Závěrečná zpráva NI-PDP

Leoš Tobolka

LS 2022/23

Obsah

1 Zadání

Úloha MRO: Minimální hranový řez hranově ohodnoceného grafu

Vstupní data

- n = přirozené číslo představující počet uzlů grafu G, $150 > n >= 10$
- k = přirozené číslo představující průměrný stupeň uzlu grafu G, $3n/4 > k >= 5$
- $G(V, E)$ = jednoduchý souvislý neorientovaný hranově ohodnocený graf o n uzlech a průměrném stupni k, váhy hran jsou z intervalu $< 80, 120 >$
- a = přirozené číslo, $5 =< a =< n/2$

Úkol

Nalezněte rozdělení množiny uzlů V do dvou disjunktních podmnožin X, Y tak, že množina X obsahuje a uzlů, množina Y obsahuje $n - a$ uzlů a součet ohodnocení všech hran u, v takových, že u je z X a v je z Y (čili velikost hranového řezu mezi X a Y), je minimální.

Výstup algoritmu

Výpis disjunktních podmnožin uzlů X a Y a ohodnocených hran minimálního řezu mezi nimi a celková váha tohoto minimálního řezu.

2 Popis implementace

2.1 Sekvenční algoritmus

První implementovaný algoritmus, který slouží k nalezení minimálního řezu grafu. Z tohoto algoritmu vychází i následující paralelní verze, které jsou pouze vylepšenou verzí této implementace.

Algoritmus je typu BB-DFS, tj. prohledávání do hloubky s průběžným prořezáváním větví. Návrat z větví se provádí, pokud je mezistav vyhodnocen jako nepřípustný nebo pokud se dosáhne maximální hloubky. Stavový prostor je reprezentován jako binární strom; počáteční stav se větví na další stavy až do hloubky n.

Prohledávání binárního stromu je implementováno jako rekurzivní funkce. V každé úrovni se uzel přiřadí do jedné ze dvou skupin X a Y. Pro každý stav jsou počítány metriky, které slouží pro optimalizaci; metriky jsou následující:

Počet přiřazených uzlů ve skupině je počet přiřazených uzlů stavu ve skupině X nebo Y.

Průběžná váha řezu je dosavadní váha řezu pro ohodnocené uzly.

Dolní odhad váhy zbývajícího řezu lze spočítat tak, že pro každý zatím nepřiřazený uzel grafu v daném mezistavu s částečným řezem vypočteme, o kolik by se váha řezu zvýšila, pokud by tento uzel patřil do X, o kolik by se zvýšila, pokud by tento uzel patřil do Y a vezmeme menší z těchto dvou hodnot a tato minima posčítáme pro všechny nepřiřazené uzly.

Již zmiňované prořezávání je založeno na těchto metrikách. Pokud je počet přiřazených uzlů v jedné skupině ostře větší než vstupní parametr a, je jisté že další větve z tohoto stavu se nemůžou dostat do přípustného řešení, proto lze z takového stavu udělat návrat. V případě že průběžná váha řezu stavu je menší než již minimální nalezené řešení, není možnost z tohoto stavu dalším větvením vyrobit stav s menší váhou. Přidáváním dalších

uzlů do skupin váhu může pouze zvednout. Stejně tak i dolní odhad váhy zbývajícího řezu stavu, pokud váha přesáhne už již nalezené řešení, lze tuto větev ukončit.

V případě že se rekurzivní volání dostane až do hloubky n, stav má všechny uzly ohodnocené a pokud je jeho průběžná váha menší než nejlepší nalezené řešení, nastaví se tento stav jako nejlepší.

2.2 Paralelní algoritmus – task paralelismus

Pomocí knihovny OpenMP jsem rozšířil sekvenční algoritmus, aby byl schopný pracovat paralelně. V tomto případě jsem toho docílil pomocí task paralelismu. Tento způsob se zakládá v rozdělení kódu na nezávislé části (úlohy), které mohou běžet paralelně.

Sekvenční řešení využívá k nalezení řešení rekurzivní funkci. Z toho důvodu paralelizace nebyla obtížná; stačilo pouze přidat OpenMP direktivy nad volání funkcí. Jediný problém který bylo třeba vyřešit je současný přístup k proměnné s nejlepším výsledkem; tedy vytvoření kritické sekce při aktualizaci nejlepšího řešení.

Tento způsob paralelizace působí přirozeně a pro většinu lidí je to pravděpodobně první způsob který je napadne. Avšak je potřeba přidat důležitou optimalizaci vzhledem k možnému velkému množství podúloh, které vyžadují velkou režii ke správě vláken. Tento problém se dá optimalizovat tím, že pokud se větev výpočtu blíží ke konci, lze toto větev dopočítat sekvenčně. Tímto způsobem se výrazně zlepší problém s nadměrnou režií. Pro tento konkrétní algoritmus jsem nastavil, aby se větev dopočítala sekvenčně, pokud zbývá pouze 15 neohodnocených uzlů ve stavu.

2.3 Paralelní algoritmus – datový paralelismus

Alternativní způsob k task paralelismu je datový paralelismus. Princip této metody je rovnoměrné rozdělení dat, tak aby každý procesor prováděl synchronně výpočet nad přidělenou částí.

V této úloze je pouze jeden počáteční stav (neohodnocené všechny uzly). Pro rozdělení úkolů do více vláken je ale potřeba mít více stavů, které se jednotlivým vláknům přiřadí a ty prohledají všechny následující větve. Z toho důvodu jsem z počátečního prázdného stavu nejdříve sekvenčně vygeneroval pomocí BFS algoritmu další stavy, které se následovně vláknům přiřazují. BFS algoritmus hledá až do hloubky 10, tedy vygeneruje maximálně 1024 stavů.

Problém se současným přístupem k proměnné s nejlepším řešením jsem vyřešil stejně jako u task paralelismu.

S tímto způsobem paralelizace nejsou tolik spojeny problémy s nadměrnou režií jako u task alternativy. Každému procesoru je přidělena část stavového prostoru, kterou může sekvenčně prohledat.

2.4 Paralelní algoritmus – MPI

Pomocí knihovny OpenMPI lze rozdistribuovat výpočet na více výpočetních uzlů. Jednotlivé uzly spolu komunikují pomocí zpráv a práci rozděluje master proces mezi slave procesy. Pro vytvoření zprávy, která může být odeslána na jiný uzel, musí být objekt stavu převeden na primitivní datové typy a po přijetí převeden zase zpátky. Slave procesy provádí paralelní výpočet. Jedná se tedy pouze o rozdistribuování data paralelismu na více uzlů.

Po spuštění programu master proces vygeneruje z počátečního prázdného stavu několik dalších stavů, které poté postupně rozděluje mezi slave procesy. Slave proces z přijatého stavu opět pomocí BFS algoritmu vygeneruje několik dalších stavů a ty zpracovává pomocí data paralelismu na více jádrech. Po dokončení práce slave vrátí masteru zprávu že

dokončil výpočet a že očekává další práci. Pokud master proces stále má stavy k rozdělení, pošle slavu další stav. Pokud další práci k rozdělení nemá, oznámí slavu konec, slave pošle masteru svůj nejlepší stav a ukončí se.

3 Naměřené výsledky

Pro měření byl využit školní výpočetní cluster *star.fit.cvut.cz*, který pro tuto úlohu poskytuje až 4 uzly, kde každý uzel má 20 jader.

Pro měření jsem zvolil instance *graf*_30_20, *graf*_35_17, *graf*_40_8. Jejich velikost odpovídá tomu, aby je sekvenční řešení dokázalo vyřešit do 10 minut. Paralelní algoritmy jsou měřeny na 2, 4, 8, 16 a 20 jádrech. Distribuovaný MPI algoritmus je měřen na 2, 3 a 4 uzlech; pokaždé na 20 jádrech na každém uzlu.

Každé měření na daném počtu jader/uzlů jsem spustil 3x, vypsané časy jsou jejich průměrem.

Parametr a jsem pro každou instanci zvolil následovně:

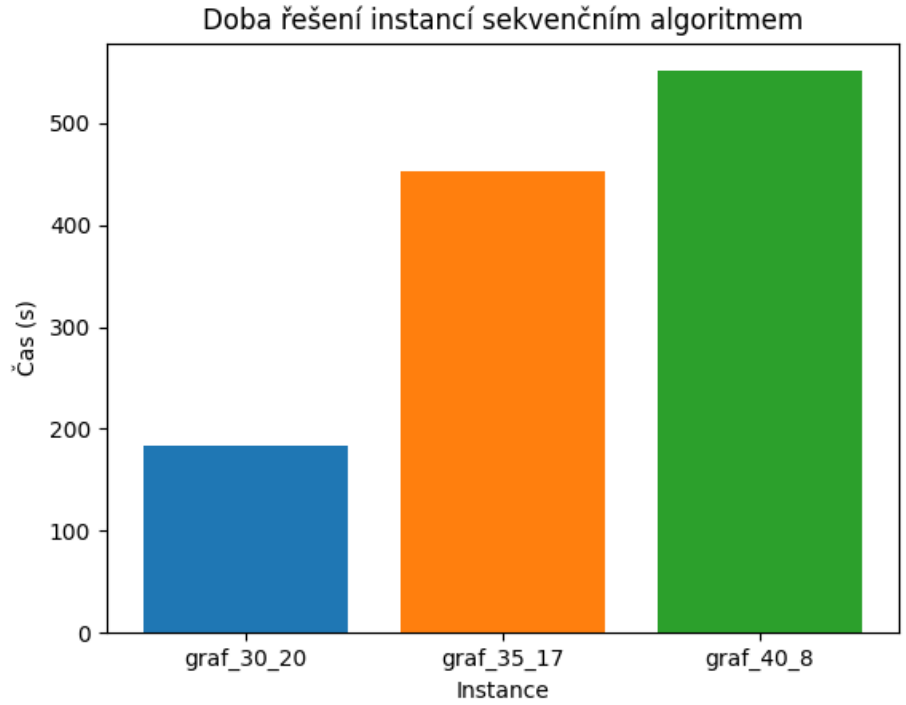
Zvolený parametr a jednotlivých instancí	
graf_30_20	15
graf_35_17	12
graf_40_8	8

Tato volba parametru také vychází z doby řešení sekvenčního algoritmu, aby doběhl do 10 minut, ale zároveň neskončil příliš brzy.

3.1 Sekvenční algoritmus

Je zřejmé že velikost instance přímo úměrně ovlivňuje dobu běhu. Po vyzkoušení algoritmu na několika různých instancí vyplynulo, že i průměrná váha vrcholů a parametr a výrazně ovlivňuje dobu běhu. Tento fakt ale není tolik podstatný, důležité je jak dlouho sekvenčnímu algoritmu trvalo najít správné řešení, aby u paralelní verzí algoritmu bylo vidět, o kolik se čas zlepšil.

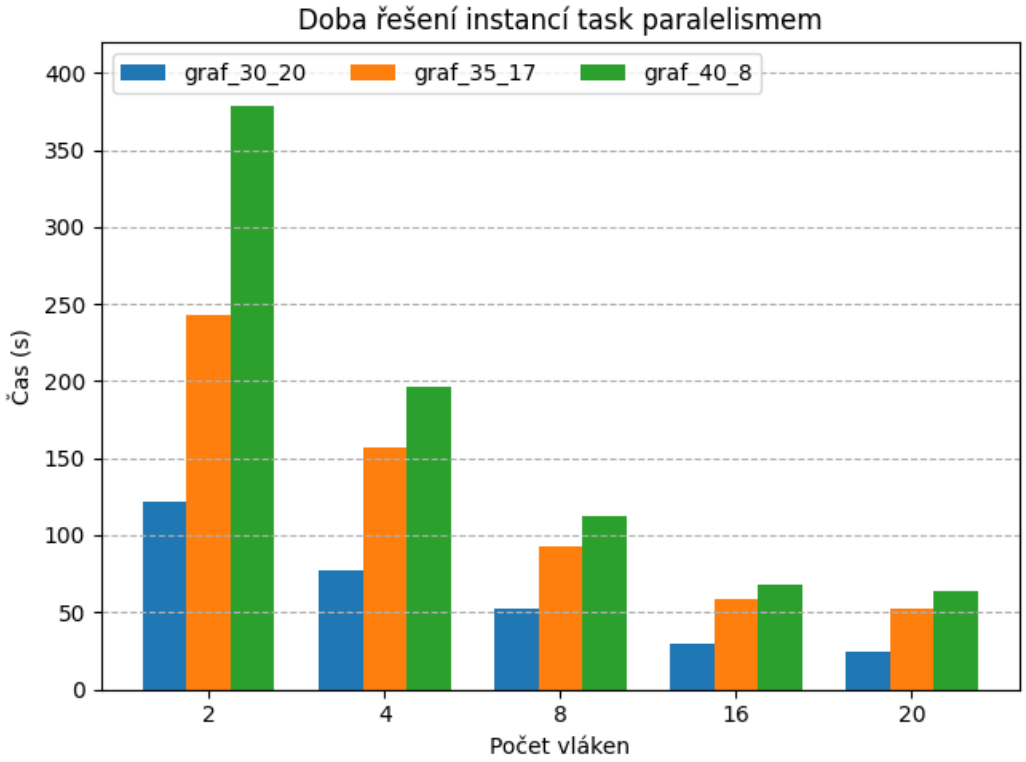
Doba řešení instancí **sekvenčním** algoritmem (v sekundách)	
graf_30_20	184.173
graf_35_17	452.601
graf_40_8	551.193



3.2 Paralelní algoritmus – task paralelismus

Task paralelismus znatelně výpočet zrychlil, avšak ne tolik aby se výsledek dal považovat za uspokojivý. V nejlepším případě na 20 vláknech je zrychlení 8.71 a efektivnost pouze 0.43. Zajímavým jevem je že s rostoucím počtem vláken je zrychlení lepší pro větší instance.

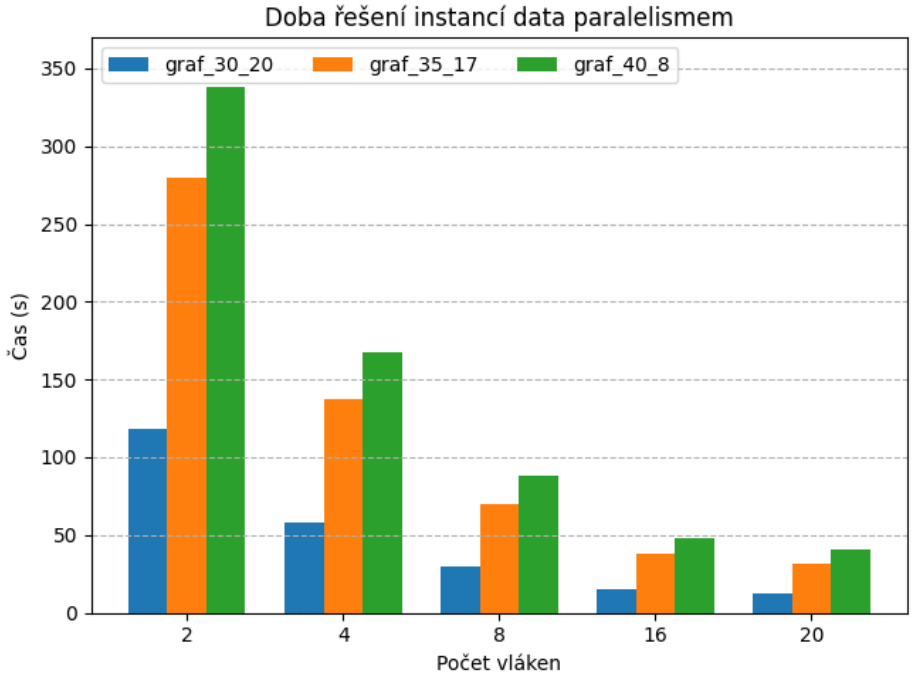
Doba řešení instancí **task** paralelismem v sekundách (a zrychlení)					
Instance	2	4	8	16	20
graf_30_20	121.648 (1.51)	77.122 (2.39)	52.610 (3.5)	29.226 (6.3)	24.108 (7.64)
graf_35_17	243.0466 (1.86)	156.513 (2.89)	92.569 (4.89)	59.0554 (7.67)	52.457 (8.63)
graf_40_8	378.924 (1.46)	196.432 (2.81)	112.575 (4.9)	68.285 (8.08)	63.287 (8.71)



3.3 Paralelní algoritmus – data paralelismus

U datového paralelismu je efektivita na 20 vláknech v nejlepším naměřeném případě 0.735. To je výrazně lepší než u task verze. Z popisu implementace je jisté, že hlavním důvodem je nižší režie u této verze implementace. Avšak pro maximální počet vláken, zrychlení klesá s rostoucí složitostí instance; to je rozdíl oproti task verzi algoritmu, ale to je pouze minimální faktor vzhledem k velkému rozdílu zrychlení a efektivity. Z toho důvodu je pro tuto úlohu datový paralelismus lepší volba.

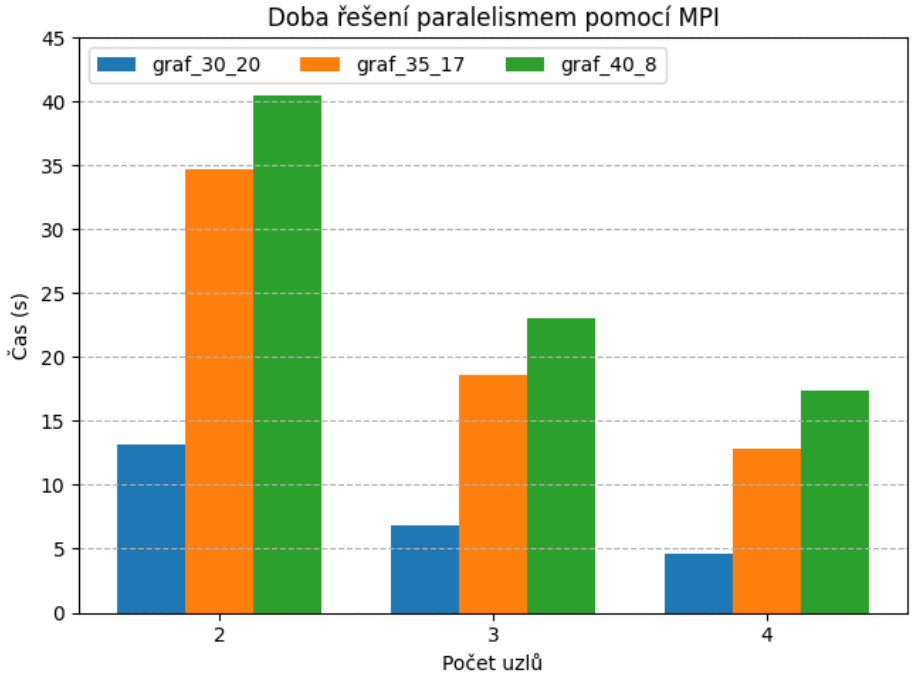
Doba řešení instancí **data** paralelismem v sekundách (a zrychlení)					
Instance	2	4	8	16	20
graf_30_20	118.182 (1.56)	58.330 (3.16)	29.344 (6.28)	15.400 (11.96)	12.520 (14.71)
graf_35_17	279.685 (1.62)	137.807 (3.29)	70.277 (6.44)	37.717 (12)	31.344 (14.44)
graf_40_8	337.837 (1.63)	167.836 (3.29)	88.011 (6.27)	48.166 (11.45)	40.559 (13.6)



3.4 Paralelní algoritmus – MPI

Naměřený čas pro pro dva uzly, kde jeden z uzlů je master, který nic nepočítá, je překvapivě dobrý; má skoro stejné zrychlení jako data paralelismus na 20 jádrech. Komunikace mezi masterem a slavem zhoršuje efektivitu pouze minimálně. Čas výpočtu pro 3 a 4 uzly je také uspokojivá. Zrychlení je až 39x pro nejmenší instanci. Avšak zrychlení klesá vzhledem ke složitosti instance.

Doba řešení paralelismem pomocí **MPI** v sekundách (a zrychlení)			
Instance	2	3	4
graf_30_20	13.142 (14.02)	6.775 (27.19)	4.666 (39.48)
graf_35_17	34.657 (13.06)	18.628 (24.3)	12.814 (35.33)
graf_40_8	40.421 (13.64)	23.010 (23.97)	17.404 (31.69)



4 Vyhodnocení výsledků

Z měření je vidět že tento problém je vzhledem ke své stromové struktuře dobře paralelizovatelný na více vláken/uzlech. U obou paralelních verzí, task a datové, bylo naměřeno výrazné zrychlení. A to i pro distribuované řešení.

Datový paralelismus byl pro tuto úlohu vhodnější. Jeho zrychlení a efektivita stoupá s rostoucím počtem vláken výrazně lépe než pro task paralelismus. V MPI verzi jsem využil také datového paralelismu.

Rychlost MPI je už na 3 (2 výpočetních) uzlech výrazně lepší než než jakýkoliv výsledek z datového, či task paralelismu. Z toho lze usoudit že komunikace mezi uzly algoritmus nijak výrazně neomezuje rychlost a úspěšně lze algoritmus distribuovat na více uzlů; to potvrzuje i výsledek pro 4 uzly.

Ačkoliv bylo při zvyšování počtu vláken dosahováno lepších výsledků, nebylo zrychlení superlineární, neboť se průměrný čas nezkracoval více než k-krát při k násobném zvětšení výpočetních jader.

Pro nejtěžší instanci s MPI implementací na 4 uzlech bylo naměřeno zrychlení 31.69, což je efektivita 0.528 (pokud se nepočítají jádra z master uzlu).

5 Závěr

V rámci tohoto úkolu jsem pracoval na řešení problému minimálního hranového řezu. Nejprve jsem vytvořil sekvenční řešení, které jsem pomocí knihovny OpenMP transformoval na taskový a datový paralelismus. Datový paralelismus byl výrazně efektivnější, proto jsem ho využil i v distribuovaném řešení pomocí knihovny OpenMPI, pomocí které se dalo problém řešit na několika workerech zároveň.

Při implementaci jsem nenarazil na žádné vetší obtíže. Synchronizace ke sdíleným zdrojům šla jednoduše vyřešit pomocí OpenMP direktiv. Pro rozdistribuování výpočtu na více procesorů bylo nutné stav výpočtu převést na primitivní datové typy, aby šlo pomocí OpenMPI přenést takový stav zprávou na jiný uzel.

Určitě by šlo jednotlivé implementace více optimalizovat, avšak pro demonstraci paralelismu pro tento typ úlohy byly algoritmy postačující.