Paralelní a distribuované algoritmy
Projekt č. 2 - Implementace Enumeration Sort

Daniel Dušek, xdusek21@stud.fit.vutbr.cz

9. dubna 2017

1 Rozbor a analýza algoritmu

Algoritmus *Enumeration Sort* realizovaný za pomoci lineárního pole n procesorů se společnou sběrnicí, je paralelním řadícím algoritmem. Každý z procesorů obsahuje 4 registry X, Y, Z a C, jejichž účel je popsán níže v této sekci.

Princip paralelního algoritmu Enumeration Sort

1. Všechny C registry se nastaví na hodnotu 1.
2. Následující činnosti se opakují 2n krát 1 ≤ k ≤ 2n:
   - Je-li na vstupu prvek x_i, vloží se sběrnicí do registru X_i a lineárním spojením do Y_1.
   - Každý procesor, jehož registry X a Y nejsou prázdné provede jejich porovnání a je-li $X > Y$[1] inkrementuje svůj registr C
   - Obsah všech registrů Y se posune doprava.
   - Je-li vstup vyčerpán ($k > n$), procesor P_{k-n} pošle sběrnicí obsah svého registru X procesoru P_{Ck-n} a ten jej uloží do svého registru Z
3. V následujících n cyklech procesory posouvají obsah svých registrů Z doprava na procesor P_n, který produkuje seřazenou posloupnost.

Z výše popsaného principu algoritmu plyne, že jeho asymptotická složitost lze vyjádřit rovnicí $O(1) + O(2 * n) + O(n)$. Proměnná n zde reprezentuje počet řazených hodnot. Členy rovnice v takovém pořadí, v jakém jsou uvedeny, odpovídají krokům výše uvedeného algoritmu, tedy *počáteční nastavení registrů*, *distribuce a porovnávání hodnot* a nakonec *produkce hodnot na kořenovém procesoru*. Potom tedy platí, že čas potřebný k řešení úlohy (v krocích) $t(n) = O(n)$.

Dále víme, že cena paralelního řešení je vyjádřena vztahem $c(n) = p(n) * t(n)$, kde $p(n)$ je počet procesorů, z toho vyvodíme, že cena tohoto paralelního algoritmu je rovna $O\left(n^2\right)$ a tedy není optimální.

Registr X slouží k uchování hodnoty přiřazené procesoru na začátku před řazením. Skrze registr Y prochází hodnoty, které jsou v průběhu řazení porovnávány a posouvány směrem doprava. Během těchto porovnávání je inkrementován registr C, jehož hodnota určuje počet prvků menších než je hodnota registru X. Do registru Z se po konci první fáze kroku algoritmu umístí hodnota odpovídajícího procesoru X tak, aby posuvy ke kořenovému procesoru došlo k vygenerování seřazené posloupnosti.

[1] V implementované verzi algoritmu je navíc zavedena podmínka $X \geq Y$, dle článku *The Parallel Enumeration Sorting Scheme for VLSI od autorů Hiroto Yasuura, Naofumi Takagaki, Shuzu Yajima*, kterou je zajištěna funkčnost i při výskytu více shodných čísel

2 Implementace

Jako implementační jazyk byl zvolen jazyk C++ s využitím knihovny Open MPI.

Kód programu je rozdělen do dvou význačných proudů - větve pro řídící, tzv. kořenový, proces, jehož běh simuluje běh řídícího procesoru a větve, kterou se vydávají procesy „pracovníků“, které simulují „pracovnické“ procesory.

Kořenový proces čte sekvenčně ze souboru *numbers* posloupnost náhodných čísel v rozsahu 0-255 a posílá je postupně skrze sběrnici podřízeným procesům. Po přečtení každého čísla jej také pošle lineárním spojením do registru Y_1. Během načítání hodnot a jejich distribuce navíc kořenový proces postupně vypisuje načtená čísla v takovém pořadí, v jakém je získal. Následně již kořenový proces jen vyčkává než mu ostatní, podřízené procesy, začnou hlásit seřazené hodnoty posloupnosti. Tyto hodnoty převezme a vypíše na standardní výstup.

Pracující procesy přebírají hodnoty od kořenového procesu a provádí řazení hodnot tak, jak je dáno principem algoritmu.

Z konkrétních funkcí knihovny Open MPI je používáno pro meziprocesorovou komunikaci zejména funkcí `MPI_Recv()` a `MPI_Send()` a na jednom místě programu navíc funkce `MPI_Barrier()`, jako synchronizačního prostředku, který nepropustí žádný proces dál, dokud všechny ostatní procesy z komunikační skupiny nedorazily do stejného bodu. Bariéra je použita před začátkem posílání hodnot do kořenového procesu, aby bylo se 100% jistotou zajištěno, že každý proces odesílá seřazenou hodnotu.

3 Měření a experimenty

Implementace algoritmu byla podrobena měření, kdy byl měřen čas potřebný k seřazení různého počtu hodnot. Toto měření probíhalo na autorově pracovní stanici, kdy v průběhu celého měření bylo dbáno na stejné konstatní zatížení prostředí, aby docházelo k minimálnímu počtu chyb způsobených prostředím. Pro každou získanou hodnotu bylo provedeno 30 měření. Z naměřených 30 hodnot byly odstraněny očividné chyby měření a ponechané hodnoty byly zprůměrovány.

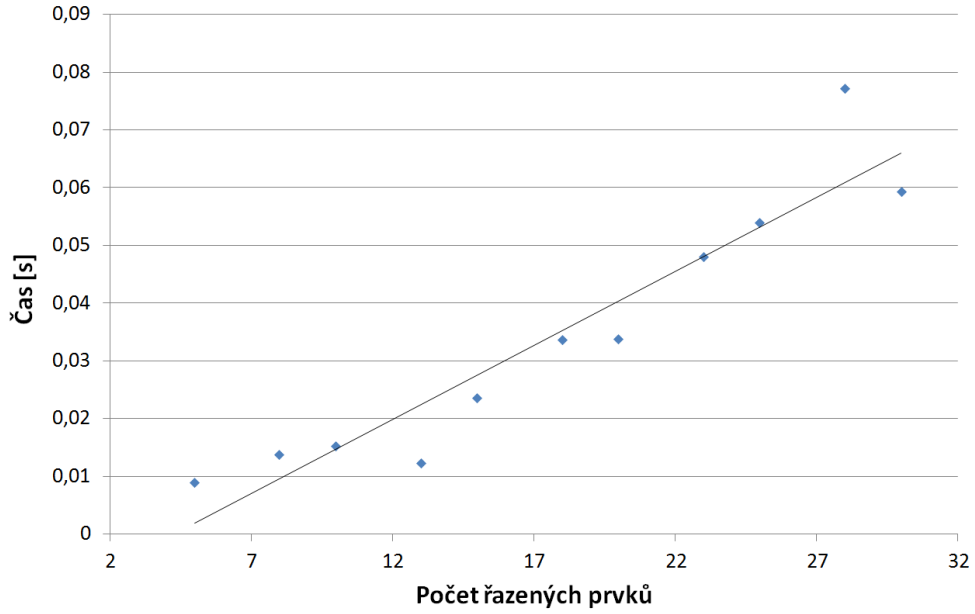
Stejné experimenty byly provedeny i na školním testovacím server *Merlin* a výsledný časový trend byl takřka shodný s trendem naměřeným na autorově stanici. Časově se hodnoty ovšem lišily - na školním serveru byly řádově nižší (toto si autor vysvětluje jako rozdíl v HW konfiguraci).

Pro měření doby běhu skriptu bylo využito funkce `MPI_Wtime()` volané na vhodných místech v kódu - a sice před začátkem řadícího algoritmu a po jeho konci, zevnitř těla kořenového procesu. Umístění funkcí do kořenového procesu je dostatečné a oprávněné, neb moment, kdy kořenový procesor zná správné pořadí řazených prvků je zároveň i moment, kdy algoritmus doběhl. Protože kořenový proces čeká na odpověď od všech podřazených procesů, je opravdu zachycená celá doba běhu algoritmu. Dvě získané hodnoty od sebe potom byly odečteny a tím byla získána opravdová doba běhu algoritmu.

V Tabulce 1 je vyobrazena závislost času potřebného k seřazení hodnot, na počtu těchto hodnot (*Pozn. tabulka je zde uvedena pouze pro úplnost*). Hodnoty v tabulce jsou právě výše zmíněné, naměřené a zprůměrované výsledky. Na základě těchto hodnot byl také vytvořen graf zobrazující časovou závilost na počtu prvků.

Tabulka 1: **Závislost času potřebného k seřazení na počtu řazených prvků**

počet prvků	2	3	5	8	10	13	15	18
čas [s]	0.0031	0.0045	0.0090	0.0138	0.0152	0.0123	0.0237	0.0337
počet prvků	20	23	25	28	30			
čas [s]	0.0337	0.0480	0.0539	0.0772	0.0594			



Obrázek 1: Graf závislosti času potřebného k seřazení prvků na jejich počtu

4 Komunikační protokol procesů

Meziprocesorová komunikace je simulována pomocí funkcí poskytovaných MPI knihovnou `MPI_Recv()` a `MPI_Send()`. Pomocí těchto funkcí je simulována jak komunikace po sběrnici, tak lineární propojení procesorů. Knihovna MPI navíc poskytuje další užitečné funkce jako je `MPI_BCast` či `MPI_Gather`, které by mohly být využity při dalších možných optimalizacích implementovaného algoritmu (popsáno níže). Sekvenční diagram meziprocesové komunikace je zachycen na obrázku 2.

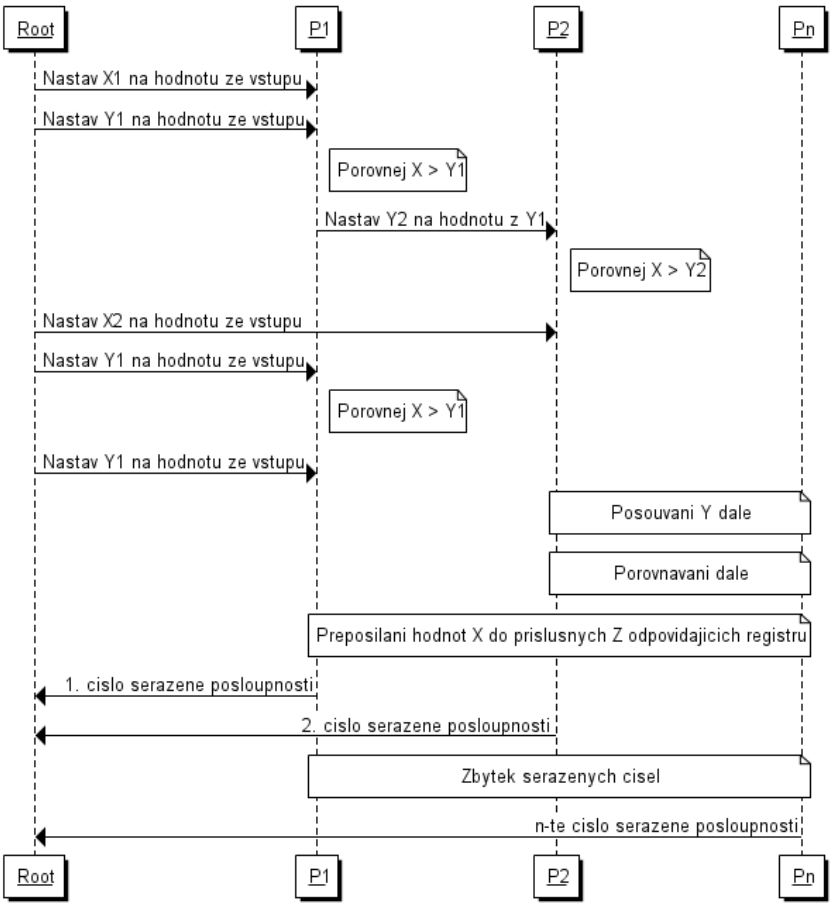
5 Navrhované modifikace algoritmu

Při modifikované implementaci algoritmu *enumeration sort* by bylo možné využít funkcí knihovny MPI, a to `MPI_Scatter()` a `MPI_Bcast()` pro efektivnější distribuci čísel k podřazeným procesorům. Dále by pak navíc bylo při předávání výsledků možné zrychlit předávání hodnot zpět ke kořenovému procesoru využitím funkce `MPI_Gather()`. Odhadované snížení složitosti při použití těchto funkcí by mohlo být až o $O(n) + O(n)$.

Autor by rád odůvodnil překročení rozsahu 3 stran právě touto sekcí, kterou na základě informací z fóra považuje za důležitou k uvedení navíc.

6 Závěr

Měřením a experimentováním nad odevzdávanou implementací *enumeration sort* algoritmu byla potvrzena lineární asymptotická složitost tohoto algoritmu. Implementovaný algoritmus narozdíl od základního algoritmu *enumeration sort* dokáže řadit i posloupnosti obsahující více stejných hodnot. Naměřené hodnoty jsou k vidění v Sekci 3, komunikační protokol procesů pak v Sekci 4 - Komunikační protokol procesů.



Obrázek 2: Sekvenční diagram meziprocesové komunikace.