Cvičení 9

Příklad 1: Důležitou součástí algoritmu Merge-sort je procedura MERGE, která spojí dvě setříděné posloupnosti do jedné setříděné posloupnosti.
Můžeme se na to dívat tak, že tato procedura řeší následující problém:

VSTUP: Dvě setříděné posloupnosti $A = (a_0, a_1, \ldots, a_{n-1})$ a $B = (b_0, b_1, \ldots, b_{n-1})$.

VSTUP: Setříděná posloupnost $C = (c_0, c_1, \ldots, c_{2n-1})$, která vznikne spojením posloupností A a B.

Popišme si jednu z možných implementací procedury MERGE.
Budeme předpokládat, že prvky posloupností jsou indexovány od nuly, tj. první prvek má index 0, druhý 1, atd. Sudé indexy jsou tedy 0, 2, 4, …, a liché indexy 1, 3, 5, …
Pro jednoduchost v tomto popisu budeme také předpokládat, že n je mocnina dvojky.

Procedura MERGE bude pracovat rekurzivně:

- Pokud je $n = 1$, porovná prvky a_0 a b_0 a vrátí setříděnou posloupnost tvořenou těmito dvěma prvky.
- Pokud je $n > 1$, provede následující kroky:
  - Z posloupnosti $A = (a_0, a_1, \ldots, a_{n-1})$ vytvoří posloupnosti A' a A'' délky $n/2$, tvořené prvky se sudými a lichými indexy, tj.

    $$A' = (a_0, a_2, \ldots, a_{n-2}) \qquad A'' = (a_1, a_3, \ldots, a_{n-1})$$

  - Podobně z posloupnosti $B = (b_0, b_1, \ldots, b_{n-1})$ vytvoří posloupnosti B' a B'' délky $n/2$, tvořené prvky se sudými a lichými indexy, tj.

    $$B' = (b_0, b_2, \ldots, b_{n-2}) \qquad B'' = (b_1, b_3, \ldots, b_{n-1})$$

  - Procedura MERGE se zavolá rekurzivně na posloupnosti A' a B' a na posloupnosti A'' a B''. Jako výsledek dostaneme posloupnosti

    $$C' = (c'_0, c'_1, \ldots, c'_{n-1}) \qquad C'' = (c''_0, c''_1, \ldots, c''_{n-1})$$

    kde $C' = \text{MERGE}(A', B')$ a $C'' = \text{MERGE}(A'', B'')$. (Obě tyto posloupnosti budou délky n.)
  - Posloupnosti C' a C'' se spojí do posloupnosti C tak, že prvky posloupnosti C' budou na pozicích se sudými indexy a prvky posloupnosti C'' na pozicích s lichými indexy, tj.

    $$C = (c'_0, c''_0, c'_1, c''_1, \ldots, c'_{n-1}, c''_{n-1})$$

    Posloupnost C má délku $2n$.
  - V posloupnosti C se pro i, kde $1 \leq i < n$, porovnají prvky s indexy $2i - 1$ a $2i$, tj. prvky na pozicích 1 a 2, 3 a 4, …, $2n - 3$ a $2n - 2$.
    Pokud budou porovnané prvky ve špatném pořadí, prohodí se.

Ukažte, že tato varianta procedury MERGE je korektní implementací spojování dvojice posloupností.

Navrhněte její vhodnou paralelní implementaci pro stroj PRAM typu EREW, která s n procesory bude mít časovou složitost $\mathcal{O}(\log n)$.

Příklad 2: Popišme si paralelní algoritmus, který by měl řešit následující problém:

VSTUP: Neorientovaný graf $G = (V, E)$.

VÝSTUP: Rozklad grafu G na souvislé komponenty.

Daný algoritmus bude pracovat na stroji PRAM typu CRCW ARBITRARY.

Předpokládáme, že graf $G = (V, E)$ je zadán jako:

- pole vrcholů V velikosti n
- pole hran E velikosti m — předpokládá se, že každá hrana je v tomto poli uvedena dvakrát, jednou jako (u, v) a jednou jako (v, u). Pořadí hran není důležité.

Algoritmus bude používat následující tři pole velikosti n, jejichž indexy odpovídají vrcholům z množiny V:

- D — indexy rodičů jednotlivých vrcholů ve vytvářených stromech
- D_{prev} — uložení hodnot z pole D z předchozí iterace
- Q — booleovské hodnoty indikující, že pro daný vrchol došlo ke změně

Dále bude použita globální proměnná *changed* typu Bool.
Inicializace:

- pro každý vrchol $v \in V$ se paralelně provede:

```
D[v] := v
```

- procesor 0 provede:

```
changed := TRUE
```

Hlavní cyklus se bude provádět, dokud bude platit *changed* = TRUE.

Jedna iterace hlavního cyklu:

- *Krok 1* — pro každý vrchol $v \in V$ se paralelně provede:

```
Q[v] := FALSE
D_prev[v] := D[v]
```

- *Krok 2* — pro každý vrchol $v \in V$ se paralelně provede:

```
D[v] := D_prev[D_prev[v]]
if D[v] ≠ D_prev[v] then
    Q[D[v]] := TRUE
```

- *Krok 3* — pro každou hranu $(u, v) \in E$ se paralelně provede:

```
if D[u] = D_prev[u] and D[v] < D[u] then
    D[D[u]] := D[v]
    Q[D[v]] := True
```

- *Krok 4* — pro každou hranu $(u, v) \in E$ se paralelně provede:

```
if D[u] = D[D[u]] and Q[D[u]] = False then
    if D[u] ≠ D[v] then
        D[D[u]] := D[v]
```

- *Krok 5* — pro každý vrchol $v \in V$ se paralelně provede:

```
D[v] := D[D[v]]
```

- *Krok 6* — procesor 0 provede:

```
changed := False
```

- *Krok 7* — pro každý vrchol $v \in V$ se paralelně provede:

```
if Q[v] = True then
    changed := True
```

Rozeberte detailně činnost tohoto algoritmu:

a) Ukažte, že tento algoritmus je korektní.

b) Analyzujte jeho časovou složitost.

Příklad 3: Vezměme si následující problém:

VSTUP: Posloupnost přirozených čísel $a_0, a_1, \ldots, a_{n-1}$.

VÝSTUP: Součet $a_0 + a_1 + \cdots + a_{n-1}$.

Tento problém budeme chtít řešit na stroji PRAM typu EREW, který ale bude mít následující omezení. Globální sdílená paměť bude obsahovat vstup uložený v poli A. Toto pole bude pouze pro čtení. Obsah jeho buněk není možné měnit. Kromě buněk, ve kterých bude uloženo vstupní pole A, bude globální sdílená paměť obsahovat jedinou buňku, do které je možné zapisovat a jejíž obsah je možné číst. V jednom okamžiku může k této buňce přistupovat jen jediný procesor. Žádná další sdílená globální paměť není k dispozici.

Procesorů máme k dispozici neomezené množství a každý z nich má k dispozici neomezené množství lokální paměti, do které je možné zapisovat a ze které je možné číst. Rovněž předpokládáme, že všechny procesory mají na začátku výpočtu uloženy ve své lokální paměti délku vstupu n.

Ukažte, že i s takto omezeným strojem PRAM je možné řešit výše uvedený problém v čase, který je asymptoticky lepší než lineární.

Nápověda: Zkuste navrhnout paralelní algoritmus, který při použití $\mathcal{O}(\sqrt{n})$ procesorů bude mít časovou složitost $\mathcal{O}(\sqrt{n})$.

Příklad 4: Navrhněte synchronní distribuovaný algoritmus, který bude implementovat ***prohledávání do šířky*** (BFS — breadth-first search) na grafu komunikační sítě. Komunikační síť je silně souvislý orientovaný graf $G = (V, E)$. V každém vrcholu grafu G běží jeden proces, který může posílat zprávy dalším procesům přes hrany komunikační sítě G. Předpokládejte, že na začátku je jeden z vrcholů označen jako počáteční. Označme jej s.

Cílem je vytvořit strom T, který bude zahrnovat všechny vrcholy grafu G. Kořenem stromu T bude vrchol s a hrany stromu T budou podmnožinou množiny hran grafu G. Navíc pro každý vrchol v v tomto stromě bude platit, že vzdálenost z s do v ve stromě T bude stejná, jako vzdálenost z s do v v grafu G.

Můžete předpokládat, že každý proces má na začátku přiděleno unikátní ID (UID).

Výsledkem běhu algoritmu bude to, že každý proces bude znát UID svého rodiče a svých potomků ve stromě T a vědět, kterými hranami je spojen s jednotlivými potomky.

a) Uvažujte nejprve jednodušší případ, kdy v grafu G jsou jednotlivá spojení mezi vrcholy obousměrná (tj. kdy ke každé hraně (u, v) existuje hrana (v, u) a procesy ve vrcholech u a v vědí, jak jsou tyto dvojice vzájemně si odpovídajících si hran spolu spárovány).

b) Analyzujte časovou a komunikační složitost algoritmu navrženého v předchozím bodě.

c) Navrhněte algoritmus pro obecný případ, kdy G je libovolný silně souvislý graf.

d) Analyzujte časovou a komunikační složitost algoritmu navrženého v předchozím bodě.

Příklad 5: Podobně jako v předchozím příkladě bude i v tomto příkladě úkolem navrhnout synchronní distribuovaný algoritmus, který v grafu komunikační sítě zjistí pro zadaný vrchol s, jak vypadají nejkratší cesty z vrcholu s do všech ostatních vrcholů grafu.

Předpokládejme však nyní, že hrany grafu G mají přiřazeny váhy a délka cesty se nepočítá jako počet hran na této cestě, ale jako součet vah těchto hran.

Můžete předpokládat, že váhy jsou nezáporné, a že na začátku zná každý proces váhy těch hran, které vycházejí z vrcholu, ve kterém se daný proces nachází.

Navrhněte příslušný algoritmus a analyzujte jeho časovou a komunikační složitost.

a) Uvažujte nejprve jednodušší případ, kdy všechny procesy na začátku znají celkový počet vrcholů n.

b) Navrhněte, jak problém řešit v obecném případě, kdy procesy celkový počet vrcholů na začátku neznají.