Priradenie poradia preorder vrcholom

Adam Bezák, xbezak01@stud.fit.vutbr.cz

22. dubna 2018

1 Rozbor a analýza algoritmu

V prvom rade je potreba spracovať vstup a vytvoriť si internú reprezentáciu stromu v programe. Každý z prechodov stromu (inorder, postorder, preorder) vyžaduje vytvoriť Eulerovskú cestu. Teda Eulerovská cesta je jeden zo spôsobov ako prechádzať binárnym stromom a výsledkom je, že sa vytvorí zoznam v ktorom sú obsadené všetky uzly a hrany. Pracuje na princípe, že zo vstupného binárneho stromu sa vytvorí orientovaný graf tak, že každá hrana (u,v) je nahradená dvomi orientovanými hranami (u,v) a (v,u). Výsledná orientovaná kružnica prejde všetkými hranami, každou hranou prejde len jeden krát. Na implementáciu sa využíva funkcia následníka, ktorá každej hrane priradí hranu `ETour(e)`, ktorá následuje hranu `e`. Na implementáciu binárneho stromu sa využíva `Adjency list`.

Na prechod preorder využijeme vytvorenú Eulerovskú cestu a to tak, že vždy navštívime najprv otca a potom oboch synov. Od prechodu Eulerovskov cestou sa líši tým, že ignoruje spätné hrany a využíva len dopredné. V paralelnom cykle, je najprv potrebné vytvoriť váhové pole. Pokiaľ je hrana dopredná, jej váha je 1. V opačnom prípade je 0. Nad týmto polom vytvoríme sumu suffixov, ktorá sčíta len dopredné hrany označené váhou 1 a ignoruje hrany označené váhou 0. Vo výslednom poli sumy suffixov máme sčítané jednotlivé prvky, ktoré odpovedajú dopredným hranám od **konca**. Preto je treba ešte spraviť korekciu a to tak, že od veľkosti vstupu odpočítame váhu danej hrany a získame finálne poradie.

2 Teoretická zložitosť algoritmu

Na vytvorenie Eulerovej cesty potrebujeme lineárnych $2 * n - 2$ procesorov, kde n je veľkosť vstupu, a využívame paralelný cyklus. V ideálnom prípade sa pri využití Adjency listu neprechádza celým zoznamom ale pristupuje sa len k prvému prvku zoznamu tak dosahujeme konštantnú zložitosť $O(1)$. Celkovú Eulerovskú cestu sme taktiež schopný vytvoriť v konštantnom čase $O(1)$. Ohodnotenie hrán váhami má taktiež konštantnú časovú zložitosť $O(1)$. Algoritmus suffix sum má teoretickú časovú zložitosť $O(n * \log n)$. Výsledný preorder algoritmus má teda teoretickú časovú zložitosť $c(n) = O(n * \log n)$.

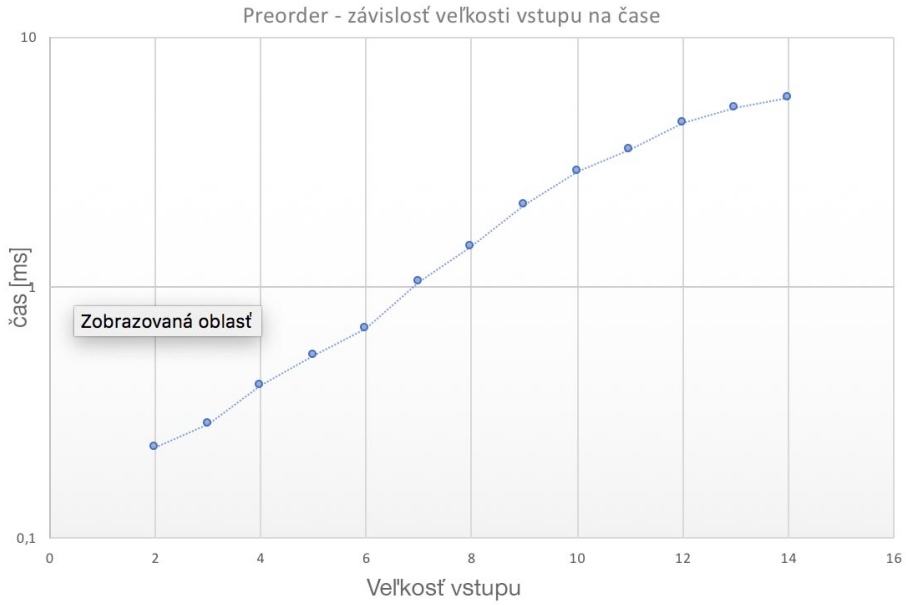
3 Implementácia

Vždy na začiatku práce s knižnicou MPI je potrebné spraviť inicializáciu pomocou `MPI_Init(&argc, &argv)`. Procesor s rankom 0 vytvorí internú reprezentáciu Adjency listu. Je reprezentovaná vektorom `vector<struct AdjChar>`. Pre každý uzel sa vytvorí jedna štruktúra `AdjChar`, ktorá pozostáva z ďalšieho vektoru štruktúr jednotlivých sousedností. Napr. pre uzol sa vytvorí najprv ľavá sousednost, potom pravá sousednost a ak sa nejedná o koreň tak sa vytvorí aj spätná sousednosť na nadradený uzol. Po vytvorení Adjency listu sa nainicializuje MPI dátový typ štruktúry `ProcEdge`, pomocou ktorej sa rozpošlú jednotlivé hrany všetkým procesorom na základe funkcie `ETour`. Táto funkcia je implementovaná ako prechod vektorom a na základe poradového čísla hrany vráti nasledujúcu hranu v Eulerovskej ceste. Ak dotazovaná hrana na indexe `i` nemá žiadneho následníka vo vektore sousednostní, tak sa vráti hrana na indexe 0, v opačnom prípade sa vráti následník dotazovanej hrany s indexom `i + 1`. Ak už

každý procesor ma priradenú hranu môže sa začať počítať pole suffix sum. V prvom kroku si každý procesor paralelne vypočíta svoju váhu na základe doprednej hrany. Pole suffix sum sa počíta paralelne v každom procesore na základe jeho váhy a priradenej hrany. Posledný procesor s rankom `i` ma priradenú poslednú hranu v Eulerovskej ceste a odošle procesoru s rankom `i - 1` neutrálny prvok (0) sčítaný s jeho váhou. Procesor s rankom `i - 1` príjme túto hodnotu, pripočíta svoju váhu a odošle ďalej na procesor s rankom `i - 2`. Prvý procesor už len príjma poslednú hodnotu a nikam neposiela. Takýmto spôsobom sa vypočíta hodnota suffix sum pre každý procesor. Následne všetky procesory odošlú svoju štruktúru hrany, ktorá obsahuje aj hodnotu suffix sum procesoru s rankom 0. Tento procesor príjme tieto štruktúry do vektoru a postará sa o výpis jednotlivých uzlov v poradí preorder na základe doprednej hrany a hodnoty suffix sum. Vypíše najprv koreň a potom koncový uzel každej preorder hrany.

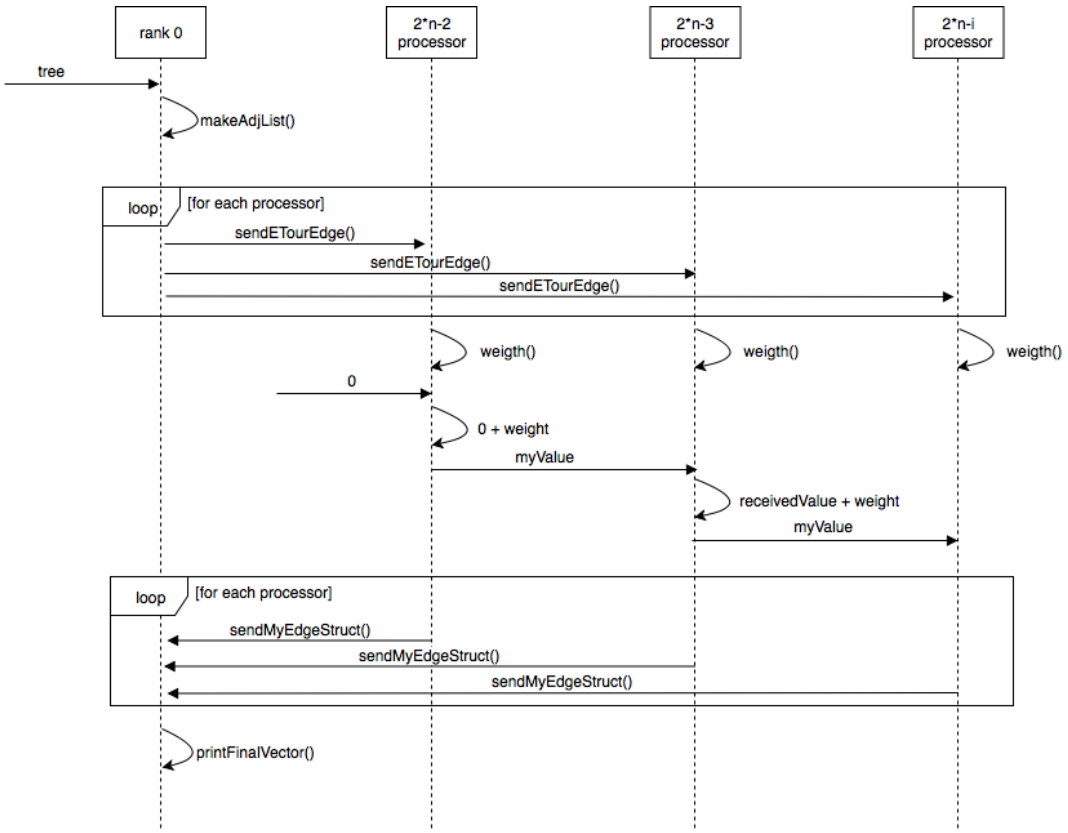
4 Meranie časovej zložitosti

Meranie časovej zložitosti prebiehalo na referenčnom stroji `Merlin` a na meranie som využil vstavanú funkciu `MPI_Wtime()`, ktorá merala čas od začiatku vytvárania Adjency listu až po konečnú distribúciu hodnôt suffix sum procesoru s rankom 0. Na obrázku č. 1 je možné vidieť výsledné časy v logaritmickom merítku. Testovaná bola množina vstupných hodnôt od veľkosti 2 po 14. Pre každý vstup prebehlo 6 meraní, z ktorých sa následne vypočítala priemerná hodnota.



Obrázek 1: Graf časovej závislosti.

5 Sekvenčný diagram



Obrázek 2: Sekvenčný diagram.

6 Záver

Z naimplementovaného algoritmu som dosiahol lineárnu časovú zložitosť. Teoretická časová zložitosť sa mojou implementáciu nepotvrdila. Je možné, že to spôsobouje zlý výber implementácie Adjency listu a funkcie ETour, kedže v naprogramovanom riešení sa prechádza celý Adjency list sekvenčne a hladá sa nasledujúca hrana funkcie ETour. Ak by funkcia ETour bola naprogramovaná pomocou poľa a pristupovalo by sa k nej skrz index, tak sa domnievam, že by bola dosiahnutá požadovaná teoretická zložitosť.