Paralelní a distribuované algoritmy – 2. projekt

Ondřej Ondryáš (xondry02), 22. dubna 2023

Algoritmus

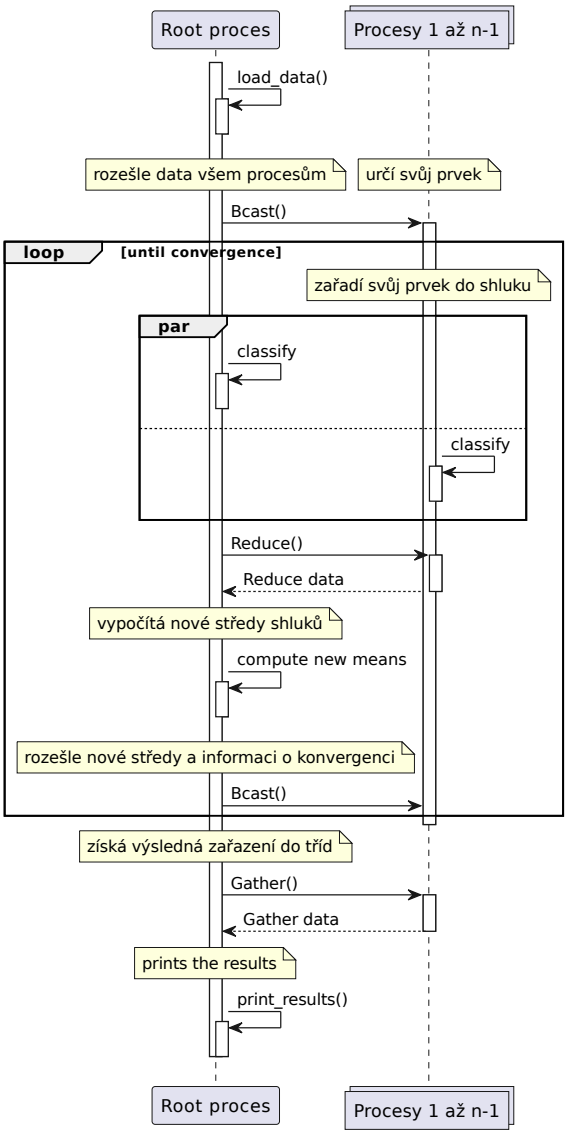
Pro vypracování byl využit algoritmus, který oproti sekvenční verzi paralelizuje hledání středů, kterým jednotlivé prvky vstupu přísluší. Algoritmus je uveden v pseudokódu níže, obrázek 1 obsahuje sekvenční diagram komunikace.

Kořenový proces data načte ze souboru a následně je operací *Broadcast* rozešle všem procesům. Každý proces k chodu potřebuje znát pouze jemu přiřazenou hodnotu a počáteční hodnoty středů, bylo by zde tedy alternativně možné využít operaci *Scatter* pro rozeslání hodnoty a pomocí *Broadcast* distribuovat pouze 4 hodnoty počátečních středů. Asymptotickou časovou složitost celého algoritmu by to však nevylepšilo, zůstal jsem tedy u tohoto řešení.

Každý proces zjistí, do kterého shluku má být jeho hodnota zařazena. Procesy tvoří dvě pole *sums* a *counts* o k prvcích (k je počet shluků, tedy 4). Všechny položky polí jsou na počátku nulové, po nalezení shluku se na pozici v prvním poli odpovídající indexu shluku zařadí hodnota přiřazená procesu, do druhého pole se přiřadí jednička.

Následně procesy provedou pro obě pole redukci s operací sumy. V kořenovém procesu vzniknou pole součtů a pole počtů hodnot pro každý shluk:

```
        sums                 counts
p0:   [0,  0, 10,  0] | [0, 0, 1, 0]
p1:   [0,  0,  9,  0] | [0, 0, 1, 0]
p2:   [0, -3,  0,  0] | [0, 1, 0, 0]
p3:   [0,  0,  0, 90] | [0, 0, 0, 1]
->p0: [0, -3, 19, 90] | [0, 1, 2, 1]
```



Obrázek 1: Sekvenční diagram.

Kořenový proces provede $k = 4$ kroků, ve kterých zjistí novou hodnotu každého středu vydělením položek z obou polí. Přiřadí ji do pole středů, zároveň ověřuje, zda v poli nastala změna. Hodnotu typu bool s touto informací pomocí *Broadcast* šíří všem uzlům. Pokud nějaká změna nastala, kořen dále pomocí *Broadcast* šíří nové středy.

Po ukončení hledání středů je použita operace *Gather*, která z procesů do kořene přenese finální index shluku pro každou vstupní hodnotu.

Algoritmus 1: K-means

```
1:  def K-means():
2:      if rank == 0 then let data = LoadData() else let data = ∅
3:      Broadcast(data) // rozešle hodnoty všem proc.
4:      let means[0..3] = data[0..3], sums[0..3] = ∅, counts[0..3] = ∅
5:      let val = data[rank]
6:      let converged = False
7:      repeat
8:          sums[0..3] = 0, counts[0..3] = 0 // vynuluje pomocná pole
9:          najdi index min_i středu z means nejbližšímu k val
10:         sums[min_i] = val, counts[min_i] = 1 // v poli každého procesu právě jedna obsazená položka
11:         Reduce(sums, SUM) // do root procesu sečte po položkách pole sums a counts
12:         Reduce(counts, SUM)
13:         if rank == 0 then
14:             spočítej nové hodnoty means[i] jako sums[i]/counts[i]
15:             pokud došlo ke změně, nastav converged = True
16:         Broadcast(converged) // rozešle informaci o pokračování výpočtu
17:         Broadcast(means) // rozešle nové středy
18:     until converged
19:     let min_indices[0..n] // posbírá do root procesu informace o příslušnosti hodnot do shluků
20:     Gather(min_i, min_indices)
21:     PrintResults(data, means, min_indices)
```

Složitost

Celkovou **časovou** složitost algoritmu vyjádřeme pomocí výrazu:

$$t(n) = \mathcal{O}(n) + \mathcal{O}(B(n,n)) + i \cdot \mathcal{O}(2R(K,n) + K + B(1,n) + K + B(K,n)) + \mathcal{O}(G(n))$$

kde

- $B(a,b)$ označuje časovou složitost operace *Broadcast* při rozeslání a prvků mezi b procesů,
- $R(a,b)$ označuje časovou složitost operace *Reduce* při rozeslání a prvků mezi b procesů,
- n označuje počet prvků a v souladu se zadáním zároveň počet procesů,
- K označuje počet shluků, tedy podle zadání konstantu 4,
- i označuje horní mez počtu iterací před nalezením výsledku pro daný vstup.

První člen představuje složitost načtení n hodnot ze souboru. Druhý člen představuje složitost operace *Broadcast* na řádku 3. Třetí člen označuje složitost hlavní smyčky, ve které se i-krát provede K porovnání, 2 operace *Reduce* na řádcích 11 a 12, K dělení a *Broadcast* na řádcích 16 a 17. Čtvrtý člen označuje složitost operace *Gather* na řádku 20.

Uvážíme-li, že K je nízká konstanta, a předpokládáme-li, že implementace MPI zvládne provést operace *Broadcast* a *Reduce* pro jeden prvek v čase $O(\log n)$, můžeme prostřední člen zjednodušit:

$$t(n) = \mathcal{O}(n) + \mathcal{O}(B(n,n)) + \mathcal{O}(i \log n) + \mathcal{O}(G(n))$$

V prvním kroku se distribuuje n prvků, s využitím předchozího předpokladu se tedy pro tuto operaci dostáváme na čas $\mathcal{O}(n \log n)$. Dále uvažme, že operaci *Gather* je možné provést přinejhorším v čase $\mathcal{O}(n)$:

$$t(n) = \mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(i \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n) + \mathcal{O}(i \log n) = \mathcal{O}((n + i) \log n)$$

Pro další zpřesnění analýzy by bylo možné rozvést, jaké hodnoty může nabývat člen i, který zřejmě bude možné omezit v závislosti na n, ale jeho přesná hodnota bude záviset také na charakteru vstupních dat. Pro účely srovnání se sekvenční verzí algoritmu na ní však zcela nezáleží, protože paralelizace zde nijak počet iterací algoritmu neovlivní.

Cena algoritmu $c(n)$ je zde rovna $t(n) \cdot n$, tedy:

$$c(n) = n \cdot \mathcal{O}((n + i) \log n) = \mathcal{O}(n^2 \log n + ni \log n)$$

Co se **prostorové** složitosti týče, algoritmus využívá na každém procesoru pole o n prvcích pro uložení vstupu a několik polí o velikosti K pro mezivýpočty a hodnoty středů. Toto je asymptoticky zanedbatelné. Prostorovou složitost dále ovlivní konkrétní implementace MPI operací, nicméně nepředpokládám, že by se dostala dál než za $\mathcal{O}(n)$.

Poznámka pro pousmání: že zadání klade horní mez na n, konkrétně 32. V časové analýze by se tedy dalo tvrdit, že je asymptotická časová složitost algoritmu konstantní, protože při růstu n do nekonečna vždy končí po 33 krocích. Předpokládám však, že toto *zjištění* není cílem tohoto projektu.

Implementační detaily

Program předpokládá existenci souboru `numbers` s obsahem podle zadání. V odevzdané verzi programu jsou hodnoty ze vstupu považovány za znaménkové bajty, změnou hodnoty makra `USE_SIGNED_VALUES` v počáteční části kódu je možné toto chování změnit a interpretovat vstup jako bezznaménkové bajty.

V případě, kdy se hodnota od dvou středů nachází stejně daleko (typicky v případě opakování některé z prvních čtyřech hodnot na vstupu), se pro její zařazení do jednoho ze středů použije pseudonáhodná (tedy de facto nedeterministická) volba.

Diskuze

Časová složitost sekvenčního algoritmu je $\mathcal{O}(niK) = \mathcal{O}(ni)$. Pokud je počet iterací vyšší než $\mathcal{O}(n)$, což je intuitivně očekávatelné, paralelní algoritmus s výše uvedenou časovou složitostí je rychlejší $\mathcal{O}(\frac{n}{\log n})$-krát. Paralelní verze však není optimální, neboť cena je ve srovnání se sekvenčním algoritmem $\mathcal{O}(\log n)$-krát větší (navíc předpokládám, že tento sekvenční algoritmus sám není pro řešení problému optimální).