Semestralní projekt MI-PAR 2014/2015:

Paralelní algoritmus pro řešení problému maximální klika grafu

Tomáš Šabata
Phú Hải Bùi

magisterské studijum, FIT ČVUT, Kolejní 550/2, 160 00 Praha 6

29. listopadu 2014

1 Definice problému a popis sekvenčního algoritmu

1.1 Definice:

Klika grafu G je jeho *maximální úplný podgraf*, t.j. takový podgraf, ktery není obsažen v žádném větším podgrafu. Velikost kliky je počet jejich vrcholů.

1.2 Úkol:

Zjistit, zda graf G obsahuje kliku o velikosti alespoň (rovnou nebo větší) r*n a nalézt největší takovou kliku.

1.3 Vstupní data:

G(V,E) = jednoduchý souvislý neorientovaný neohodnocený graf o n uzlech a m hranách
n = přirozené číslo představující počet uzlů grafu G, n >= 5
k = přirozené číslo řádu jednotek představující průměrný stupeň uzlu grafu G, n >= k >= 3
r = kladné reálné číslo, 0 < r < 1

Graf byl reprezentován pomocí matice sousednosti, který byl pomocí generátoru neorientovaných grafu vygenerován do souboru. V programu byl tento graf uchováván pomocí dvourozměrného pole.

1.4 Výstup algoritmu:

Seznam uzlů tvořící kliku a velikost kliky, popřípadě konstatování, že klika neexistuje.

1.5 Sekvenční algoritmus:

Sekvenční algoritmus typu *BB-DFS* s hloubkou stavového stromu omezenou na n. Cena řešení, která se maximalizuje, je velikost kliky vzhledem k zadané podmínce. Horní mez ceny řešení není známa. Algoritmus skončí, až prohledá celý stavový prostor.

Dolní mez je 2, pokud graf obsahuje aspoň 1 hranu.
Horní mez není známá, ale dá se odhadnout takto: pokud G obsahuje kliku o velikosti x, pak musí obsahovat x vrcholů se stupněm větším nebo rovným x-1.

1.6 Paralelní algoritmus:

Paralelní algoritmus je typu *PBB-DFS-V*.

1.7 Tabulka naměřených časů pro sekvenční řešení

Tabulka 1: Naměřené hodnoty pro sekvenční řešení

Instance	Čas [s]
55 uzlový	281
65 uzlový	615
70 uzlový	901

2 Popis paralelního algoritmu a jeho implementace v MPI

Paralelní algoritmus jde rozdělit do několika navzájem disjunktních fází. Některé fáze jsou rozděleny barierami napříč přes všechny procesy. Jednotlivé fáze jsou popsány v sekci 2.1 Fáze.
Struktura programu:

- načtení dat (načtení grafu z textového souboru)
- bariera (všechny procesy načetly data)
- rozeslání (přijímání) počáteční práce
- bariera
- vlastní výpočet
- získaní nejlepšího výsledku

2.1 Fáze

2.1.1 Načtení dat

V této fázi všechny procesory uloží graf ze souboru a načtou důležité konstanty do singletonu. Jedná se o konstantu velikosti zásobníku, který je napříč celým programem stejně velký. Konstatní velikost zásobníku usnadní zasílaní bufferu zásobníku pomocí knihovny MPI.

2.1.2 Rozesílaní (přijímaní) počáteční práce

Procesor s *id=0* začne rozesílat postupně procesorům práci. Každému procesoru se pošle jeden počáteční uzel stavového prostoru, který si ve svém výpočtu expanduje. Expanze stavového prostoru je totožná se sekvenčním algoritmem. Procesory s *id≠0*. Naslouchají na příchozí práci. V případě, že je více procesorů než práce, zasílá se procesorům prázdný zásobník.

2.1.3 Vlastní výpočet

Vlastní výpočet obsahuje hlavní smyčku která se řídí příznakem status, tedy zda je procesor *active* nebo *idle*. V případě, že procesor se nachází ve stavu active, provádí *SolveSubtree()*, která expanduje 100 stavů stavového prostoru, a provadí *checkMessages()*, kde odpovídá na zprávy od ostatních procesorů. V případě, že se procesor nachazí ve stavu *idle*, provádí *tokenStart()* (v případě procesoru s id=0), *getNewJob()* a *checkMessages()*.

Na počátku této fáze se většina procesorů nachází ve stavech *active*. Proto většina procesorů expanduje stavy a počítá kliku. Po každých 100 zpracovaných uzlech skočí zpátky do hlavní smyčky za účelem zpracovaní příchozích zpráv.

Soubežně s výpočtem se zahajuje algoritmus distribuvovaného ukončení výpočtu (ADUV), který spouští proces s id=0 po zpracovaní celého stvého stavového podprostoru.

2.1.4 Získaní nejlepšího výsledku

Protože v algoritmu *ADUV*, zasíláme kromě tokenu i idProcesoru s nejlepším možným výsledkem pak procesor s id=0 tomuto procesoru může okamžitě zaslat zprávu se žádostí o uzly kliky. Toto by se dalo úplně odstranit pokud by se při algoritmu ADUV zasílala i klika.

2.2 Vyvažování zátěže

2.2.1 Dělení zásobníku

Dělení zásobníku se provádí jinak, než je uvedeno v návodech na stránce EDUXu. Stavy se expandují a zpracovávají preorder, tedy po vyexpandování jednoho stavu se rovnou zpracovává. Dělení zásobníku tedy probíhá tak, že pokud je počet stavů nutných k expandovaní z akutalního stavu větší než konstanta k, pak se tento stav předá procesu žádajícímu o práci. Proces, který byl o práci žádán pokračuje tak, jako kdyby podstrom, který předal, již zpracoval.

2.2.2 Algoritmus hledání dárce

Algoritmus hledání dárce cyklicky rotuje doprava modulo počet procesů.

2.3 výčet důležitých funkcí

- *divideStack()* - rozdělí zásobník pokud je práce dostatek
- *WorkDone()* - převadí proces do stavu idle a volá metodu token()
- *getNewWork()* - žádá o novou práci
- *checkMessages()* - příjme všechny zprávy a obslouží je
- *tokenStart()* - procesor s id=0 startuje ADUV
- *Token()* - zpracovává a odesílá token algoritmu ADUV
- *JobRequest()* - žádá o novou práci
- *JobReceived()* - zpracovaní bufferu v případě, že došla nová práce
- *NoJobReceived()* - žádáný procesor nepřidělil novou práci

- *SendClique()* - rozesílá kliku ve fázi získání nejlepšího výsledku
- *isClique()* - zjištuje zda daný stav stavového prostoru je klika. Složitost $O(n^2)$
- *solveSubtree()* - řeší dany podstrom, zastaví se po vyšetření všech uzlů v podstromu. Každých 100 zpracovaných uzlů se přerušuje pro zpracovaní zpráv
- *askerID()* - získa id procesu, ktereho se ma zeptat na praci. Rotuje s id procesů.
- *sendWorkAtStart()* - distribuje praci ve fázi rozesílaní (příjímaní) počáteční práce
- *void listenAtStart()* - naslouchа na praci ve fázi rozesílaní (příjímaní) počáteční práce
- *void startComputing()* - zahajeni fáze vlastní výpočet
- *void printResults()* - sebrani vysledku a interpretace. Jedná se o fázi získaní nejlepšího výsledku

3 Naměřené výsledky a vyhodnocení

Hledání potřebných instancí, které mají časovou složitost 5, 10 a 15 minut, jsem prováděl na výpočetním svazku STAR. Sekvenční měření složitosti začalo, až když byl načten graf ze souboru a následné spuštení výpočtu. Výsledné hodnoty jsou aritmetické průměrné hodnoty pro 3 krát změřené hodnoty. Našel jsem tyto grafy, které se svojí složitostí (viz. Tabulka 2: Naměřené hodnoty) blížily hledaným hodnotám, a to:

- 55 uzlový graf s průměrný stupněm uzlu grafu 44
- 65 uzlový graf s průměrný stupněm uzlu grafu 52
- 70 uzlový graf s průměrný stupněm uzlu grafu 55

Pro jednotlivé měření instancí paralelního času jsem použil i = 2, 4, 8, 16 a 32 procesorů pro porovnání se složitostí sekvenčního algoritmu. Následující tabulka je pro zvolenou konstantu *500* (viz. Tabulka 2: Naměřené hodnoty).

Tabulka 2: Naměřené hodnoty

Počer procesorů / instance	**55 uzlový**	**65 uzlový**	**70 uzlový**
sekvenční	**281 s**	**615 s**	**901 s**
2	214 s	797 s	925 s
4	216 s	727 s	704 s
8	177 s	328 s	326 s
16	171 s	90 s	132 s
32	178 s	95 s	134 s

3.1 Graf zrychlení

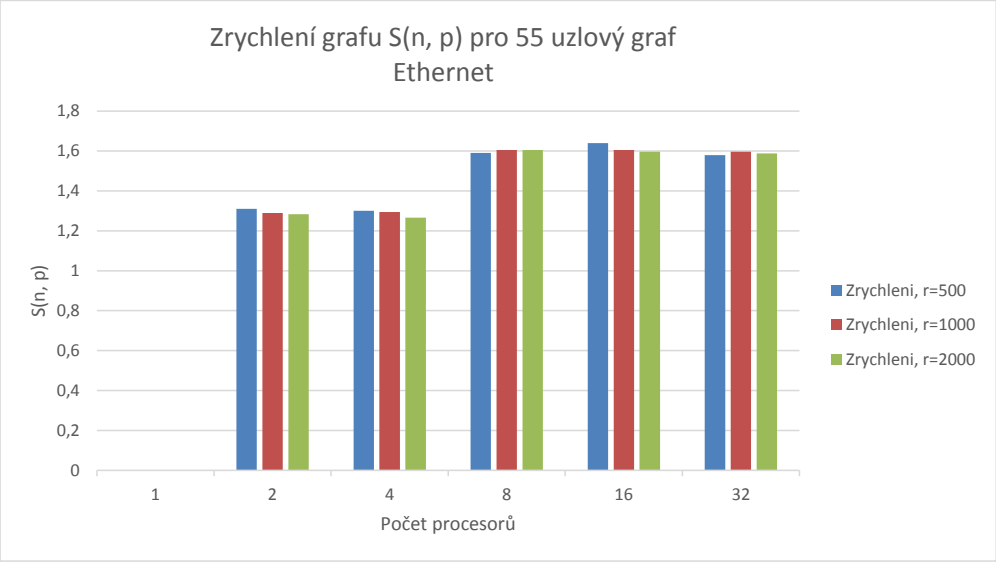
Udělaji jsme pro každý problém instance různé konstanty r=500, 1000 a 2000. Konstanta r je číslo, které udává velikost problému, kdy se má zásobník dělit.

3.1.1 Problém a) 55 uzlový graf

Tabulka 3: Naměřené hodnoty pro graf o 55 uzlech

Počer procesorů	**1**	**2**	**4**	**8**	**16**	**32**
Čas [s], r=500	281	214	216	177	171	178
Zrychlení, r=500	x	1,31	1,30	1,59	1,64	1,58
Čas [s], r=1000	281	218	217	175	175	176
Zrychlení, r=1000	x	1,29	1,29	1,61	1,61	1,60
Čas [s], r=2000	281	219	222	175	176	177
Zrychlení, r=2000	x	1,28	1,27	1,61	1,60	1,59

Obrázek 1: Zrychlení grafu S(n, p) pro 55 uzlový graf - Ethernet

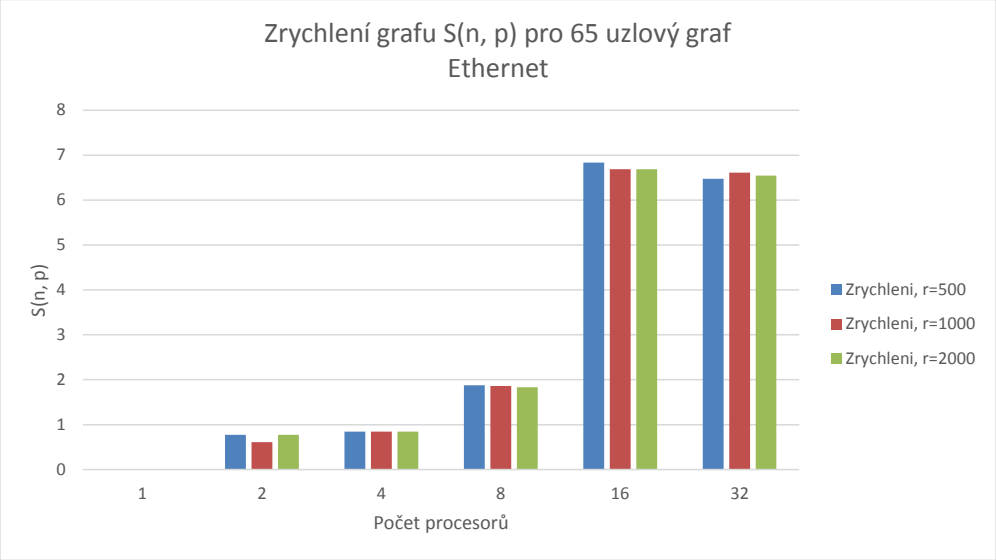


3.1.2 Problém b) 65 uzlový graf

Tabulka 4: Naměřené hodnoty pro graf o 65 uzlech

Počer procesorů	**1**	**2**	**4**	**8**	**16**	**32**
Čas [s], r=500	615	797	727	328	90	95
Zrychlení, r=500	x	0,77	0,85	1,88	6,83	6,47
Čas [s], r=1000	615	1007	729	331	92	93
Zrychlení, r=1000	x	0,61	0,84	1.86	6,68	6,61
Čas [s], r=2000	615	794	728	336	92	94
Zrychlení, r=2000	x	0,77	0,84	1,83	6,68	6,54

Obrázek 2: Zrychlení grafu S(n, p) pro 65 uzlový graf - Ethernet

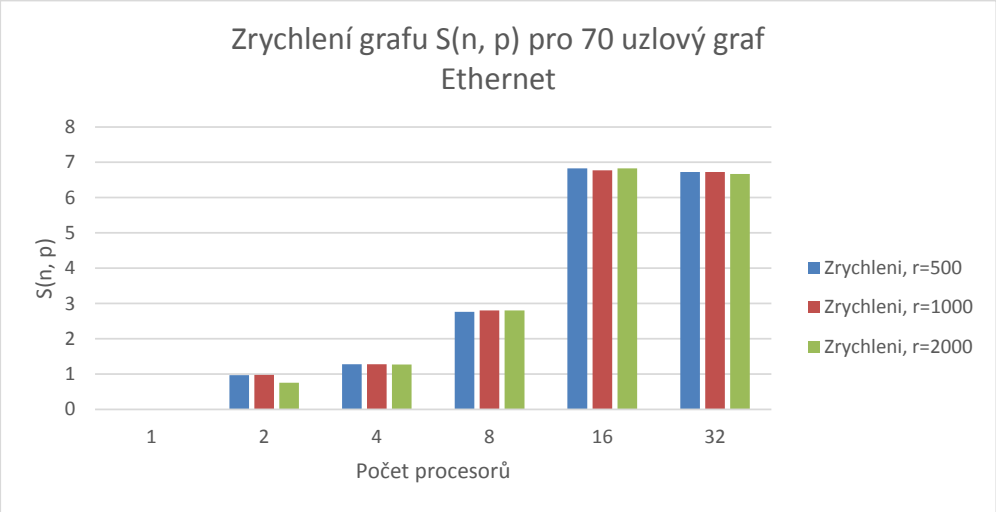


3.1.3 Problém c) 70 uzlový graf

Tabulka 5: Naměřené hodnoty pro graf o 70 uzlech

Počer procesorů	**1**	**2**	**4**	**8**	**16**	**32**
Čas [s], r=500	901	925	704	326	132	134
Zrychlení, r=500	x	0,97	1,28	2,76	6,83	6,72
Čas [s], r=1000	901	924	705	321	133	134
Zrychlení, r=1000	x	0,98	1,28	2,81	6,77	6,72
Čas [s], r=2000	901	1191	708	321	132	135
Zrychlení, r=2000	x	0,76	1,27	2,81	6,83	6,67

Obrázek 3: Zrychlení grafu S(n, p) pro 70 uzlový graf - Ethernet



3.2 Komunikační složitost

Z naměřených dat jsme vypozorovali, že pro p>16 procesorů začíná být síťová komunikace velká, kde se to odrazilo ve zrychlení paralelního výpočtu. Je to způsobeno nenalezení správné konstanty pro dělení práce pro vysoký počet procesorů, kde pak je komunikace po síti četnější. Pro tuto úlohu jsme nenašli instanci, pro kterou by existovalo superlineární zrychlení.

3.3 Granularita

Granulita pro tuto úlohu nemá cenu dělit více než na 16 procesorů, protože pak už vychází podobné výsledky s malou odchylkou. Je to způsobeno vysokou cenou za síťovou komunikaci a výpočet pro p>16 procesorů začíná být cenově (paralelně) nevýhodný. Výpočty jsou díky lineárnímu zrychlení pořád cenově optimální.

Pro malé grafy, které běží na sekvenčním řešení přibližně 5 minut dosahuje stále lineárního zrychlení, ale paralelní cena už je opravdu nízka, kde stojí za zváženou, jestli nechat výpočet běžet na více procesorech.

4 Závěr

Implementace sekvenčního algoritmu a následné porovnání naměřených výsledků paralelního algoritmu potvrdilo fakt, že dojde ke zrychlení výpočtu konkrétního problému. Paralelní algoritmus by se dal určitě zrychlit při nalezení správné konstanty pro dělení práce pro vysoký počet procesorů, v našem případě p>16, kde by nedocházelo k přílišné komunikaci po síti. V určité vysoké zvolené konstanty by nedocházelo k přerozdělování práce mezi procesory a pak by docházelo ke zpomalení výpočtu.