1 Preorder

Algoritmus přiřazení pořadí preorder vrcholům stromů (nejdříve otec, pak levý syn a následně pravý syn) se skládá ze tří částí:

1. Vytvoření pole hodnot
2. Suma suffixů
3. Korekce výsledků ze sumy suffixů

1.1 Eulerova cesta

Pro správnou funkčnost sumy suffixů se nejdříve musí vypočítat Eulerova cesta. Algoritmus spočívá v tom, že každá hrana se nahradí dvojicí orientovaných hran (dopřednou a její reverzní) čímž ze stromu vznikne eulerovský graf, který obsahuje orientovanou kružnici, která prochází každou hranou právě jednou. Takový to graf je pak reprezentován seznamy sousednosti, kde pro každý uzel se tvoří struktura seznamu sousedů.

Tato struktura pak obsahuje informaci, jaká hrana jde od daného uzlu k jeho sousedu, následně pak její reverzní hranu a pokud uzel má více než 1 souseda, tak i odkaz na následující strukturu souseda.

Pro takovéto seznamy sousednosti se pak počítá eulerovská cesta, kde každý procesor se stará o jednu hranu. Pro svou hranu pak nalezne její reverzní, v případě, že pro reverzní hranu existuje soused, tak je hlavní hrana daného souseda výsledkem algoritmu. Pokud souseda nemá, tak je výsledkem hlavní hrana na začátku daného seznamu.

Výsledné pole Etour se pak ještě musí opravit (musí se zavést kořen stromu). To se provede tím, že hodnota v poli Etour pro hranu, která vede do kořene stromu (poslední, koncová hrana) se nahradí danou hranou.

1.2 Vytvoření pole hodnot – pole vah

Pro sumu suffixů se také musí ještě vypočítat i pole hodnot. Jednotlivé hodnoty se počítají paralelně, pro každou hranu jedna hodnota. Pokud je hrana dopředná, tak je její hodnota v poli 1 v opačném případě 0.

1.3 Suma suffixů

Suma suffixů se provádí nad polem vah v pořadí eulerovské cesty. Algoritmus se provádí paralelně, kde každý procesor se stará o hodnotu z pole cesty a pole vah. Je rozdělen na tři části. V první se provede úprava hodnoty pole vah v procesoru, který se stará o poslední hranu – nastaví svou hodnotu váhy na 0. Ve druhé části se provádí cyklus, který se opakuje až do `log n`. V tomto cyklu se k hodnotě váhy přičte váha následníka dané hrany (o jakého následníka se jedná se zjistí z hodnoty pole cesty) a následně se do hodnoty následníka přiřadí hodnota následníka mého následníka (jako by se změnil odkaz). V poslední se provádí úprava hodnot vah, v případě, že původní hodnota váhy v posledním procesoru (první část algoritmu) se lišila od nuly, tak se tato původní hodnota přičte ke všem hodnotám vah.

1.4 Korekce výsledků

Pořadí uzlů v preorder průchodu určují jednotlivé hodnoty vah pro dané hrany. Výsledky jsou ovšem v opačném směru (od konce), proto se musí provést korekce. Korekce se opět provádí paralelně s tím, že v ní berou účast pouze procesory, které se starají o dopředné hrany. Výsledné pořadí pro danou hranu se tedy vypočte jako: $n - weight + 1$, kde n je celkový počet hran. Nakonec se ještě kořenové hraně přiřadí hodnota 1.

1.5 Analýza složitosti

Algoritmus se skládá z několika částí. Tvorba seznamu sousednosti a Eulerova cesta má složitost `O(n)`. Oprava Eulerovy cesty, tvorba pole vah a korekce výsledků po sumě suffixů má složitost `O(1)`. Samotná suma suffixů pak `O(log n)`. Celková časová složitost je tedy `O(n log n)`. Počet procesorů potřebných pro výpočet je `2n-2`.

2 Implementace

K implementaci se použil jazyk C++ a knihovna Open MPI.

2.1 Načtení hodnot uzlů a jejích zpracování

Hodnoty uzlů načítá a zpracovává první (nultý) proces. Následně pro ně vytvoří hrany a seznam sousednosti – funkce `createSousedy()`. Hrana je reprezentována objektem třídy `Edges`, která obsahuje id dané hrany a startovací a ukončovací uzel (podle směru orientované hrany). Jeden prvek v seznamech sousednosti je reprezentován objektem třídy `Soused` a osahuje vlastnosti pro id hlavní a reverzní hrany.

Celý seznam sousednosti je pak ve výsledku vektor vektorů objektů Soused, kde pro každý uzel existuje jeden vektor sousedů. Odkazy na sousedy jsou prováděny pomoci indexace, kdy se ví, že pokud má vektor délku větší než jedna, tam má více než jednoho souseda a hodnoty pro souseda jsou na následujícím indexu ve vektoru.

Následně první proces odešle všem ostatním procesům id hran, o které se mají starat a také všem odešle celý seznam sousedů (před odesláním se jednotlivé vektory převedou na pole). Přijímající proces seznam sousedů opět poskládá do původní podoby (jedna dvojice je jeden Soused, pokud je jich více jedná se o další objety Soused).

2.2 Eulerova cesta

Funkce `eulerTour()` provádí algoritmus Eulerovy cesty. Tuto funkci provádí každý proces samostatně, tím, že ve vektoru vektorů sousedů vyhledává vektor, jehož jedna z hlavních hran obsahuje id hrany, o kterou se daný proces stará. Při nalezení pak postup opakuje tentokrát pro reverzní hranu. Výsledkem je pak hodnota souseda (pokud má reverzní hrana ještě nějakého souseda – není na posledním indexu vektoru) nebo hodnota prvního souseda ve vektoru.

2.3 Pole hodnot – pole vah

Provádí každý procesor samostatně ve funkci `getWeight()`, kde prohledává vektory sousedů a podle nich určuje, zda je hrana dopředná nebo zpětná, a přiřazuje si hodnotu 1 nebo 0. Proces si také nastaví booleovskou proměnnou, o jaký druh hrany se stará.

2.4 Korekce Eulerovy cesty

Po vypočtení hodnot Eulerovy cesty odešlou všechny procesy své hodnoty do nultého procesu. Jenom ten ví, která hrana je poslední, a tak může provést korekci. Následně hodnoty odešle zpět.

2.5 Suma suffixů

Opět každý proces provádí samostatně ve funkci `suffixSum()`.

V první částí proces, který se stará o poslední hranu nastaví svoji váhu na 0 (svou původní hodnotu si uloží bokem) a označí se jako koncový proces.

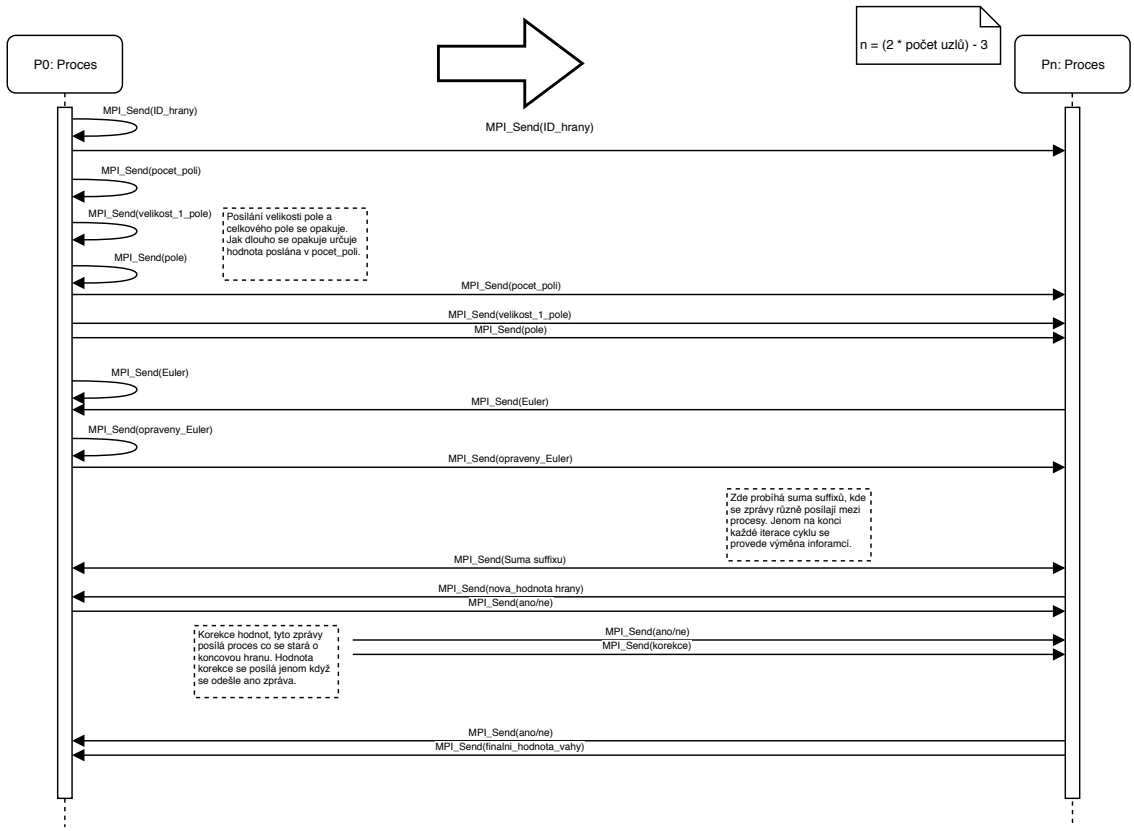
Následně se opakuje cyklus, ve kterém si procesory navzájem posílají zprávy s žádostmi o hodnoty. Menší výjimky tvoří procesy, co se starají o první a poslední hranu. Prvního se nikdo nebude ptát na jeho hodnoty, takže ani nemá očekávat od někoho zprávy. Naopak poslední musí očekávat proměnný počet dotazů na jeho hodnoty – v první iteraci cyklu proces odpovídá 2krát, v dalším 3krát a poté 5krát, takhle pokračuje až do celkového počtu všech hran. To kolikrát proces musí v dané iteraci odpovědět se počítá pomoci vzorce $(x * 2) - 1$, s tím, že se začíná na hodnotě 2 a do x se dosazuje hodnota z předcházející iterace. Dalším problémem, který se v cyklu musel řešit bylo to, že ačkoliv na začátku některé procesy dostaly žádost o hodnoty, tak v dalších iteracích se jich již nikdo neptal – procesy se zasekly na čekání na zprávu a dále nepracovaly. Proto na konci cyklu všechny procesy posílají své nové hodnoty hran do nultého procesu a ten následně rozešle všem informaci, zda má daný proces v další iteraci cyklu čekat, že se ho někdo zeptá na jeho hodnoty. Na konci cyklu se ještě nachází synchronizační bariéra (`MPI_Barrier()`), která zabrání procesům přepsat svoje hodnoty, dokud všechny nebudou mít svoje nové hodnoty (mohlo se stát, že proces měl už novou hodnotu, ale jiný proces chtěl po něm ještě tu starou nebo, že 2 procesy byly v rozdílných iteracích cyklu).

Ve třetí části algoritmu koncový proces odešle všem zbývajícím informaci o tom, zda se provádí korekce výsledných hodnot vah.

2.6 Korekce výsledků pro preorder a výpis uzlů

Každý proces pak ve funkci `/preOrder()` provádí korekci výsledných hodnot. Následně pak všechny pošlou svoje výsledky do nultého procesu. Ten zpracuje jejích pořadí (používá k tomu třídu `Node`), přidá kořenový uzel a výsledek vypíše.

2.7 Sekvenční diagram komunikačního protokolu



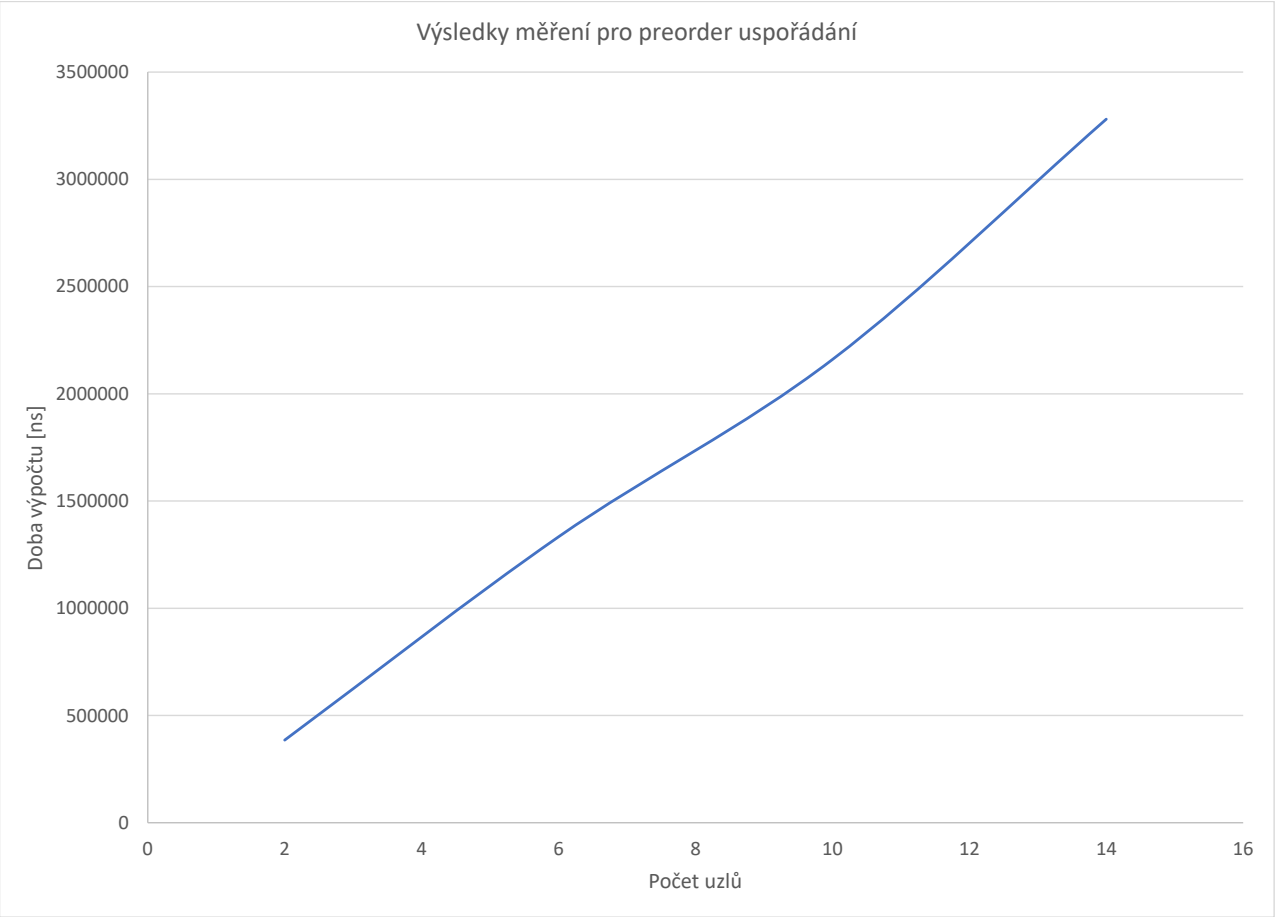
Obrázek 1: Sekvenční diagram komunikačního protokolu

3 Experimenty se vstupy

Experimentování a měření probíhalo na školním serveru Merlin. Merlin je víceuživatelský systém obsahující jeden procesor s 12 logickými procesory, výsledné časové hodnoty z experimentování tedy mohou být zkreslené z důvodu vytížení systému.

K měření času probíhání výpočtu se použila funkce `int clock_gettime(clockid_t clk_id, struct timespec *tp)`, algoritmus měření byl převzat z Profiling Code Using clock_gettime.

Měření probíhalo s počtem uzlů 2, 6, 10 14. Pro každou hodnotu se provedlo 7 měření, odebralo se nejlepší a nejhorší výsledek a ostatní se zprůměrovaly. Na obrázku 2 lze pak nalézt graf s průměrnými výsledky doby výpočtu pro jednotlivý počet uzlů.



Obrázek 2: Výsledky experimentů

4 Závěr

Naměřené výsledky přímo neodpovídají teoretické časové složitosti algoritmu . Může to být způsobeno nepřesnou implementací požadovaného algoritmu nebo, jak již bylo dříve zmíněno, vytížením serveru merlin. Vliv na zvětšení časové složitosti může mít i to, že se musela zajišťovat dodatečná synchronizace mezi jednotlivými procesy.