Autor: Jan Lorenc
(xloren15)

Datum: 23.04.2022

PRL - Paralelní a distribuované algoritmy

2. projekt – Přiřazení pořadí preorder vrcholům

Algoritmus

Předpokladem pro algoritmus je již spočtená Eulerova cesta (dále Etour). Pro tu je třeba celkem *2n - 2* procesů, kde *n* je délka vstupu a lze ji získat dle následujícího pseudokódu:

Algoritmus 1:

Vstup: Pole sousedů „Sousedi"
Výstup: Pole Etour s položkamu pro každou hranu (proces) e = (u, v)

```
1. Parallel.ForEach i in range(1, 2n-2) {
2.         Etour[e] = next(e_r) != null ? next(e_r) : Sousedi[v]
3. }
```

Přiřazení preorder pořadí lze provést ve 3 krocích. Nejprve inicializujeme váhy hran na 1 pro dopředné hrany a 0 pro reverzní. Dále spočítáme sumu sufixů, čímž získáme váhu pro každou hranu. Na závěr se pro dopředné hrany (reverzní nás nezajímají) provede korekce, což je prakticky sestupné seřazení dle vah. Následující pseudokód ukazuje tento algoritmus:

Algoritmus 2:

Vstup: Pole Etour
Výstup: Pole s preorder pořadím

```
1.  // 1 – inicializace vah
2.  Parallel.ForEach e in edges {
3.      weights[e] = e is forwardEdge ? 1 : 0
4.  }
5.  // 2 – suma sufixů
6.  weights = SuffixSum(Etour, weights)
7.  // 3 – korekce
8.  Parallel.ForEach e in edges {
9.      if (e is forwardEdge) {
10.         result[v] = n – weights[e]  // +1 pokud indexujeme od 1
11.     }
12.     result[root] = 0  // 1 pokud indexujeme od 1
13. }
```

Analýza složitosti

Jelikož se algoritmus skládá z několika částí, lze časovou složitost odvodit takto:

1) Konstrukce Etour je v konstantním čase, neboť každý procesor si zvládne zpracovat potřebné věci sám - *O(c)*.
2) Inicializace vah je taktéž konstantní, poněvadž procesor pouze kontroluje dopřednost hrany - *O(c)*.

3) Suma sufixů má časovou složitost *O(log n).*
4) Korekce podobně jako inicializace spočívá jen v kontrole dopřednosti hrany a následnému přiřazení - *O(c).*

Celková časová složitost je tedy $t(n) = O(c) * O(c) * O(\log n) * O(c) = O(\log n)$

Počet procesů již byl zmíněn a získá se jako $p(n) = 2n - 2 \sim O(n)$

Celková cena algoritmu je tedy $O(n * \log n)$.

Implementace

Algoritmus je implementován v jazyce C++ s využitím knihovny Open MPI. Byly implementovány pomocné funkce `Send()`, `Receive()`, `AllGather()`, `Barrier()` nad MPI funkcemi `MPI_Send()`, `MPI_Recv()`, `MPI_Allgather()` a `MPI_Barrier()` pro snazší práci, ošetření chybových stavů a především redukci boilerplate kódu.

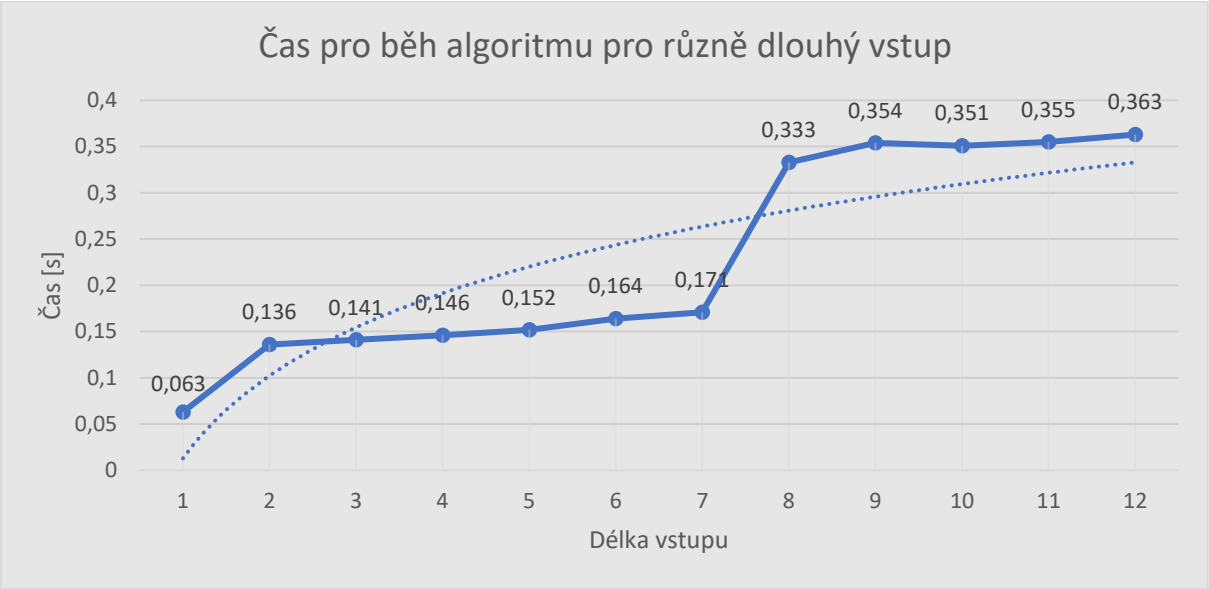
Na začátku programu dojde ke kontrole, zda vstup nemá délku 1. V tomto případě nemá cenu nic počítat a vrátí se opět vstup, neboli kořen. Navíc by pak docházelo k chybám s indexací, kdyby se pokračovalo. Poté se teprve inicializuje MPI. Každý proces reprezentuje jednu hranu a jako první si zjistí, kam vůbec ukazuje (dopředná na následníka, reverzní na předchůdce). Následně se spočítá seznam sousedů (pouze těch, které daný proces potřebuje), Etour a reverzní Etour. K tomu slouží funkce `getNeighbors()`, `getEtour()` a `getReversedEtour()`. Funkce `getEtour()` provádí řádek 2 z algoritmu 1 tak, že přiřadí ob jednoho následujícího souseda (přeskočí svůj reverse) pokud existuje, jinak přiřadí prvního ze sousedů (začátek seznamu). Pro získání reverzní Etour se jen Etour zašle a přijme obráceně. Aby měl každý proces celou Etour (i reverzní) a nejen svou část, volá se `AllGather()`, která ji sesbírá.

Následuje zavolání funkce `preorder()` implementující algoritmus 2. Sudým procesům neboli dopředným hranám se přiřadí váha 1, jinak 0. Následuje suma sufixů, která pro každý proces spočívá v iterativních žádostech na váhu následníka, obdržení odpovědi a přičtení vrácené váhy ke své stávající. Toto se provádí, dokud proces nedojde na konec Etour. Podotkněme, že se proces na konec nedostane později, než za $\log_2$(počet procesů). Poté dojde ke korekci váhy. Oproti algoritmu z přednášky zde indexujeme od 0, tedy nedochází k přičtení +1. Výsledná vrácená váha je již tedy pořadím.

Na závěr jsou váhy pomocí `MPI_Gather()` sesbírány a zprostředkovány master procesu. Toto se odehrává ve funkci `gatherResults()`. Master proces pak vytiskne výsledný řetězec na standardní výstup bez konce řádku.

Experimenty

Program byl spouštěn pro různé délky vstupů a tedy s rozdílnými počty procesorů. Školní server Merlin dovolil maximálně 22 procesů, tedy délku vstupu 12. Měření délky vykonání programu bylo prováděno linuxovým příkazem `time`. Pro každý vstup byl program spuštěn 20x a výsledek zprůměrován. Následující graf udává délku výpočtu v sekundách nad vstupy délek 1-12 a tedy počtu procesorů 1, 2, 4, 6 … 22 (2n – 2). Lze vidět, že navzdory výraznému skoku mezi vstupem délky 7 a 8 (počtem procesorů 12 a 14) se graf drží logaritmického trendu. Onen skok by naopak mohl být vnímán i jako jakási korekce, neboť před ním vývoj grafu trochu zaostává a po něm má naopak tendenci se k trendu blížit.



Závěr

Byl implementován algoritmus přiřazení pořadí preorder vrcholům pro generickou velikost vstupu. Pro řešení bylo zásadní vymyslet komunikaci procesů pro výpočet sumy sufixů. Experimenty byla prokázána časová složitost algoritmu.