PRL
(Paralelní a Distribuované algoritmy)

Implementace algoritmu "Enumeration Sort"

Historie			
Verze	**Datum**	**Kdo**	**Poznámka**
1.0	29.03.2014	Kirill Gaevskii, xgaevs01	

Obsah

1 Zadání....3
1.1 Vstup....3
1.2 Výstup....3
2 Popis algoritmu....3
3 Analýza algoritmu....3
4 Implementace....4
4.1 Eliminace duplicitních hodnot....4
4.2 Enumeration Sort....4
5 Experimenty....4

1 Zadání

Pomocí knihovny Open MPI implementujte algoritmus Enumeration Sort na lineárnítopologii.

1.1 Vstup

Soubor "numbers" obsahující čísla velikosti 1 byte, která jdou bez mezery za sebou.

1.2 Výstup

Výstup na *stdout* se skládá ze dvou částí:

- Jednotlivé načtená čísla v jednom řádku oddělená mezerou;
- jednotlivé seřazená čísla oddělená novým řádkem.

2 Popis algoritmu

Enumeration Sort je paralelní řadící algoritmus, pracující na lineární architektuře, doplněnou o společnou sběrnici, schopnou přenést v každém kroku jednu hodnotu. Každý procesor P_i má 4 registry:

- X_i – prvek vstupní posloupností x_i;
- Y_i – postupně x_1, x_2, … x_n;
- C_i – počet prvků menších než x_i;
- Z_i – seřazený prvek Y_i.

Algoritmus řazení vstupního pole X o velikosti n:

1. Všechny registry C - na hodnotu 1;
2. 2n krát opakovaní následovní činností:
   a) Pokud vstup není vyčerpán, vstupní prvek xi se vloží do registru X_i, pomocí sběrnice a také do Y_1 pomocí lineárního spojení. Dále se doprava posune obsah všech registrů Y ;
   b) Každý procesor provede porovnání těchto registry X a Y provede porovnání těchto registrů a v případě, že X > Y provede se inkrementace registru C;
   c) Po vyčerpání vstupu, tedy i > n procesor Pi-n pošle sběrnicí obsah svého registru X procesoru P_c, který jej uloží do registru Z.
3. V následných n-cyklech procesory posouvají obsah svých registrů Z doprava a procesor P_n udelavá výslednou seřazenou posloupnost.

3 Analýza algoritmu

Celková časová složitost algoritmu je součtem dílčích časových složitostí v jednotlivých krocích uvedených výše.

1. O(1) – nastavení registru C je v konstantním čase;
2. O(2n) – distribuce hodnot a porovnání má lineární časovou složitost;
3. O(n) – distribuce výsledků má lineární časovou složitost.

Výsledná časová složitost je: t(n) = O(n). Pro vykonání algoritmu je potřeba n procesoru: p(n) = n. Cena algoritmu, získaná ze vztahu p(n) * t(n), je: c(n) = $O(n^2)$.

Cena algoritmu Enumeration Sort není optimální.

4 Implementace

Algoritmus byl implementován v jazyce C za pomocí knihovny pro podporu paralelního řešení výpočetních problémů OpenMPI.

4.1 Eliminace duplicitních hodnot

Algoritmus funguje správně jen s posloupnosti, kde jsou výskyty všech čísel jedinečné. Generovaná posloupnost hodnot, však tuto podmínku nezaručuje a proto bylo potřeba odstranit vícenásobný výskyt čísel ze vstupní posloupnosti.

Pro odstranění duplicitních hodnot byla využita metoda, která postupně nachází čísla, která jsou v posloupnosti vícekrát. Tyto čísla pak nahradí následující hodnotu, kteráse v posloupnosti nevyskytuje. Tato metoda negativně ovlivňuje časovou složitost programu, proto samotné měření časové složitosti algoritmu, bylo provedeno vždy až po eliminaci duplicitních hodnot.

4.2 Enumeration Sort

Program pomocí knihovny OpenMPI simuluje lineární topologii se společnou sběrnicí, proto byl algoritmus řazení lehce poupraven. Lineární propojení i sběrnice jsou simulovány pomocí blokující funkce MPI_Send. Ta umožňuje z jednoho procesoru zaslat hodnotu druhému procesoru, který ji však musí převzít pomocí blokující funkce MPI_Recv. Parametrem těchto funkcí je i tag, který určuje typ zprávy, čímž je rozlišeno, jestli se posílá hodnota registru X (tag = X_TAG), registru Y (tag = Y_TAG), nebo registru Z (tag = Z_TAG).

Po 2n-krocích (algoritus krok 2) je vstupní posloupnost vzestupně seřazena. V dalších n-krocích sehodnoty posouvají doprava, s tím že vždy poslední procesor tuto hodnotu pošle hlavnímu procesoru a ten když přijdou všichny hodnoty jejich tiskne na výstup. Následující sekvenční diagram ukazuje komunikaci mezi různými procesory v různých krocích algoritmu (Obrázek 1).

5 Experimenty

Pro měření časové složitosti jsem použil funkci **gettimeofday**, která vrací aktuální čas. Časovou známku jsem uložil na začátku řazení a na konci řazení. Rozdíl těchto časových známek udává dobu řazení.

Pro různé délky vstupních posloupností jsem měření opakoval vždy 10 krát, a nad těmito hodnotami jsem vypočítal ořezaný průměr (nebyla započítána nejmenší a největší naměřená hodnota).

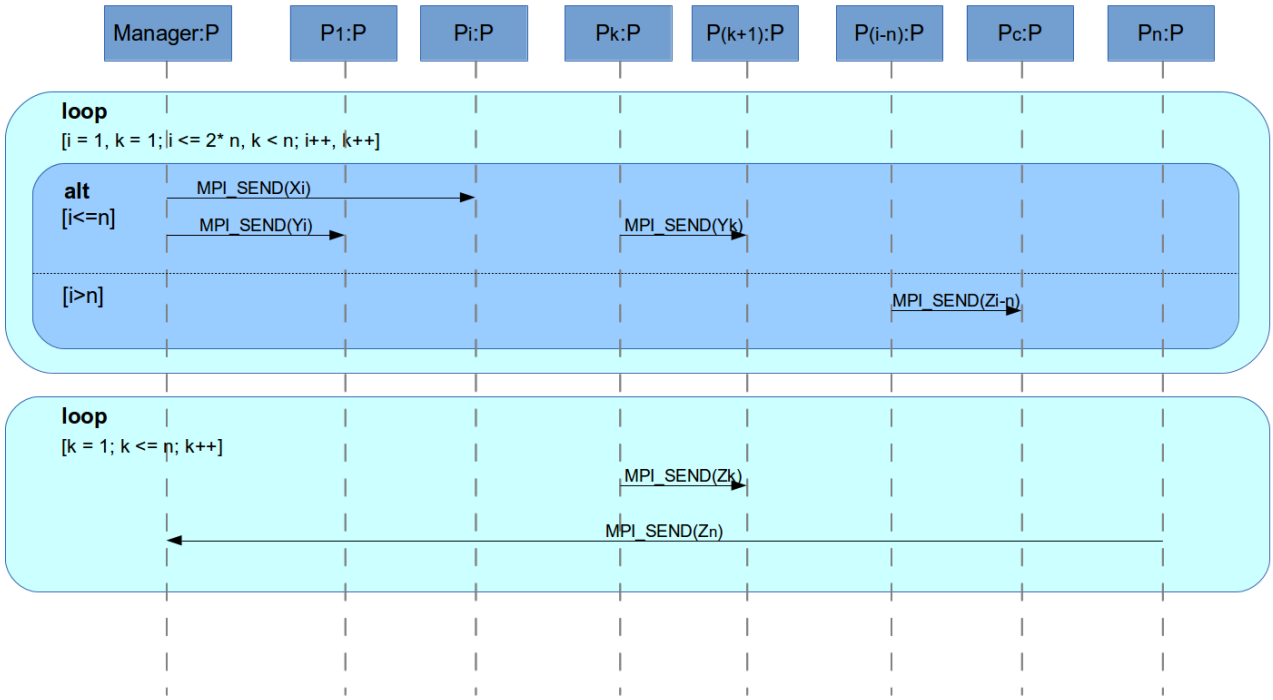
Následující tabulka ukazuje naměřené hodnoty pro různé délky vstupní hodnot:

Počet hodnot	Ořezaný průměr času [ms]
10	13,706875
20	42,766875
30	77,33525
40	120,636625
50	167,1515
60	196,358

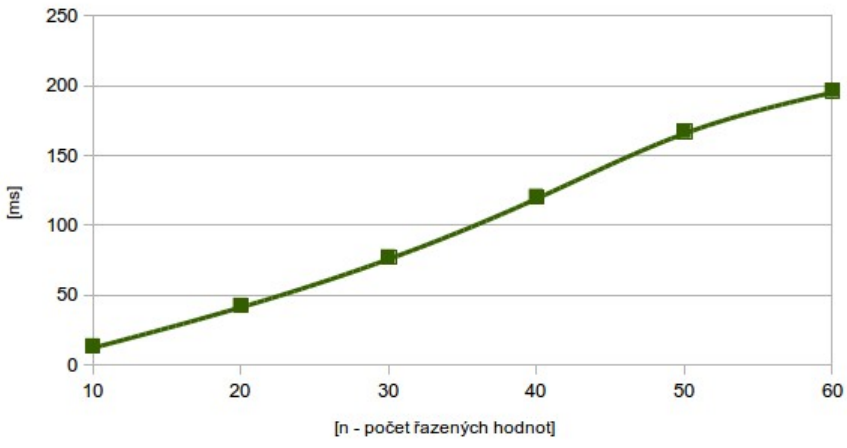
Tabulka 1: Čas pro různé počty řazených hodnot

Z tabulky je zřejmé, že se vzrůstajícím počtem řazených hodnot roste čas lineárně. To odpovídá teoretické časové složitosti algoritmu.

Z naměřených hodnot byl pro lepší přehlednost sestaven graf (Obrázek 2).



Obrázek 1: Sekvenční diagram komunikace procesorů



Obrázek 2: Výsledky měření