Paralelní algoritmy pro bisekci grafu

Libor Smékal

23. května 1999

Obsah

Kapitola 1

Úvod

V současné době jsme svědky prudkého rozvoje výpočetní techniky. S nástupem internetu rychle roste počet lidí, kteří počítače používají, a tak není divu, že se všichni výrobci výpočetní techniky snaží na trhu se svými výrobky co nejvíce prosadit. Důsledkem toho jsou nejen neustále nové modely (např. firmy Intel, AMD, Cyrix přicházejí téměř každý půlrok s vylepšeným, příp. úplně novým procesorem), ale také pokles cen. Víceprocesorové sestavy, které byly dříve dostupné jen největším firmám a vědeckým ústavům, nejsou dnes i mezi běžnými uživateli žádnou výjimkou.

Při srovnání počítače s jedním a se dvěma procesory se nabízí představa, že na dvouprocesorovém počítači běží všechny programy dvakrát rychleji. Tento omyl vychází z předpokladu, že při řešení úlohy podle daného algoritmu bude možno vždy plně využít všech procesorů, navíc se zanedbává nutná meziprocesorová komunikace. Většina používaných algoritmů však není určena pro paralelní zpracování a proto dnes roste význam paralelních algoritmů, které dokáží výkon víceprocesorového systému využít.

Tato práce se zabývá paralelními algoritmy pro bisekci grafu. V praxi má tato úloha široké použití. Jedná se např. o návrh obvodů VLSI, kde je jednou z nejpoužívanějších metod při řešení rozmísťovacího problému. Dále se využívá při přidělování procesorů, kdy je třeba jednotlivé úlohy rozmístit na procesory tak, aby se minimalizovala meziprocesorová komunikace. Další velice časté použití je při řešení problému segmentování paměti.

Bisekce grafu chápaná jako optimalizační problém je přesně tím případem, který byl zmíněný ve druhém odstavci. Jsou sice známé algoritmy, které na sériovém počítači tuto úlohu řeší s dobrými výsledky, přesto je však není možné vzhledem ke své sekvenční podstatě uspokojivě paralelizovat. Jedná se především o algoritmus simulovaného žíhání a o Kernighan-Lin algoritmus.

V roce 1991 byla v [2] publikována nová, tzv. mob[†] heuristika, která je vhodná pro paralelní zpracování. Tato heuristika, analýza jejího chování a vlastností, návrh dalších modifikací včetně jejich implementace jsou hlavními tématy této diplomové práce. Výsledkem testování na síti pracovních stanic a na počítači IBM SP-2 je určení vhodnosti dané heuristiky pro konkrétní typ grafu, chování heuristiky v závislosti na hodnotách nejdůležitějších vstupních parametrů a vyhodnocení zrychlení a efektivnosti. Heuristiky jsou porovnávány především podle dosaženého času, bisekční šířky a schopnosti odhalit v grafu "úzké místo".

U čtenáře této práce se předpokládá, že je seznámen se základy teorie grafů a se základy teorie paralelních systému.

[†]Protože autorovi není znám vhodný překlad tohoto anglického slova, nebude v průběhu celého textu překládáno.

Kapitola 2

Minimální bisekce grafu

2.1 Definice základních pojmů a značení

Mějme neorientovaný **graf G** $= (V, E)$, kde V je množina uzlů a E množina hran. Množina uzlů resp. hran grafu G bude označována $V(G)$ resp. $E(G)$. Hrana incidující s uzly u a v, $u, v \in V(G)$ bude zapisována jako $\langle u, v \rangle$.

Stupeň uzlu $u \in V(G)$, $deg_G(u)$ je počet hran incidujících s tímto uzlem. Množinu stupňů všech uzlů grafu G, $\{deg_G(u), u \in V(G)\}$ udává $deg(G)$.

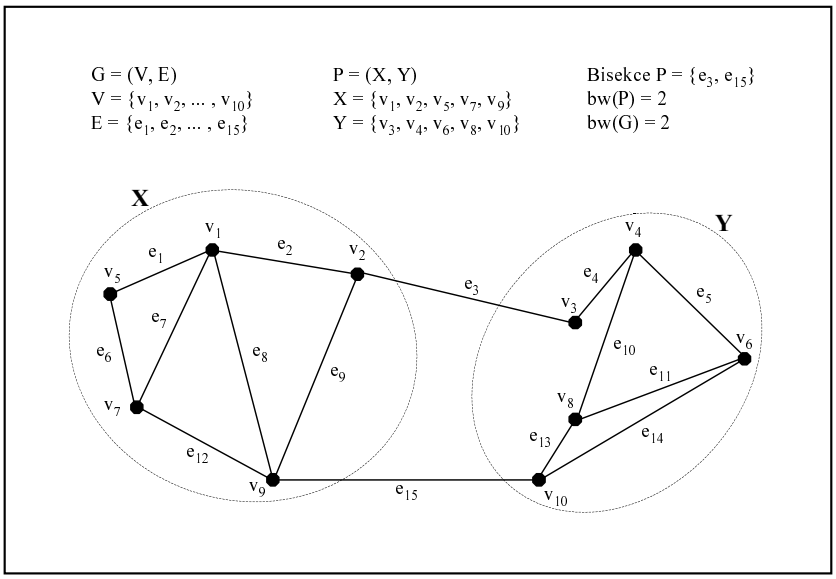
Hodnota $\Delta(G) = max(deg(G))$ je **(maximálním) stupněm** grafu G, hodnota $\delta(G) = min(deg(G))$ je **stupněm minimálním**. Jestliže $\Delta(G) = \delta(G) = k$, nazýváme G k**-regulární graf**.

Rozdělením P $= (X, Y)$ grafu G rozumíme rozdělení množiny uzlů na dvě stejně velké poloviny X a Y (příp. lišící se svou velikostí o 1, pokud je $|V|$ lichá).

Bisekcí rozdělení P je množina hran spojující X a Y.

Bisekční šířkou bs(P) **rozdělení** P rozumíme počet hran bisekce rozdělení P.

Bisekční šířka bs(G) **grafu** G je minimální $bs(P)$ přes všechna možná rozdělení P grafu G.



Obrázek 2.1: Bisekce grafu

2.2 Heuristické algoritmy

Nalezení minimální bisekce grafu patří mezi často řešené problémy teorie grafů. Protože jde o NP-úplný problém [1], nepoužívají se v praxi algoritmy, které hledají nejlepší možné řešení. Úlohy se řeší pomocí různých heuristických metod navrhovaných tak, aby bylo v "rozumném" (polynomiálním) čase co nejčastěji nalezeno vyhovující řešení, které je ale jen málokdy nejlepší možné. Heuristiky se obvykle nesrovnávají s algoritmy pro hledání optimálního řešení, ale porovnávají se mezi sebou s cílem nalézt nejlepší z nich. Kritériem kvality takových algoritmů jsou především praktické zkušenosti při jejich používání. Při výpočtu bisekční šířky dosahují v praxi nejlepších výsledků heuristiky založené na simulovaném žíhaní a Kernighan-Lin heuristika. Jejich základní principy budou proto objasněny v následujících odstavcích.

2.3 Kernighan-Lin heuristika

KL-okolí KL-N(P_0) rozdělení $P_0 = (X_0, Y_0)$ je množina rozdělení $\{P_i = (X_i, Y_i)\}$, kde X_i a Y_i jsou sjednocení dvou disjunktních množin: $X_i = F_i^X \cup R_i^X$, $Y = F_i^Y \cup R_i^Y$. Nechť $a \in R_{i-1}^X$ a $b \in R_{i-1}^Y$ jsou uzly, jejichž prohozením bychom dosáhli největšího zmenšení (příp. nejmenšího zvětšení, jestliže zmenšení nelze dosáhnout) bisekční šířky $bs(P_{i-1})$. Rekurzivní předpis pro vytvoření P_i z rozdělení P_{i-1} je následující:

$$
\begin{aligned}
F_0^X &= \emptyset & F_0^Y &= \emptyset \\
R_0^X &= X_0 & R_0^Y &= Y_0 \\
F_i^X &= F_{i-1}^X \cup \{b\} & F_i^Y &= F_{i-1}^Y \cup \{a\} \\
R_i^X &= R_{i-1}^X - \{a\} & R_i^Y &= R_{i-1}^Y - \{b\}
\end{aligned}
$$

Počet rozdělení v okolí KL-N(P) je $|V|/2$. Kernighan-Lin heuristika je pro dané počáteční rozdělení P popsána následující procedurou:

```
procedure KernighanLin(P);
begin
    Q = nejlepšíRozděleníKLN(P);
    while bs(Q) < bs(P) do begin
        P = Q;
        Q = nejlepšíRozděleníKLN(P);
    end;
    KernighanLin = P;
end;
```

Funkce *nejlepšíRozděleníKLN(P)* vrací rozdělení s nejmenší bisekční šířkou z KL-N(P).

2.4 Simulované žíhání

Simulované žíhání není jen jednou z metod pro nalezení minimální bisekce grafu, ale jedná se o často používanou obecnou metodou pro řešení složitých optimalizačních úloh. V chování této heuristiky hraje důležitou roli ochlazovací plán.

Ochlazovací plán CS délky L je posloupnost $[CS_1, CS_2, \ldots, CS_L]$ pravděpodobnostních funkcí, kde $CS_t : Z \to (0, 1)$. Mnoho implementací simulovaného žíhání používá ochlazovací plán definovaný $CS_t(x) = k_1 e^{-x/k_2 T(t)}$, kde $T(t)$ je "teplota" v kroku t, k_1 a k_2 jsou konstanty.

Předpokládejme, že algoritmus proběhne v $q(n)$ krocích, $q(n)$ je polynomiální v závislosti na n. Dále předpokládejme, že uzly v množinách X, Y jsou očíslované $1, 2, \ldots, n/2$. V poli RX resp. RY je uloženo $q(n)$ náhodně vybraných hodnot - indexů uzlů z množin X, resp. Y. Pole RV obsahuje $q(n)$ náhodných hodnot z intervalu $(0, 1)$. Všechny uvedené náhodné veličiny generujeme s rovnoměrným rozdělením, to znamená, že pravděpodobnost vygenerování hodnoty menší nebo rovno p z intervalu $(0, 1)$ je rovna p. Algoritmus simulovaného žíhání je pro dané počáteční rozdělení P popsán následující procedurou:

```
procedure SimulovanéŽíhání(P);
begin
    t = 1;
    while t ≤ q(n) do begin
        Q = swap(P, RX[t], RY[t]);
        delta = bs(Q) - bs(P);
        if (delta < 0) or (RV[t] < CS[t](delta)) then P = Q;
        t = t + 1;
    end;
    SimulovanéŽíhání = P;
end;
```

Funkce *swap* prohodí uzel s indexem $RX[t]$ z množiny X s uzlem s indexem $RY[t]$ z množiny Y. Ukončení algoritmu nemusí být vázáno vždy jen na předem daný počet iteračních kroků. Další možností je např. ukončení v případě, že v m po sobě jdoucích krocích nedošlo ke zlepšení bisekční šířky.

Kapitola 3

Mob heuristika

Vlastnosti Kernighan-Lin heuristiky a simulovaného žíhání byly podrobně rozebrány v článku [2]. Výsledkem těchto analýz bylo zjištění, že Kernighan-Lin heuristika patří do třídy P-úplných a simulované žíhání do třídy P-těžkých problémů. To znamená, že obě heuristiky patří do kategorie špatně paralelizovatelných algoritmů [3]. V článku [2] byla také navržena nová, tzv. mob heuristika. Ta obsahuje některé rysy předchozích heuristik, odlišuje se ale dobrou paralelizací.

3.1 Definice mob heuristiky

Mějme $n = |V|, m < n/2$ a $P_0 = (X_0, Y_0)$. Mob-N(P_0, m) je množina všech rozdělení získaných prohozením m uzlů z množiny X_0 s m uzly množiny Y_0. Dvě množiny uzlů velikosti m vyměňované mezi X_0 a Y_0 nazýváme *mob*. Mob je optimální, jestliže záměna způsobí největší zmenšení bisekce.

Pro dané rozdělení $P = (X, Y)$, dvě náhodně zvolená reálná čísla r_x, r_y z intervalu (0, 1) a velikost mobu m je algoritmus $A(P, m, r_x, r_y)$ pro nalezení nejlepšího rozdělení z mob-N(P, m) následující:
Pro každý uzel v nalezneme zisk(v, P), který je roven změně bisekční šířky, pokud bychom samotný uzel v přehodili na druhou stranu. Zisk(v, P) je kladný, pokud by došlo ke zmenšení bisekční šířky. Zavedeme $\mathcal{X}(g) = \{v \in X | \text{zisk}(v, P) \geq g\}$ Zvolíme g_x tak, že $|\mathcal{X}(g_x + 1)| < m \leq |\mathcal{X}(g_x)|$. Nechť $m_x = |\mathcal{X}(g_x)|$. Dále očíslujeme uzly v $\mathcal{X}(g_x)$ čísly 0 až $m_x - 1$. Do mobu velikosti m je vybrán uzel i, jestliže platí, že hodnota $(i + r_x \cdot m_x)$ *modulo* m_x je menší než m. Analogicky postupujeme i pro množinu Y.

Mob plán M délky L je sekvence $[m_0, \ldots, m_{L-1}]$ taková, že

$n/2 > m_0 > \ldots > m_{L-1}$.

Podrobněji bude mob plán rozebrán v následující sekci.

Mob heuristika začíná s libovolným počátečním rozdělením P_0. Dále je použito algoritmu $A(P_0, m_0, r_{x0}, r_{y0})$ k nalezení nejlepšího rozdělení P_1 z mob-N(P_0, m_0). Jestliže je $bs(P_1) \leq bs(P_0)$, hledání opakujeme v mob-N(P_1, m_0), jinak je následující rozdělení hledáno v mob-N(P_1, m_1) za použití algoritmu $A(P_1, m_1, r_{x1}, r_{y1})$. Hledání nového rozdělení je jeden iterační krok. Mob heuristika končí vybráním poslední hodnoty m_{L-1} z mob plánu nebo po pevně stanoveném počtu iteračních kroků.

3.2 Mob plán

Celý algoritmus je založen na prohazování dvou množin uzlů - mobu mezi oběma polovinami rozděleného grafu. Velikost mobu v průběhu algoritmu je určena právě mob plánem. Ten ovlivňuje chování velmi významně a proto v závislosti na použitém mob plánu hovoříme o následujících heuristikách:

Lineární heuristika - používá mob plán, pro jehož prvky platí:

$$m_i = \lfloor (1 - \frac{i}{L}) m_0 \rfloor \qquad i = 0, 1, \ldots, L-1$$

Exponenciální heuristika - používá mob plán, pro jehož prvky platí:

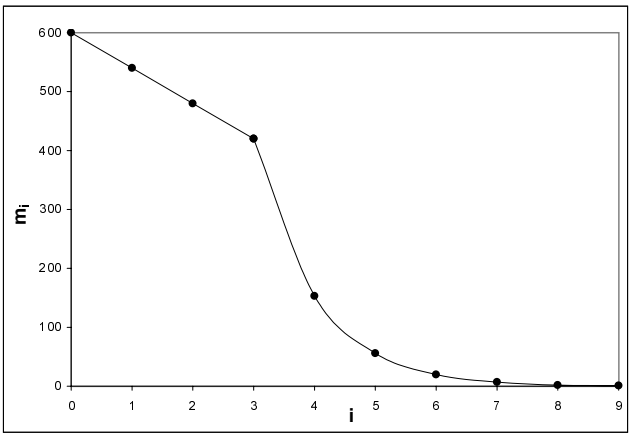
$$m_i = \lfloor m_0^{(1-\frac{i}{L-1})} \rfloor \qquad i = 0, 1, \ldots, L-1$$

Lineární i exponenciální mob plán je jednoznačně určen parametry L, m_0. Kombinovaný lin-exp mob plán je definován navíc parametrem c, který určuje počet prvků lineární části mob plánu.

Kombinovaná heuristika - používá mob plán, pro jehož prvky platí:

$$m_i = \lfloor (1 - \frac{i}{L}) m_0 \rfloor \qquad i = 0, 1, \ldots, c-1$$

$$m_i = \lfloor q^{\frac{L-i-1}{L-c}} \rfloor \qquad i = c, c+1, \ldots, L-1$$
$$q = (1 - \frac{c-1}{L}) m_0$$



Obrázek 3.1: Kombinovaný mob plán, $L = 10$, m_0=600, c=4

Kapitola 4

Paralelní algoritmus

4.1 Definice základních pojmů

Procesorem bude v následujícím textu označován *virtuální procesor*, který je v implementaci reprezentován jedním procesem. Několik *virtuálních procesorů* tedy může běžet na jednom *fyzickém procesoru*. Počet procesorů bude označován p, procesory budou číslovány $0, 1, \ldots, p-1$.

Všesměrové vysílání *(broadcast)* je způsob komunikace, kdy jeden procesor posílá stejnou zprávu všem ostatním procesorům.

Nechť $\oplus$ je zadaná binární asociativní operace v doméně hodnot D a mějme vstupní pole $A = A_1, A_2, \ldots, A_n$, $n > 1$, hodnot z D.

Paralelní redukce pole A je hodnota $S = A_1 \oplus \cdots \oplus A_n$.

Prefixový součet pole A je pole B s prvky $B_1, B_2, \ldots, B_n$, kde $B_i = A_1 \oplus \ldots \oplus A_i$.

Postfixový součet pole A je pole B s prvky $B_1, B_2, \ldots, B_n$, kde $B_i = A_i \oplus \cdots \oplus A_n$.

Paralelní časová složitost všech těchto tří operací na EREW PRAM a většině distribuovaných paralelních počítačů s p procesory je $T(n,p) = O(n/p + \log p)$. Proto pro $p = \Omega(n/\log n)$ je $T(n,p) = O(\log n)$.

Podrobněji jsou uvedené termíny rozebrány v [3].

4.2 Volba vhodné topologie

Jak plyne z definice heuristiky, každý uzel grafu G musí mít informaci o tom, na kterou stranu patří jeho sousedé. Při náhodném rozdělení uzlů na jednotlivé procesory (a zvláště pak při větším stupni grafu k) je velmi pravděpodobné, že se sousedé daného uzlu budou nacházet na větším počtu procesorů. Běžně je počet procesorů p mnohem menší než počet uzlů n a tak je na jednom procesoru větší počet uzlů, které musí komunikovat se všemi svými sousedy. Komunikace mezi procesory se tedy

blíží komunikaci "každého s každým" a proto se jeví zvolená **topologie úplného grafu** jako velmi výhodná. Dalším důvodem pro její volbu byla snadná implementace v prostředí PVM.

4.3 Mapování uzlů grafu na procesory

Při paralelizaci úlohy je velmi důležité ji správně rozdělit na části, které jsou pak zpracovávány jednotlivými procesory. Vhodným rozdělením - mapováním můžeme docílit výrazného zlepšení algoritmu.

Cílem tohoto algoritmu je nalézt minimální bisekci obecného grafu, který není nijak přesněji specifikován. Při mapování uzlů na procesory tudíž nebylo možno vyjít ze znalosti grafu a proto bylo navrženo mapování minimalizující alespoň meziprocesorovou komunikaci. Ta obsahuje především zprávy s informacemi, v které polovině se nachází uzly daného procesoru. Jde tedy o větší počty 1-bitových hodnot. Pořadí bitů v paketu odpovídá lokálnímu očíslování uzlů na každém procesoru a tak je možno při komunikaci přenášet jen vlastní 1-bitové hodnoty a mapováním je jednoznačně dáno, o jakém bodu daný bit informuje. Mapování s těmito vlastnostmi je mnoho, s ohledem na jednoduchost implementace bylo vybráno **cyklické mapování**.

Je-li p počet procesorů, pak i-tý uzel na j-tém procesoru je uzel číslo

$$f_1(i,j) = i \cdot p + j$$

Naopak, je-li x číslo uzlu grafu, pak tento uzel je uložen v procesoru

$$f_2(x) = x \ modulo \ p$$

jako lokální uzel číslo

$$f_3(x) = \lfloor x/p \rfloor$$

4.4 Popis algoritmu

Paralelní algoritmus je rozložen do jednoho hlavního a p výpočetních procesů. Nejdříve je aktivní hlavní proces, v němž dojde ke zpracování vstupních parametrů, vytvoření komunikační a výpočetní topologie a rozeslání dat výpočetním procesům. Ty pak provádí vlastní paralelní výpočet. Hlavní proces čeká na ukončení výpočtu a poté přijímá výsledky, které požadovaným způsobem zpracuje.

Hlavní proces :

1. **begin**
2. načteníParametrů;
3. vygenerováníGrafu;
4. rozděleníDatAZasláníVýpProcesům;

Výpočetní proces (prováděno na p procesorech paralelně) :

5. **begin**
6. příjemDatZHlProcesu;
7. vygenerováníPočátečníhoRozdělení;
8. bs = bisekčníŠířka;
9. nr = aktuálníRozdělení;
10. m = mobPlán.dalšíHodnota;
11. **while** $m > 0$ **do begin**
12. výpočetZiskuKaždéhoUzlu;
13. stanoveníMnožinyKandidátůNaProhození;
14. vybráníUzlůKProhození;
15. prohozeníUzlů;
16. b = bisekčníŠířka;
17. **if** $b < bs$ **then begin**
18. $bs = b$;
19. nr = aktuálníRozdělení;
20. **end else** m = mobPlán.dalšíHodnota;
21. **end;**
22. zaslánívýsledkůHlProcesu(bs, nr);
23. **end.**

Hlavní proces :

24. příjemVýsledkůOdVýpProcesů;
25. zpracováníVýsledků;
26. **end.**

4.5 Detailnější rozbor některých kroků

4.5.1 Generování počátečního rozdělení

Vygenerování počátečního rozdělení (řádek 7) je možné provést několika způsoby. V případě náhodného generování by tato operace musela být v hlavním procesu a všechny procesory informovány o všech uzlech. Druhý způsob, který byl zvolen i pro tento algoritmus, definuje funkci $f_4(x)$, která pro zadaný uzel x vrací jeho pozici (stranu 0 nebo 1). Funkcí, které rozdělí množinu uzlů na dvě stejně velké části, existuje mnoho, v našem případě byla použita:

$f_4(x) = \lfloor \frac{2 \cdot x}{n} \rfloor$

Generování poč. rozdělení tímto způsobem má tu výhodu, že není zapotřebí žádná meziprocesorová komunikace.

4.5.2 Výpočet bisekční šířky

Při výpočtu bisekční šířky (řádky 8, 16) se nejdříve na každém procesoru zjistí pro každý jeho uzel, kolik má sousedů v opačné polovině grafu, a tyto hodnoty se sečtou. Výsledky jsou pak pomocí paralelní redukce sečteny. Kořenový procesor redukce (procesor 0) výsledný součet vydělí 2 (protože každá hrana byla započítána dvakrát) a výsledek zašle pomocí všesměrového vysílání ostatním procesorům.

4.5.3 Výpočet zisku každého uzlu

Zisk uzlu je hodnota, o kterou by se zmenšila bisekční šířka, pokud bychom tento uzel přesunuli do opačné poloviny grafu. Jeho výpočet (řádek 12) se provádí tak, že se zjistí počet sousedů uzlu v opačné a ve stejné polovině grafu, ve které daný uzel leží, a tyto hodnoty se odečtou. Výpočet je proveden lokálně pro všechny uzly daného procesoru.

4.5.4 Výběr uzlů k prohození pomocí globální heuristiky

V závislosti na tom, jak stanovíme množinu kandidátů na prohození (řádek 13) a které potom prohodíme (řádek 14), dostáváme další dvě varianty, které se liší především v nárocích na komunikaci a také kvalitou výsledné bisekce. Lepšími výsledky, ale také větší meziprocesorovou komunikací se vyznačuje globální heuristika.

Jde o variantu, kdy je přesně dodržena definice mob heuristiky. Globální je nazývána proto, že stanovení množin kandidátů a výběr uzlů k prohození probíhá globálně, tzn. podle určitého kritéria ze všech uzlů bez ohledu na to, na kterém procesoru jsou umístěny. V dalším textu bude vždy popisován jen postup pro množinu X

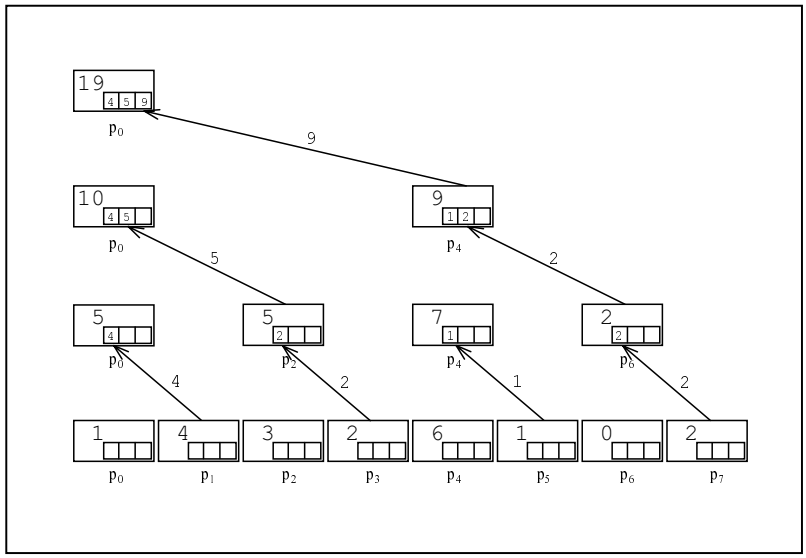
rozdělení $P = (X, Y)$, pro množinu Y je postup analogický.

Nejdříve je třeba určit g_x, protože tato hodnota jednoznačně určuje množinu kandidátů $\mathcal{X}(g_x)$. Na každém procesoru je vytvořeno pole S o velikosti $2k + 1$ s prvky $S_{-k}, S_{-k+1}, \ldots, S_k$ (k je stupeň grafu). Výsledek výpočtu zisku každého uzlu je použit jako index do tohoto pole, které je na pozici dané indexem zvýšeno o 1. Nyní je pro pole S v novém poli T vypočítán postfixový součet, tudíž na pozici s indexem i je v tomto poli počet uzlů, které mají zisk $\geq i$. Následuje součet těchto polí pomocí paralelní redukce (součtem dvou polí A a B se rozumí taková operace, jejímž výsledkem je pole C o stejné velikosti s prvky $C_i = A_i + B_i$).

Z výsledného pole, označme jej $TSUM$, kořenový procesor paralelní redukce určí hledanou hodnotu g_x. Ta je rovna největšímu indexu i, pro nějž platí $TSUM_i \geq m$. Následně jsou hodnoty g_x, $TSUM_{g_x}$ a náhodně vygenerované celé číslo r_x z intervalu $\langle 0, TSUM_{g_x} - 1\rangle$ zaslány všesměrovým vysíláním všem ostatním procesorům. Všem je tedy známo, že $\mathcal{X}(g_x)$ tvoří uzly, které mají zisk $\geq g_x$ a že počet uzlů v $\mathcal{X}(g_x)$ je roven $TSUM_{g_x}$. Hodnota r_x slouží k náhodnému výběru m uzlů z množiny $\mathcal{X}(g_x)$.

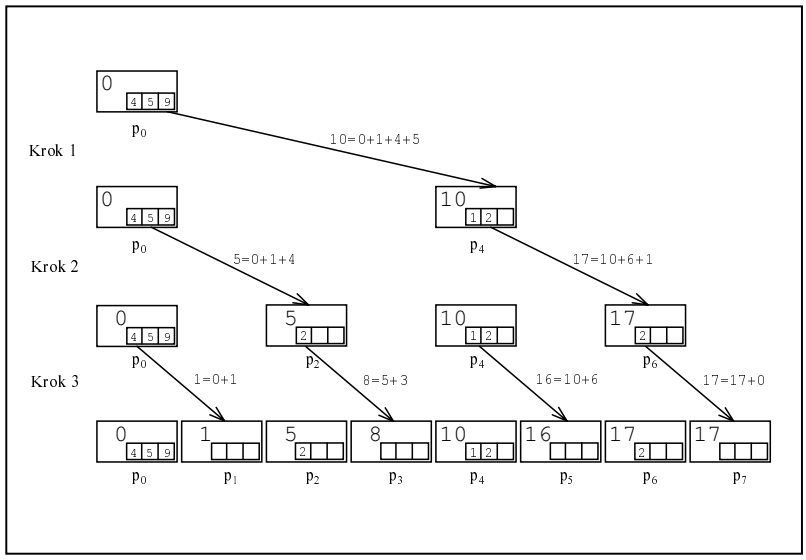
Další operací je globální očíslování uzlů v $\mathcal{X}(g_x)$. Pro každý procesor i je třeba určit hodnotu $first^i$, která bude globálním indexem v $\mathcal{X}(g_x)$ jeho prvního uzlu z této množiny. Standardním řešením tohoto problému by mohl být výpočet prefixového součtu způsobem uvedeným v [3]. Po ukončení tohoto výpočtu by ale potřebná hodnota $first^i$ byla uložena na procesoru $i - 1$ a byla by tedy nutná ještě dodatečná komunikace. Proto byl navržen způsob, jak vypočítat paralelní prefix tak, aby hodnota $first^i$ byla po ukončení výpočtu uložená na procesoru i. Toto vylepšení je vyváženo větší paměťovou složitostí $O(\log p)$. Jde o nejsložitější operaci algoritmu a proto bude objasnění provedeno na příkladu:

Předpokládejme, že $p = 8$, $TSUM_{g_x} = 19$. Hodnota T_{g_x} na procesoru i bude dále označována $T^i_{g_x}$. V tomto příkladě jsou hodnoty $T^0_{g_x}, T^1_{g_x}, \ldots, T^7_{g_x}$ postupně 1, 4, 3, 2, 6, 1, 0, 2. První fází je běžná paralelní redukce jen s tím rozdílem, že procesory si v pomocném poli pamatují hodnoty, které jim od jiných procesorů během redukce přišly. Situaci znázorňuje obrázek 4.1.



Obrázek 4.1: První fáze očíslování $\mathcal{X}(g_x)$

Ve druhé fázi dochází k šíření hodnot opačným směrem. Během této fáze obdrží každý procesor (kromě procesoru 0) jednu hodnotu a to právě hledanou $first^i$ (pro procesor 0 je $first^0 = 0$). Označme pomocné pole na procesoru i jako TT^i s prvky $TT^i_0, TT^i_1, \ldots, TT^i_d$, kde $d = \lfloor \log_2(p-1) \rfloor$. V prvním kroku zašle procesor 0 hodnotu $first^0 + T^0_{g_x} + \sum_{i=0}^{d-1} TT^0_i$ procesoru 2^d. Obecně v kroku j zašle procesor p^e hodnotu $first^e + T^e_{g_x} + \sum_{i=0}^{d-j} TT^e_i$ procesoru $p^{e+2^{d-j+1}}$, $e = 0 \cdot 2^{(d-j+2)}, 1 \cdot 2^{(d-j+2)}, \ldots$, přičemž musí platit $(e + 2^{(d-j+2)}) < p$, $j = 1, 2, \ldots, d+1$. Celá situace je zachycena na obrázku 4.2.



Obrázek 4.2: Druhá fáze očíslování $\mathcal{X}(g_x)$

V okamžiku, kdy procesor zná index svého prvního kandidáta v $\mathcal{X}(g_x)$, může vzestupně očíslovat i své další uzly z této množiny. Toto lokální číslování probíhá až při následné operaci přehazování uzlů (viz podsekce 4.5.6).

4.5.5 Výběr uzlů k prohození pomocí lokální heuristiky

Při analýze meziprocesorové komunikace zjistíme, že zdánlivě jednoduchá operace náhodného výběru uzlů k prohození je nejnáročnější komunikační operací. Tato skutečnost byla hlavní motivací k vývoji lokální heuristiky. Cílem v globální heuristice je vybrat do $\mathcal{X}(g_x)$ uzly s nejlepším ziskem. Pokud předpokládáme, že všechny tyto "nejlepší" uzly nejsou lokalizovány na jednom procesoru, ale nachází se rovnoměrně na všech procesorech, je možné komunikaci mezi procesory dramaticky snížit. Stejně jako u globální heuristiky, algoritmus bude popsán jen pro množinu X rozdělení $P(X,Y)$, pro množinu Y je postup analogický.

Vybrání celkového počtu m uzlů lze zajistit tak, že zvolíme m/p nejvhodnějších uzlů z každého procesoru. Protože vše má být pokud možno lokální, je na každém procesoru i určena maximální hodnota g_x^i tak, aby velikost lokální množiny kandidátů $\mathcal{X}^i(g_x^i)$ byla větší nebo rovna m/p. Výsledná množina kandidátů $\mathcal{X}$ je pak rovna sjednocení těchto lokálních množin:

$$\mathcal{X} = \mathcal{X}^0(g_x^0) \cup \mathcal{X}^1(g_x^1) \cup \ldots \cup \mathcal{X}^{p-1}(g_x^{p-1})$$

Stanovení lokálních množin $\mathcal{X}^i(g_x^i)$ se provádí jako u globální heuristiky jen s tím rozdílem, že se neprovede globální redukce polí se ziskem a hodnotu g_x^i si určí každý procesor lokálně.

Drobný problém vzniká při výběru uzlů v případě, jestliže čísla m a p jsou nesoudělná. Pak některé procesory musí vybrat $\lfloor m/p \rfloor$ uzlů, zbylé $\lfloor m/p \rfloor + 1$ uzlů. Aby některé procesory trvale nevybíraly o jeden uzel navíc, je volba provedena náhodně. Procesor 0 všesměrovým vysíláním rozešle všem zbylým procesorům náhodně vygenerované celé číslo rp_x z intervalu $\langle 0, p-1 \rangle$. Každý procesor i si vypočítá hodnotu m^i, která udává, kolik uzlů se má na daném procesoru prohodit, musí samozřejmě platit $m = \sum_{i=0}^{p-1} m^i$. Tyto podmínky splňuje např. m^i dané vztahem:

$$m^i = \lfloor \tfrac{1}{p}(m - ((i + rp_x) \; modulo \; p) - 1) \rfloor + 1$$

Lokálně je každým procesorem i vygenerována také náhodná hodnota r_x^i z intervalu $\langle 0, 1 \rangle$, která je pak použita k výběru uzlů. Nechť $m_x^i = |\mathcal{X}^i(g_x^i)|$. Každému uzlu z $\mathcal{X}^i(g_x^i)$ je přidělen index (celé číslo z intervalu $\langle 0, m_x^i - 1 \rangle$). Aby byl uzel s indexem j vybrán, musí platit, že hodnota $(j + r_x^i \cdot m_x^i) \; modulo \; m_x^i$ je menší než m^i.

Komunikace při selekci uzlů tímto způsobem zahrnuje jen jedno všesměrové vysílaní jednoho čísla. Toto podstatné zjednodušení je vyváženo tím, že nemusí být k prohození vybrány ty nejvhodnější uzly. Je tu ovšem ještě jeden rozdíl. U globální heuristiky je výsledná bisekce nezávislá na počtu použitých procesorů, u lokální dostáváme různé výsledky pro různé hodnoty p. S rostoucím p v množině kandidátů $\mathcal{X}$ přibývá uzlů, které by se tam při globálním výběru nedostaly a tak se také zvětšuje pravděpodobnost, že k prohození nebudou vybrány ty nejvhodnější uzly.

4.5.6 Prohození uzlů

Součástí této operace (řádek 15) je také lokální očíslování kandidátů na daném procesoru. Před touto operací je již znám index prvního kandidáta. Každý procesor postupně probírá všechny své uzly. Pokud se jedná o kandidáta, vzestupně jej očísluje a přesně podle kritéria z definice globální příp. lokální heuristiky jej přehodí nebo nepřehodí na opačnou stranu. Nakonec zašle informace o novém rozložení všech svých uzlů sousedním procesorům. *Sousedním procesorem* procesoru i je vlastník alespoň jednoho uzlu spojeného hranou s libovolný uzlem procesoru i.

4.6 Analýza složitosti algoritmu

Analýzou složitosti paralelního algoritmu rozumíme zjistit závislosti základních veličin (čas, cena, zrychlení, efektivnost) na velikosti vstupních dat (v našem případě se jedná o počet uzlů n, příp. o počet hran grafu n_e) a počtu procesorů p. Nejde o přesné vyjádření daných funkcí, ale o jejich asymptotické chování.

Mějme funkce $f, g : N^+ \to R^+$. Potom říkáme, že funkce f je stupně nejvýše $O(g(n))$ (zapsáno $f(n) = O(g(n))$), jestliže

$$\exists c \in R^+ \quad \exists n_0 \in N^+ \quad \forall n \geq n_0 \ (f(n) \leq c \cdot g(n)).$$

Funkce g je v tomto případě horním odhadem asymptotického chování funkce f. Při určování škálovatelnosti algoritmu bude třeba i dolní odhad:

Funkce f je stupně nejméně $\Omega(g(n))$ (zapsáno $f(n) = \Omega(g(n))$), jestliže:

$$\exists c \in R^+ \quad \exists n_0 \in N^+ \quad \forall n \geq n_0 \ (f(n) \geq c \cdot g(n)).$$

Podrobněji je tato problematika rozebrána v [3].

4.6.1 Paralelní čas globální heuristiky

Paralelní čas je měřen počtem kroků dvojího druhu:

1. Výpočetní kroky (porovnávání, sčítání, záměna, ...)
2. Komunikační kroky (přenos a výměna informací mezi procesory)

Nejdříve určíme složitost operací, které jsou použity v průběhu jedné iterace algoritmu. Závislosti budou vyjadřovány nejen na n a p, ale také na k, což nám později snadno umožní určit složitosti v závislosti na počtu hran n_e:

1. Výpočet bisekční šířky na jednom procesoru: $T_1(n, p, k) = O(k \cdot \frac{n}{p})$
2. Paralelní redukce při součtu bisekčních šířek: $T_2(n, p, k) = O(\log p)$

3. Broadcast výsledné bisekční šířky: $T_3(n,p,k) = O(1)$
4. Výpočet zisků na jednom procesoru: $T_4(n,p,k) = O(k \cdot \frac{n}{p})$
5. Výpočet postfixového součtu pole se ziskem: $T_5(n,p,k) = O(k)$
6. Paralelní redukce při součtu polí se ziskem: $T_6(n,p,k) = O(k \cdot \log p)$
7. Broadcast výsledných hodnot: $T_7(n,p,k) = O(1)$
8. První fáze očíslování $\mathcal{X}(g_x)$: $T_8(n,p,k) = O(\log p)$
9. Druhá fáze očíslování $\mathcal{X}(g_x)$: $T_9(n,p,k) = O(\log p)$
10. Přehození uzlů na druhou stranu: $T_{10}(n,p,k) = O(\frac{n}{p})$

Složitost jednoho iteračního kroku je součtem složitostí uvedených operací:

$$T_s(n,p,k) = T_1(n,p,k) + T_2(n,p,k) + \ldots + T_{10}(n,p,k) = O(k \cdot \log p + k \ \cdot \frac{n}{p})$$

Zbývá ještě odhadnout počet iterací. Nejhorší případ je ten, když na začátku jsou v bisekci všechny hrany a každou iterací dojde ke zlepšení o 1 až na výslednou nulovou bisekční šířku. Přičíst by se mělo minimálně i $L-1$ "nezlepšujících" iterací, které jsou nutné k vybrání všech hodnot z mob plánu. Pokud hledáme závislost na n, je ale horní odhad dán především počtem hran grafu n_e. V případě k-regulárního grafu platí, že $n_e = k \cdot n/2$. Vynásobením T_s touto hodnotou dostáváme následující vztah:

$$T^{'}(n,p,k) = O(k^2 \cdot n \cdot \log p + k^2 \cdot \frac{n^2}{p})$$

Protože $n_e = k \cdot n/2$, je možné při výpočtu složitosti nahradit součin $k \cdot n$ hodnotou n_e:

$$T^{'}(n_e,p,k) = O(k \cdot n_e \cdot \log p + \frac{n_e^2}{p})$$

Při vyhodnocování všech dalších veličin budeme předpokládat, že $k = O(1)$. Časové složitosti globální heuristiky jsou pak rovny:

$$T(n,p) = O(n \cdot \log p + \frac{n^2}{p})$$

$$T(n_e,p) = O(n_e \cdot \log p + \frac{n_e^2}{p})$$

Protože jsou obě vypočtené složitosti stejné, budou další veličiny vycházející z paralelního času vyjadřovány jen závislostí na n a p.

4.6.2 Paralelní čas lokální heuristiky

Pro výpočet paralelního času lokální heuristiky je podstatné, že se neprovádí paralelní redukce při součtu polí se ziskem (operace 6) a tak složitost vyjádřená i v závislosti na stupni grafu k vyjde jednodušší:

$$T_l^{'}(n,p,k) = O(k \cdot n \cdot \log p + k^2 \cdot \frac{n^2}{p})$$

Pokud opět uvažujeme $k = O(1)$, je časová složitost lokální heuristiky:

$$T_l(n,p) = O(n \cdot \log p + \frac{n^2}{p})$$

Protože jsou časové složitosti globální i lokální heuristiky stejné, budou další veličiny vycházející z paralelního času počítány bez rozlišení těchto heuristik.

4.6.3 Výpočet ceny

Cena algoritmu $C(n,p)$ je definována:

$$C(n,p) = p \cdot T(n,p)$$

Po dosazení za $T(n,p)$ dostáváme výslednou cenu:

$$C(n,p) = O(n \cdot p \cdot \log p + n^2)$$

4.6.4 Výpočet zrychlení

Pro výpočet zrychlení algoritmu je nutné znát složitost sekvenčního řešení $SU(n)$. Zrychlení $S(n,p)$ je pak dáno vztahem:

$$S(n,p) = \frac{SU(n)}{T(n,p)}$$

V sekvenčním řešení neprobíhá samozřejmě žádná komunikace a tak z 10 operací uvedených při výpočtu složitosti globálního řešení zbydou jen operace 1, 4, 5, 10. Složitost sekvenčního algoritmu je tedy:

$$T(n) = O(n^2)$$

Zrychlení je pak rovno:

$$S(n,p) = \frac{O(n^2)}{O(n \cdot \log p + \frac{n^2}{p})} = O(\frac{p \cdot n}{p \cdot \log p + n})$$

4.6.5 Výpočet efektivnosti

Efektivnost algoritmu je dána vztahem:

$$E(n,p) = \frac{SU(n)}{C(n,p)} = \frac{S(n,p)}{p}$$

Po dosazení dostáváme:

$$E(n,p) = O(\frac{n}{p \cdot \log p + n})$$

4.6.6 Škálovatelnost algoritmu

Škálovatelnost algoritmu nám dává odpověď na otázku, jaká je optimální velikost vstupních dat n pro zadaný počet procesorů p, příp. obráceně, kolik procesorů zvolit pro zadaná vstupní data tak, aby procesory byly optimálně využity. Problém lze formulovat i takto: Jak je třeba při změně jednoho parametru (n nebo p) upravit ten druhý, aby zůstala zachována efektivnost algoritmu ? Cílem je tedy určit vztah mezi n a p, aby platilo $E(n,p) = O(1)$. Podrobněji je tato problematika rozebrána v [3]. V našem případě tedy musí platit:

$$\frac{n}{p \cdot \log p + n} = konst$$

Jednoduchou úpravou tohoto vztahu lze dostat přímo výsledné závislosti:

$$n_{opt} = \Omega(p \cdot logp) \qquad\qquad p_{opt} = O(\frac{n}{\log n})$$

Kapitola 5

Reprezentace a generování grafů

5.1 Reprezentace grafu

Reprezentace grafu v hlavním a ve výpočetním procesu nejsou stejné. Tato odlišnost plyne především ze zcela jiných požadavků na datové struktury reprezentující graf v dané situaci. Zatímco při počítání bisekce ve výpočetním procesu je nutno znát především okolí zadaného uzlu, při generování grafu v hlavním procesu je nejdůležitější informace o existenci zadané hrany. Z tohoto hlediska je určitě nejvhodnějším řešením matice sousednosti, kde přístup k požadované hraně má konstantní časovou složitost. Problém nastává s paměťovou náročností matice pro grafy s velkým počtem uzlů, navíc pokud se jedná o řídké grafy, je toto řešení značně neefektivní. Pro rozsáhlé grafy je tedy i při generování použito řešení z výpočetního procesu, kdy máme pole uzlů a každý uzel vlastní seznam svých sousedů. To je ideální řešení pro výpočet bisekční šířky a výpočet zisku, přístup k hranám při generování grafu však probíhá s lineární časovou složitostí. Grafy se ale generují jen za účelem získání vstupních dat pro heuristiku, měření času začíná až po jejich rozeslání do výpočetních procesů a tak toto pomalejší řešení nemá žádný vliv na výkon heuristiky. Uvedené pole se seznamy sousedů je reprezentováno objektovou strukturou, která prostřednictvím svých metod umožňuje stejný styl práce jako s maticí sousednosti.

5.2 Generování k-regulárního grafu

Pro testování bylo nutné navrhnout a implementovat algoritmus generující k-regulární grafy, které tvoří vstupní data. Algoritmus by měl generovat grafy s n uzly podle definice uvedené v sekci 2.1, tedy každý uzel inciduje přesně s k hranami, přičemž množina sousedů je náhodná a nejsou dovoleny vícenásobné hrany. Výjimku tvoří případ, kdy n a k jsou lichá čísla, v tomto případě má jeden uzel stupeň $k-1$. Posledním parametrem algoritmu je hodnota r, která udává maximální počet neúspěšných

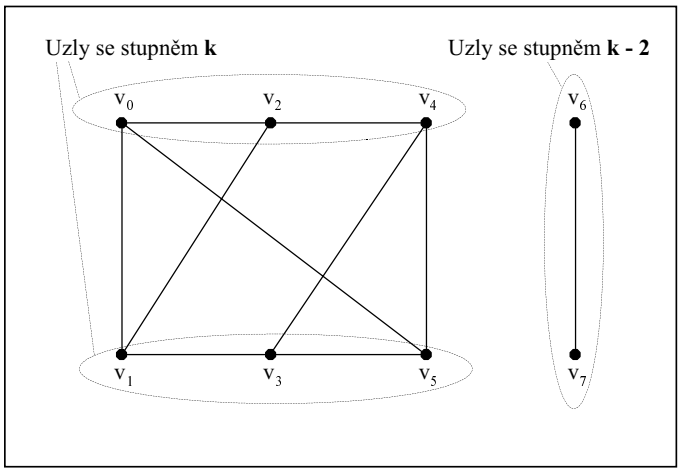
pokusů při přidávání jedné hrany. Výsledný graf je uložen ve dvojrozměrném poli $MS[n][n]$, které představuje matici sousednosti. Pro výpočet je nutné pomocné pole $U[n]$, jehož prvky jsou dvojice čísel:

```
record {
    int id;
    int k;
end;
```

Prvky pole nesou informaci o jednotlivých uzlech, id udává číslo uzlu, k jeho aktuální stupeň. Při inicializaci se provede nastavení všech prvků pole $U[i].id = i$, $U[i].k = 0$, vynulování všech prvků matice sousednosti a inicializace pomocné proměnné s na hodnotu n.

Přidání jedné hrany probíhá tak, že se náhodně vyberou dva různé indexy i, j z intervalu $\langle 0, s-1\rangle$ tak, aby mezi uzly $U[i].id$ a $U[j].id$ neexistovala hrana. Tato hrana se vytvoří a zvýší se o 1 hodnoty $U[i].k$ a $U[j].k$. Pokud je $U[i].k$ resp. $U[j].k$ rovno k, provede se záměna prvku $U[s-1]$ s prvkem $U[i]$ resp. $U[j]$ a hodnota s se sníží o 1. Tak je zajištěno, že v poli U na pozicích $0, 1, \ldots, s-1$ jsou vždy jen prvky reprezentují uzly se stupněm menším než k a na pozicích $s, s+1, \ldots, n-1$ jsou prvky reprezentující uzly se stupněm k. Graf je celý vygenerován, jestliže $s = 0$ nebo je $s = 1$ a $U[0].k = k-1$.

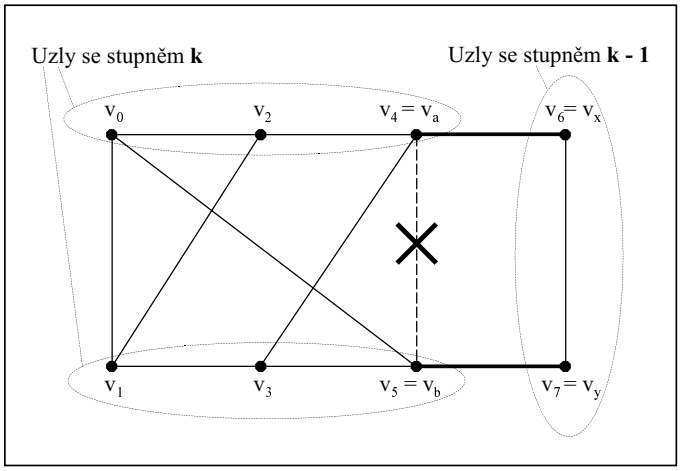
Tento způsob generování je velmi efektivní pro řídké grafy. Se stoupajícím stupněm grafu může ke konci generování docházet k tomu, že počet pokusů na náhodné zvolení dvou uzlů nespojených hranou bude příliš velký. Ovšem i v řídkých grafech může ke konci generování nastat situace, kdy hranu uvedeným způsobem nelze přidat. Příklad je uveden na obrázku 5.1, kde je znázorněna možná situace při generování grafu s parametry $n = 8$ a $k = 3$.



Obrázek 5.1: Problémová situace při přidávání hrany

Je vidět, že jen uzly v_6, v_7 mají stupeň menší než k. Jsou však spojeny hranou a tak již do tohoto grafu není možné žádnou hranu přidat. Přitom ale zadaný graf ještě není vygenerován. Naštěstí jde tato problémová situace poměrně snadno vyřešit.

Pokud jsou v r pokusech za sebou vygenerovány jen dvojice uzlů, mezi které hranu přidat nelze, provedeme s poslední dvojicí uzlů v_x, v_y následující operaci. Nalezneme dva jiné uzly v_a, v_b se stupněm k spojené hranou tak, aby neexistovaly hrany mezi v_x a v_a a mezi v_y a v_b. Dalším krokem je zrušení hrany $\langle v_a, v_b \rangle$ a přidání hran $\langle v_x, v_a \rangle$ a $\langle v_y, v_b \rangle$. Stupeň uzlů v_a a v_b tak zůstávává zachován k a do grafu je přidána jedna hrana (stupeň uzlů v_x, v_y je zvýšen o 1). Situace z minulého obrázku by tedy mohla být vyřešena takto:



Obrázek 5.2: Řešení předchozí problémové situace

Z obrázku je vidět, že sice byla přidána hrana, ale nastala také nová problémová situace a proto by k dokončení generování zadaného grafu bylo třeba popsanou operaci přerovnání hran ještě jednou zopakovat.

5.3 Generování grafu s "úzkým místem"

Graf s "úzkým místem" byl použit pro testování úspěšnosti heuristik. Pro tento typ úloh bylo nutné vygenerovat graf, který má malou, v našem případě nulovou, bisekční šířku. To jde velmi snadno udělat např. tak, že z k-regulárního grafu odstraníme hrany mezi lichými a sudými uzly. Graf se tak rozpadne na dvě části - v první jsou hranami spojené jen sudé uzly a v druhé jen liché. Mezi oběma částmi nevede žádná hrana a bisekční šířka tohoto grafu je tudíž nulová. Odebráním poloviny hran jsme ale snížili průměrný stupeň uzlů grafu na polovinu. Abychom tedy dostali graf, jehož uzly mají průměrný stupeň k, je nejdříve nutné vygenerovat algoritmem z předchozí podkapitoly k-regulární graf se stupněm $2 \cdot k$ a teprve potom provést uvedené odebrání hran.

Kapitola 6

Implementace algoritmu

Algoritmus uvedený v předchozí kapitole byl i se všemi svými variantami implementován v jazyce C++ s využitím knihovny funkcí systému PVM. Implementace byla provedena přesně podle algoritmu uvedeného v sekci 4.4 a proto zde budou popsány především nejdůležitější datové struktury a třídy.

6.1 Nejdůležitější datové struktury a třídy

Základní třídou, reprezentující jeden výpočetní proces, je *GraphPartProcC*. Hlavní metodou, která výpočet obsahuje v té podobě, jak byl popsán v sekci 4.4, je metoda *run()*. Z ní jsou volány další metody této třídy představující dílčí kroky algoritmu. V následujících odstavcích budou popsány všechny instanční proměnné mající rozhodující význam pro funkci algoritmu.

6.1.1 Implementace bitových polí a paketů

Proměnné *pointSideP*, *sendPackP*, *recvPackP*, *resultPackP* jsou ukazateli na instanci třídy *BitPacketC*. Tato třída reprezentuje bitové pole, x-tý bit lze nastavit na hodnotu 0 resp. 1 metodou *reset(int x)* resp. *set(int x)*. Oproti klasickému bitovému poli je *BitPacketC* rozšířen o hodnotu, která je používána jako adresa (pro její nastavení resp. zjištění slouží metoda *setAddr(int addr)* resp. *addr()*). Instance této třídy jsou totiž těmi strukturami, v kterých jsou při meziprocesorové komunikaci uloženy informace o tom, v které polovině se nachází uzly daného procesoru a adresa je jednoznačným identifikátorem odesílajícího procesoru. Proměnná *sendPackP* slouží k odesílání těchto informací, které jsou druhou stranou přijímány do proměnné *recvPackP*. V proměnné *resultPackP* je ukládáno prozatímní nejlepší řešení. Význam jednotlivých bitů byl popsán v sekci 4.3. Proměnná *pointSideP* není využívána jako packet, ale jako klasické bitové pole. Jsou zde uchovávány informace o tom, do které

poloviny patří daný uzel. Na tuto instanci vlastní ukazatel všechny uzly procesoru a tak je zajištěno, že každému uzlu jsou k dispozici informace o všech ostatních a přitom jsou tyto informace na každém procesoru uloženy jen jednou.

6.1.2 Implementace grafu

Graf G je ve výpočetním procesu reprezentován instancemi třídy *PointC*. Ukazatelem na pole těchto instancí je proměnná *pointPP*. Ve třídě *PointC* je instanční proměnnou pole *neibourP* o velikosti k, ve kterém jsou uložené ty uzly grafu, se kterými je uzel reprezentovaný danou instancí spojen hranou. Počet těchto sousedních uzlů je v instanční proměnné *neibourNum*.

Metoda *side()* vrací, do které poloviny grafu uzel náleží. Návratovou hodnotou metody *bw()* je příspěvek (počet hran), kterým uzel přispívá do výsledné bisekce, viz. podsekce 4.5.2. Zisk uzlu vrací metoda *countGain()*. Popis výpočtu zisku je podrobněji popsán v podsekci 4.5.3.

6.1.3 Implementace mob plánu

Mob plán je reprezentován proměnnou *scheduleP*. Jde o proměnnou typu ukazatel na instanci třídy *ScheduleC*. Tato třída je abstraktním předkem všech tříd implementujících mob plán. Metoda *nextValue()* slouží k získání další hodnoty mob plánu. Typ mob plánu je dán virtuální funkcí *func()*. Tato metoda je ve třídě ScheduleC tzv. čistou virtuální funkcí (viz. [11]) a implementována je až ve třídách *LinearScheduleC*, *ExponentialScheduleC*, *LinExpScheduleC*, které jsou potomky třídy *ScheduleC* a představují po řadě typy mob plánu uvedené v sekci 3.2. Volba mob plánu se provádí při inicializaci vytvořením instance příslušné třídy a při dalším používání se bez jakéhokoliv rozlišování typu mob plánu pracuje s uvedenou proměnnou *scheduleP*. Toto řešení využívající polymorfismu také umožňuje velice snadné vytvoření dalšího mob plánu - stačí jen vytvořit nového potomka třídy *ScheduleC* a předefinovat virtuální metodu *func()*.

6.1.4 Implementace polí se ziskem

Při globálním očíslování množiny kandidátů bylo nutné implementovat dvě (operace se provádí zvlášť pro každou polovinu grafu) pole se ziskem. Tato dvě pole reprezentuje instance třídy *GainC*. Nabízí se otázka, proč nebyla navržena třída reprezentující jen jedno pole a použity dvě její instance. Je to proto, že při paralelní redukci jsou obě pole zasílána pomocí jedné komunikační operace a tak bylo vhodné z obou polí vytvořit jednu souvislou datovou oblast. Třída obsahuje metody pro inicializaci

polí, pro přístup k prvkům pomocí indexů z rozsahu $-k \ldots k$, výpočet postfixového součtu, součet polí, a tak i při implementaci mohla být použita téměř stejně jako odpovídají pole při popisu algoritmu.

6.2 Uživatelské rozhraní

Výpočetní proces je představován programem *bisekp*, hlavnímu procesu odpovídá program *bisekm*. Program *bisekp* nemá žádné parametry a je automaticky spouštěn systémem PVM. Výpočet se spouští programem *bisekm* a jeho parametry jsou v uvedeném pořadí následující:

- p - počet procesorů
- n - počet uzlů grafu
- k - stupeň grafu
- L - délka mob plánu
- m_0 - počáteční velikost mobu
- t - typ heuristiky, 0 = globální lineární, 1 = globální exponenciální, 2 = globální kombinovaná, 4 = lokální lineární, 5 = lokální exponenciální, 6 = lokální kombinovaná
- $par1$ - generovaný graf - nezáporná celočíselná hodnota použitá pro inicializaci generátoru náhodných čísel při generování grafu
- c - má význam jen pro kombinované heuristiky, udává počet kroků lineární části mob plánu

6.3 Systém PVM

PVM je softwarový systém, který umožňuje použít počítačovou síť (dokonce heterogenní) jako jeden velký paralelní počítač. Na jednom systému PVM lze vytvořit několik virtuálních paralelních počítačů, přičemž jejich počet není závislý na počtu fyzických strojů v počítačové síti. Pro vytváření programů určených ke spuštění na těchto virtuálních počítačích je určena knihovna funkcí, které zpřístupňují operace potřebné při paralelním programování. Jde především o vytváření a rušení paralelní procesů, jejich vzájemnou komunikaci a synchronizaci. Další informace jsou uvedeny v [6].

6.4 Výpočetní prostředky použité při testování

Prvním systémem, na kterém probíhalo testování, byla sít pracovních stanic s operačním systémem Linux. Každá ze stanic byla osazena procesorem Pentium Pro 200 MHz a pamětí RAM 64 MB. Stanice byly propojeny sítí Ethernet (100 Mbit).

Část úloh byla také otestována na počítači IBM SP-2 s operačním systémem AIX 4.1. Tento počítač je tvořen několika uzly propojenými speciální sítí. Instalace v průběhu měření obsahovala 23 uzlů - 1 s procesorem POWER 2, 8 s procesorem P2SC a 14 s novějšími procesory P2SC. Celkový diskový prostor byl asi 180 GB, celková paměť RAM měla kapacitu 6 GB.

Nejmarkantnější rozdíl oproti linuxovému systému je právě ve zmiňované propojovací síti. Jedná se o vysoce výkonný přepínač *(High Performance Switch)*, který je schopen přenášet data mezi kterýmikoliv uzly rychlostí až 40 MB/s. Přepínač je složen z několika křížových přepínačů 4x4 a uvedená šířka pásma 40 MB/s je kapacitou (v každém směru) jedné propojovací linky mezi těmito křížovými přepínači.

Dalším rozdílem, především ve stylu práce, je samotné spouštění úloh na některých uzlech. To je umožněno prostřednictvím plánovače *(Load Leveler)*, který se podle zadaných požadavků automaticky postará o přidělení procesorů a spuštění úlohy v příslušném módu. Podrobnější informace o počítači IBM SP-2 jsou uvedeny v [7]. Pro plné využití tohoto hardwarového vybavení bylo nutno použít speciální verzi systému PVM (PVMe) od firmy IBM.

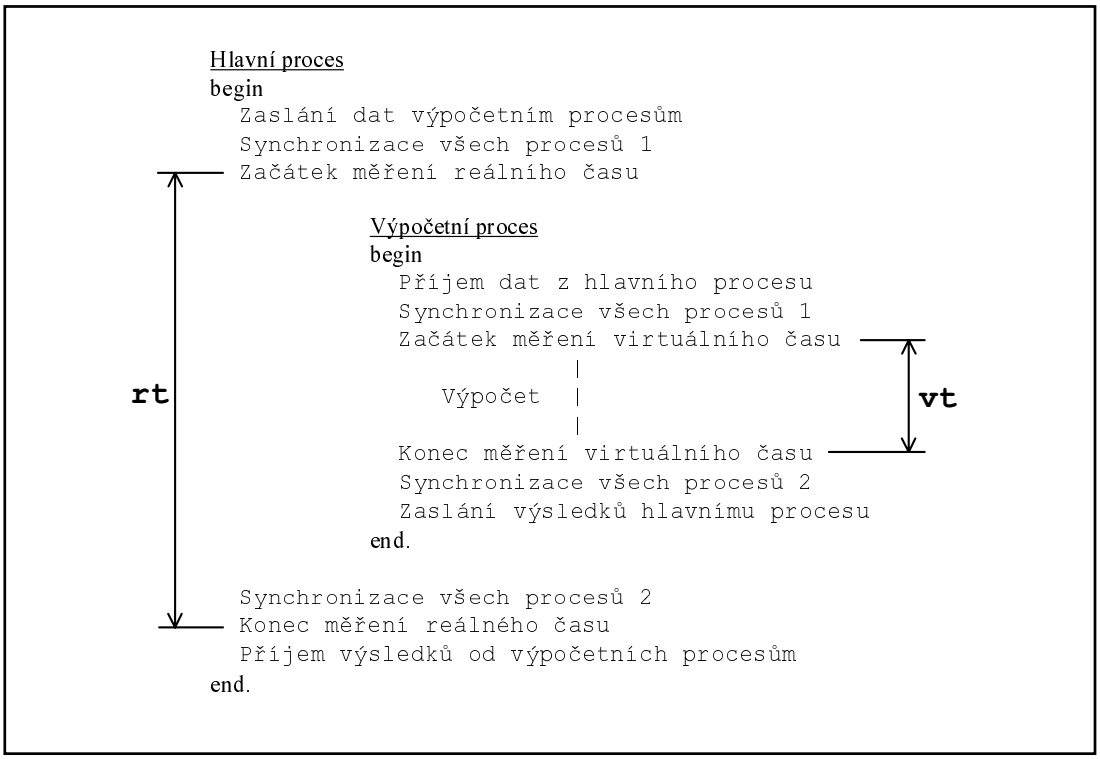
6.5 Měřené veličiny při testování

Výsledkem jednotlivých měření byly tyto základní veličiny:

- Reálný čas rt
- Virtuální čas vt
- Bisekční šířka bw
- Počet iterací it

Reálný čas udává celkovou dobu, po kterou výpočet probíhal. Do tohoto času nejsou započítány časy potřebné na počáteční rozeslání dat výpočetním procesům a zpětné zaslání výsledků. Pro zadaný graf by tyto hodnoty měly být vždy stejné, v praxi jsou však výrazně ovlivněny momentálním vytížením propojovací sítě a tak by při jejich zahrnutí do výsledného času mohlo dojít ke zkreslení času vlastního výpočtu. Čas je měřen v hlavním procesu tak, jak to ukazuje obrázek 6.1.

Virtuální čas je skutečný čas, který byl spotřebován procesorem na výpočet. Je měřen v každém výpočetním procesu (viz obrázek 6.1) a jeho výsledná hodnota je maximem ze všech naměřených hodnot.



Obrázek 6.1: Měření reálného a virtuálního času

Počet iterací je počet průchodů cyklem while (viz popis algoritmu, sekce 4.4). Tato hodnota je přímo úměrná virtuálnímu času. ale narozdíl od něj je nezávislá na systému, na kterém probíhalo měření.

Kapitola 7

Výsledky a jejich vyhodnocení

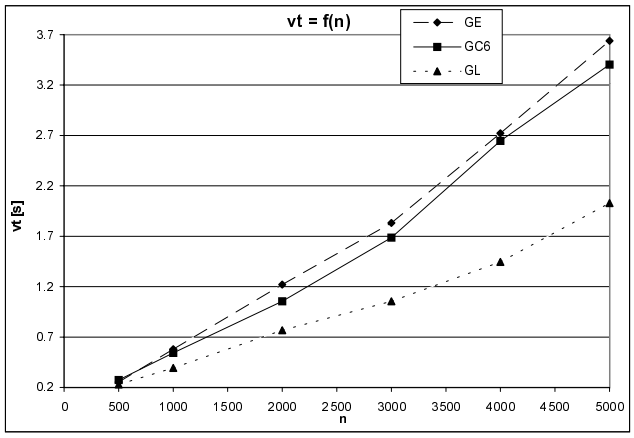
Pokud nebude explicitně uvedeno jinak, znamená to, že všechna měření byla prováděna s globální heuristikou. Symbol GL označuje globální lineární, GE globální exponenciální a GCc globální kombinovanou heuristiku. Proměnná c v symbolu GCc odpovídá stejné proměnné, která byla použita při definici kombinované heuristiky v sekci 3.2.

7.1 Vhodnost heuristiky pro daný typ grafu

Tato měření byla provedena za účelem zjistit, kterou heuristiku (dle typu mob plánu) je nejvhodnější použít pro daný typ grafu. Heuristiky byly porovnávány podle virtuálního času, dosažené bisekční šířky a schopnosti odhalit "úzké místo grafu". Měření byla provedena na síti Linuxových pracovních stanic, hodnoty jsou průměrem z 10 měření.

7.1.1 Hustý k-regulární graf

Následující graf vyjadřuje pro tři různé heuristiky závislost virtuálního času vt na počtu uzlů n, vstupem je k-regulární graf stupně $k = 100$. Ostatní parametry jsou $p = 4$, $L = 10$, $m = 0.1 \cdot n$.

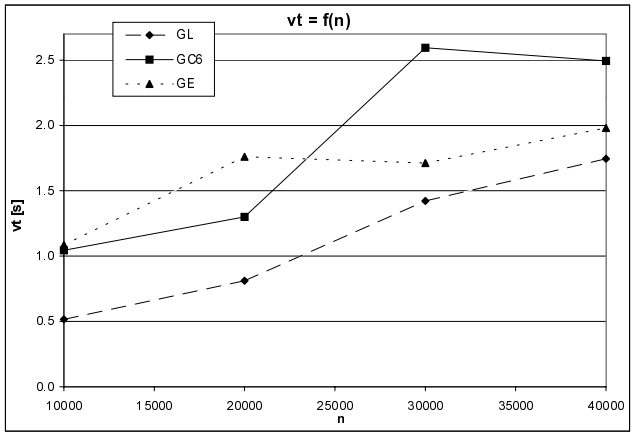


Obrázek 7.1: Závislost $vt = f(n)$ pro k-regulární graf stupně $k = 100$

Z grafu je vidět, že pro všechna n dosahuje lineární heuristika nejlepších výsledků. Pro $n = 5000$ je virtuální čas lineární heuristiky menší o 44% než vt heuristiky exponenciální a o 40% než vt heuristiky GC6. Rozdíly bisekčních šířek uvedených heuristik nepřesahují 0.3% a tak lze říci, že pro výpočet bisekční šířky hustého k-regulárního grafu **je nejvhodnější lineární heuristika**. Poslední velmi malé hodnoty mob plánu exponenciální a kombinovaných heuristik zapříčiňují velké zvýšení počtu iterací (k výsledku se dochází po menších krocích) a tedy i virtuálního času. Zlepšení výsledné bisekční šířky je však minimální.

7.1.2 Řídký k-regulární graf

Ve všech měřeních bylo opět dosaženo nejlepších výsledků **s lineární heuristikou**. Se zvyšujícím se počtem uzlů se snižoval rozdíl vt oproti exponenciální heuristice až na 12% při $n = 40000$, s lineární heuristikou bylo dosaženo i nejlepší bisekční šířky (pro $n = 40000$ rozdíl činil 2.7%).

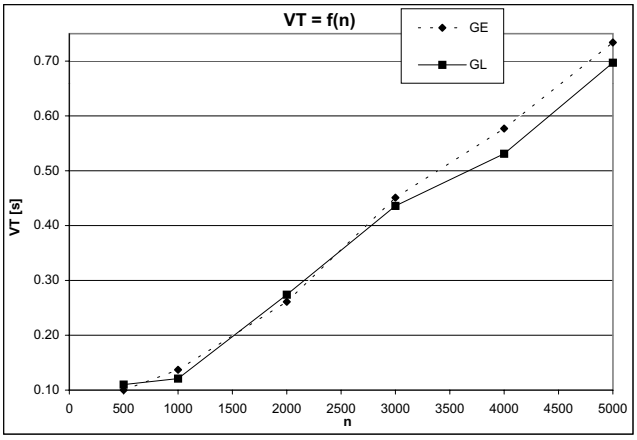


Obrázek 7.2: Závislost $vt = f(n)$ pro k-regulární graf stupně $k = 5$

Zajímavé je také zjištění, že pro všechna n dosáhla lineární heuristika nejlepších výsledků vždy, další pořadí kombinovaných heuristik a heuristiky exponenciální nelze jednoznačně stanovit. Analýzou počtu iterací v průběhu algoritmu bylo zjištěno, že značně kolísá počet iterací s moby nejmenších velikostí a proto exponenciální a kombinované heuristiky mají méně vyrovnané výsledky než heuristika lineární. Tento jev se samozřejmě projevil i u hustých k-regulárních grafů z předchozího měření.

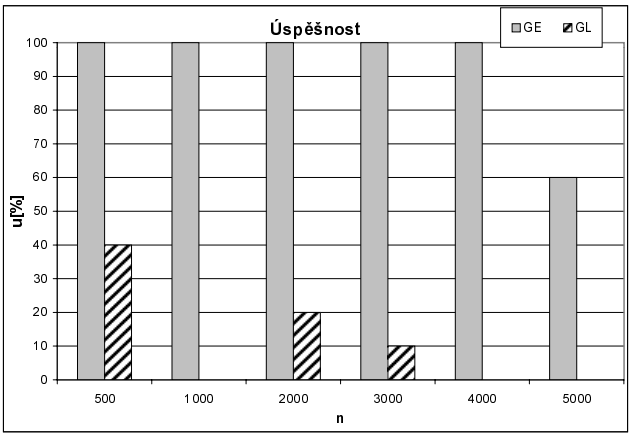
7.1.3 Hustý graf s "úzkým místem"

V tomto případě jsou rozdíly ve virtuálním čase podstatně menší než u hustých grafů bez úzkého místa, lineární heuristika je pro $n = 5000$ lepší než exponenciální jen o 5%.



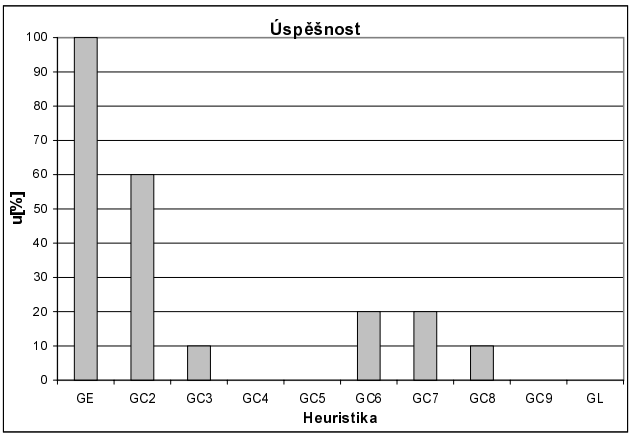
Obrázek 7.3: Závislost $vt = f(n)$ pro graf s "úzkým místem", $k = 50$

Při testování grafů s "úzkým" místem byly heuristiky srovnávány nejen podle virtuálního času a bisekční šířky, ale i podle úspěšnosti, s jakou toto úzké místo (nulovou bisekci) dokázaly odhalit. Jak ukazuje následující graf, dosáhla zde vynikajících výsledků exponenciální heuristika, která jen pro $n = 5000$ nedosáhla 100% úspěšnosti. Naproti tomu schopnosti lineární heuristiky odhalit "úzké místo" jsou velmi nízké.



Obrázek 7.4: Závislost $u = f(n)$ pro graf s "úzkým místem", $k = 50$

Úspěšnost jednotlivých heuristik pro $n = 4000$ ukazuje následující graf:

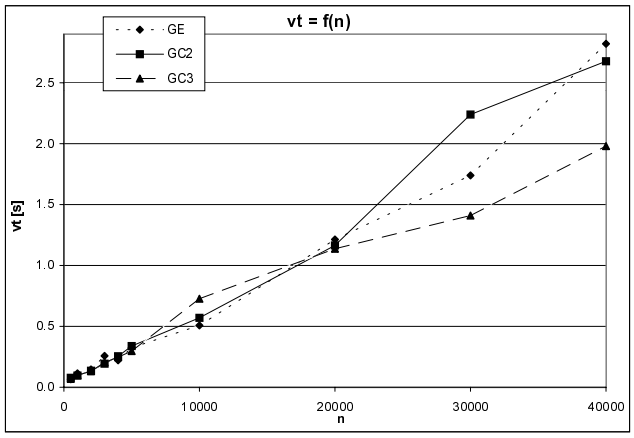


Obrázek 7.5: Úspěšnost heuristik, graf s "úzkým místem", $k = 50$

Při hledání "úzkého místa" jsou důležité moby s malou velikostí, protože jen prohazováním malých množin uzlů ke konci heuristiky je možné přesně dosáhnout malých bisekčních šířek. Pro husté grafy s "úzkým místem" je tedy nejvhodnější **exponenciální heuristika**.

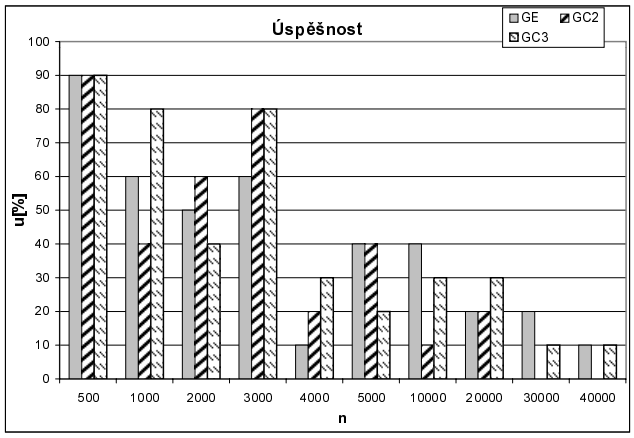
7.1.4 Řídký graf s "úzkým místem"

Stejně jako v předchozím případě, lineární heuristika nemá šanci odhalit "úzké místo". Výsledky exponenciální heuristiky se mírně zhoršily, nebylo ani jednou dosaženo 100% úspěšnosti. Došlo však k výraznému zlepšení u všech kombinovaných heuristik a heuristiky GC2 a GC3 dosáhly srovnatelných výsledků s exponenciální heuristikou. Virtuální čas heuristiky GC3 byl pro $n = 40000$ o 29% lepší než čas heuristiky exponenciální a o 26% lepší než čas heuristiky GC2.



Obrázek 7.6: Závislost $vt = f(n)$ pro graf s "úzkým místem", $k = 5$

Měření s řídkými grafy byla provedena pro větší počet uzlů, což v tomto případě prokázalo pokles úspěšnosti s rostoucím počtem uzlů grafu.



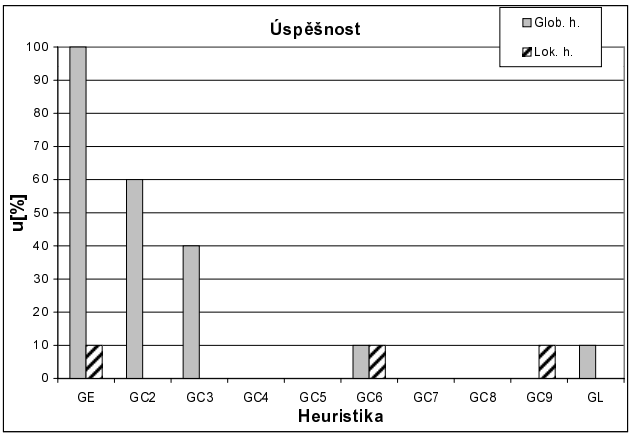
Obrázek 7.7: Závislost $u = f(n)$ pro graf s "úzkým místem", $k = 5$

Z grafu je vidět, že úspěšnost všech tří sledovaných heuristik je poměrně vyrovnaná. Vzhledem k výsledkům virtuálního času je ale pro tento typ grafu **nejvhodnější heuristika GC3**. Lepšího času bylo zřejmě dosaženo díky větším mobům v prvních iteracích heuristiky, otázkou zůstává, proč v hustých grafech tento typ heuristiky má tak malou úspěšnost. Zajímavé je také nedosažení 100% úspěšnosti ani v jednom případě. To je zřejmě způsobeno tím, že stupeň grafu se příliš neliší od velikosti úzkého místa a tak v průběhu heuristiky není jasně dán jednoznačný "směr zlepšení", jak je tomu u hustých grafů s úzkým místem.

7.2 Srovnání globální a lokální heuristiky

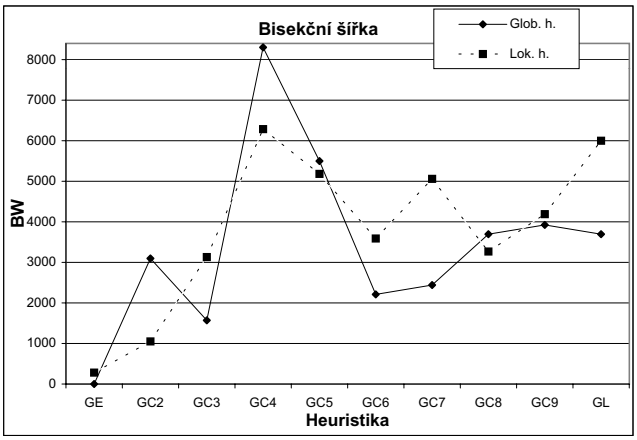
Měření bylo provedeno na linuxové síti pracovních stanic, parametry heuristiky byly následující: $p = 4$, $n = 6000$, $m_0 = 600$, $l = 10$. Testovaný graf byl rozdělen na

4 stejně velké komponenty, mezi nimiž nevedou žádné hrany, a na každý procesor byla umístěna jedna čtvrtina uzlů z každé komponenty. Toto, pro lokální heuristiku velmi nepříznivé rozdělení, přineslo očekávané výsledky v úspěšnosti heuristik:



Obrázek 7.8: Závislost $u = f(n)$, srovnání glob. a lok. heuristiky

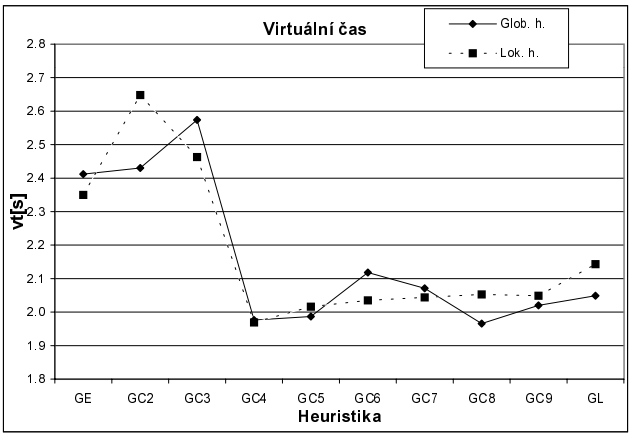
Je vidět, že lokální heuristika najde správnou bisekční šířku jen výjimečně. Dalším kritériem pro srovnání byla průměrná bisekční šířka:



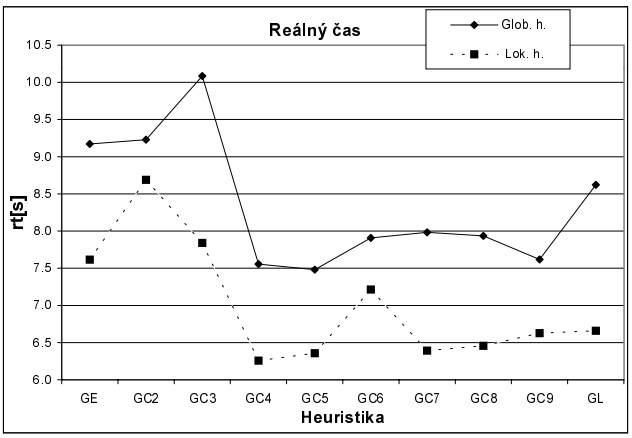
Obrázek 7.9: Bisekční šířka pro různé typy heuristik

Toto měření nepřineslo jednoznačné výsledky. Lokální heuristika byla dokonce ve čtyřech případech úspěšnější, v dalších dvou případech dosáhla globální heuristika lepších výsledků jen velmi těsně. K přesnějšímu určení této závislosti by bylo třeba provést další měření (závislosti pro různě velké grafy různých typů).

Další srovnání byla provedena pomocí virtuálního a reálného času:



Obrázek 7.10: Virtuální čas pro různé typy heuristik

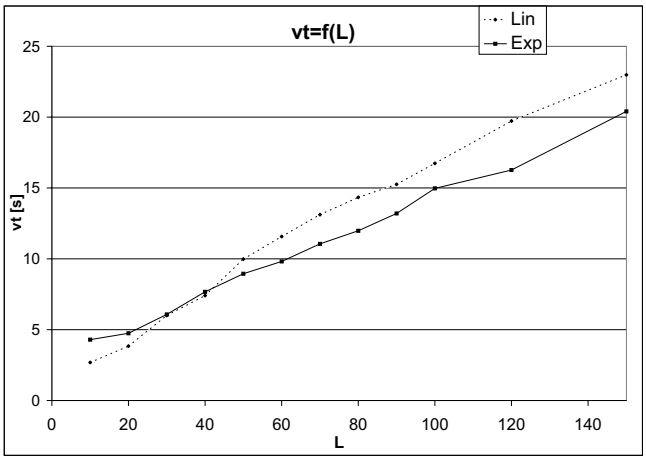


Obrázek 7.11: Reálný čas pro různé typy heuristik

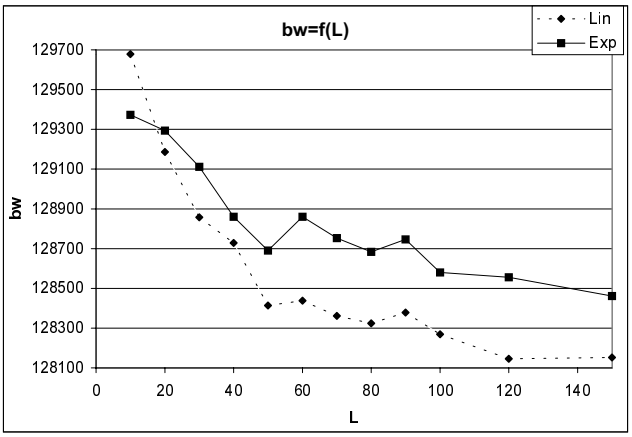
Tyto závislosti přesně splnily očekávání. Zatímco virtuální časy obou heuristik jsou přibližně stejné (maximální rozdíl 0.2 s), rozdíly v reálných časech jsou podstatně větší (až 2.2 s). Ve virtuálním čase je totiž zahrnut jen čas výpočetních kroků, zatímco v reálném čase je zahrnuta i doba potřebná ke komunikaci. Na síti pracovních stanic s operačním systémem Linux byl tedy reálný čas lokální heuristiky až o 22.8% lepší, minimální zlepšení bylo 5.8%. Přitom je však nutno počítat s tím, že všechny lokální heuristiky prakticky nejsou schopny najít "úzké místo" grafu.

7.3 Závislost na délce mob plánu

Měření bylo provedeno na linuxové síti pracovních stanic, parametry heuristiky byly následující: $p = 4$, $n = 6000$, $m_0 = 10$, k-regulární graf stupně $k = 100$ (hustý graf).



Obrázek 7.12: Závislost $vt = f(L)$ pro k-regulární graf stupně $k = 100$

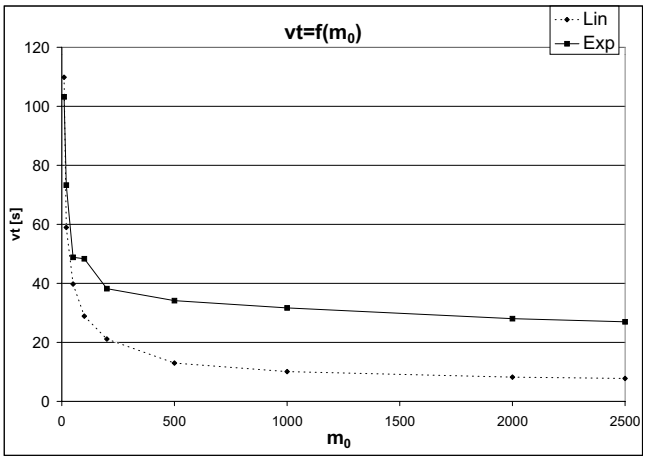


Obrázek 7.13: Závislost $bw = f(L)$ pro k-regulární graf stupně $k = 100$

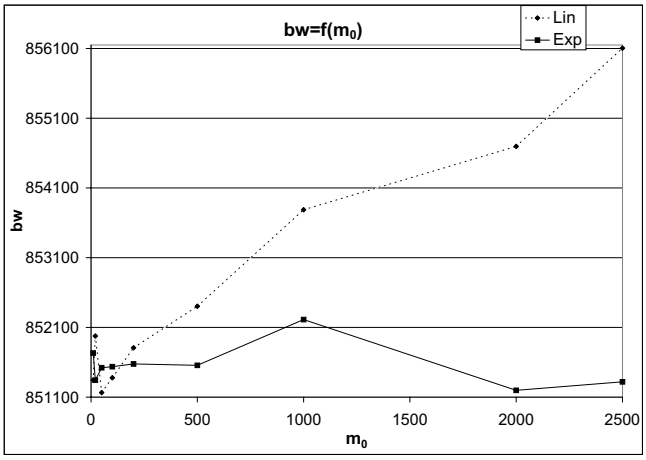
Ukázalo se, že virtuální čas roste lineárně s rostoucí délkou mob plánu, zdvojnásobení délky způsobí přibližně 1.8-násobné zvětšení času. S rostoucí délkou mob plánu také dochází ke zlepšení bisekční šířky. Toto zlepšení je ovšem minimální (přibližně 1%) a proto zvětšování délky mob-plánu (alespoň pro tento typ grafu) není příliš výhodné.

7.4 Závislost na počáteční velikosti mobu

Měření bylo provedeno na linuxové síti pracovních stanic, parametry heuristiky byly následující: $p = 4$, $n = 6000$, $l = 10$, k-regulární graf stupně $k = 100$ (hustý graf).



Obrázek 7.14: Závislost $vt = f(m_0)$ pro k-regulární graf stupně $k = 100$



Obrázek 7.15: Závislost $bw = f(m_0)$ pro k-regulární graf stupně $k = 100$

Je vidět, že po počátečním prudkém poklesu klesá virtuální čas dále jen velmi pomalu. Bisekční šířka v případě lineární heuristiky mírně roste, (asi o 0.5% při 5-násobném zvětšení m_0), u exponenciální heuristiky se bisekční šířka prakticky nemění. Parametr m_0 je tedy důležité nastavit minimálně na tu hodnotu, v kterém končí prudký pokles virtuálního času, další zvětšování již nepřináší výrazný efekt. Pro měření, která byla provedena, se jako vhodná ukázala hodnota $m_0 = 0.1 \cdot n$. Velká hodnota virtuálního času pro malá m_0 je způsobena zřejmě tím, že díky malým mobům se dochází k výsledku po malých krocích, roste tedy počet iterací a tudíž i virtuální čas.

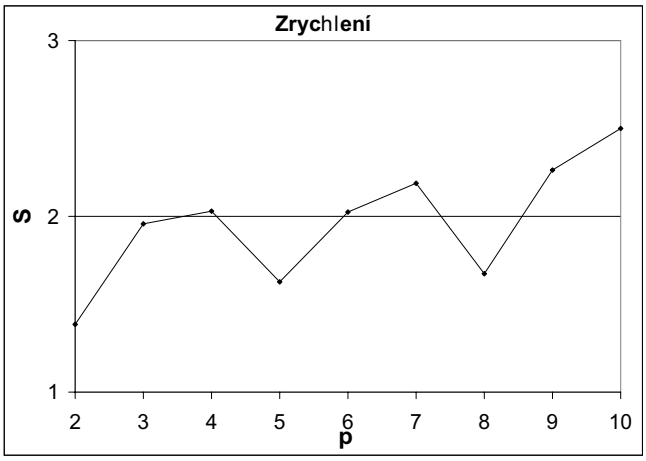
7.5 Zrychlení a efektivnost

Jak plyne z definic v podsekcích 4.6.4 a 4.6.5, pro určení uvedených veličin je třeba znát čas sekvenční heuristiky. Z časových důvodů nebyla provedena úplně nová sekvenční implementace, ale jen upraveno paralelní řešení. Rozdíl spočívá především v odstranění veškeré komunikace a přípravy komunikačních paketů. Čistě sekvenční řešení by se zřejmě lišilo především jiným návrhem datových struktur, které by již nemusely respektovat potřebu meziprocesorové komunikace.

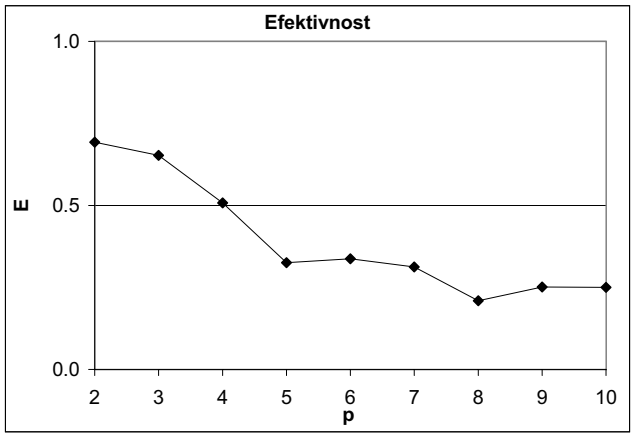
Jak již bylo zmíněno, při testování byly měřeny dva časy - reálný a virtuální. Nastává otázka, který z nich použít pro výpočet zrychlení a efektivity. Virtuální čas odpovídá vlastnímu výpočtu, z komunikační části jsou v něm zahrnuty jen přesuny do komunikačních bufferů, při vlastní komunikaci ale neběží (proces je "uspán"). Na unixových systémech reálný čas odpovídá skutečnému trvání všech úloh, které zde v daném okamžiku běží. Pokud by byla spuštěna jen měřená úloha, byl by reálný čas pro výpočet efektivnosti a zrychlení vhodnější. To ovšem v době měření na linuxové síti prac. stanic nebylo možné. Tyto podmínky měly být zajištěny automaticky pomocí plánovače na počítači IBM SP-2. Tomu však neodpovídaly naměřené výsledky. V některých případech byl reálný čas větší než virtuální jen 2%, jindy se lišil až o 100%. Po těchto problémech byl tedy pro výpočet zvolen čas virtuální.

7.5.1 Ověření zrychlení a efektivnosti na IBM SP-2

První ověření proběhlo na počítači IBM SP-2 až pro 10 procesorů pro k-regulární graf stupně $k = 100$, $n = 8000$, $l = 10$, $m_0 = 800$ a lineární heuristiku. Výsledky byly následující:



Obrázek 7.16: Ověření zrychlení na IBM SP-2

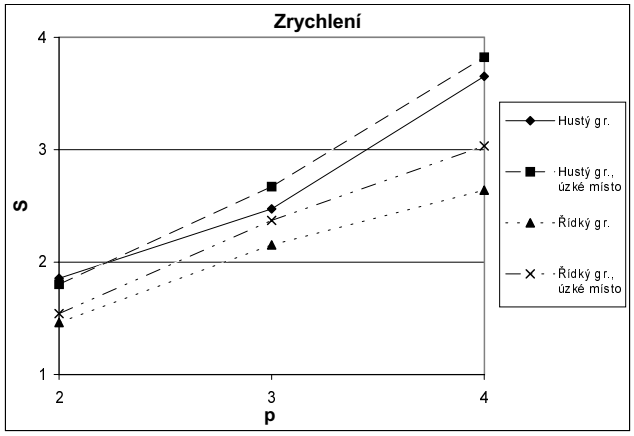


Obrázek 7.17: Ověření efektivnosti na IBM SP-2

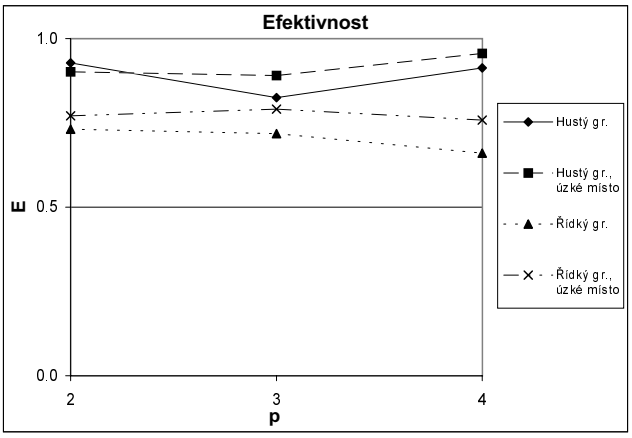
Výkyvy na grafu zrychlení jsou způsobené zřejmě tím, že hodnoty, z nichž bylo zrychlení vypočítáno, jsou průměrem jen ze tří měření. Zrychlení s rostoucím počtem procesorů mírně roste, k maximální hodnotě p se však ani jednou nepřibližuje. Efektivnost dosahuje nejlepší hodnoty pro $p = 2$, pod 0.5 klesá při použití více než 4 procesorů.

7.5.2 Zrychlení a efektivnost pro daný typ grafu

Měření byla provedena pro exponenciální heuristiku na linuxové síti pracovních stanic, $p = 4$, $l = 10$, $m_0 = 0.1 \cdot n$. Husté grafy byly měřeny pro $n = 5000$, řídké pro $n = 10000$, grafy s "úzkým místem" pro $k = 5$, grafy bez "úzkého místa" pro $k = 100$.



Obrázek 7.18: Zrychlení v závislosti na typu grafu

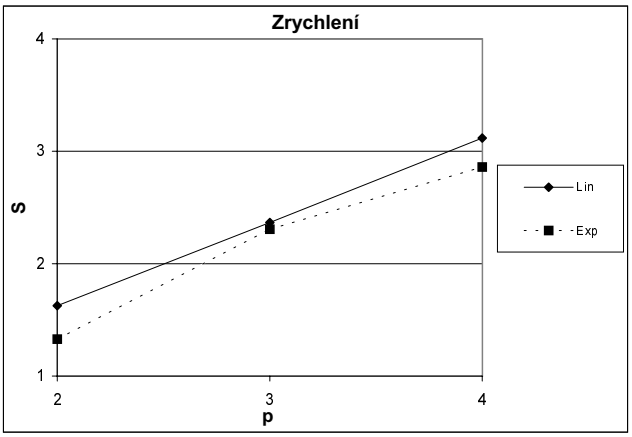


Obrázek 7.19: Efektivnost v závislosti na typu grafu

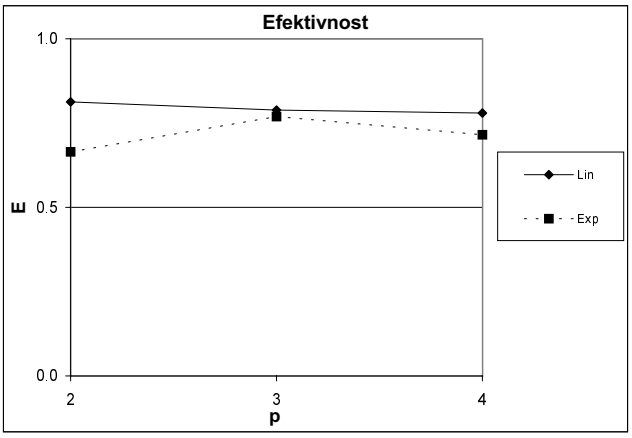
Narozdíl od měření na IBM SP-2, bylo dosaženo velice dobrého zrychlení a tudíž i efektivnosti. Lepší výsledky byly naměřeny pro husté grafy (které ale měly menší počet uzlů). Všechny tyto výsledky závisí na časech sekvenční heuristiky. Naskýtá se tedy otázka, zda hodnoty efektivity blízké 1 nebyly dosaženy díky stavu, ke kterému dochází při tzv. superlinearitě. V této situaci je sekvenční řešení zpomalováno např. tím, že není možné udržovat, narozdíl od paralelního, všechna data v paměti a tak je možné naměřit zrychlení větší než p a efektivitu větší než 1. Výsledky by asi také nebyly tak dobré v případě, pokud by bylo možno použít k výpočtu reálný čas, který lépe odráží skutečnou dobu potřebnou pro meziprocesorovou komunikaci.

7.5.3 Zrychlení a efektivnost lokální heuristiky

Měření byla provedena pro lineární i exponenciální heuristiku, parametry byly následující: $n = 5000$, $k = 100$, $l = 10$, $m_0 = 500$.



Obrázek 7.20: Zrychlení lokální heuristiky



Obrázek 7.21: Efektivnost lokální heuristiky

V tomto případě se jen potvrdily výborné výsledky měření na linuxové síti pracovních stanic, s lineární heuristikou bylo dosaženo mírně lepších výsledků, efektivita se pohybovala v intervalu $\langle 0.66, 0.81 \rangle$.

Kapitola 8

Závěr

V této práci byly navrženy nové heuristiky pro výpočet minimální bisekce grafu. Základní heuristikou, která byla východiskem pro další varianty, se stala mob heuristika uveřejněná v [2]. Modifikace představují především kombinované heuristiky, jejichž první část odpovídá chování lineární a druhá část chování exponenciální heuristiky. Další modifikací základní mob heuristiky byla lokální heuristika, která byla navržena s cílem minimalizovat meziprocesorovou komunikaci.

Za účelem testování byla provedena implementace mob heuristiky a všech uvedených variant. K implementaci bylo využito prostředí PVM, které umožňuje použít heterogenní počítačovou síť jako paralelní počítač s distribuovanou pamětí. Implementace byla provedena pro dva různé systémy. Prvním z nich byla počítačová síť pracovních stanic s operačním systémem Linux, dalším systémem byl 23-procesorový počítač IBM SP-2. Při této implementaci bylo využito speciální verze PVM od firmy IBM, která umožňuje plně využít veškerého hardwarového vybavení.

Ověřování probíhalo především na linuxovém systému. Prvním důvodem k tomu byl mnohem jednodušší průběh samotného testování. Jednotlivá měření byla automaticky spouštěna pomocí unixového skriptu a tak mohlo být provedeno rozsáhlé ověření vlastností všech uvedených heuristik. Spouštění paralelních úloh na IBM SP-2 je nutné provádět prostřednictvím plánovače, který automaticky rozmístí jednotlivé procesy na procesory. To bohužel při našem ověřování neprobíhalo bezchybně, byla nutná neustála kontrola průběhu měření a proto na tomto systému byla ověřena jen menší část úloh. Byly zde také v některých případech naměřeny nevěrohodné hodnoty reálného času, což následně ovlivnilo především vyhodnocení zrychlení a efektivnosti.

Při ověřování vlastností heuristik byly získány cenné zkušenosti, které je možné využít i v praxi. Heuristiky byly testovány na různých typech grafů a výsledky vyhodnoceny na základě dosažených časů, bisekční šířky, příp. schopnosti přesně odhalit "úzké místo" grafu. Bylo také ověřeno chování heuristik pro různé hodnoty vstupních

parametrů jako je délka mob plánu a počáteční velikost mobu. Vyhodnocení zrychlení a efektivnosti prokázalo, že mob heuristika a její varianty patří mezi dobře paralelizovatelné algoritmy. Lepší výsledky vyhodnocení uvedených veličin na linuxovém systému mohou být zapříčiněny použitím virtuálního času při výpočtu. Virtuální čas totiž nezahrnuje celou dobu potřebnou pro meziprocesorovou komunikaci, která je na IBM SP-2 podstatně rychlejší.

I když měření byla poměrně rozsáhlá, vzhledem k velkému počtu parametrů heuristiky a velkému počtu různých typů grafu nebylo možno ani zdaleka ověřit všechny možné závislosti. Především zjištění úspěšnosti nalezení "úzkého místa" grafu v závislosti na délce mob plánu a počáteční velikosti mobu by mohlo přinést zajímavé výsledky. Dále by bylo vhodné prověřit vlastnosti lokální heuristiky i pro další typy grafů. Kombinované lineárně exponenciální heuristiky vznikly vhodnou modifikací mob plánu. Mob plán však může tvořit libovolná (neklesající) posloupnost a tak se otevírá prostor pro vývoj dalších zajímavých variant. Mob heuristika používá libovolné počáteční rozdělení. Dalšího zlepšení by bylo možno dosáhnout použitím vhodného algoritmu, který by konstruktivní metodou určil počáteční rozdělení "rozumněji" - provedl by jednoduchý odhad minimální bisekce.

Literatura

[1] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, San Francisco, 1979.

[2] J. E. Savage, M. G. Wloka. Heuristic for Parallel Graph-Partitioning. Technical Report No. CS-89-41, Brown University, 1991

[3] P. Tvrdík. Parallel systems and algorithms. Vydavatelství ČVUT, Praha, 1997.

[4] L. Smékal, M. Šoch. Evaluation of the mob heuristics in parallel graph partitioning. In Poster 99, pages IC-16. Faculty of Electrical Engineering, CTU Prague, 1999.

[5] M. Šoch, P. Tvrdík, M. Wolf. Parallel graph-partitioning using the mob heuristic. In M. Bubak, J. Dongarra, J. Wasniewski, editors. Recent Advances in Parallel Virtual Machines and Message Passing Interface, number 1332 in Lecture Notes in Computer Science, pages 383-389. Springer Verlag, 1997.

[6] A. Geist, A. Bequelin, J. Donqarra, W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Networked Parallel Computing, MIT Press, Massachusetts Institute of Technology, 1994.

[7] IBM Systems Journal, Vol. 34, No. 2, 1995.

[8] J. Hlavička. Architektura počítačů. Vydavatelství ČVUT, Praha, 1996.

[9] L. Kučera. Kombinatorické algoritmy. SNTL, Praha, 1983.

[10] J. Plesník. Grafové algoritmy. Veda, Bratislava, 1983.

[11] P. Herout, V. Rudolf, P. Šmrha. ABC programátora v jazyce C. Nakladatelství KOPP, České Budějovice, 1992.

Příloha A

Zdrojové texty

Zdrojové texty implementovaných algoritmů se nachází na přiložené disketě.