Paralelní a distribuované algoritmy - 2018/2019
Bucket sort

Petr Nodžák
xnodza00@stud.fit.vutbr.cz

30. března

1 Popis algoritmu

Bucket sort je jeden z paralelních algoritmů, který pracuje na stromové architektuře, pro n prvků potřebuje $2 * log_2(n) - 1$ procesorů, z nich je právě m procesorů listových $m = log_2(n)$ a každý listový procesor obsahuje n/m prvků. Každý nelistový procesor má 2 potomky a spojuje jejich seřazené posloupnosti do jedné.

Algoritmus se skládá ze 4 kroků, v 1. kroku pracuje jen kořenový procesor, načítá a rozesílá části vstupní posloupnosti listovým procesorům. Ve 2. kroku je provedeno seřazení posloupnosti každým listovým procesorem. 3. krok obnáší spojení dvou seřazených posloupností od potomků, v posledním kroce je provedeno spojení do jedné seřazené posloupnosti.

- **Kořenový procesor** v počáteční fázi načítá hodnoty ze vstupu a rovnoměrně rozešle prvky na jednotlivé listové procesory. V závěrečné fázi čeká na seřazené posloupnosti od potomků, ty spojí do výsledné posloupnosti a vypíše. (krok 1 a 4)
- **Listový procesor** čeká na prvky od kořenového procesoru, poté je seřadí optimálním sekvenčním algoritmem a odešle rodiči. (krok 2)
- **Nelistový procesor** čeká na prvky od obou potomků, až je oba přijme, začne spojovat dvě seřazené posloupnosti optimálním sekvenčním algoritmem do jedné. (krok 3)

1.1 Složitost algoritmu

n - počet vstupů
m - počet listových procesorů

- **1. krok:** kořenový procesor postupně načítá a rozesílá hodnoty ze vstupu listovým procesorům $t(n) = O(n)$

- **2. krok:** každý listový procesor seřadí svoji posloupnost heap sortem, který má časovou složitost $O(r*logr)$ a $r = n/log(n)$, tedy časová složitost kroku 2 je $t(n) = O(n/log_2(n) * log_2(n/log_2(n))) = O(n)$
- **3. krok:** každý nelistový procesor, kromě kořenového, spojí seřazené posloupnosti svých potomků do jedné seřazené spoloupnosti, na i-té úrovni stromu to bude $O(2^{log_2(m)-i} * (n/m + 1))$ a pro všechny úrovně:
$$\sum_{i=1}^{log_2(m)-1} 2^{log_2(m)-i} * (n/m + 1) = O(n)$$
- **4. krok:** kořenový procesor spojí spojí dvě posloupnosti o délce $n/2$ tedy časová složitost posledního kroku je $O(n)$

Časová složitost u výše uvedených kroků nepřesahuje $O(n)$, potom teda $t(n)$ algoritmu je $t(n) = O(n)$ a prostorová složitost $p(n)$ je $p(n) = O(log_2(n))$, potom se jeho cena $c(n)$ rovná:
$c(n) = t(n) * p(n) = O(n) * O(log_2(n)) = O(n * log_2(n))$

2 Implementace

Algoritmus byl implementovaný v jazyce c++ za pomocí knihovny Open MPI.

Stromová architektura je reprezentovaná jako pole procesorů, které jsou vhodně indexované (rodič i, potomci $i * 2 + 1$ a $i * 2 + 2$), tím vzniká pomyslná stromová architekruta.

Ve skriptu *test.sh* probíhá veškerý **výpočet počtu procesorů**, který bude program potřebovat. Bylo nutné ošetřit krajní podmínky a to pro malé a nevalidní n. Pokud $n <= 0$, program se vůbec nespustí, pro $n <= 3$ je programu přidělen jen jeden procesor a nakonec pro $n > 3$ je spočítán počet procesorů jako $pocet\_proc = 2*(log(n)/log(2))-1$, vše za pomocí *bash calculatoru* (pro výpočet logaritmu), a jednoduché aritmetiky. Samotné rozdělení na listové a nelistové procesory probíhá již v programu *bks.cpp*

Kořenový procesor postupně načítá, odesílá pole $n/log_2(n)$ prvků každému listovému procesoru a vypisuje přečtenou posloupnost.

Listové procesory začínají na indexu, vypočítaném jako celkový počet procesorů vydělený dvěma $numprocs/2$. Listové procesory seřadí přijaté pole prvků heap sortem. Pokud je procesor P_i sudý (resp. lichý), jeho rodič je $P_{(i-1)/2}$ (resp. $P_{i/2}$).

Analogicky, co není listový procesor, to je **nelistový**, tedy procesory s indexem menším než $numprocs/2$. Nelistový procesor P_x čeká na seřazenou posloupnost od pravého P_{x*2+2} a levého P_{x*2+1} potomka, po přijetí spojí obě posloupnosti do jedné a pokud není procesor kořenový, odešle posloupnost svému rodiči.

Pokud **kořenový procesor** příjme posloupnosti od svých potomků, spojí je a postupně výpíše na výstup.

3 Experimenty

Algoritmus byl testován na stroji Merlin. K získání doby běhu algoritmu byla použita funkce *clock_gettime()* a její makro *CLOCK_PROCESS_CPUTIME_ID*, které zajistí měření procesorového času. Měření spouští kořenový procesor na začátku algoritmu a ukončuje po spojení posledních posloupností.

n	1	4	8	16	256	1024	8192	16383	16384	32767
t[ms]	0,13	0,25	0,3	0,34	0,68	1,09	1,86	2,93	2,15	3,65

Table 1: Vybrané naměřené hodnoty

Pro více vstupů než 32 768 nebylo možné algoritmus spustit, protože Merlin nedovolil program spustit s více jak 27 procesory. Tudíž se testovalo jen v rozsahu $< 1, 32767 >$. Pro různé vstupy bylo provedeno 100 běhů, z toho nebylo bráno v úvahu 5 nejhorších a 5 nejlepších běhů. 90 běhů bylo zprůměrováno a vyneseno do grafu 1.

Figure 1: Naměřená časová složitost

4 Komunikační protokol

Komunikace mezi procesory je znázorněna na obrázku 2.

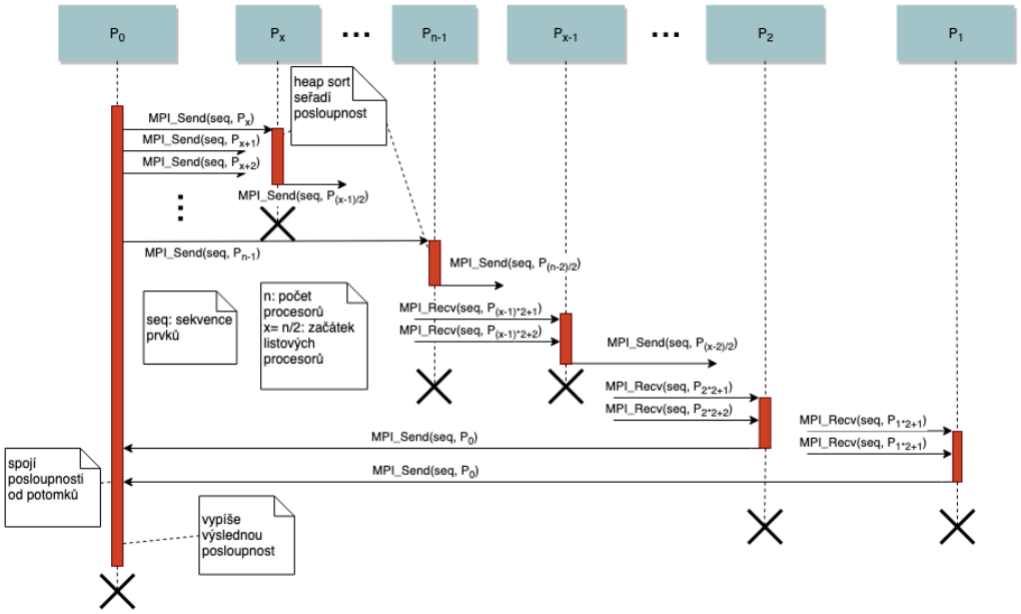

Figure 2: Komunikační protokol

5 Závěr

V grafu jsem se snažil vyobrazit přechod z nejhoršího stavu (nejvíce prvků na procesor), při počtu procesorů 25 na nejlepší stav (nejméně prvků na procesor) u počtu 27 procesorů. Tento jev je pozorovatelný v tabulce 1 při počtu prvků 16383 a 16384 (pilovitý tvar v grafu 1).

Časová složitost algoritmu je podobná funkci $f(x) = 120 * x + 500000$, což je lineární funkce, tudíž implementovaný algoritmus téměř splňuje teoretickou časovou složitost. Je velmi těžké, až skoro nemožné dosáhnout teoretické časové složitosti, jelikož reálně naimplementované algoritmy jsou často závislé na řadě faktorů (vyrovnávací paměť, vytíženost jednotlivých procesorů, atd.)