Semestralní projekt MI-PAR 2011/2012:

Paralelní algoritmus pro řešení problému:
KGM - kostra grafu s minimálním stupněm

Antonín Daněk
Patrik Kompuš

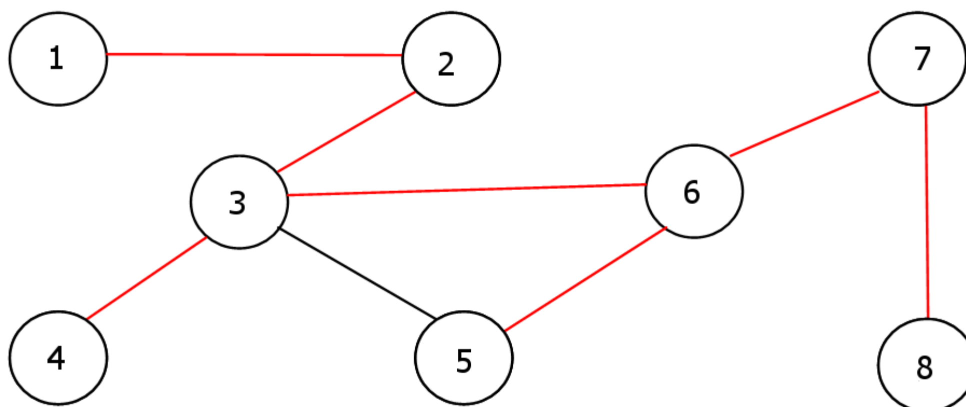
magisterské studijum, FIT ČVUT, Thákurova 9, 160 00 Praha 6

5. prosince 2011

1 Definice problému, vstup a výstup

1.1 Zadání

Úkolem je nalézt kostru (libovolného) grafu G s minimálním stupněm, kde stupeň kostry grafu je definován jako maximální stupeň ze všech uzlů kostry. Řešení existuje vždy, protože lze vždy sestrojit kostru grafu.



Obrázek 1: Červené hrany zobrazují kostru stupně 3, která je také minimální kostrou tohoto grafu.

1.2 Vstup

Vstupem je jednoduchý, souvislý, neorientovaný, neohodnocený graf *G(V,E)* 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 $>= 3$

Pro generování vstupních grafů byl použit generátor s volbou typu grafu **"-t AD"**, který generuje neorientované, neohodnocené grafy. U těchto grafů byla následně ověřena jejich *souvislost*, neboť tu generátor nezajišťuje.

Příkladem vstupu je:

01000000
10100000
01011100
00100000
00100100
00101010
00000101
00000010

Jedná se o graf zobrazený na obrázku 1. Přímý výstup z programu *generátor* navíc obsahuje na prvním řádku hodnotu, která udává počet vrcholů. Mý používáme vstup bez této informace.

1.3 Výstup

Výstupem je výpis kostry grafu G a hodnota stupně této kostry. Pro vstup uvedený jako příklad vstupu vypadá výstup takto:

Input has 8 edges.
Graph has 8 vertexes.
Best spanning tree degree is: **3**
[1 - 2][2 - 3][3 - 4][3 - 6][6 - 5][6 - 7][7 - 8]

2 Popis sekvenčního algoritmu a jeho implementace

Sekvenční algoritmus je typu BB-DFS[1]. Přípustný koncový stav je vytvořená kostra s minimálním stupněm. Přístupný mezistav je vytvořená kostra, o které nevíme, jestli je minimálního stupně.

Algoritmus končí po prohledání celého prostoru či při dosažení dolní meze. Těsná dolní mez je rovna 2. Horní mez je rovna nejvyššímu stupni v grafu G. Cena, kterou minimalizujeme, je stupeň kostry.

2.0.1 Implementace a průběh programu

Algoritmus je implementován pomocí zásobníku, přičemž jeho základní částí je **generátor kombinací** a metody pro ověření, zda je daná kombinace kostrou grafu.

Po načtení vstupu ze souboru dochází k inicializaci generátoru kombinací, která spočívá ve vytvoření výchozí kombinace a její vložení na vrchol zásobníku. Tato výchozí kombinace se skládá ze všech vstupních hran.

Následně jsou procházeny všechny kombinace generátoru, který však vrací pouze **kombinace velikosti V-1** (víme, že kostra grafu má vždy takový počet hran) a navíc má libovolný uzel této **kombinace stupeň menší, než stupeň nejlepší kostry**, jakou jsme již našli. Podrobněji o generování kombinací v 2.0.2.

Z takto získané kombinace je vytvořen graf a zbývá pouze ověřit, zda daný graf **neobsahuje cyklus a zda je spojitý** (získaná kombinace hran nemusí tvořit spojitý podgraf, i když původní graf spojitý je).

V případě úspěchu těchto testů jsme nalezli novou kostru s menším stupněm a algoritmus tehdy uloží hrany a stupeň kostry právě zpracovávaného grafu. Hodnotu stupně kostry také předáme (pro účely optimalizace hledání dalších kombinací) generátoru. V této situaci může dojít k ukončení výpočtu a to v případě, že právě nalezené řešení má stupeň roven **dolní mezi**.

Pakliže nalezené řešení dolní mezí není, algoritmus pokračuje další kombinací, dokud nejsou zkontrolovány **kombinace všechny**.

[1] Branch-and-Bound Depth-First Search

2.0.2 Princip generování kombinací a prořezávání

Jak je možné vidět na obrázku 2, generátor pracuje s kombinačním vektorem **(a,b)**, který se skládá ze dvou množin čísel. První množina (**a**) představuje samotnou kombinaci, přičemž množina druhá (**b**) se používá ke generování kombinací dalších (při použití kombinace **a** jako prefix).

Nevýhody:

Hlavní nevýhodou je nemožnost generování pouze kombinací délky **k**, jako to lze udělat u algoritmu popsaném *Kennethem H. Rosenem*. Generujeme proto poměrně hodně kombinací hran, které nikdy nemohou být kostrou grafu, protože jejich počet **není k=V-1** (na obrázku jsou označny žlutě kombinace velikosti k=4). Tato vlastnost se však dá dobře využít k optimalizaci.

Výhody:

Největší výhodou (pro nás nutný požadavek) je možnost snadného **vyjmutí kombinačního vektoru** ze stromu kombinací a tento vektor poté použít ke generování podstromu kombinací, jehož kombinace se nenacházejí v žádném jiném podstromu. Díky tomu je možné problém paralelizovat (podrobněji viz 3).

Dále je možné aplikovat některé heuristiky, které proces generování optimalizují. Víme, že v žádném podstromu uzlu nemůže být kombinace velikosti **k**, jestliže:

- Velikost množiny čísel **a** je větší než **k**.
- Součet velikostí množiny **a** a množiny **b** je menší než **k**.

Kromě toho je generování kombinací optimalizováno tzv. prořezáváním, které můžeme aplikovat až tehdy, známe-li libovolnou hodnotu stupně kostry.

Pro příklad se můžeme opět podívat na obrázek 2, kde je červeně znázorněna kombinace 1-3-4. Představme si, že máme graf, ve kterém jsme již nalezli kostru stupně 3 a nyní tento stupeň používáme k prořezávání. Algoritmus v tomto případě rozpozná (pokud to tak je), že hrany 1-3-4 tvoří uzel stupně 3 (např. by mohlo jít o hrany [2-5][2-6][2-7]) a tudíž takový graf nemůže vytvořit lepší řešení, než jaké již máme. Je možné pozorovat, že kombinace uzlu slouží jako prefix všech kombinací v jeho podstromech a tudíž je možné na základě tohoto zjištění uzavřít celý podstrom tohoto uzlu.

Je třeba si uvědomit, že příklad na obrázku 2 je velmi malý (a ani zde nejsou zobrazeny všechny kombinace) a u větších grafů s miliardami kombinací metoda **prořezávání výpočet velmi zrychluje**.

2.1 Měření sekvenčního algoritmu

Jak je možné vidět v tabulce 1, doba běhu programu je velmi závislá nejen na počtu hran a uzlů, ale také na faktu, jestli se v grafu nachází dolní mez.

V obou případech rychlost závisí na rozmístění jednotlivých koster v kombinačním stromu, přičemž v případě grafu s dolní mezí může být rozdíl mnohem patrnější. To je možné vidět např. na grafu s id *_17.4min.sh*, jehož vyřešení zabralo přibližně 17,5 minuty, ale vyřešení grafu *graf14k4_2.sh*, který má ještě o hranu více, ale obsahuje dolní mez, pouze 1,2 minuty.

Může se stát, že i u velmi velkého grafu nalezneme **kostru stupně 2** v počátečních kombinacích stromu a můžeme tedy **hned výpočet ukončit**. Podobně pokud u grafu bez dolní meze nalezneme **na začátku kostru nízkého stupně**, můžeme rychleji a **efektivněji prořezávat** strom kombinací.

Také je možné u grafů bez dolní meze (kde je nutné zkontrolovat celý prostor kombinací) pozorovat velmi strmý nárůst složitosti v závislosti na počtu hran a vrcholů. To je způsobeno časovou složitostí algoritmu generátoru O(n!). Pokud bychom použili generátor, který umí generovat pouze kombinace velikosti k, pak by složitost byla O(n! / (r! (n - r)!)), kde n=počet hran a r=počet vrcholů - 1. Počty kombinací je také možné vidět v tabulce 1.

3 Popis paralelního algoritmu a jeho implementace v MPI

Paralelní algoritmus je typu G-PBB-DFS-D[2], přičemž základní princip je stejný, jako u sekvenčního algorimut 2 - tedy dochází ke generování nových kombinací hran a ty jsou ověřovány. Je zde však důležitý rozdíl, že prostor kombinací není zpracováván pouze na jednom procesoru. Při realizaci tohoto paralelního algoritmu byla použita knihovna MPI[3], pomocí které probíhá komunikace mezi jednotlivými procesy.

[2]Global Parallel Branch-and-Bound Depth-First Search Dynamic Data Exchange
[3]Message Passing Interface

3.1 Inicializace výpočtu

Počátek algoritmu je (kromě načtení vstupu) **odlišný** pro proces typu master (může být libovolný proces, ale je volen proces s id=0) a **pro procesy typu slave**.

Master proces vloží na zásobník iniciální kombinační vektor (stejným způsobem jako je tomu u sekvenčního algoritmu 2.0.1, struktura vektoru je popsána v 2.0.2) a expanduje jej za účelem **odeslání práce slave procesům**. Problém je, že při tomto generování práce vznikají kombinace, které je třeba také zkontrolovat - ty se proto ukládají na tzv. master zásobník pro pozdější kontrolu. Po vygenerování (a odeslání) práce master proces zkontroluje kombinace v master zásobníku a poté pracuje stejným způsobem, jako procesy slave.

Slave proces na začátku pouze **čeká, dokud mu není zaslán kombinační vektor**. Až se tak stane (stane se vždy, i když není nalezeno dostatek práce pro všechny - pak je procesu zaslán vektor prázdný), vloží si tento vektor na vrchol svého doposud prázdného zásobníku a začne jej dále expandovat a zpracovávat.

3.2 Komunikace pomocí MPI

Práce s MPI byla v kódu vyexternalizována do statické třídy *MPIUtils*.

Kromě inicializace výpočtu pak veškerá komunikace s ostatními procesy probíhá v rámci **centrální komunikační smyčky**, kde je vždy jednou za určitý počet lokálně zpracovaných grafů (daný konstantou) zkontrolováno, zda současnému procesu nepřišla zpráva a v případě potřeby je na ni náležitě zareagováno.

V programu existují tyto zprávy:

3.2.1 MSG_WORK_REQUEST

Tuto zprávu proces odešle v případě, že zpracoval celý svůj lokální zásobník. Podrobněji o výběru příjemce zprávy v 3.3.

3.2.2 MSG_WORK_SENT

Zpráva je jednou ze dvou možností odpovědi na zprávu 3.2.1, která je zaslána v případě, že **proces má dostatek práce a může se o ni podělit**.

Práce (kombinační vektor) je serializována do pole integerů, přičemž jednotlivé množiny jsou ukončeny oddělovačem -1. Práce je posílána neblokujícím MPI_Isend, protože je pravděpodobné, že data budou větší než 1kB a pro data větší než 1kB hrozí u MPI_send deadlock.

Na straně příjemce pak probíhá deserializace, k čemuž jsou opět využity oddělovače přidané do zprávy. Problém je, že některé implementace MPI nepodporují zasílání velikosti dat ve status kódu MPI zprávy, proto příjemce dat musí alokovat tak velký zásobník, jaká největší data mohou dorazit. Nejdelší data mohou být délky počtu hran + 2 (pro oddělovače).

Podrobněji o výběru práce v 3.2.1.

3.2.3 MSG_WORK_NOWORK

Tato zpráva je druhou možnou odpovědí na zprávu 3.2.1 a je procesem odeslána v případě, kdy sám žádnou práci nemá resp. jí nemá dostatek, aby mělo smysl ji rozdělit mezi více procesů.

3.2.4 MSG_FOUND_SOLUTION

Zprávu odesílá vždy ten proces, který našel **nové lepší řešení** a to všem ostatním procesům. Do zprávy se ukládá pouze jedna hodnota, která vyjadřuje stupeň nově nalezené kostry.

Při příjmu je vždy zkontrolováno, zda-li toto nové řešení je opravdu lepší, protože se může stát, že než došlo k přijetí zprávy, bylo lokálně nalezeno řešení ještě lepší - pak bychom začali optimalizovat horším řešením, než jakým ve skutečnosti můžeme, ačkoliv na funkčnost algoritmu by toto vliv nemělo.

Účel této zprávy je dvojí. V první řadě je použita k rozesílání dílčích řešení mezi jednotlivými procesy, aby všechny mohly co nejlépe optimalizovat proces hledání kombinací (princip prořezávání je popsaný v 2.0.2). V druhé řadě se týká pouze master procesu, který může vyvolat **ukončení výpočtu** a to v případě, že přijaté řešení je **dolní mezí (2)**. Podorbněji viz následující zprávy.

3.2.5 MSG_END_CALCULATION

Tato zpráva může být odeslána **výhradně master procesem** a je v ní uložena hodnota nejlepšího řešení, jaké se na síti nachází. Každý slave pak jednoduše ověří, jestli on toto řešení má. Pokud ano, **pošle jej zpět a ukončí svou činnost** (jinak pouze ukončí svou činnost).

3.2.6 MSG_END_BEST_SOLUTION

Tato zpráva je jednou ze dvou možných odpovědí na zprávu 3.2.5 - v případě, že proces disponuje požadovaným nejlepším řešením.

Nejlepší řešení je odesláno **pouze v podobě textu**, protože s ním zde nepotřebujeme dále pracovat. Pokud bychom chtěli nalezenou kostru dále strojově využít, bylo by vhodné doimplementovat serializaci do pole čísel a především deserializaci na straně příjemce.

3.2.7 MSG_END_DONT_HAVE

Tato zpráva je jednou ze dvou možných odpovědí na zprávu 3.2.5 - v případě, že proces požadované nejlepší řešení nemá.

3.2.8 MSG_END_TOKEN

Tento token odešle každý proces, který dokončil zpracování své fronty a všechny ostatní procesy mu oznámily, že nemají práci navíc. Token se posílá vždy **následujícímu procesu** - *(id+1) mod pocet*. Při odesílání či přeposílání tokenu proces přechází do pasivního módu, ve kterém už pouze kontroluje zprávy na síti a odpovídá na ně.

Důležité je, že se do zprávy ukládá informace o tom, **odkud token pochází** (jinak by mohlo dojít k předběhnutí jiným tokenem, který ještě nenavštívil všechny procesy) a také **stupeň nejlepšího nelezeného řešení**.

Každý další proces si tento token buď uloží na později (v případě, že má ještě práci ke zpracování), a nebo (pokud se již nachází v pasivním módu) jej přepošle dál. Při přeposílání tokenu dochází u každého procesu ke kontrole, zda současný proces nenalezl řešení nižšího stupně, než jaké je uložené v tokenu. Pokud ano, pak hodnotu přepíše. Poté co token projde přes všechny procesy a vrátí se ke svému odesílateli, můžeme prohlásit dvě věci:

- V tokenu je uložena hodnota kostry s nejnižším stupněm.
- Všechny procesy dokončily práci a přešly do pasivního módu.

Použití tohoto tokenu teoreticky provází problém předčasného vyřazení některých procesů, zatímco na sítí se ještě stále nachází práce. Představme si následující případ:

- Proces **A** dokončí zpracování svého zásobníku a postupně žádá všechny ostatní procesy o práci.

- Procesy odpovídají, že už práci nemají, ale minimálně jeden z nich (**B**) ve skutečnosti ještě práci má, jenom se z důvodu malého počtu kombinací na zásobníku rozhodl, že procesu A práci nepošle.

- Proces **A** přejde do pasivního módu, tedy už pouze kontroluje MPI zprávy a neprovádí žádný výpočet.

- Proces **B** mezi tím rozexpandoval svou původní malou práci do velké a proces **A** tak zbytečně neprovádí žádnou práci, i když by mohl procesu B pomoci.

Popsaný problém však může vzniknout jen v případě, že nemůžeme odhadnout velikost rozexpandovaného podstromu kombinací. V našem případě to ale lze a problém tudíž nehrozí - podrobněji viz 3.4.

Pokud bychom si chtěli být naprosto jistí, mohli bychom použít ověřený algoritmus ADUV[4], který pracuje na principu obarvování tokenu.

3.2.9 MSG_END_TOKEN_IS_BACK

Tuto zprávu odresovanou master procesu odešle slave, pokud se k němu vrátil jeho token.

Je v ní uložena **hodnota nejlepší kostry na síti** a master proces na základě této zprávy rozešle ukončovací výzvu 3.2.5 - čímž získá nejlepší kostru od toho, kdo ji našel.

3.3 Algoritmus hledání dárce

Pro hledání dárce byl použit algoritmus NV-AHD[5], přičemž generátor náhodných čísel je v každém procesu inicializován jiným semínkem (*time(0) + myProcessId*).

Vždy když procesu dojde práce, začne žádat ostatní procesy o práci následujícím způsobem:

1. Vytvoří se seznam id všech procesů (kromě id žadatele).

[4] Algoritmus pro distribuované ukončení paralelního výpočtu
[5] Náhodné Výzvy - Algoritmus pro Hledání Dárce

2. Je vygenerováno náhodné číslo v rozsahu indexů tohoto seznamu.

3. Takto vybraný proces je odebrán ze seznamu a je mu zaslána žádost o práci.

4. Pokud proces práci nemá, pokračuje se znovu bodem 2.

3.4 Algoritmus dělení zásobníku

Protože se v našem případě dá **odhadnout množství kombinací v jednotlivých podstromech** stromu kombinací, nevyužíváme přímo žádný algoritmus navrhovaný na webu předmětu. Nejblíže je však náš algortimus algoritmu D-ADZ[6].

Při pohledu na grafické znázornění stromu kombinací 2 je vidět, jak velikost podstromu určitého uzlu závisí na velikosti množiny b kombinačního vektoru. Konkrétně lze spočítat jako (size(b)-1)! + (size(b)-2)! + ... + 1, neuvažujeme-li žádné optimalizace v podobě heuristik či prořezávání.

Na základě tohoto se lze poměrně dobře rozhodnout, zda-li má smysl ještě dále práci rozdělovat. Konkrétně procházíme celý zásobník vektorů a ukládáme si celkový součet velikostí všech množin b. Navíc si také ukládáme, který vektor má množinu b největší. Následně v závislosti na těchto hodnotách provedeme rozhodnutí (kontrolujeme, jestli je největší b množina dostatečně velká a jestli je jejich **součet dostatečně velký**).

Pakliže se rozhodneme práci zaslat, vrátíme žadateli **vektor s největší b množinou**.

3.5 Ukončení výpočtu

Vše podstatné je popsané u jednotlivých zpráv: 3.2.5, 3.2.6, 3.2.7, 3.2.8 a 3.2.9, přičemž ukončení může nastat za různých podmínek a to konkrétně:

- Master proces nalezne dolní mez.
- Slave proces nalezne dolní mez a informuje o tom master proces zprávou 3.2.4.
- Master procesu se vrátil jeho ukončovací token.

[6] půlení u Dna-Algoritmy pro Dělení Zásobníku

- Master proces dostal zprávu o tom, že někomu jinému se vrátil jeho ukončovací token.

Všechny zmíněné případy vyusťují ve stejný závěr, kdy **master proces disponuje informací o hodnotě nejlepšího řešení na síti** a pomocí zprávy 3.2.5 postupně od všech slave procesů toto řešení vyžaduje a zároveň jim tím signalizuje konec (ne pouze výpočtu ale i programu samotného).

Zajímavostí je, že se může na síti v některých případech vyskytovat těchto nejlepších řešení i více (často existuje více koster např. stupně 3 apod.). To nepředstavuje problém a je pak použito řešení poslední přijaté.

3.6 Naměřené výsledky a vyhodnocení

Pro měření byly použity **stejné instance problému**, jaké byly použity u měření sekvenčního algoritmu (jehož výsledky je možné vidět v tabulce 1). Při měření byly zrušeny všechny pomocné a informační výpisy do konzole.

Všechna měření byla provedena pouze při použití komunikační sítě InfiniBand. Měření na Ethernetu nebylo provedeno z důvodu ušetření kapacit výpočetního klastru Star (a s vědomím, že rozdíly jsou zanedbatelné).

3.6.1 Parametry pro škálování algoritmu

Měření škálování algoritmu bylo měřeno pouze na grafu s id **_14.5min.sh** - k obdobným výsledkům bychom došli i u jiných grafů.

Algoritmus je možné škálovat pomocí dvou konstants. První nese název **CHECK_MSG_AMOUNT** a ovlivňuje, jak často algoritmus kontroluje nově příchozí MPI zprávy. Výsledky měření je možné vidět na obrázku 3.

Jak je vidět, hodnota tohoto parametru **výrazně neovlivňuje rychlost** algoritmu. Projevovat se začíná až u větších počtů procesů (v závislosti na velikosti vstupního grafu), kde je vhodnější volit **nižší hodnoty**. Pokud pustíme relativně malý graf na velkém počtu procesorů, bude algoritmus nucen více komunikovat, protože jednotlivé procesy nebudou mít dostatečně velké podstromy kombinací a budou nuceni častěji žádat o práci.

Druhou konstantou pro škálování je **WORK_SPLIT_CONSTANT**, která ovlivňuje dělbu práce mezi procesy. Čím větší hodnota konstanty, tím více práce musí proces mít, aby byl ochotný se o ni podělit s jiným procesem.

Jak je vidět na grafu 4, opět je vhodné nastavit **nízkou hodnotu**, při které se procesy budou dělit o práci i pro poměrně malé počty kombinací.

Je zde také vidět, že pokud nastavíme dostatečně velkou hodnotu, nedochází od určitého počtu procesů už k žádnému zrychlení, protože procesy zpracují pouze práci přidělenou od master procesu na začátku - dále už pouze čekají na konec, protože jim další práce není přidělena.

3.6.2 Zrychlení

Naměřená data s optimálními konstantami CHECK_MSG_AMOUNT=100 a WORK_SPLIT_CONSTANT=30 je možné vidět v tabulce 2 a grafu 5. Graf zrychlení ($S(n,p) = SU(n)/T(n,p)$) jsem z důvodu názornosti rozdělil do dvou, protože grafy s dolní mezí (7) vykazují oproti těm bez dolní meze (6) podstatně větší zrychlení a spadají tak do jiného měřítka.

Na grafech je možné vidět, že ve dvou ze čtyř případů došlo k **superlineárnímu zrychlení** u grafů bez dolní meze a na grafech s dolní mezí došlo k superlineárnímu zrychlení vždy.

K tomuto zrychlení dochází v případě grafů s dolní mezí z důvodu, že není nutné procházet všechny kombinace hran, ale pouze je třeba najít toto nejlepší řešení. Hledaná kombinace se může nacházet kdekoliv ve stromu kombinací, ale čím **více procesů** k prohledávání použijeme, tím **větší šanci** máme, že tuto kombinaci najdeme dříve (za předpokladu, že rovnoměrně rozdělíme práci).

Z podobného důvodu dochází k superlineárnímu zrychlení u některých grafů bez dolní meze. Zde se využívá **prořezávání stromu** kombinací a čím dříve najdeme (co nejnižší) řešení, tím dříve (a efektivněji) můžeme začít metodu aplikovat.

3.6.3 Posouzení paralelního algoritmu

Komunikační složitost algoritmu je poměrně nízká, program nevyžaduje díky dobrému rozdělování práce příliš komunikace. Nejvíce program začíná komunikovat blízko konci výpočtu, kdy začínají mít všechny procesy nedostatek práce a předávané vektory už nepředstavují tak velké podstromy. Je zde však stále prostor pro optimalizaci, což je vidět na naměřených výsledcích, kde ne u všech grafů bylo dosaženo superlineárního zrychlení.

Algoritmus **hledání dárce** pracuje s generátorem náhody, což z něj dělá poměrně kvalitní nástroj, ale i zde je prostor pro optimalizaci. Teoreticky lepším řešením by bylo nejdříve ověřit velikost práce na všech procesech a poté o práci požádat toho, kdo jí má nejvíce. Prakticky by to však znamenalo

čekat na odpověď od všech procesů při každé žádosti o práci, což by mohlo způsobit zpomalení.

Algoritmus **dělení zásobníku** pracuje s velikostí množiny **b** kombinačního vektoru (2.0.2), která vyjadřuje množství kombinací v kombinačním podstromu a tím umožňuje učinit informované rozhodnutí, zda práci poslat (a jakou).

Nedostatek paralelní aplikace vidím částečně v algoritmu na ukončení práce, kdy se na konci obvykle stane, že je na síti velké množství tokenů. Nepředstavuje to však funkční problém.

Jako **návrh na zlepšení** vidím implementaci algoritmu pro nalezení libovolné kostry grafu ještě před začátkem hledání kostry s nejnižším stupněm. To by mohlo algoritmus znatelně urychlit, protože bychom mohli začít prořezávat strom kombinací hned od začátku. Bylo empiricky ověřeno, že nejvíce času zabere nalezení první kostry (s dostatečně nízkým stupněm) a dále už algoritmus díky prořezávání pracuje mnohem rychleji. Implementací by však byl do programu vnesen další prvek „náhody“, kdy by záleželo na tom, jakou kostru (s jakým stupněm) by se podařilo na začátku nalézt.

3.6.4 Limit paralelismu

Problém nelze paralelizovat do nekonečna a při určitém počtu procesů (v závislosti na velikosti grafu) začnou náklady na komunikaci převažovat urychlení paralelním výpočtem.

3.6.5 Sémantika příkazové řádky pro spouštění programu

V programu nebylo implementováno načítání konstant pro škálování algoritmu z příkazové řádky. Bylo ověřeno, že ty námi zvolené jsou optimální a není třeba je měnit. Spuštění programu vypadá takto:
./program graf.graph
Kde *program* je zkompilovaný program (může jít o sekvenční i paralelní verzi) a *graf.graph* je soubor s maticí typu 1.2.

4 Závěr

Tato sekce navazuje na sekci 3.6.3, kde je možné nalézt *Posouzení paralelního algoritmu.*

Semestrální práci hodnotíme jako zdařilou, ačkoliv v případě reálného využití na velkém množství procesorů by pravděpodobně bylo třeba **provést další optimalizace** např. algoritmu na dělení práce a žádosti o práci, kvůli kterým u některých grafů nedochází k superlineárnímu zrychlení.

Za velmi zdařilou považujeme implementaci **prořezávání** stromu kombinací a detekci **nalezení dolní meze**, což jsou věci, které výpočet velmi urychlují. Stejně tak považujeme za zdařilou zvolenou **formu uchování stavů** a jejich předávání, kdy lze pomocí jediného vektoru poslat práci o velikosti celého podstromu.

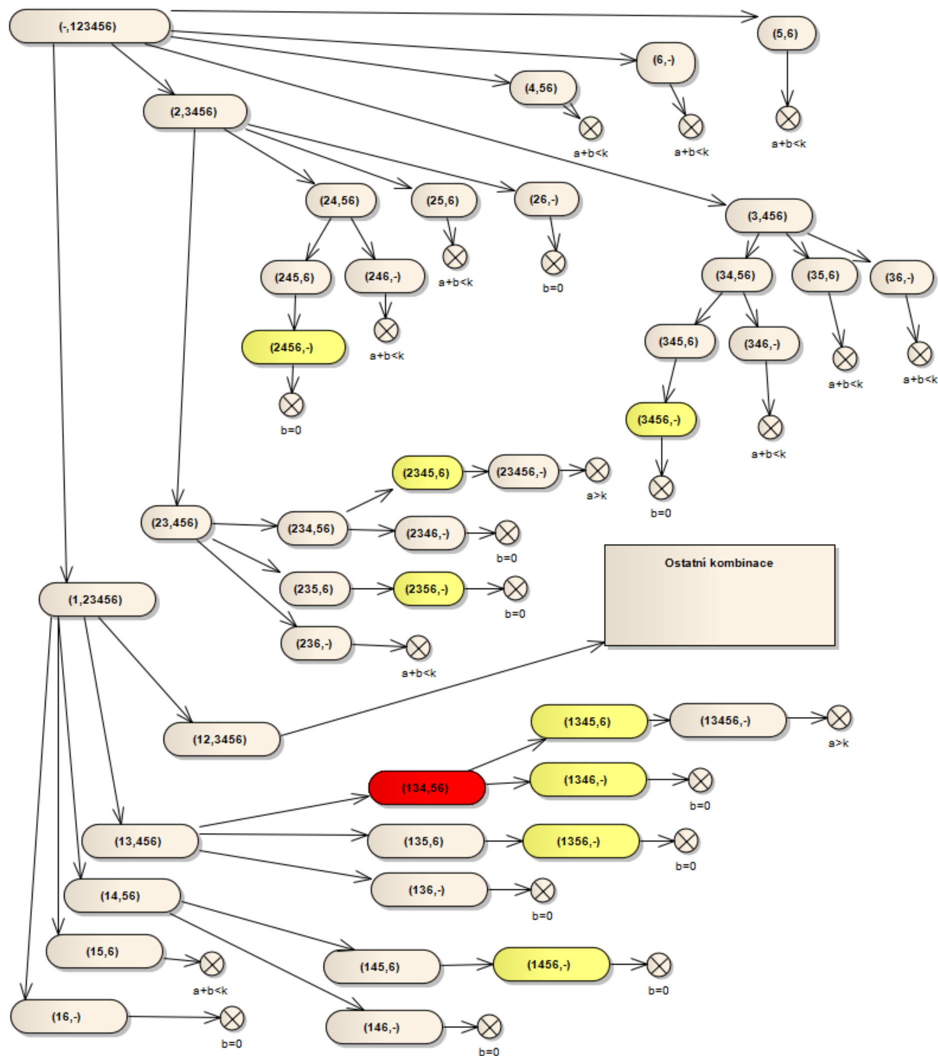
Osobní přínos je v podobě pokročilejšího zvládnutí C++ a vyzkoušení si knihovny MPI. Příjemně nás překvapilo, že díky MPI není samotná paralelizace výpočtu až tak velký problém, ačkoliv tato semestrální práce byla velmi časově náročná.

5 Literatura

1. Sekce *Laboratoře* na stránkách předmětu - https://edux.fit.cvut.cz/courses/MI-PAR/labs/start

2. Programovaní pod MPI - https://users.fit.cvut.cz/ soch/mi-par/

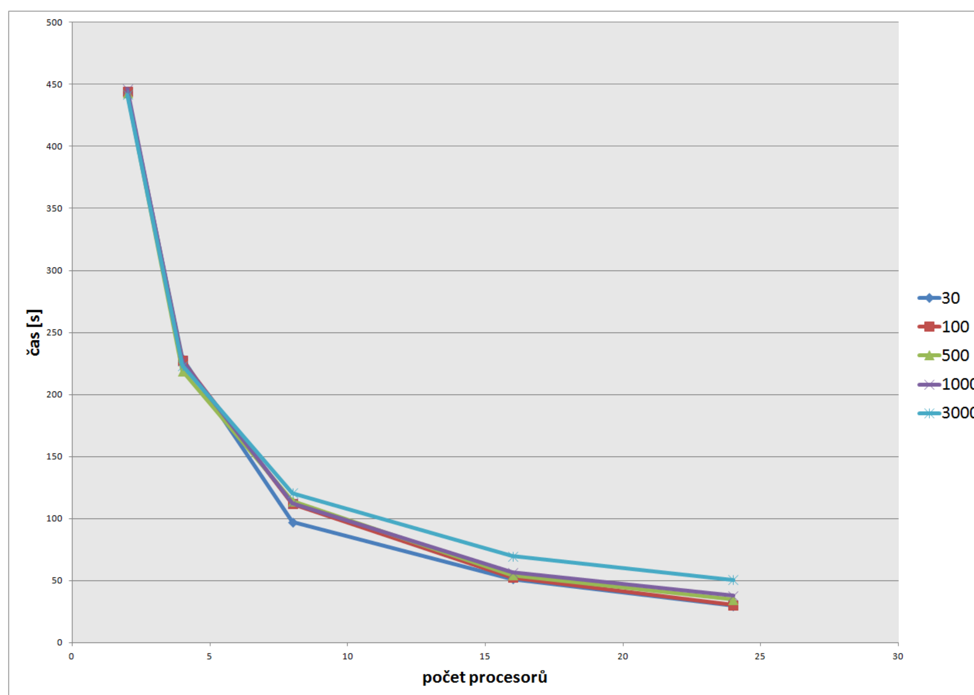
3. Výpočetní klastr Star - http://star.fit.cvut.cz/dokuwiki/



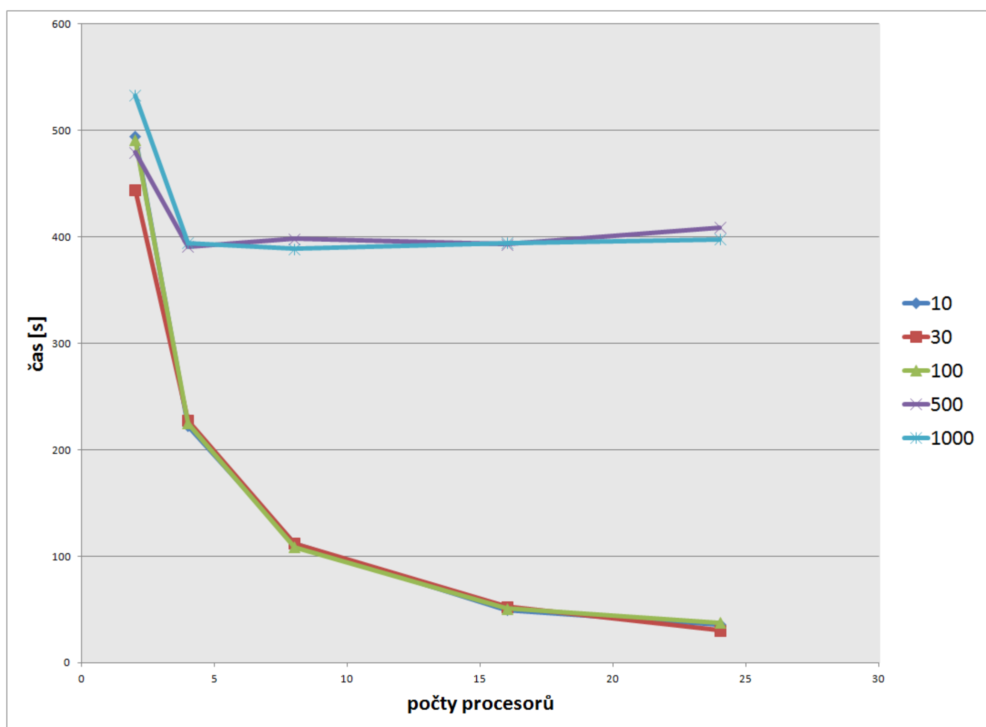
Obrázek 2: Grafické znázornění způsobu generování kombinací v programu.

id	počet uzlů	počet hran	nejnižší stupeň kostry	počet kombinací délky v-1	čas
_4.9min.sh	14	25	4	5 200 300	267,7s
_7.7min.sh	13	26	4	9 657 700	468,72s
_14.5min.sh	13	27	4	17 383 860	842,13s
_17.4min.sh	14	27	4	20 058 300	1049,89s
graf14k4_2.sh	14	28	**2** (dolní mez)	37 442 160	73,11s
graf14k4_1.sh	14	28	**2** (dolní mez)	37 442 160	141,96s
graf15k4_2.sh	15	30	**2** (dolní mez)	145 422 675	1219,63s

Tabulka 1: Měření doby běhu sekvenčního algoritmu



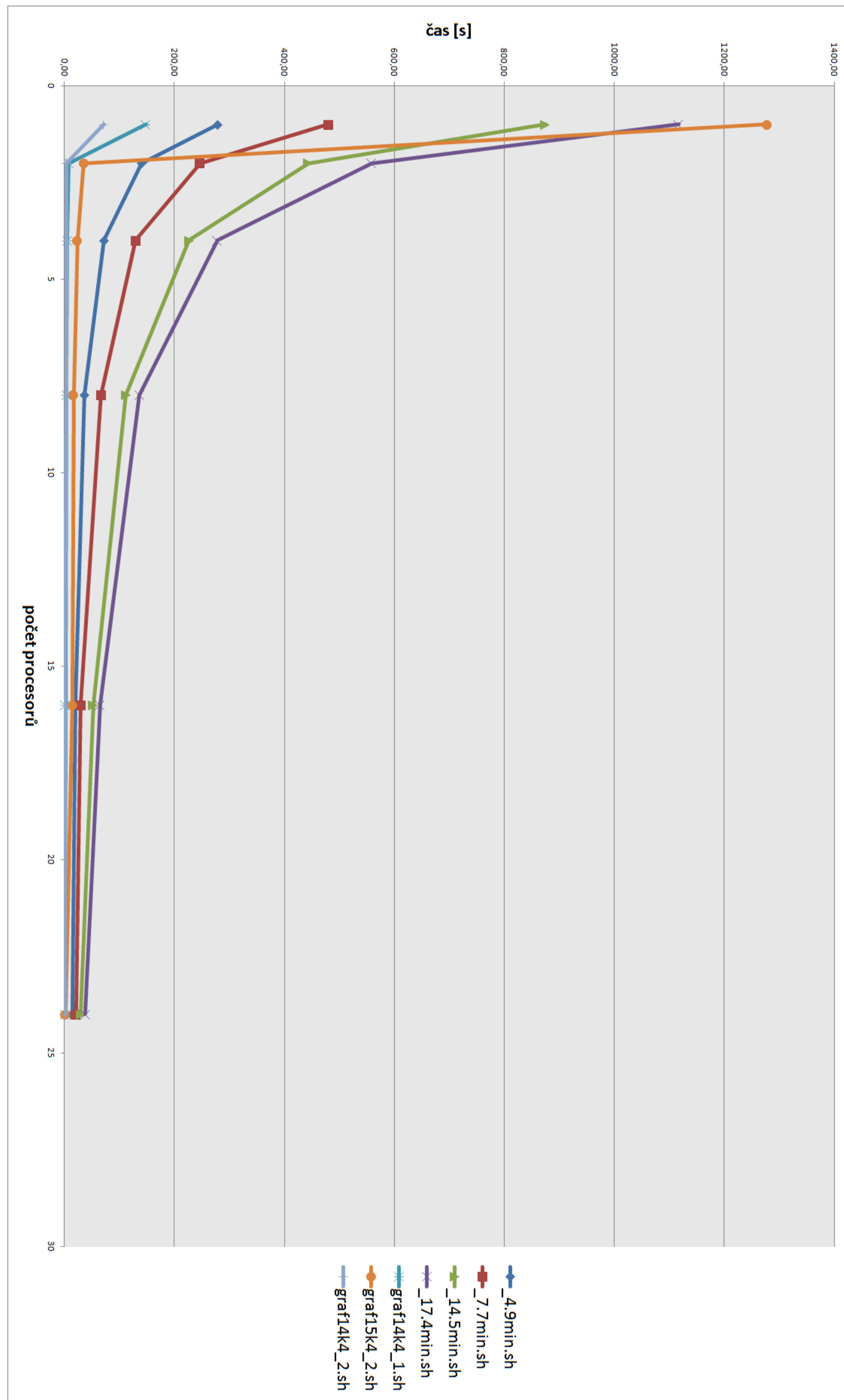
Obrázek 3: Rychlost zpracování grafu v závislosti na počtu procesorů pro různé konstanty CHECK_MSG_AMOUNT.



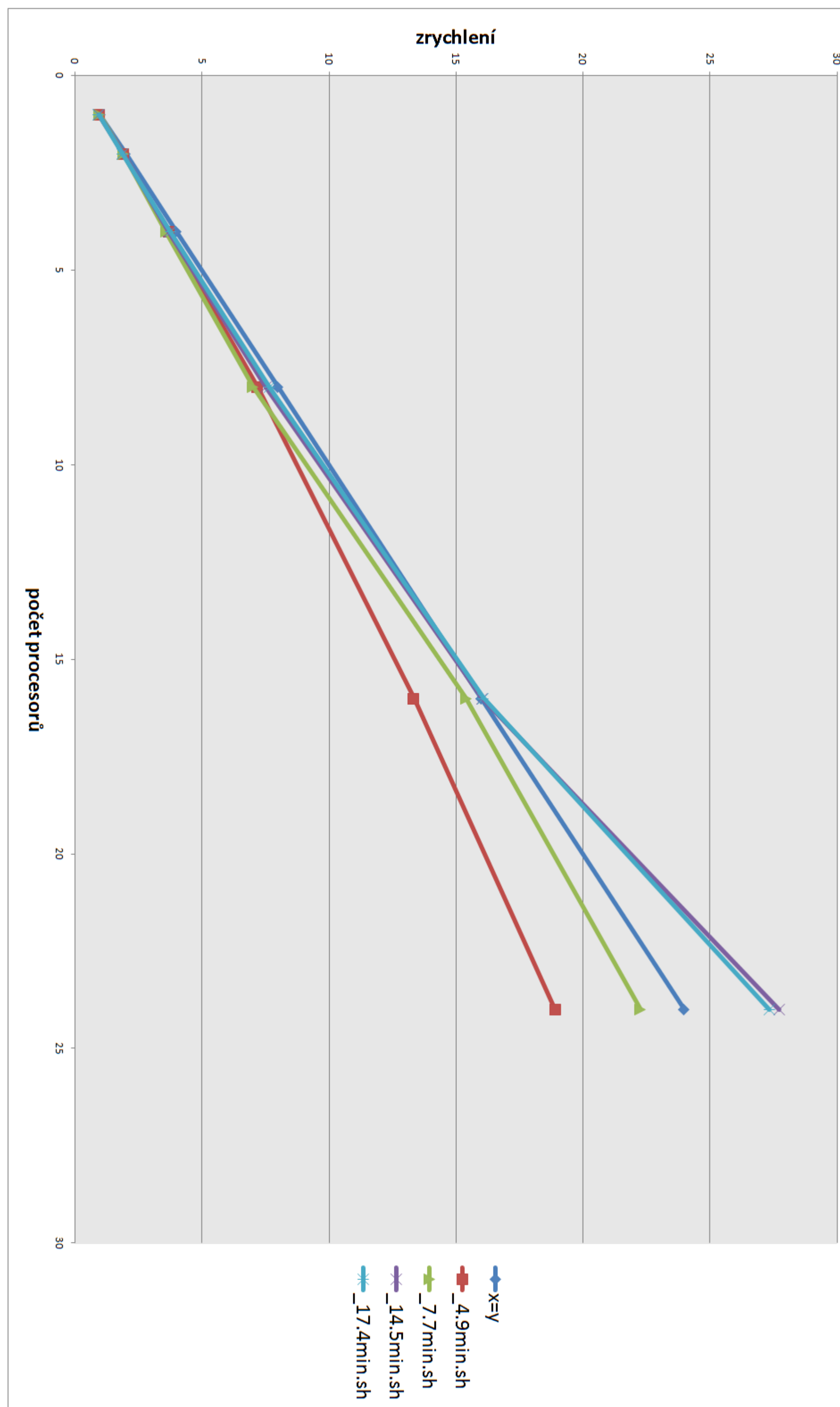
Obrázek 4: Rychlost zpracování grafu v závislosti na počtu procesorů pro různé konstanty WORK_SPLIT_CONSTANT.

počet procesů / id	1	2	4	8	16	24
_4.9min.sh	279,99s	140,83s	72,49s	37,23s	20,06s	14,146s
_7.7min.sh	480,242s	247,005s	130,26s	66,825s	30,424s	21,048s
_14.5min.sh	874,58s	444,19s	227,46s	111,917s	52,528s	30,362s
_17.4min.sh	1116,97s	558,849s	278,264s	136,958s	65,253s	38,397s
graf14k4_2.sh	71,927s	4,429s	4,508s	3,814s	1,729s	0,973s
graf14k4_1.sh	148,26s	8,261s	5,227s	3,998s	2,088s	1,398s
graf15k4_2.sh	1277,53s	35,685s	24,084s	17,367s	15,169s	2,416s

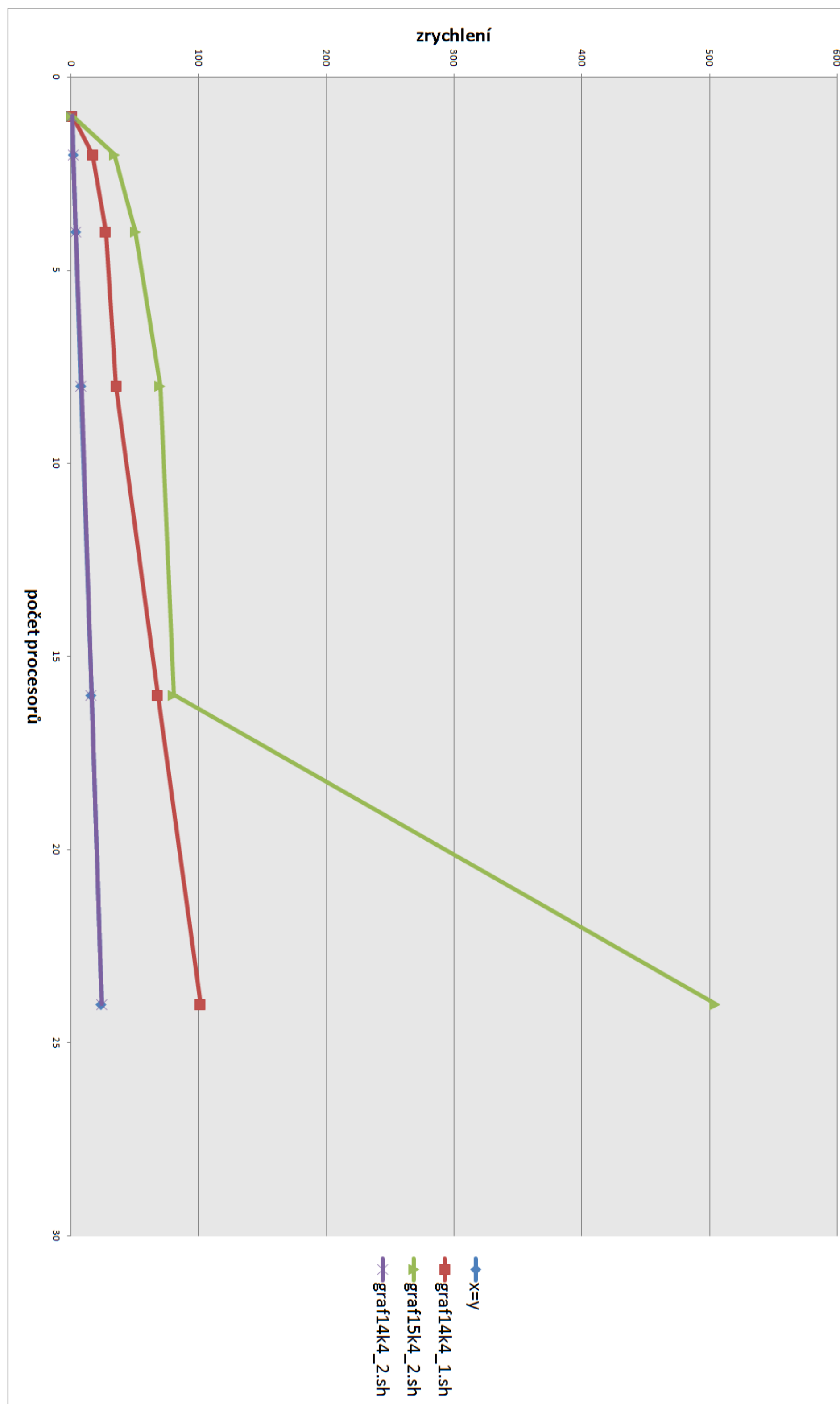
Tabulka 2: Měření doby běhu paralelního algoritmu



Obrázek 5: Rychlosti zpracování grafů v závislosti na počtu procesorů.



Obrázek 6: Zrychlení ($S(n,p) = SU(n)/T(n,p)$) výpočtu v závislosti na počtu procesorů pro grafy bez dolní meze.



Obrázek 7: Zrychlení ($S(n,p) = SU(n)/T(n,p)$) výpočtu v závislosti na počtu procesorů pro grafy s dolní mezí.