Meno a priezvisko: Vladimír Marcin
Login: xmarci10

1 Úvod

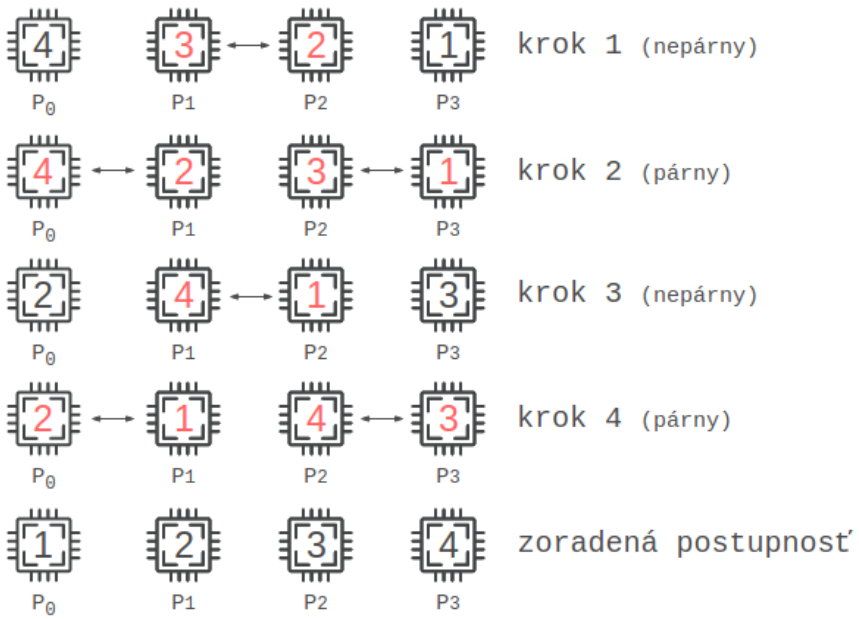
Úlohou tohoto projektu bolo implementovať v jazyku `C/C++` pomocou knižnice `Open MPI` algoritmus *Odd-even transposition sort* na lineárnom poli o n procesoroch. Okrem samotnej implemntácie algoritmu bolo nutné vytvoriť riadiaci skript, ktorý bude riadiť testovanie.

2 Odd-even transposition sort

2.a Popis algoritmu

Algoritmus *Odd-even transposition sort* je variantou algoritmu *Bubble-sort*. Rovnako ako v algoritme Bubble sort sú prvky porovnávané v pároch a v prípade nutnosti sú vymenené. Avšak v našom algoritme sú tieto porovnania a výmeny robené v dvoch fázach: *párna* a *nepárna*. Nech $a_0, a_1, \ldots, a_n$ je radená sekvencia. Počas párnej fázy sú porovnávané prvky s párnymi indexmi s ich pravým susedom: $(a_0, a_1), (a_2, a_3), \ldots, (a_{n-1}, a_n)$ (predpokladajme, že n je párne). Podobne v nepárnej fáze sú porovnávané prvky s nepárnymi indexmi s ich pravým susedom: $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-2}, a_{n-1})$. V prípade, že sú porovnávané prvky v nesprávnom poradí, sú tieto prvky vymenené.

Ak tento algoritmus prevedieme do paralelného prostredia, porovnania a výmeny v rámci jednej fázy môžu byť vykonávané súčasne. Uvažujme, že každý prvok v radenej postupnosti patrí jednému procesoru. Nech n je počet procesorov (a taktiež počet radených prvkov) a predpokladajme, že procesory sú usporiadané v jednorozmernom poli. Potom každý prvok a_i je priradený procesoru P_i, kde $i = 0, 1, 2, \ldots, n$. V párnej fáze sa každý procesor s párnym indexom spojí so svojím pravým susedom, porovnajú svoje hodnoty a v prípade potreby si ich vymenia. Podobne prebieha aj nepárna fáza. Priebeh paralelného radenia je ilustrovaný na obrázku 1. Algoritmus *Odd-even transposition sort* garantuje, že najneskôr po n fázach ($n/2$ párnych + $n/2$ nepárnych), kde n je počet radených prvkov, bude postupnosť zoradená.



Obr. 1: Ukážka *Odd-even transposition sort* so 4 procesormi a 4 radenými hodnotami

2.b Analýza algoritmu

1. Z popisu algoritmu v sekcii 2.a je zrejmé, že počet procesorov potrebných pre náš algoritmus je rovný počtu radených prvkov a teda $\mathbf{p(n) = n}$.

2. Počas každej fázy algoritmu *Odd-even transposition sort*, buď párny alebo nepárny procesor vykoná operáciu `compare-exchange` (viz Obr. 2) so svojím pravým susedom. Táto operácia vyžaduje čas $\mathcal{O}(1)$. Spolu sa takýchto fáz zopakuje n, čize celková časová zložitosť nášho algoritmu je $\mathbf{t(n) = \mathcal{O}(n)}$.

3. Je zrejmé, že na vykonanie porovnania v operácii `compare-exchange` si každý procesor musí dočasne uchovať hodnotu svojho suseda. Z čoho vyplýva, že priestorová zložitosť nášho algoritmu je $\mathbf{s(n) = \mathcal{O}(n)}$.

4. Cena paralelného algoritmu je obecne definovaná ako $c(n) = p(n) * t(n)$ a z toho vyplýva, že cena nášho algoritmu je $\mathbf{c(n) = \mathcal{O}(n^2)}$.

Aby bol paralelný radiaci algoritmus optimálny musí platiť, že $t_{seq}/c(n) = 1$, kde t_{seq} je časová zložitosť optimálneho sekvenčného radiaceho algoritmu a $c(n)$ je cena paralelného riešenia. A keďže časová zložitosť optimálneho sekvenčného radiaceho algoritmu je $\mathcal{O}(n * log(n))$ a cena nášho algoritmu je $\mathcal{O}(n^2)$, algoritmus *Odd-even transposition sort* **nieje optimálny**.

3 Implementácia

Algoritmus je implementovaný v jazyku `C` za použitia knižnice `Open MPI`. Na samotnom začiatku sa spustí funkcia `MPI_Init` na inicializáciu `MPI` prostredia. Zároveň si každý proces zistí svoje id (`rank`) a celkový počet procesorov.

Ďalšou významnou časťou algoritmu je funkcia `read_numbers`. V tejto funkcii procesor s `rankom` 0 alokuje potrebné miesto, načíta vstup zo súboru `numbers` a zároveň ho vypíše na štandardný výstup. Keďže čísla na vstupe sú z rozsahu 0 až 255, v implementácii sa používa dátový typ `unsigned char` (v `MPI` je to typ `MPI_BYTE`). Po načítaní a vypísaní sú hodnoty rozdistribuované medzi jednotlivé procesory. Keďže v našom algoritme každý procesor dostane rovnaké množstvo dát z alokovaného poľa (práve `1 byte`), na distribúciu hodnôt je použitá funkcia `MPI_Scatter`. Na záver ešte procesor, ktorý alokoval pamäť pre hodnoty zo vstupu uvoľní alokovanú pamäť, keďže každý z procesorov už má hodnotu uloženú vo svojej ,,lokálnej" premennej.

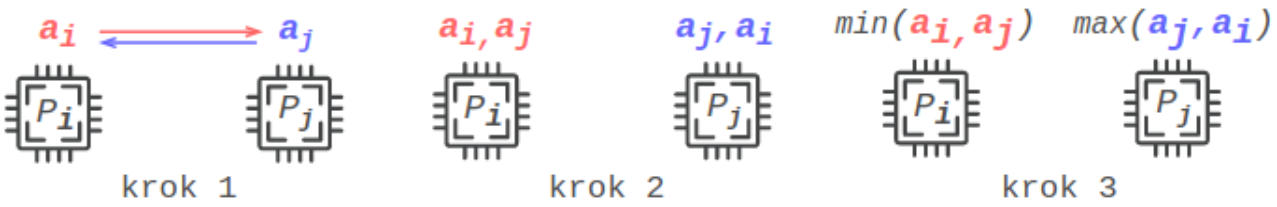
Po tom ako každý procesor obdrží svoju hodnotu prichádza na rad samotné radenie. To je implementované vo funkcii `sort`. Na začiatku tejto funkcie si najprv každý procesor spočíta svojho suseda, a to ako pre párnu tak aj nepárnu fázu algoritmu:

```
1     if (my_rank % 2) { /* Odd rank */
2         even_partner = my_rank - 1;
3         odd_partner = my_rank + 1;
4         if (odd_partner == proc_num) { odd_partner = MPI_PROC_NULL; }
5     } else { /* Even rank */
6         even_partner = my_rank + 1;
7         if (even_partner == proc_num) { even_partner = MPI_PROC_NULL; }
8         odd_partner = my_rank - 1;
9         if (odd_partner == -1) { odd_partner = MPI_PROC_NULL; }
10    }
```

Listing 1: Výpočet suseda pre jednotlivé fázy algoritmu *Odd-even transposition sort*

Ak niektorý z procesorov bude mať hodnotu svojho suseda -1, alebo rovnú počtu procesorov potom vieme, že tento procesor má byť v danej fáze nečinný (idle) a hodnota jeho suseda je nastavená na `MPI_PROC_NULL`. `MPI_PROC_NULL` je konštanta definovaná v `MPI`, ktorá ak je použitá ako zdroj alebo cieľ nejakej komunikácie, pokus o nadviazanie takejto komunikácie jednoducho skončí.

Keďže každý z procesorov má určeného suseda pre párnu aj nepárnu fázu algoritmu prichádza na rad samotné radenie. Tam si v každej fáze procesory pošlú hodnoty pomocou funkcie `MPI_Sendrecv` a v prípade potreby si ich vymenia (Obr. 2). Výhodou funkcie `MPI_Sendrecv` je, že `MPI` implementácia riadi komunikáciu a tak nemôže dôjsť k zablokovaniu, alebo pádu programu spôsobeného nesprávnym poradím operácií `MPI_Send` a `MPI_Recv`.



Obr. 2: Paralelná `compare-exchange` operácia. Procesory P_i a P_j pošlú svoje hodnoty jeden druhému. Procesor P_i si potom u seba nechá menšiu hodnotu a naopak procesor P_j väčšiu.

Poslednou časťou implementácie je výpis zotriedenej postupnosti, ktorý je implementovaný vo funkcii `print_numbers`. V tejto funkcii zozbierame hodnoty od všetkých procesorov pomocou funckie `MPI_Gather` a procesor s rankom 0 následne vypíše zoradené hodnoty.

4 Experimenty

Cieľom experimentov bolo overiť časovú zložitosť paralelného algoritmu *Odd-even transposition sort*. Pre účely zistenia časovej zložitosti bolo potrebné eliminovať rušivé vplyvy, ktoré nesúvisia so samotným algoritmom. Preto sme ignorovali čas potrebný na načítanie a distribúciu hodnôt medzi procesory, ako aj čas potrebný na spätnú propagáciu zoradených hodnôt a výpis. V konečnom dôsledku sme tak merali iba čas strávený v hlavnom triediacom cykle popísanom v sekcii 2.a, na základe ktorého bola vykonaná aj analýza algoritmu v sekcii 2.b.

Pri meraní času paralelných programov, ale musíme byť trochu opatrnejší, pokiaľ ide o spôsob merania času. V našom programe je časť kódu, ktorú chceme odmerať, spúšťaná viacerými procesmi a tým pádom pri meraní času získame množinu niekoľkých časov (každý proces = samostatný čas). Avšak, na to aby sme mohli overiť časovú zložitosť potrebujeme jeden čas a to čas, ktorý uplynie od okamihu, keď prvý proces začne vykonávať meraný úsek kódu až po čas kedy posledný proces skončí vykonávanie meraného kódu. Konkrétnu implementáciu tohto procesu môžte vidieť v Listingu 2.

```
    /* Synchronize all processes */
    MPI_Barrier(MPI_COMM_WORLD);
    local_start = MPI_Wtime();
    /* Code to be timed */
    sort(my_rank, &local_value, comm_sz, MPI_COMM_WORLD);

    local_finish = MPI_Wtime();
    local_elapsed = local_finish - local_start;
    MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

#   if (DEBUG > 0)
    if (my_rank == 0)
        printf("Sorting time = %e miliseconds\n", elapsed * 1000);
#   endif
```

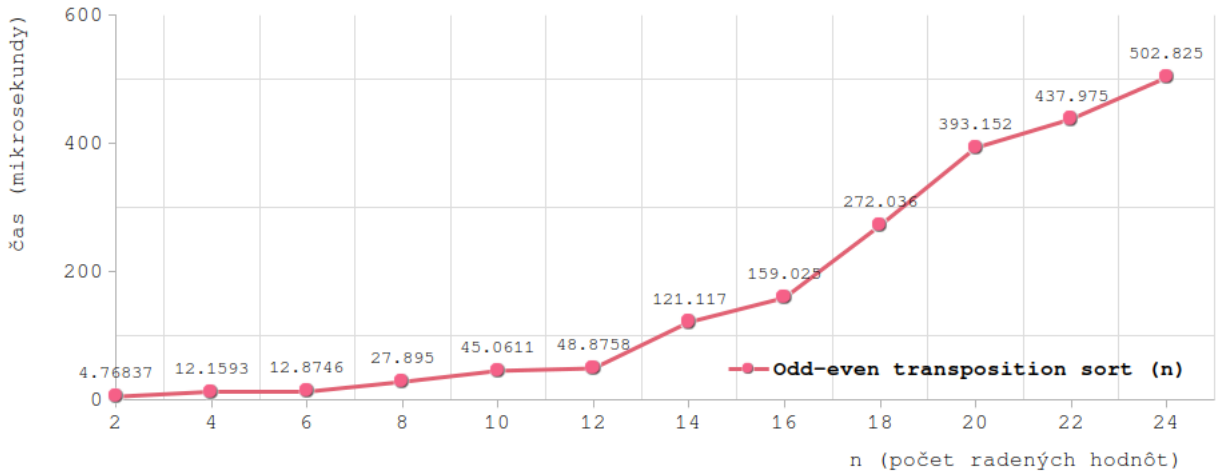
Listing 2: Meranie času testovaného algoritmu

Na implementáciu sme použili funkcie, ktoré ponúka knižnica `Open MPI`. Prvým krokom je volanie `MPI_Barrier`, ktorá aspoň približne zosynchronizuje všetky procesy a následne každý z nich zistí svoj čas pomocou funkcie `MPI_Wtime`. Na záver každý proces volá fukncu `MPI_Reduce`, ktorá

vráti čas najpomalšieho procesu. V prípade, že úroveň `DEBUG` módu je `1`, proces 0 vypíše zmeraný čas algoritmu.

Posledným úskalím, s ktorým sme sa museli popasovať pri meraní času je variabilita výsledných časov. Ak náš program spustíme niekoľko krát, je veľmi pravdepodobné, že výsledný čas bude zakaždým iný. A to dokonca aj v prípade, že ho spustíme na rovnakom systéme a s rovnakým vstupom. Ako riešenie sa ponúka použiť ako výsledný čas priemer alebo medián z nameraných výsledkov. Avšak je veľmi nepravdepodobné, že rýchlosť nášho algoritmu by mohla byť zlepšená nejakou vonkajšou udalosťou a tak namiesto strednej hodnoty alebo mediánu, ako výsledok reportujeme najnižšiu z nameraných hodnôt [3].

Samotné experimenty prebiehali tak, že pre každú zvolenú veľkosť vstupného súboru čísel bolo vykonaných 50 meraní a z nich bola tá najmenšia hodnota vybraná ako výsledná. Vstupná postupnosť čísiel na experimenty bola získaná z `/dev/random`. Výsledky experimentov sú zobrazené v grafe na Obrázku 3.



Obr. 3: Experimentálne overenie časovej zložitosti.

5 Zhodnotenie experimentov

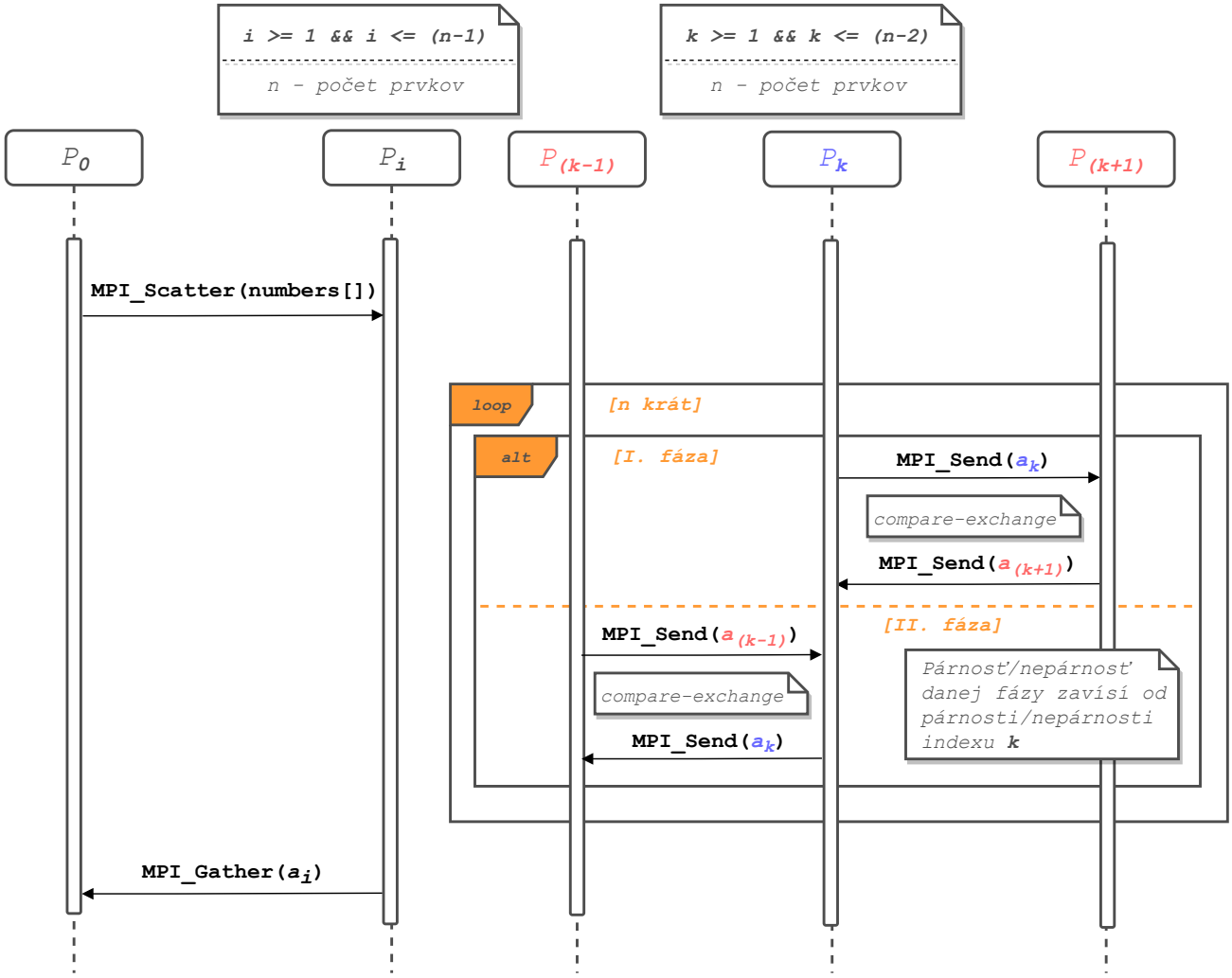
Výsledky meraní zobrazené na obrázku 3 **nie sú** v súlade s odvodenou teoretickou časovou zložitosťou, ktorá je lineárna $\mathcal{O}(n)$ (viz sekcia 2.b). Z nameraných výsledkov si môžme všimnúť, že s narastajúcim počtom prvkov (procesov), čas potrebný na radenie rastie rýchlejšie ako lineárne.

Obecne platí, že dnešné procesory potrebujú viac času na odoslanie prvku z jedného procesu do druhého, než na porovnanie prvkov. To má za následok, že hocijaký paralelný radiaci algoritmus, ktorý používa toľko procesov ako je počet radených prvkov bude mať veľmi slabé výsledky (čo sa času týka), keďže veľkú časť celkovej doby algoritmu bude predstavovať medziprocesová komunikácia [2].

Presne tento problém, sa prejavil aj v našom prípade, čo môžme vidieť na už spomínanom obrázku 3. S narastajúcim počtom procesov sa aj čas potrebný na komunikáciu medzi nimi zvyšoval, čo malo za následok rýchlejší ako očakávaný (lineárny) nárast časovej zložitosti.

Možným riešením tohto problému je modifikácia algoritmu tak, že každý procesor bude vlastniť n/p prvkov (n-počet radených prvkov, p-počet procesorov), čím sa redukuje čas strávený medziprocesorovou komunikáciou. Presne takáto modifikácia odpovedá algoritmu *Merge-splitting sort*, viz [1].

6 Komunikačný protokol



Obr. 4: Komunikačný protokol zasielania správ. Definícia operácie `compare-exchange` viz Obr. 2.

Literatúra

[1] *Distribuované a paralelní algoritmy a jejich složitost, algoritmy řazení* (prednáška PRL). 2007. URL `https://www.fit.vutbr.cz/study/courses/PDA/private/www/h003.pdf`

[2] Grama, A.; Karypis, G.; Kumar, V.; aj.: *Introduction to Parallel Computing*. Addison-Wesley, druhé vydání, 2003, ISBN 0201648652 9780201648652.

[3] Pacheco, P.: *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011, ISBN 9780123742605 0123742609.