Paralelní počítání
aneb stokrát nic spočítalo výsledek

(verze 1.2 z 2021-05-13)

0. Paralelní počítače

Vývoj výpočetní techniky jde stále mílovými kroky kupředu. Už nyní je ale velmi náročné zvyšovat počet operací, které jeden procesor za jednotku času zvládne, a hrozí, že v ne až zas tak vzdálené budoucnosti narazíme na fyzikální limity (možná se to může zdát překvapivé, ale existují). Je zde ale jednoduché řešení – pořídíme si víc procesorů. A nebudeme se nijak uskromňovat, nezůstaneme u dvou, čtyř, nebo třeba u tisíce; počet procesorů, které budeme používat, bude závislý na velikosti problému, který budeme řešit.

Každý procesor bude umět zhruba to, co byste od běžného procesoru čekali (více v technické odbočce dále). Při víceprocesorovém počítání ale narazíme na několik problémů. Prvním z nich je synchronizace. Tu vyřešíme jednoduše: Naše procesory budou všechny vykonávat ten samý program, a dokonce budou v jeho vykonávání vždy na tom samém místě. To například znamená, že je-li v našem programu konstrukce typu `if podmínka then ... else ...`, nejprve budou procesory, které vyhodnotily `podmínku` jako nesplněnou, čekat na ty, které ji vyhodnotily jako splněnou, než dokončí větev `if`, a pak budou naopak ty, které vyhodnotily `podmínku` jako splněnou, čekat, až ostatní dokončí vykonávání větev `else`. Podobně máme-li v programu cyklus, procesory, které ho opustí dříve, čekají, dokud ho neopustí všechny.

Jenže ups, to by pak naše procesory nebyly o nic lepší, než jeden jediný procesor, vždyť by dělaly vždy všechny přesně to samé. Proto řekneme, že každý procesor obsahuje (read-only) registr `id`, v němž je napsané jeho číslo. Dále má každý procesor ještě svoji vlastní paměť, v níž může mít data, ke kterým má přístup pouze on. Domluvíme se, že lokální proměnné budeme pojmenovávat pouze malými písmeny, kdežto proměnné ve společné paměti všech procesorů budeme pojmenovávat velkými písmeny.

Takže například následující funkce přičte ke všem prvkům pole jedničku: každý procesor si vezme na starost jednu buňku. Zvládnou to v konstantním čase, ale potřebujeme tolik procesorů, kolik je prvků pole.

```
def PrictiJedna(POLE):
    a = POLE[id]
    a = a + 1
    POLE[id] = a
```

Totéž bychom samozřejmě mohli napsat i bez proměnné `a` na jediném řádku.

V praxi se ale většinou nesetkáme s problémy, které jdou vyřešit na každém prvku vstupu zvlášť, bez toho, aby si procesory musely předávat mezivýsledky. To nás přivádí k druhému problému, co kdyby se víc procesorů pokoušelo současně přistupovat k tomu samému místu v paměti? Proto zavedeme 4 modely, ve kterých budeme úložky řešit.

- **EREW** neboli Exclusive Read Exclusive Write. V tomto modelu náš program zahlásí chybu (a počítač se se skřípěním zastaví), pokud se dva procesory pokusí současně přistoupit k téže buňce paměti.
- **CREW** neboli Common Read Exclusive Write. V tomto modelu je povoleno, aby více procesorů četlo z jedné buňky, nikdy však nesmí více procesorů do jedné buňky zapisovat.
- **CRCW Common.** V tomto modelu je povoleno, aby více procesorů četlo z jedné buňky, a do jedné buňky smí i více procesorů současně zapisovat. Má to ale háček. Pokud se pokusí víc procesorů současně zapisovat do jedné buňky, musí se všechny shodnout – tedy zapisovat to samé číslo. **Pokud se více různých procesorů současně pokusí zapsat různá čísla do téže buňky, tak nejen, že zápis neproběhne, ale celý program spadne.**
- **CRCW Priority.** V tomto modelu mohou z jedné buňky procesory libovolně číst i do ní zapisovat. Pokud se dva procesory pokusí současně zapsat do téže buňky, zůstane v ní to, co tam zapsal procesor s nejnižším `id`.

Technická odbočka

Pokud máte pocit, že chápete, co by měly naše procesory umět, nebo znáte výpočetní model RAM, můžete tuto sekci s klidem přeskočit. Jinak zde ale v krátkosti vysvětlíme, co všechno by měl počítač umět.

V našich problémech budeme potřebovat pouze tři typy proměnných: číselné, logické a pole. Číselná proměnná je proměnná, do níž se vejde přirozené číslo velké maximálně polynomiálně ve velikosti vstupu. To znamená, že například součet všech čísel na vstupu se vám do jedné buňky vejde, jeho druhá, třetí, či libovolná jiná konstantní mocnina také, ale třeba součin všech čísel na vstupu už nemusí. Logická proměnná je prostě „Ano“ nebo „Ne“. Pole může obsahovat buď číselné nebo logické položky a může být jednorozměrné i vícerozměrné.

Počítač umí vyhodnocovat výrazy. Co to je výraz? Samotné číslo je rozhodně výraz. Také samotné označení číselné proměnné je výraz. Pokud máme dva výrazy, jejich součet, rozdíl, celočíselný podíl, modulo i součin je také výraz. Pokud `p` je

pole, `p[výraz]` je také výraz (značící `výraz`-tou položku pole `p`). Nakonec umí i bitové operátory `AND`, `OR`, `XOR`, `>>` a `<<`. Operátory `>>` a `<<` jsou bitové posuvy, tedy `a<<b` je $a \cdot 2^b$ a `a>>b` je $\lfloor a/2^b \rfloor$, kde `a` i `b` jsou výrazy. `AND`, `OR` a `XOR` aplikují příslušný logický operátor na číslo po bitech, tedy 1100_2 `AND` 1010_2 `=` 1000_2, 1100_2 `OR` 1010_2 `=` 1110_2 a 1100_2 `XOR` 1010_2 `=` 0110_2. Opět mohou být oba argumenty těchto operátorů libovolné výrazy.

Počítač umí vyhodnocovat i logické výrazy. Samotné „Ano“ nebo „Ne“ jsou logické výrazy, stejně jako logické proměnné. Pokud máme dva výrazy, můžeme je porovnat pomocí operátorů $=$, $\neq$, $<$, $>$, $\leq$ i $\geq$ a dostaneme logický výraz. Máme-li logický výraz, můžeme ho znegovat a dostaneme logický výraz. Dále pokud máme dva logické výrazy, můžeme je spojit pomocí logických spojek `AND`, `OR` nebo `XOR` a dostaneme opět logický výraz.

V obou typech výrazů smíme samozřejmě používat závorky.

Počítači můžeme napsat funkci a tu následně vyhodnotit. Pokud nadefinujeme funkci `f(x)`, můžeme ji později nechat vyhodnotit. Samozřejmě si můžeme pořídit i funkci o více parametrech. Pokud má funkce návratovou hodnotu, je `f(x)`, kde `x` může být výraz, logický výraz i pole, samo o sobě výraz. Funkci můžeme volat z našeho programu i opakovaně. Počítač sám od seme umí funkci `délka(pole)`, která jako parametr přijímá pole a v konstantním čase vrátí jeho délku.

Počítač umí vyhodnotit přiřazovací příkaz `a=b`, kde `a` je buď číselná proměnná nebo `p[výraz]` pro pole `p`, a `b` je libovolný výraz. Dále umí konstrukci typu `if (logický výraz) then ... else ...` a konstrukci typu `while (logický výraz) do ...`. Protože většinou nebudeme psát celé programy, ale pouze funkce, vstup a výstup nebudeme řešit, vstup budou prostě parametry funkcí a výstup jejich návratové hodnoty. Ve funkci můžeme použít `return a`, který ukončí vykonávání funkce a vrátí `a` jako návratovou hodnotu (za každý procesor zvlášť). Opět všechny procesory čekají, dokud poslední z nich vykonávání funkce neukončí.

Počítač si umí pořídit novou proměnnou, jak lokální, tak sdílenou, a to jakéhokoli typu. Pro zjednodušení budeme předpokládat, že i pole (libovolné velikosti) si umíme pořídit v konstantním čase.

K závěru této odbočky ještě poznámka k modelům – nikde se nepíše, zda můžeme z jedné buňky v témže kroku číst i do ní zapisovat. To povolíme ve všech modelech s tím, že všechna čtení proběhnou před všemi zápisy.

1. První sada úložek

V této sekci se budeme snažit vyřešit naše problémy primárně co nejrychleji a sekundárně pomocí co nejmenšího počtu procesorů ve všech čtyřech modelech. Nemůžeme si však dovolit úplně libovolné množství procesorů. Později si ukážeme, že v takovém případě by šel téměř jakýkoli problém vyřešit v konstantním čase. Bude nám muset stačit polynomiálně mnoho procesorů – tedy musí existovat polynom P takový, že pokud máme vstup délky n, budeme potřebovat $P(n)$ procesorů. Naopak ještě podotkneme, že bychom se měli snažit hledat algoritmy rychlejší, než lineární.

Počet procesorů, které má naše funkce k dispozici v závislosti na délce vstupu, je součástí její specifikace (do řešení to můžete prostě napsat před váš kód). Počet procesorů, který bude na daný vstup potřeba, by ale měl jít z jeho délky spočítat jedním procesorem v konstantním čase. Formálně si to můžeme představit tak, že se na začátku naší funkce spustí jen jeden procesor, a ten si nejprve řekne, kolik procesorů se má spustit, a až potom začne samo vykonávání vaší funkce.

Vstup i výstup si budeme předávat v proměnných ve sdílené paměti. Pokud je výstupem jediné číslo, budeme ho zapisovat do proměnné `RESULT`. Všimněte si, že vracet ho jako výsledek z funkce `return`em nefunguje, protože tehdy může každý procesor vracet jiné číslo.

Úkol 1.1: Na vstupu máte pole `P`. Zjistěte, zda se v něm nachází jednička.

Úkol 1.2: Na vstupu máte pole `P` jedniček a nul. Najděte nejnižší index `i` takový, že `P[i]=1`.

Úkol 1.3: Na vstupu máte pole čísel `P`. Spočítejte jejich součet.

Úkol 1.4: Na vstupu máte pole čísel `P`. Spočítejte hodnotu jejich minima.

Úkol 1.5: Na vstupu máte dvě binární čísla (zadaná jako pole jedniček a nul s tím, že jejich nultá buňka obsahuje cifru na místě jednotek). Tato dvě čísla porovnejte na nerovnost (tj. vraťte „Ano“, pokud je první větší, jinak vraťte „Ne“).

Úkol 1.6: Opět jsou dána dvě binární čísla. Sečtěte je (s výsledkem vráceným podobně jako se vstupem po cifrách v poli).

Úkol 1.7: A dotřetice dvě binární čísla. Vynásobte je.

Dál pokračujte až poté, co vyřešíte alespoň část úloh z této kapitoly.

2. Intermezzo o práci

Jak jste si asi všimli, problémy umíme v našem modelu řešit opravdu rychle. Co kdyby ale někdo naše paralelní programy pustil na normálním stroji s jedním procesorem (tak, že by procesor nejprve vykonal první krok každého vlákna, pak druhý a tak dále). Oj, to bychom se na výsledek načekali. Mohli bychom zkusit psát naše programy tak, aby byly rychlé v paralelním světe, a přitom (alespoň asymptoticky) použitelné i ve světě běžném, sekvenčním. Proto nadefinujeme práci, pomocí které budeme měřit, jak dobrý by algoritmus byl, kdyby se pustil sekvenčně.

Definice: *Práce,* kterou algoritmus vykoná, je součin celkového počtu procesorů, které využije, a počtu instrukcí, než poslední z nich doběhne a algoritmus vrátí výstup.

To znamená, že například přímočarý paralelní algoritmus na výpočet součtu pole, vykoná práci $\mathcal{O}(n \log n)$, a to přesto, že vlastně pouze v prvním kroku dělají všechny procesory něco smysluplného, a kdybychom počítali čas pouze přes procesory, které zrovna něco dělají, dostali bychom $\mathcal{O}(n)$.

Úkol 2.1: Na vstupu máte pole čísel `P`. V modelu CREW zjistěte hodnotu jeho minima v logaritmickém čase, ovšem pouze s lineární prací.

Úkol 2.2: Řešte předchozí úlohu v modelu CRCW Common. Zkuste najít algoritmus rychlejší než logaritmický (byť ne nutně konstantní), který ale použije pouze lineárně mnoho procesorů.

3. Druhá sada úložek

Jak jste si na první sadě úložek nejspíše vyzkoušeli, model EREW je typicky stejně silný jako CREW, jen je mnohem otravnější. Proto ho nadále nebudeme zkoumat. Podobně CRCW Priority je obvykle stejně rychlý jako CRCW Common, a navíc jsou v něm řešení často méně zajímavá. Proto řešte následující úložky pouze v modelech CRCW Common a CREW.

Omezení z první řady úložek (polynomiálně mnoho procesorů, potřebný počet procesorů spočitatelný v konstantním čase) stále platí.

Úkol 3.1: Na vstupu máte pole `P` čísel. Setřiďte jej. Pokud vám to pomůže, můžete předpokládat, že čísla v poli jsou po dvou různá. Výsledek můžete vrátit rovnou v původním poli.

Úkol 3.2: Na vstupu máte pole `P` čísel délky `n`. Spočítejte jeho pole prefixových součtů, tedy pole `S` délky `n` takové, že pro každé `k<n` platí

$$\texttt{S[k]} = \sum_{\texttt{i}=0}^{\texttt{k}} \texttt{P[i]}.$$

Úkol 3.3: Na vstupu máte graf zadaný pomocí matice sousednosti (tj. máte dvojrozměrné pole `E` takové, že `E[i,j]` je jednička, právě když mezi vrcholy `i` a `j` je hrana). Zjistěte, zda je graf souvislý.

4. Dost bylo praxe

Na předchozích úložkách jste si osahali, jak modely fungují. Teď by bylo pěkné o nich dokázat něco obecného.

Úkol 4.1: Zkuste nějak dát do souvislosti rychlosti programů v jednotlivých modelech. Tvrzení, která hledáme, zhruba zapadají do následující šablony:

„Pokud model ____ vyřeší nějaký problém s n procesory v čase t, pak ho model ____ umí vyřešit v čase $f(t,n)$ s $g(t,n)$ procesory, kde $f(t,n) =$ ____ a $g(t,n) =$ ____."

Úkol 4.2: Dokažte, že vaše řešení úkolu 1.1 na CREW je nejrychlejší možné (asymptoticky).

5. Myslíte, že $\Theta(\log n)$ bylo rychlé?

Jak jsme vám v první kapitole slíbili, na závěr se podíváme, co by se stalo, kdybychom mohli mít i exponenciálně mnoho procesorů. Pro potřeby těchto úložek je nutné mírně přiohnout model. V technické odbočce jsme se zmínili, že do jedné číselné buňky se vejdou čísla pouze polynomiálně velká vzhledem k velikosti vstupu. Jenže v takovém případě by při exponenciálně mnoha procesorech mnohé z nich nedokázaly počítat se svým vlastním `id`. Povolíme tedy v číselných buňkách libovolně velká čísla.

Protože práce s libovolně velkými vstupy je často nepohodlná, můžete tam, kde se vám to bude hodit, předpokládat, že velikost vstupu je mocnina dvojky (mj. proto, že spočítat na jednom procesoru a^n trvá pro obecné a $\mathcal{O}(\log n)$ času, ale pro a mocninu dvojky to zvládneme v konstantním čase pomocí operátoru `<<`). Rozmyslete si také, že na našem paralelním počítači umíme v konstantním čase spočítat horní celou část dvojkového logaritmu čísla.

Úkol 5.1: Na vstupu máte číslo zadané po bitech v poli podobně jako v úkolu 1.5. V modelu CRCW Common spočtěte jeho hodnotu v konstantním čase. Pokud n je délka pole, použijte $n \cdot 2^n$ procesorů.

Dál pokračujte až poté, co si alespoň zkusíte rozmyslet předchozí úkol.

K předchozímu úkolu drobná poznámka. Možná jste si všimli, že počet procesorů není optimální, a to ani asymptoticky. Smutnou zprávou ale je, že (dokud zůstáváme u exponenciálního množství procesorů) nic takového jako asymptoticky nejnižší množství procesorů nutné na vyřešení problému v čase $\mathcal{O}(1)$ neexistuje. Pokud máme program který běží v čase $t(n)$ s $p(n)$ procesory, můžeme jistě navrhnout program, který poběží v čase $t(n) + \mathcal{O}(1)$ s $2p(n/2)$ procesory tak, že $p(n/2)$ procesorů vyřeší spodní půlku čísla, jiných $p(n/2)$ vyřeší horní půlku, a pak nějaký procesor horní půlku vynásobí příslušnou mocninou dvojky a obě půlky sečte. Jenže pro (super)exponenciální funkce $2p(n/2) \notin \Theta(p(n))$, například v našem případě jsme zmenšili počet procesorů z $n \cdot 2^n$ na $n \cdot 2^{n/2} = n \cdot \sqrt{2}^n$, což je zjevně zlepšení o víc než o konstantu.

Ale ouha, kdybychom toto zlepšení uplatnili rekurzivně, až dokud se nedostaneme na $n = 1$, sice jsme snížili počet procesorů až na $2n$, ale všechna $\mathcal{O}(1)$, o která se nám při každém kroku zhoršila časová složitost, se nám posčítala na $\mathcal{O}(\log n)$ a algoritnus tedy již není konstantní. Závěrem by mohlo tedy být, že umíme problém 5.1 řešit v konstantním čase a s počtem procesorů $n \cdot a^n$, kde a je libovolné reálné číslo ostře větší než jedna. K čemu jsme však chtěli dojít, je, že ne ve všech problémech dává dobrý smysl snažit se asymptoticky optimalizovat počet procesorů.

Stejně jako předchozí otázku, i všechny další budeme řešit v modelu CRCW Common.

Úkol 5.2: Na vstupu máte graf (opět zadaný maticí sousednosti). Zjistěte, zda je souvislý.

Úkol 5.3: Na vstupu máte pole přirozených čísel menších než k. Spočítejte jeho součet. Požadovaný počet procesorů může být závislý nejen na délce pole, ale i na k.

Úkol 5.4: Na vstupu máte graf a dva jeho vrcholy. Spočtěte jejich vzdálenost (počet hran na nejkratší cestě).

Úkol 5.5: Vhodně doplňte a následně dokažte následující větu: Mějme libovolný problém, který umíme na stroji s jedním procesorem vyřešit v čase $t(n)$ a všechny mezivýsledky jsou velké nejvýše $v(n)$, kde t a v jsou funkce spočítatelné v konstantním čase na jednom procesoru. Pak tento problém umíme vyřešit v modelu CRCW v konstantním čase s ____ procesory.