Jan Vybíral, xvybir05

Dokumentace ke 2. projektu do PRL: Implementace algoritmu “Enumeration sort”

1. Princip algoritmu a implementace

Algoritmus jsem implementoval na základě tohoto pseudokódu z přednášek:

```
1) for i = 1 to n do in parallel
        C_i = 1
   endfor
2) for k = 1 to 2n do
        if k ≤ n then h = 1 else h = k - n endif
        for i = h to n do in parallel
          if (X_i nonempty and Y_i nonempty) and X_i > Y_i then C_i = C_i + 1 endif
        endfor
        for i = h to n - 1 do in parallel
          if Y_i nonempty then Y_{i+1} = Y_i endif
        endfor
        if k ≤ n then Y_1 = nextinput, X_k = nextinput endif
        if k > n then Z_{Ck-n} = X_{k-n} endif
   endfor
3) for k = 1 to n
        output = Z_n
        for i = k to n-1 do in parallel
          Z_{i+1} = Z_i
        endfor
   endfor
```

Moje implementace se od uvedeného pseudokódu liší jen v několika drobnostech:

Pro seřazení *n* vstupních hodnot používám *n+1* procesorů, přičemž první z nich slouží jako řídící a ostatní jsou “pracovní”.
Řídící procesor provádí načítání hodnot ze vstupního souboru, tyto hodnoty pak zasílá odpovídajícím procesorům, které je ukládají do svých příslušných registrů (předposlední řádek druhého cyklu ve výše uvedeném pseudokódu).
Řídící procesor po načtení každé hodnoty ještě před jejím zasláním daným procesorům tuto hodnotu nejprve vytiskne na výstup, a poté ji vynásobí hodnotou *10000* a potom přičte unikátní konstantu v rozmezí od *0* do *9999*.
Tím je zajištěno, že pokud na vstupu nebude více než *10000* hodnot, bude každá hodnota po této úpravě unikátní a algoritmus tak bude fungovat i pokud jsou na vstupu nějaké duplicitní hodnoty. Původní algoritmus z přednášek totiž není schopen řadit stejné hodnoty.
Poslední cyklus ve výše uvedeném pseudokódu posílá hodnoty na výstup v opačném pořadí než je požadováno v zadání. Hodnoty jsou proto zasílány řídícímu procesoru, který si je ukládá v opačném pořadí do pole, které je alokováno pouze pro řídící procesor. Řídící procesor také převede hodnoty zpět do původní podoby tím, že je vydělí hodnotou *10000*.
Úplně nakonec řídící procesor vytiskne seřazenou posloupnost při průchodu polem.

2. Komunikační protokol

Procesory jsou číslovány od *0*, přičemž řídící procesor má rank *0*.

V první části hlavního cyklu (*k* mezi *1* až *n*) se komunikuje takto:

Všechny procesory s rankem *1* až *n-1* pošlou hodnotu svého registru *Y* procesoru s o jedničku vyšším rankem.
Všechny procesory s rankem *2* až *n* přijmou hodnotu od svého souseda a pokud se nejedná o prázdnou hodnotu (v programu reprezentováno konstantou *-1)* uloží ji do svého registru *Y*.
Řídící procesor přečte další hodnotu ze vstupního souboru, upraví ji způsobem popsaným výše a zašle výsledek procesorům s ranky *1* a *k.*
Procesor s rankem *1* uloží přijatou hodnotu do svého registru *Y.*
Procesor s rankem *k* uloží přijatou hodnotu do svého registru *X.*

V druhé části hlavního cyklu (*k* mezi *n+1* až *2n*) se komunikuje takto:

Všechny procesory s rankem *h* až *n-1* pošlou hodnotu svého registru *Y* procesoru s o jedničku vyšším rankem.
Všechny procesory s rankem *h+1* až *n* přijmou hodnotu od svého souseda a pokud se nejedná o prázdnou hodnotu (v programu reprezentováno konstantou *-1)* uloží ji do svého registru *Y*.
Procesor s rankem *k-n* zašle všem ostatním s pomocí broadcastu svoje hodnoty z registrů *C* a *X.*
Procesor s rankem shodným s hodnotou registru *C* procesoru *k-n* uloží obdrženou hodnotu registru *X* procesoru *k-n* do svého registru Z.

V posledním cyklu v části 3) algoritmu pošle procesor s rankem *n* hodnotu svého registru *Z* řídícímu procesoru. Ten tuto hodnotu přijme a uloží ji na správnou pozici do výsledného seřazeného pole.
Procesory s ranky *k* až *n-1* pošlou hodnotu svého registru *Z* procesorům s ranky o jedničku vyššími. Ty tyto hodnoty uloží do svých registrů *Z.*

3. Analýza složitosti

Časová složitost mé implementace je stejná jako časová složitost uvedeného pseudokódu. Krok 1) je v konstantním čase, krok 2) trvá *2n* cyklů a krok 3) trvá *n* cyklů, takže celková časová složitost je

$$t(n) = O(n)$$

Počet procesorů potřebných pro řešení úlohy v mojí implementaci je *n+1*, takže

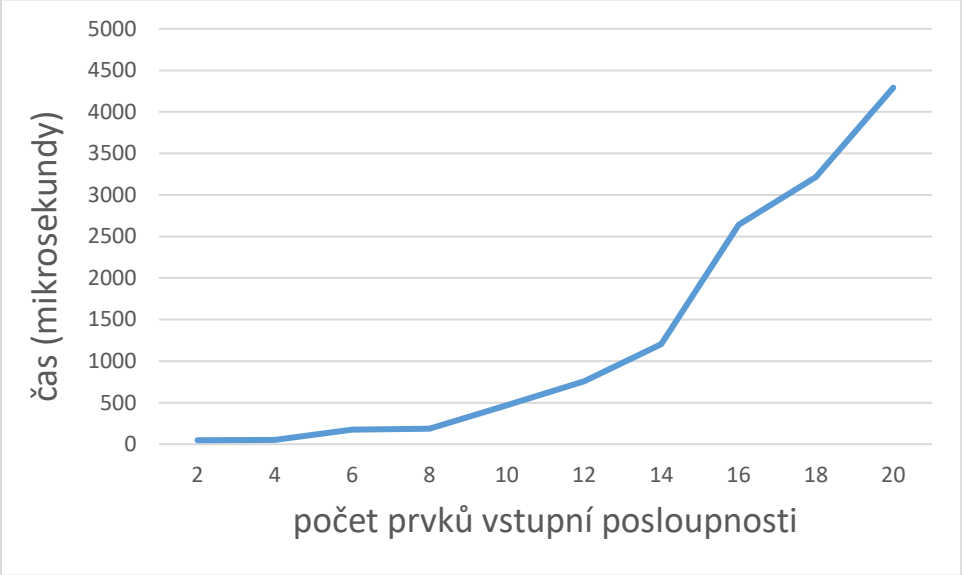
$$p(n) = O(n)$$

Celková cena implementace je tedy

$$c(n) = p(n) * t(n) = O(n) * O(n) = O(n^2).$$

4. Experimentální ověření časové složitosti

Čas potřebný k vykonání implementovaného algoritmu jsem měřil na serveru *merlin* za použití nástrojů obsažených v hlavičkovém souboru *<chrono>*, které mimo jiné umožňují měřit čas uplynulý mezi libovolnými dvěma místy v programu v mikrosekundách. Do měření nebylo zahrnuto načítání vstupů ze souboru, tisknutí výstupu, inicializace knihovny *OpenMPI* a alokace pole pro uložení výstupních hodnot. Pro každou velikost vstupu bylo provedeno 10 měření a takto získané hodnoty byly poté zprůměrovány. Následující graf znázorňuje celkový výsledek testování:



Z grafu je vidět, že očekávané lineární složitosti příliš neodpovídá. Dokonce se zdá, že čas výpočtu algoritmu může růst exponenciálně. To může být způsobeno např. tím, že byl algoritmus testován pouze na jednom procesoru, testované vstupní posloupnosti nebyly dost dlouhé (zde bylo omezení serveru *merlin*, který umožňuje vytvořit jen zhruba 20 procesů) nebo se může jednat o chybu měřících funkcí nebo dokonce chybu či neefektivitu v implementaci, které si nejsem vědom.

5. Závěr

Implementoval jsem paralelní verzi algoritmu *Enumeration Sort* a provedl měření její časové složitosti. Zjistil jsem, že výsledek zcela neodpovídá očekávané lineární složitosti, což může být způsobeno řadou faktorů.