Paralelní a distribuované algoritmy - Mesh multiplication

Michal Šrubař
xsruba03@stud.fit.vutbr.cz

2. května 2015

1 Zadání

Pomocí knihovny Open MPI implementujte v jazyce C++ algoritmus Mesh Multiplication.

1.1 Vstup a výstup programu

Vstupem jsou textové soubory *mat1* a *mat2*. Výsledná matice, která je získána operací vynásobením těchto matic je vypsána na standardní výstup ve formátu specifikovaném níže.

Jako oddělovač čísel na řádku je použita mezera, jako oddělovač jednotlivých řádků pak znak nového řádku *n*.

První řádek mat1 obsahuje počet řádků. 3 1 -1 2 2 3 3

Řazená posloupnost je uložena v binárním souboru *numbers*. Každá hodnota je velikosti jeden byte. Tento vstupní soubor je vytvořen automaticky skriptem *test.sh* pomocí linuxové utility *dd*[1].

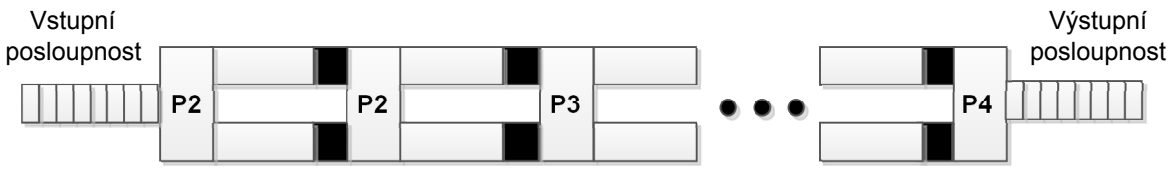
Výstup programu se skládá ze dvou částí:

1. na prvním řádku jsou mezerou oddělené vstupní hodnoty,
2. další řádky obsahují seřazené hodnoty oddělené znakem nového řádku (od nejmenšího po největší).

```
> sh test.sh 4
216 244 151 117
117
151
216
244
```

2 Pipeline merge sort

Algoritmus pipeline merge sort pracuje s lineárním polem procesorů, které jsou propojeny dvěma kanály s nenulovou kapacitou. Tato architektura je zobrazena na obrázku 3.1, kde každý procesor očekává na svých vstupních kanálech dvě posloupnosti, které spojí do výstupní **seřazené** posloupnosti na svém horním kanálu a poté vezme další dvě posloupnosti a spojí je do seřazené posloupnosti na svém dolním kanále. Takto to procesor pořád střídá, dokud je splněna podmínka práce procesoru.



Aby mohl procesor pracovat, pak musí mít na jednom svém vstupu úplnou posloupnost a na druhém alespoň jeden prvek. Vyjímku tvoří první procesor, který pouze předává prvky ze **vstupní neseřazené posloupnosti** na své výstupní kanály. Každý procesor začíná předávat nejprve na horní kanál, poté na dolní, poté na horní atd. Důležité

[1] http://man7.org/linux/man-pages/man1/dd.1.html

je zmínit, že procesor spojuje vždy pouze ty prvky které patří do **stejné posloupnosti**. Výsledkem spojení dvou posloupností je poté posloupnost dvojnásobné délky.

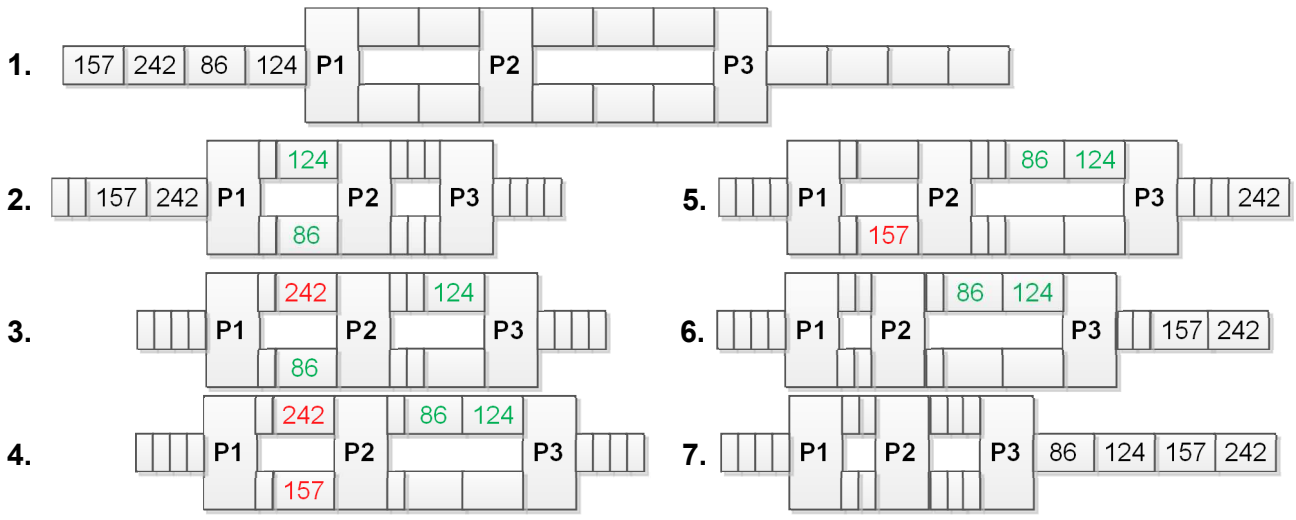
Nechť n je počet prvků vstupní neseřazené posloupnosti, pak počet procesorů, s kterými bude algoritmus pracovat, vyjádřuje vztah 1:

$$p(n) = \log_2 n + 1 \tag{1}$$

2.1 Příklad

Nechť *X={124, 86, 242, 157}* je řazená posloupnost, pak řazení, které je demonstrováno na obrázku 2.1, budou dle vztahu 1 provádět tři procesory a řazení může proběhnou následovně:

1. Na začátku má procesor P1 ve své vstupní frontě posloupnost {124, 86, 242, 157}.
2. Procesor P1 vezme ze vstupní posloupnosti první prvek s hodnotou 124 a pošle jej horním kanálem procesoru P2. Poté pošle procesoru P2 druhý prvek s hodnotou 86 přes dolní kanál. Procesor P2 má na svém vstupu dvě posloupnosti a může tedy pracovat. Provede porovnání hodnot 124 a 86 a větší z nich pošle procesoru P3 pomocí horního kanálu. Mezi tím procesor P1 posílá třetí prvek s hodnotou 242, který ovšem patří do další posloupnosti.
3. Procesor P2 teď sice má na svých vstupních kanálech dvě hodnoty, ale nemůže je porovnat, jelikož nejsou ze stejné posloupnosti. Nejprve musí dokončit porovnávání první posloupnosti, což znamená odeslat hodnotu 86 procesoru P3. Procesor P1 může zároveň poslat procesoru P2 poslední hodnotu , tj. 157.
4. Až nyní procesor P2 splňuje podmínku a může porovnat hodnoty 242 a 157, jelikož jsou ze stejné posloupnosti. Větší hodnotu, tj. 242, pošle procesoru P3.
5. Procesor P3 může hned porovnat přijatou hodnotu 157 s hodnotou 124 a větší z nich zařadit do výstupní posloupnosti.
6. Na dolní kanál procesoru P3 již nepřijde žádný prvek, takže může zkopírovat seřazenou posloupnost z horního kanálu do výstupní fronty čímž dostáváme výslednou seřazenou posloupnost *Y={86,124,157,242}*.



2.2 Analýza

Procesor P2 začne pracovat až 2 cykly po procesoru P1, jelikož procesor P1 musí nejprve poslat prvek ze vstupní posloupnosti na horní kanál a poté na dolní kanál. Procesor P3 poté začne pracovat až 3 cykly za procesorem P2, protože P2 nejprve pošle jednu seřazenou posloupnost na horní kanál a poté musí poslat alespoň jeden prvek z druhé seřazené posloupnosti a až poté muže procesor P3 začít pracovat.

Obecně tedy platí, že procesor P_i může začít pracovat až v okamžiku, kdy se na jednom jeho vstupu nachází posloupnost délky 2^{i-2} a na druhém posloupnost délky 1. Procesor P_i pak tedy začne pracovat $2^{i-2} + 1$ cyklů po procesoru P_{i-1}.

Procesor P_i obecně začne pracovat v cyklu:

$$2^{i-1} + i - 1$$

a skončí v cyklu:

$$(n-1)+2^{i-1}+i-1$$

kde n je počet řazených prvků vstupní posloupnosti. Celý algoritmus poté skončí za $2n+\log_2 n-1$ cyklů. Časová složitost se skládá z lineární a logaritmické složky a jelikož je logaritmická pro tu lineární zanedbatelná, pak celková časová složitost je lineární a tedy $t(n) = \mathcal{O}(n)$. Cena algoritmu je poté $c(n) = t(n) * p(n) = \mathcal{O}(n) * (\log_2 n + 1) = \mathcal{O}(n * \log n)$. Cena tedy je optimální.

3 Implementace

Algoritmus je implementován v jazyce C s využitím knihovny OpenMPI[2], kde jednotlivé procesory jsou simulovány procesy operačního systému. Kapacita kanálů jednotlivých procesorů je realizována pomocí linuxových front[3], kde každý procesor má svoji horní a dolní vstupní frontu kromě prvního procesoru, který pracuje pouze se vstupní frontou, kde jsou před samotným začátkem algoritmu načteny vstupní hodnoty ze souboru `numbers`. Poslední procesor má poté navíc ještě výstupní frontu kam ukládá prvky výstupní seřazené posloupnosti.

3.1 Sekvenční diagram

Nechť $X = \{x_1, x_2, x_3, \ldots, x_m\}$ je řazená posloupnost a N reprezentuje počet procesů, pak sekvenční diagram, který je zobrazen na obrázku 3.1, popisuje komunikaci mezi procesy P1 až PN. První proces P1 prochází svoji vstupní frontu a předává z ní řazené položky jednu za druhou dalšímu procesu, tj. P2, pomocí blokujícího rozhranní *MPI_Send*[4]. Jak bylo popsáno v sekci 2.2, proces P2 začne pracovat až dva cykly po procesu P1 jelikož musí počkat, až obě jeho vstupní fronty budou obsahovat hodnoty ze stejné posloupnosti.

Jakmile má proces splňuje podmínku práce, tak porovná první prvek z horní a dolní fronty a větší z nich posílá svému pravému sousedovi. Poslední proces PN taktéž provádí porovnávání prvků, ale větší z nich již neposílá pravému sousedovi, ale ukládá je do své výstupní fronty, kde se po skončení algoritmu nachází seřazená posloupnost $Y = \{y_1, y_2, y_3, \ldots, y_m\}$. Každý procesem tak projde každý prvek z řazené posloupnosti.

3.2 Výsledky experimentování

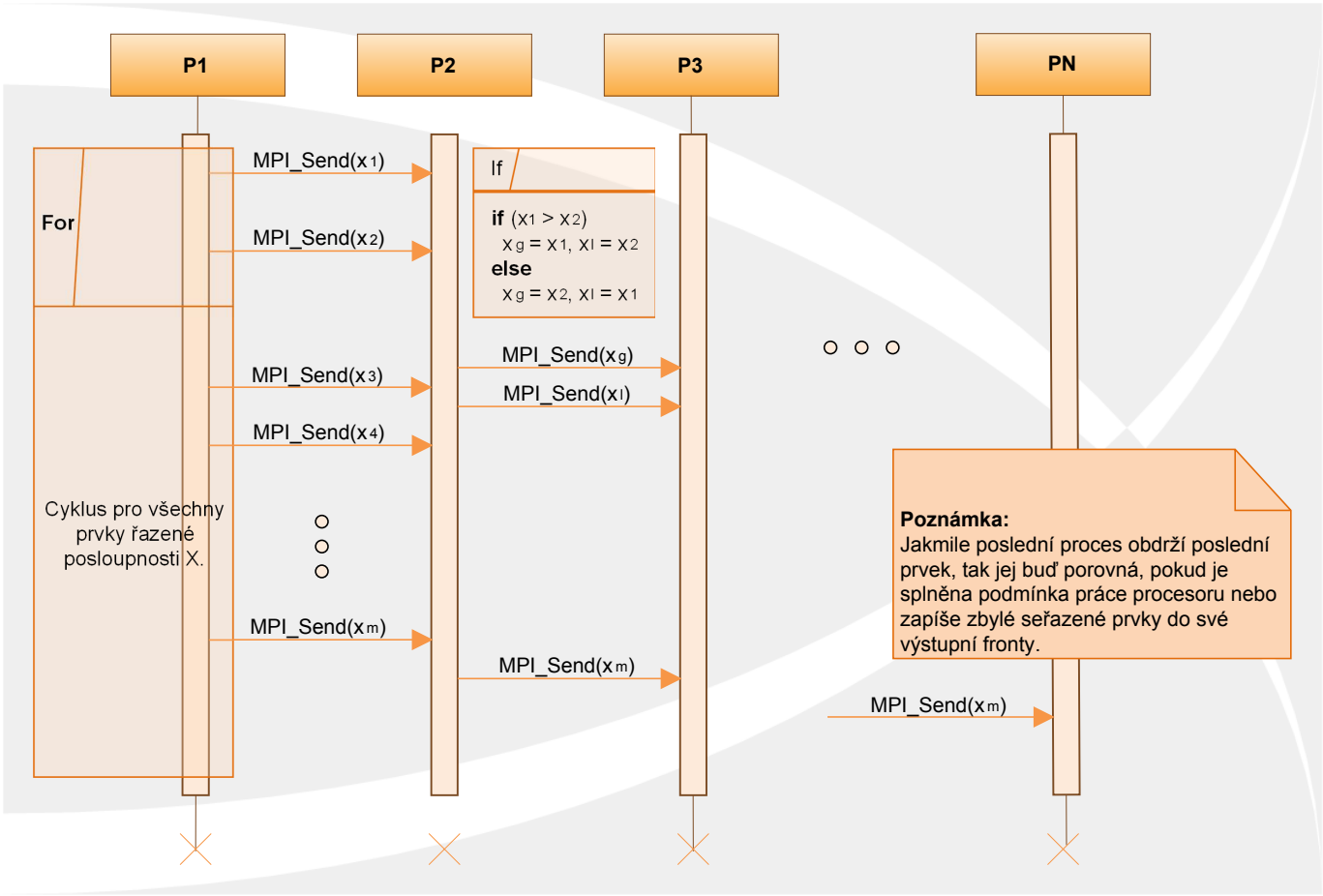
Pro měření doby práce algoritmu byla využita funkce *clock_gettime()*[5]. Při experimentech byl měřen pouze čas algoritmu a ne načítání vstupní posloupnosti ze souboru a výstup výsledků na terminál. Měření proběhlo pro vstupní posloupnosti o délkách 2, 4, 8, 16, 32, 64 a 128 prvků. Pro každou délku bylo vygenerováno celkem 100 různých vstupních posloupností ze kterých poté byla spočtena průměrná doba trvání algoritmu pro konkrétní délku vstupní posloupnosti.

Testování proběhlo na serveru *merlin.fit.vutbr.cz* pomocí skriptu *performance_test.sh*, jehož výsledky zobrazuje tabulka 3 a obrázek 2, kde sloupec n reprezentuje délku vstupní posloupnosti a čas je průměrný čas v nanosekundách.

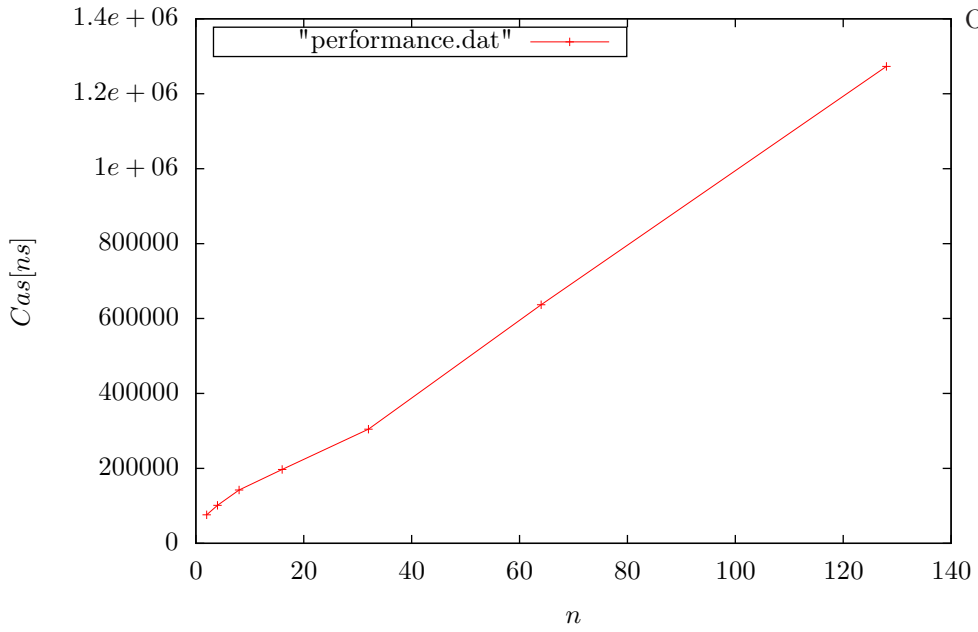
4 Závěr

Cílem tohoto projektu bylo popsat, analyzovat, implementovat a otestovat algoritmus pipeline merge sort. V sekci 2.2 bylo odvozeno, že algoritmus má lineární složitost, což bylo potvrzeno pomocí provedeného experimentu jehož výsledky zobrazuje graf na obrázku 2. Časová složitost ani nemůže být lepší, jelikož pracujeme s lineárním polem procesorů a výsledná seřazená posloupnost se musí objevit na výstupu posledního procesoru.

[2] Open Message Passing Interface, http://www.open-mpi.org
[3] http://linux.die.net/man/3/queue
[4] https://www.open-mpi.org/doc/v1.8/man3/MPI_Send.3.php
[5] http://linux.die.net/man/3/clock_gettime



Obrázek 1: Sekvenční diagram komunikace mezi procesy.



Obrázek 2: Grafické zobrazení naměřených výsledků.

Obrázek 3: Výsledky.

n	čas [ns]
2	76099
4	101573
8	142423
16	197219
32	304561
64	636753
128	1273051