KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 4 ◇ poznámky k přednášce

8. Bariéry

verze z 2. dubna 2025

1 Paralelní datové algoritmy

Podívejme se nejprve na problém součtu prefixů zadaného vektoru. Konkrétně je dán vektor `v` velikosti `n`. Například:

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Zavedeme si funkci `vsum`, která při aplikaci `(vsum` *`v m n`*`)` vrátí součet prvků vektoru *`v`* od indexu *`m`* po *`n`* (včetně). Chceme vyrobit vektor `sum` délky `n` tak, aby `(= (vref sum i) (vsum v1 0 i))` pro každé `i`.

0	1	2	3	4	5	6	7
1	3	6	10	15	21	28	36

Sekvenční program by měl lineární časovou složitost. Předpokládejme nyní, že máme k dispozici tolik procesorů, kolik je délka vektoru `v1`. S každým prvkem vektoru bude operovat proces nazývaný pracovník. Obdržíme paralelní program řešící zadaný problém:

```
let ((sum (vect n nil))) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (barrier)
  (while (< d n)
    (vset! old i (vref sum i))
    (barrier)
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
    (barrier)
    (set! d (* d 2)))
```

Výraz `(barrier)` implementuje takzvanou **bariéru** sloužící k synchronizaci pracovníků. Konkrétně pracovníci mohu projít bariérou teprve poté, co všichni k bariéře dorazili. Počet kroků výpočtu je `(ceiling (log n 2))`. Tedy výpočet probíhá v logaritmickém čase. Poznamenejme, že invariant cyklu je:

```
(= (vref sum i) (vsum v1 (max (+ (- i d) 1) 0) i))
```

Takto by vypadala simulace běhu algoritmu pro testovací vektor:

d	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8
1	1	3	5	7	9	11	13	15
2	1	3	6	10	14	18	22	26
4	1	3	6	10	15	21	28	36

Bez použití bariéry bychom paralelní datový algoritmus mohli implementovat tak, že by se v cyklu vytvářely pracovníci:

```
loop
  (co-dotimes (i n)
    úkol pro i)
```

Cyklické vytváření procesů by bylo značně neefektivní. Lepší by bylo, jak příklad s paralelním součtem prefixů nastiňuje, synchronizovat jednou vytvořené procesy:

```
pracovník (i n)
loop
  úkol pro i
  čekej, až všech n dokončí úkol
```

Tento druh synchronizace procesů nazýváme **synchronizace bariérou**.

2 Bariéra pomocí počítadla

Jednoduchou bariéru je možné vytvořit pomocí sdíleného počítadla. Označme `n` počet pracovníků, `count` počet pracovníků, kteří dorazili k bariéře, a zaveďme vektor `passed`, kde `(vref passed i)` rozhoduje, zda pracovník `i` prošel bariérou. Správnost řešení zajistíme invariantem `(-> (vref passed i) (= count n))`. Dostáváme hrubé řešení:

```
let ((count 0) (passed (vect n nil)))
pracovník (i n)
úkol pro i
[(inc! count)]
[(await (= count n))
 (vset! passed i t)]
```

Právě uvedená bariéra není znovupoužitelná. Můžeme ji implementovat pomocí složené atomické operace *Fetch-and-add*. Konkrétně pro proměnnou `var` a číslo `num` výraz `(fa var num)` atomicky provede následující:

1. Zjistí hodnotu *val* proměnné *var*,
2. zvýší hodnotu proměnné *var* o *num*
3. a vrátí hodnotu *val*.

Dostáváme jemné řešení bariéry, která ale není znovupoužitelná:

```
(fa count 1)
(await (= count n))
```

První nápad na cestě k znovupoužitelné bariéře je vynulovat počítadlo:

```
let ((count 0))
   pracovník (i n)
 loop
   úkol pro i
   [(inc! count)]
   (await (= count n)
   (set! count 0)
```

To však až k řešení nevede, protože proces, který vynuloval jako první počítadlo, může počítadlo inkrementovat před tím, než jiný proces znovu vynuluje počítadlo. Problém vyřešíme tím, že proces pouze počítadlo dekrementuje:

```
let ((count 0))
   pracovník (i n)
 loop
   úkol pro i
   [(inc! count)]
   (await (= count n))
   [(dec! count)]
   (await (= count 0))
```

Zde může ale nastat problém uváznutí v případě, že někdo dekrementuje počítadlo před tím, než všichni opustí aktivní čekání. Znovupoužitelnou bariéru obdržíme tak, že zavedeme dvě počítadla, jejichž role se budou po každém použití prohazovat:

```
let ((count1 0) (count2 n))
      pracovník (i n)
 loop
   úkol pro i
   [(inc! count1)]
   (await (= count1 n))
   [(dec! count2)]
   (await (= count2 0))
```

Pro jednoduché použití můžeme dát obě role za sebe:

```
let ((count1 0) (count2 n))
      pracovník (i n)
  loop
    úkol pro i
    [(inc! count1)]
    (await (= count1 n))
    [(dec! count2)]
    (await (= count2 0))
    [(dec! count1)]
    (await (= count1 0))
    [(inc! count2)]
    (await (= count2 n))
```

3 Vlajky

Přejdeme na synchronizaci procesů pomocí vlajek. **Vlajkami** `flags` rozumíme vektor logických hodnot. Pokud je `(vref flags i)` *Pravda*, říkáme, že vlajka `i` je **vyvěšena**, jinak říkáme, že je **sejmuta**. Pravidla pro používání vlajek jsou:

1. Ten, kdo čeká na vyvěšení vlajky, ji snímá.
2. Vlajka nesmí být vyvěšena, dokud není jisté, že je sejmuta.

Pro pohodlnější práci s vlajkami si definujeme procedury:

- `(flag-set! flags i)` ... vyvěsí vlajku
- `(flag-clear! flags i)` ... sejme vlajku
- `(flag-set? flags i)` ... rozhodne, zda je vlajka vyvěšena
- `(wait-and-clear! flags i)` ... čeká na vyvěšení vlajky a pak ji sejme:

```
(await (flag-set? flags i))
(flag-clear! flags i)
```

Koordinátor je proces, který řídí pracovníky pomocí vlajek. Budeme potřebovat dvě sady vlajek: `arrive` a `continue`. Podmínka `(flag-set? arrive i)` je splněna, jestliže pracovník `i` dorazil k bariéře. Podobně podmínka `(flag-set? continue i)` je splněna, jestliže pracovník `i` může opustit bariéru. Řešení s jediným koordinátorem:

let ((arrive (vect n nil)) (continue (vect n nil)))	
pracovník (i n)	*koordinátor*

loop
  úkol pro i
  (flag-set! arrive i)
  (wait-and-clear! continue i)
	loop
	  (dotimes (i n)
	  (wait-and-clear! arrive i))
	  (dotimes (i n)
	  (flag-set! continue i))

Problém řešení je, že není symetrické. Lepší řešení obdržíme uspořádáním pracovníků do struktury binárního stromu. Každý pracovník bude koordinátorem svých následníků. Označíme si:

- `(left i)` ... index levého následníka `(+ (* 2 i) 1)`
- `(right i)` ... index pravého následníka `(+ (* 2 i) 2)`

Listy stromu jsou pracovníci, kteří nikoho nekoordinují:

list i
(flag-set! arrive i) (wait-and-clear! continue i)

Kořen je pracovník, který koordinuje své následníky:

kořen i
(wait-and-clear! arrive (left i)) (wait-and-clear! arrive (right i)) (flag-set! continue (left i)) (flag-set! continue (right i))

Konečně vnitřní uzel koordinuje následníky a je koordinován rodičem:

vnitřní uzel i
(wait-and-clear! arrive (left i)) (wait-and-clear! arrive (right i)) (flag-set! arrive i) (wait-and-clear! continue i) (flag-set! continue (left i)) (flag-set! continue (right i))

4 Symetrické bariéry

Nejprve si zavedeme pomocnou proceduru `(wait-and-set!` *flags i*`)`, která čeká na sejmutí vlajky a následně ji vyvěsí. Synchronizaci procesu `i` s procesem `j` definujeme následovně.

proces i
`(wait-and-set! arrive i)`
`(wait-and-clear! arrive j)`

Bariéra pro `n` procesů je rozdělena do `k` fází. V každé fázi se synchronizuje proces s nějakým procesem. Každá fáze má vlastní vlajky. Tedy vlajky jsou `(vect k (vect n nil))`. Fáze číslujeme od nuly.

V **motýlí bariéře** má synchronizace `k` fází. Synchronizujeme `(= n (expt 2 k))` procesů.

Ve fázi `f` se proces `i` synchronizuje s procesem vzdáleným `(= d (expt 2 f))`. Přesněji se ve fázi `f` proces `i` synchronizuje s procesem `(+ i d)`, pokud `(< (mod i (expt 2 (+ f 1))) d)`, `(- i d)`, jinak.

Například:

proces	0	1	2	3	4	5	6	7
fáze 0	1	0	3	2	5	4	7	6
fáze 1	2	3	0	1	6	7	4	5
fáze 2	4	5	6	7	0	1	2	3

V **rozšiřovací bariéře** (**diseminační bariéře**) je `n` procesů synchronizováno v `(= k (ceiling (log n 2)))` fázích. Opět se ve fázi `f` proces synchronizuje s procesem vzdáleným`(= d (expt 2 f))`. Proces `i` ve fázi `f` se synchronizuje s procesem `(mod (- i d) n))`.

Například:

proces	0	1	2	3	4	5
fáze 0	5	0	1	2	3	4
fáze 1	4	5	0	1	2	3
fáze 2	2	3	4	5	0	1

5 Bariéra pomocí semaforů

Bariéru můžeme implementovat pomocí semaforů. Stačí nám umět synchronizovat dva procesy. Pro synchronizaci libovolného počtu procesů můžeme použít verzi motýlí bariéry. Zavedeme proměnné `arrive1` a `arrive2` určující kolikrát proces dorazil k bariéře a proměnné `depart1` a `depart2` určující kolikrát proces opustil bariéru. Náčrtek řešení:

`let ((arrive1 0) (depart1 0) (arrive2 0) (depart2 0))`	
1. pracovník	*2. pracovník*

`loop`
`  úkol pro 1. pracovníka`
`  {(= arrive1 depart1)}`
`  [(inc! arrive1)]`
`  {(= arrive1 (+ depart1 1))}`
`  [(inc! depart1)]`
`  {(= arrive1 depart1)}`
	`loop`
	`  úkol pro 2. pracovníka`
	`  {(= arrive2 depart2)}`
	`  [(inc! arrive2)]`
	`  {(= arrive2 (+ depart2 1))}`
	`  [(inc! depart2)]`
	`  {(= arrive2 depart2)}`

Vynutíme, aby podmínka `(and (<= depart1 arrive2) (<= depart2 arrive1))` byla invariantem. Obdržíme hrubé řešení:

`let ((arrive1 0) (depart1 0) (arrive2 0) (depart2 0))`	
1. pracovník	*2. pracovník*

`loop`
  úkol pro 1. pracovníka
  `{(= arrive1 depart1)}`
  `[(inc! arrive1)]`
  `{(= arrive1 (+ depart1 1))}`
  `[(await (< depart1 arrive2))`
  `(inc! depart1)]`
  `{(= arrive1 depart1)}`
	`loop`
	  úkol pro 2. pracovníka
	  `{(= arrive2 depart2)}`
	  `[(inc! arrive2)]`
	  `{(= arrive2 (+ depart2 1))}`
	  `[(await (< depart2 arrive1))`
	  `(inc! depart2)]`
	  `{(= arrive2 depart2)}`

Zavedeme dva druhy zdrojů určené rovnostmi `(= barrier1 (- arrive1 depart2))` a `(= barrier2 (- arrive2 depart1))`.

`let ((arrive1 0) (depart1 0) (arrive2 0) (depart2 0)` `(barrier1 0) (barrier2 0))`	
1. pracovník	*2. pracovník*

`loop`
  úkol pro 1. pracovníka
  `[(inc! arrive1)`
  `(inc! barrier1)]`
  `[(await (< depart1 arrive2))`
  `(inc! depart1)`
  `(dec! barrier2)]`
	`loop`
	  úkol pro 2. pracovníka
	  `[(inc! arrive2)`
	  `(inc! barrier2)]`
	  `[(await (< depart2 arrive1))`
	  `(inc! depart2)`
	  `(dec! barrier1)]`

Invarianty:

- `(and (<= depart1 arrive2) (<= depart2 arrive1))`,
- `(<= 0 barrier1)`, `(<= 0 barrier2)`

Přejdeme na řízení pomocí zdrojů:

`let ((barrier1 0) (barrier2 0))`	
1. pracovník	*2. pracovník*

`loop`
  úkol pro 1. pracovníka
  `[(inc! barrier1)]`
  `[(await (< 0 barrier2))`
  `(dec! barrier2)]`
	`loop`
	  úkol pro 2. pracovníka
	  `[(inc! barrier2)]`
	  `[(await (< 0 barrier1))`
	  `(dec! barrier1)]`

Nyní stačí použít semafory:

let ((barrier1 (sem 0)) (barrier2 (sem 0)))	
1. pracovník	*2. pracovník*

loop
  úkol pro 1. pracovníka
  (v barrier1)
  (p barrier2)
	loop
	  úkol pro 2. pracovníka
	  (v barrier2)
	  (p barrier1)

Otázky a úkoly na cvičení

1. Napište proceduru `parallel-prefix-sum`, která spočítá paralelně součet prefixů zadaného vektoru. Použijte makro `barrierd-co-dotimes` implementující bariéru. Podívejte se, jak je zde bariéra implementovaná. Například:

```
> (parallel-prefix-sum #(1 2 3 4 5 6 7 8))
#(1 3 6 10 15 21 28 36)
```

2. Napište proceduru, která v logaritmickém čase najde minimum ze zadaného vektoru čísel.
3. Implementujte bariéru pomocí sdíleného počítadla.
4. Implementujte bariéru s jediným koordinátorem.
5. Implementujte bariéru se stromovou strukturou pracovníků.
6. Implementujte motýlí bariéru.
7. Implementujte rozšiřovací bariéru.
8. Pomocí dvou vektorů `cars` a `cdrs` stejné délky můžeme reprezenotvat seznamy. Každý index odpovídá tečkovému páru. *Car* páru `i` je přímo prvek na indexu `i` vektoru `cars`. *Cdr* páru `i` je dán prvkem vektoru `cdrs` na indexu `i`: buď je to pár na zadaném indexu, nebo `nil`. Například vektory `#(1 2 3 4 5 6 7 8)` a `#(4 5 nil 6 2 0 nil 3)` reprezentují dva seznamy: `(2 6 1 5 3)` a `(8 4 7)`. Napište proceduru `find-ends`, která v logaritmickém čase pro vektor `cdrs` vrátí vektor indexů posledních párů seznamů. Například:

```
> (find-ends #(4 5 nil 6 2 0 nil 3))
#(2 2 2 6 2 2 6 6)
```