času na počtu vrcholů pro 10 procesorů a 6 vláken v paralelním algoritmu. Opět vidíme, že výpočet je časově náročnejší pro velké a husté grafy.

Závislost času na počtu procesorů v paralelním algoritmu je znázorněna obrázkem 11. Test proběhl pro 10 spuštěných vláken. Můžeme si všimnout, že křivky mají pro různé hustoty stejný



Obrázek 9: Závislost času na počtu vrcholů v sekvenčním algoritmu.

průběh. Zpočátku doba běhu klesá, ale od čtvrtého dostupného procesoru je více méně konstatní. Značí to, že algoritmus není schopný dostatečně využít dostupné zdroje.

Na dalším obrázku 12 je závislost času na počtu vláken pro 10 procesorů v paralelním algoritmu. Můžeme pozorovat, že pro velmi husté grafy je počet spuštěných vláken velmi významnou veličinou. Zajímavý je obrázek 13, který znázorňuje stejnou závislost pro jeden procesor, je velmi podobný obrázku 12. Znamená to, že algoritmus je časově efektivnější pro větší počet vláken i v případě, že má k dispozici pouze jeden procesor. Vlákna tedy nemohou běžet paralelně.

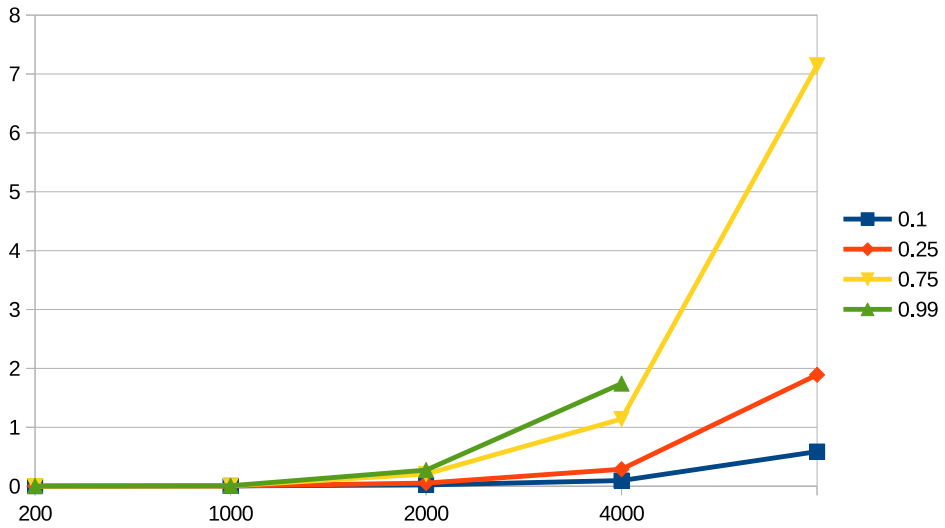
Nakonec obrázek 14 srovnává časové závislosti sekvenčního a paralelního algoritmu na počtu vrcholů. Paralelní algoritmus běžel na 12 procesorech s 15 vlákny. Vidíme, že křivky sekvenčního algoritmu jsou strmější než u paralelního algoritmu. Paralelní algoritmus má tedy lepší časovou složitost než sekvenční algoritmus.

Z uvedené experimenty v podstatě potvrdily očekávané chování navržených algoritmů. Především jsme empiricky dokázaly, že paralelní algoritmus má lepší časovou složitost než sekvenční algoritmus.

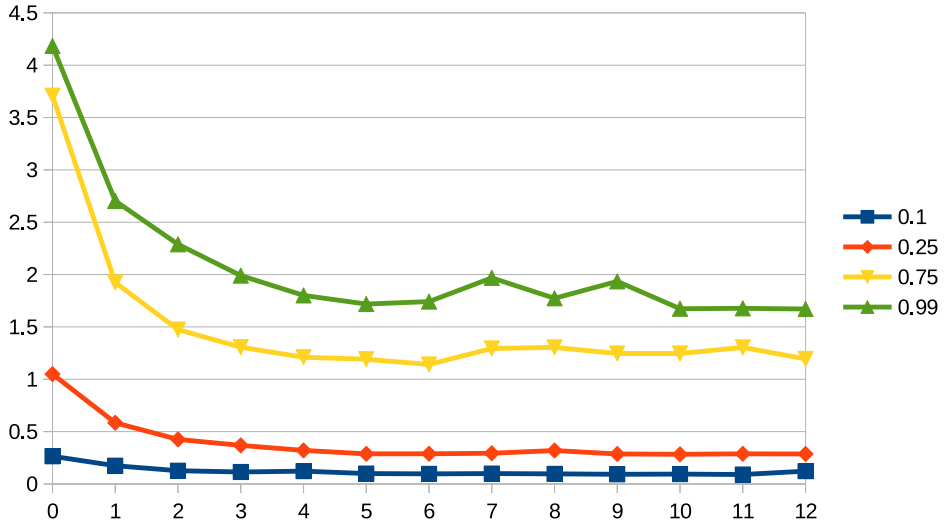
Překvapivým zjištěním byl vliv počtu vláken na dobu běhu na jednom procesoru. Takové chování si vysvětlujeme tak, že s větším počtem vláken roste počet rozpracovaných stromů a ačkoliv vlákna nemohou pracovat paralelně, zvyšuje se pravděpodobnost nalezení R-R nebo B-B konfliktu, což urychluje výpočet. Dále je třeba zohlednit omezení serveru `merlin`, který nám nedovolil pracovat s více než 20 vlákny, ačkoliv při testování na jiných počítačích se doba běhu snižovala až do počtu 50 až 100 vláken. Bylo by zajímavé zkoumat takové chování s souvislosti s větším počtem procesorů. Je tak pravděpodobné, že test, který zkoumal závislost času na počtu procesorů, není zcela korektní, neboť větší množství vláken by možná dokázalo využít všechny dostupné procesory.

6 Závěr

Implementovaly jsme Egerváryho algoritmus pro výpočet maximálního párování v bipartitních grafech a navrhly jsme a implementovaly jeho paralelní variantu. Výsledné programy jsme otestovaly pro různé vstupní programy, přičemž jsme se zaměřily především na časovou složitost. Testy potvrdily, že paralelní algoritmus má lepší časovou složitost než sekvenční. Dále se ukázalo, že počet vláken má vliv na dobu běhu programu i v případě, že program běží na jednoproceso-



Obrázek 10: Závislost času na počtu vrcholů pro 10 procesorů a 6 vláken v paralelním algoritmu.



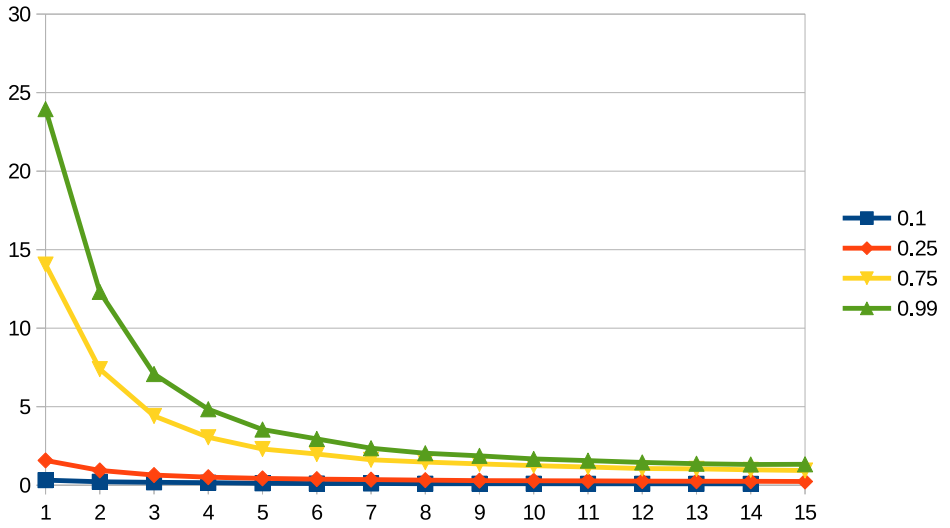
Obrázek 11: Závislost času na počtu procesorů pro 10 vláken v paralelním algoritmu.

rovém počítači.

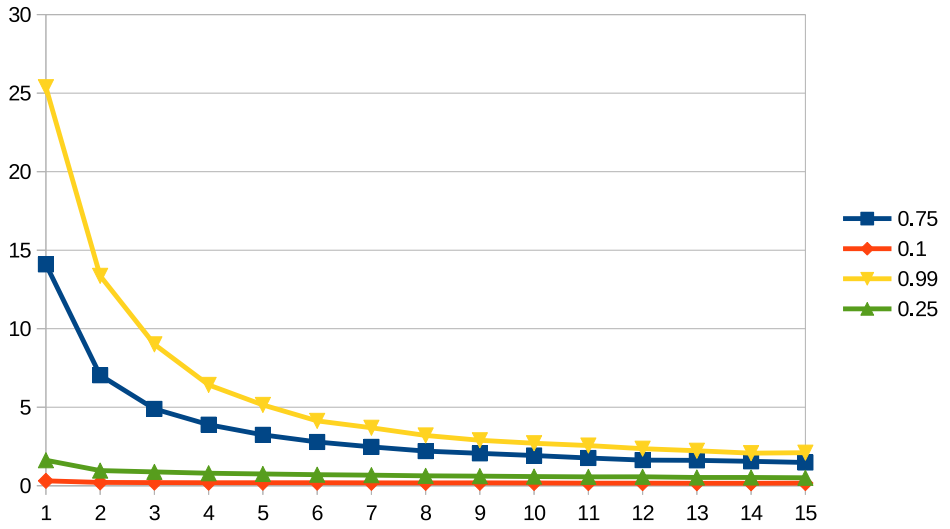
Reference

[1] BONDY, J. A. a U. S. R. MURTY. *Graph theory*. New York: Springer, 2008, xii, 651 s. ISBN 978-1-84628-969-9.

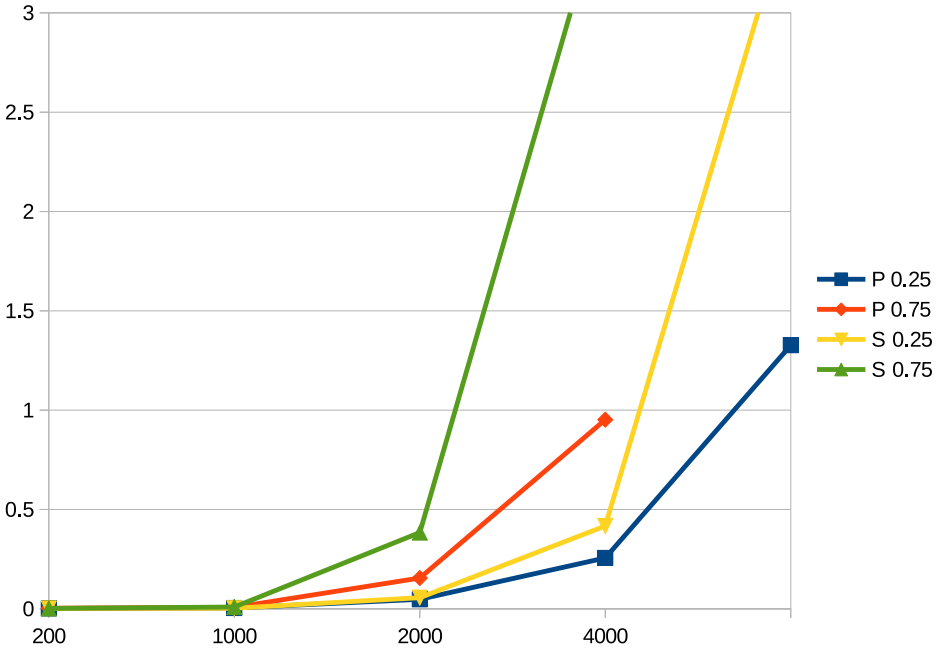
[2] *Python-igraph* [online]. ©2003-2013 [cit. 2014-12-13]. Dostupné z: http://igraph.org/python/



Obrázek 12: Závislost času na počtu vláken pro 10 procesorů v paralelním algoritmu.



Obrázek 13: Závislost času na počtu vláken pro 1 procesor v paralelním algoritmu.



Obrázek 14: Závislost času na velikosti grafu v sekvenčním algoritmu a pro 12 procesorů a 15 vláken v paralelním algoritmu.