Dokumentácia k projektu do predmetu PRL
Implementácia algoritmu *Odd-even transposition sort*

Matúš Liščinský
xlisci02@stud.fit.vutbr.cz

Úvod

Úlohou tohto projektu bolo implementovať paralelný radiaci algoritmus *Odd-even transposition sort* v jazyku C/C++ pomocou knižnice Open MPI. Okrem samotného algoritmu bolo potrebné vytvoriť testovací shell skript, ktorého cieľom je riadiť testovanie projektu.

Odd-even transposition sort

Popis algoritmu

Odd-even transposition sort je paralelný radiaci algoritmus pracujúci na lineárnej architektúre. Jeho základná myšlienka je prevzatá z algoritmu Bubble sort, ktorý postupne porovnáva a v prípade potreby zamieňa do požadovaného poradia dve za sebou idúce čísla v poli. Tieto porovnania a zámeny sa v algoritme Odd-even transposition sort riešia v dvoch fázach - párnej fáze a nepárnej fáze. V texte budeme ďalej pre jednoduchosť uvažovať radenie do vzostupnej postupnosti.

Na začiatku sa každému procesoru priradí jedna z radených hodnôt. V prvej fáze sa každý nepárny procesor p_i spojí so svojim pravým (párnym) susedom p_{i+1}, porovnajú svoje hodnoty a vymenia si ich, ak platí vzťah $p_i > p_{i+1}$. V druhej fáze sa takýmto spôsobom spoja a prípadne si vymenia hodnoty párne procesory p_i so svojimi nepárnymi susedmi p_{i+1}. Hodnoty sú v lineárnom poli n procesorov zoradené po prebehnutí maximálne n fáz.

Analýza algoritmu

Párna aj nepárna fáza v sebe zahŕňa zaslanie aktuálnej hodnoty susednému procesoru, porovnanie a následné spätné zaslanie nižšej z porovnávaných hodnôt. Pri každej z celkovo n behov fáz sa vykonávajú tri operácie (jedno porovnanie a dva prenosy). Počet týchto operácii je konštantný a nezávisí od veľkosti vstupu, čo indukuje konštantnú zložitosť $\mathcal{O}(1)$. Z toho vyplýva **časová zložitosť** $t(n) = \mathcal{O}(n) \cdot \mathcal{O}(1) = \mathcal{O}(n)$, čo je pre lineárnu topológiu najlepším možným výsledkom [1].

Za účelom porovnania prvkov si každý procesor potrebuje dočasne uchovať hodnotu susedného procesora, z čoho plynie, že **priestorová zložitosť** algoritmu $s(n) = \mathcal{O}(n)$.

Popis algoritmu jasne naznačuje, že pre radenie n prvkov potrebujeme n procesorov, preto **počet procesorov** $p(n) = n$. Pri radení veľkého množstva prvkov je z tohto dôvodu použitie tohto algoritmu prakticky nerealizovateľné.

Cena paralelného riešenia je všeobecne definovaná ako $c(n) = p(n) \cdot t(n)$, a je optimálna vtedy, ak je rovná časovej zložitosti optimálneho sekvenčného radiaceho algoritmu, teda $\mathcal{O}(n \cdot log(n))$ [1]. Je zrejmé, že cena paralelného algoritmu *Odd-even transposition sort* **nie je optimálna**, pretože $c(n) = \mathcal{O}(n^2)$.

Implementácia

Tento algoritmus je implementovaný v jazyku C++ za pomoci knižnice pre paralelné výpočty Open MPI. Na začiatok je pre inicializáciu MPI prostredia nevyhnutné zavolať knižničnú funkciu `MPI_Init`, po ktorej si už každý procesor uloží celkový počet procesorov a svoje id (rank) v rámci skupiny procesorov. Procesor s $id = 0$ následne prečíta sekvenciu bytov zo súboru `numbers` a až na prvú hodnotu, ktorú si ponechá, rozdistribuuje ostatné hodnoty medzi zvyšné procesory, pomocou `MPI_Send` funkcie. Keďže predpokladáme, že vstupom je sekvencia čísel v rozsahu 0 až 255, zo súboru sa číta po veľkosti 1B a ukladá do poľa typu `unsigned char`.

Jednotlivé procesory prijímajú priradené prvky funkciou `MPI_Recv`, a v rámci predprípravy sa vypočíta dvojica indexov pre posledné dva procesory a počet opakovaní hlavného cyklu ($n/2$), ktorý je potrebné v prípade nepárneho počtu procesov **zaokrúhliť nahor** (zapisujeme $\lceil n/2 \rceil$).

Nasleduje samotný algoritmus (Listing 1), kde sa $\lceil n/2 \rceil$ krát vystrieda nepárna fáza s párnou fázou. Obe fázy implementujú rovnakú logiku, rozdiel je iba v tom, či je iniciátor komunikácie párny alebo nepárny procesor v poradí. Komunikácia začína tak, že jeden procesor zašle svojmu susedovi svoju uloženú hodnotu, tento procesor ju príjme, porovná so svojou, uloží si vyššiu hodnotu z porovnávaných a druhú (nižšiu) odošle naspäť (funkcia `compare_and_swap` viz. Listing 2). Pre väčšiu prehľadnosť kódu sú funkcie `MPI_Send` a `MPI_Recv` obalené do nových funkcií `send_to` a `recv_from`, bez vyžadovania konštantných parametrov, ako je počet prvkov (1), dátový typ (`MPI_BYTE`), tag, komunikátor (`MPI_COMM_WORLD`), resp. adresa `MPI_Status` štruktúry.

```
# algorithm odd-even transposition sort
for _ in 1..N/2:
    # 1.phase, odd processors
    if is_odd(id) && id < last_even:
        send_to (right_neigh(id), &value)
        recv_from(right_neigh(id), &value)
    # even processors
    else if id <= last_even && id != 0:
        recv_from(left_neigh(id), &neigh_value)
        compare_and_swap(id, &value, &neigh_value)
    # 2.phase, even processors
    if (!is_odd(id) || id == 0) && (id < last_odd):
        send_to (right_neigh(id), &value)
        recv_from(right_neigh(id), &value)
    # odd processors
    else if id <= last_odd:
        recv_from(left_neigh(id), &neigh_value)
        compare_and_swap(id, &value, &neigh_value)
```

Listing 1: Implementovaný algoritmus Odd-even transposition sort zapísaný v pseudokóde.

```
# compare and swap
compare_and_swap(id, *value, *neigh_value):
    if(*neigh_value > *value){
        send_to(left_neigh(id), &(*value)));
        *value = *neigh_value;
    }
    else send_to(left_neigh(id), &(*neighval));
```

Listing 2: Pseudokód funkcie compare_and_swap

Po skončení, hlavný procesor prijíma od ostatných procesorov ich hodnoty, ktoré si ukladá do poľa a vypíše v požadovanom formáte na štandardný výstup. Následne sa po výpise použité polia uvoľnia z pamäte a zavolá sa funkcia `MPI_Finalize`, ktorá ukončí prostredie vykonávania MPI.

Experimenty

Pre overenie časovej zložitosti bol implementovaný algoritmus experimentálne otestovaný. Zaujímal nás predovšetkým výkon algoritmu bez okolitej réžie s ním spojenej, čo zahŕňa napríklad načítanie prvkov zo súboru, výpis prvkov na štandardný výstup, a podobne. Celkovo boli zozbierané údaje pre dva druhy testovania. V tom prvom sa experimentom podrobila časť kódu od počiatočného rozoslania hodnôt procesorom, až po zozbieranie zoradených prvkov z procesorov. Pri druhom testovaní sa meral výlučne čas strávený radením prvkov. Pre lepšie pochopenie, ktorý úseky kódu sa merajú, je možné nahliadnuť do pseudokódu hlavnej časti implementácie (viz. Listing 3), kde sú tučne zvýraznené miesta zbierania časových údajov. Režim testovania je v kóde možné zapnúť definovaním makra `TEST1`, resp. `TEST2`.

```
MPI_Init()
if id == 0:
    numbers = read_input()
# TEST1_start
if id == 0:
    for i in 1..N-1:
        send_to(i,&numbers[i])
else:
    recv_from(0, &value)
# TEST2_start
# algorithm odd-even transposition sort (Listing 1)
# TEST2_finish
if id == 0:
    for i in 1..N-1:
        recv_from(i, &results[i])
else:
    send_to(0, &value)
# TEST1_finish
if id == 0:
    print(results)
MPI_Finalize()
```

Listing 3: Pseudokód hlavnej časti implementácie s vyznačenými miestami merania časových údajov.

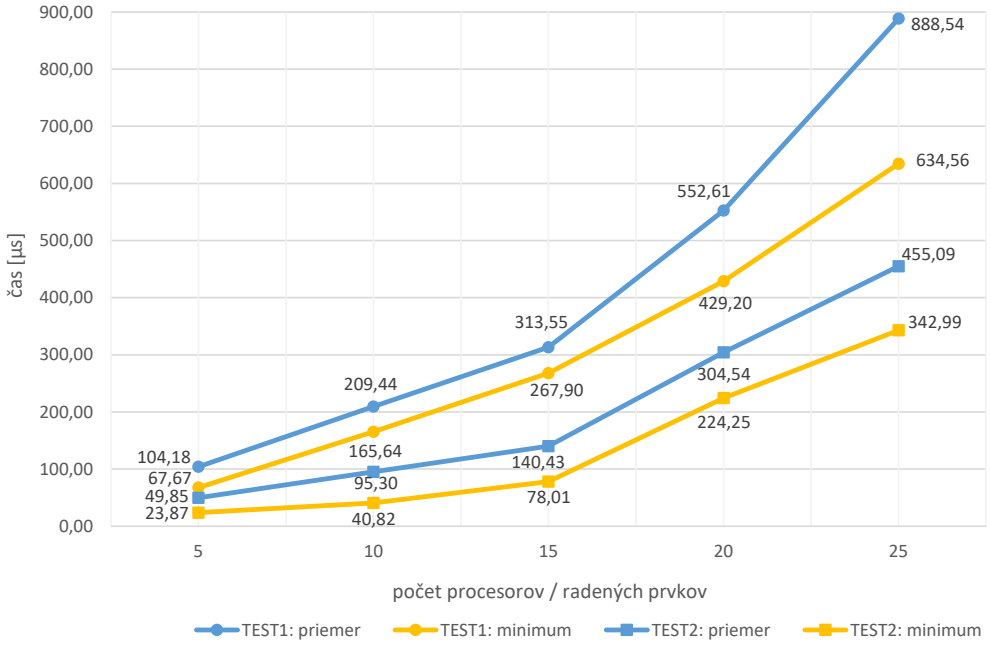
Pred začiatkom testovania je volaním funkcie `MPI_Barrier` vytvorená explicitná bariéra, aby boli všetky procesy pred spustením testovania zosynchronizované na rovnakej pozícii. Jednotlivé časy behu programu sú získané vďaka funkcii `MPI_Wtime`, rozdielom časových údajov pred a po vykonaní meraného úseku kódu. Výsledná hodnota pri jednom behu je zo všetkých procesorov zredukovaná na jednu (maximálnu) hodnotu funkciou `MPI_Reduce`. Pomocou spomínaných makier je taktiež zabezpečené, že pri behu programu pre účely experimentálneho testovania sa preskočia výpisy na `stdout`, pretože I/O operácie sú časovo náročné.

```
# TEST{1|2}_start
double start;
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

# code

# TEST{1|2}_finish
double finish = MPI_Wtime();
double loc_elapsed = finish-start;
double elapsed;
MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
if(id == 0) cout << elapsed * 1000000 << endl;
```

Listing 4: Implementácia zberu časových údajov pri testoch `TEST1`, resp. `TEST2`.

Pre účely testovania bol vytvorený bash skript, ktorý automatizovane spúšťal testovací skript s počtom prvkov 5, 10, 15, 20, 25 a pre každý z nich sa algoritmus otestoval 25-krát. Testovací skript `test.sh` je ošetrený tak, že berie práve jeden argument v podobe kladného celého čísla, v inom prípade končí chybovou hláškou. Na problém ako sa vysporiadať s variabilitou u výsledných časov prezentujeme dve riešenia: odstrániť 5 najvyšších časov a zvyšné časy spriemerovať, alebo vybrať minimum. Preto minimum, pretože je malá pravdepodobnosť, že by okolitý šum mal taký pozitívny vplyv na beh algoritmu, ktorý by viedol k lepšiemu dosiahnutému času.

Implementácia bola priebežne testovaná na lokálnom systéme *Ubuntu 19.10*, na školskom serveri *merlin*, a na superpočítači *anselm*, na ktorý sme ako študenti predmetu AVS dostali prístup. Výsledky experimentov sú práve z testovania na superpočítači, z dôvodu väčšej stability výsledných časov.



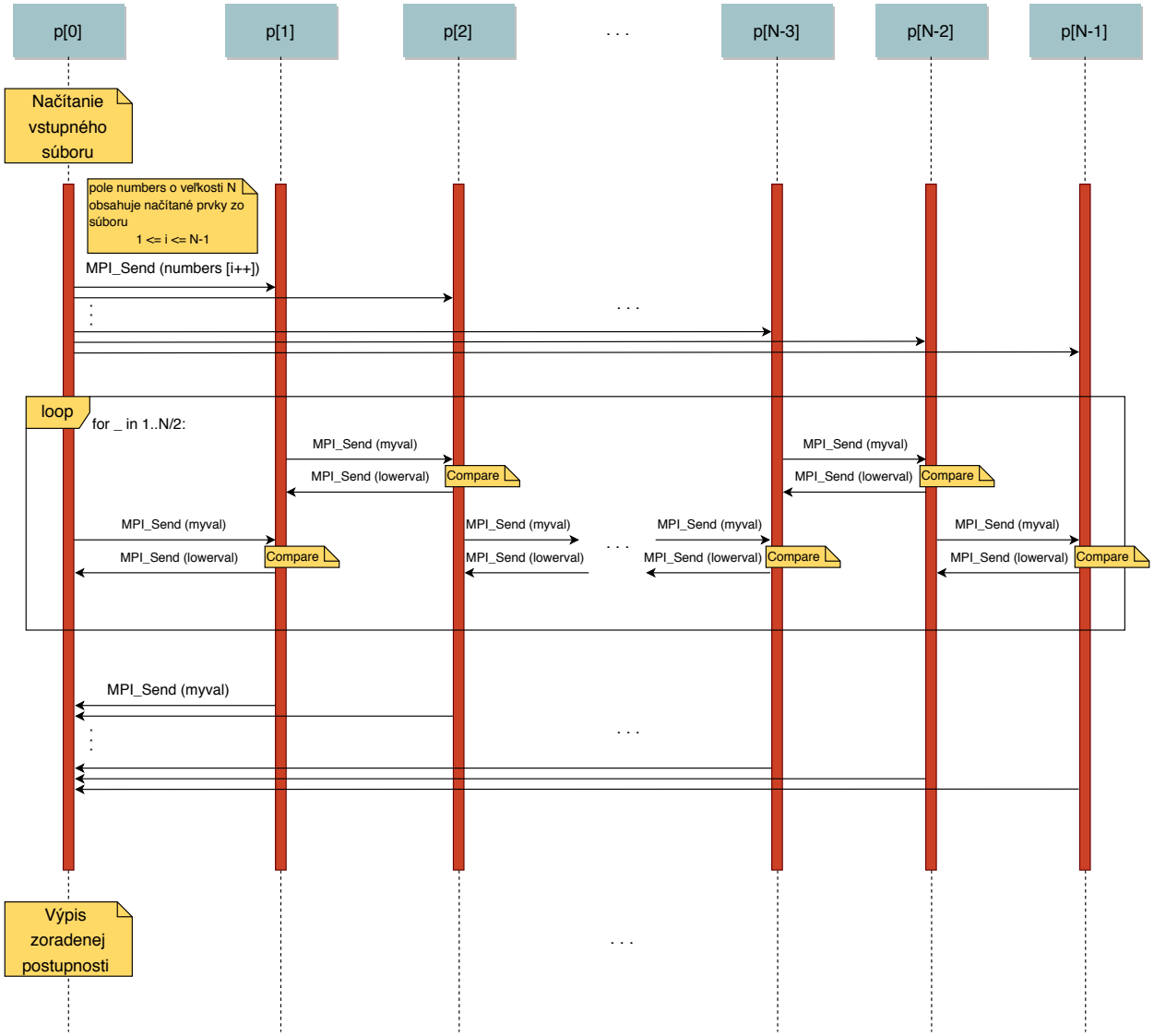
Obr. 1: Výsledky experimentálnych meraní z dvoch typov meraní a dvoch spôsobov ich vyhodnotení.

Záver

Na základe teoretických znalostí o princípe fungovania algoritmu *Odd-even transposition sort* sme vyvodili časovú zložitosť $\mathcal{O}(n)$ a cieľom experimentov bolo prakticky ju overiť. Z grafu pozorujeme, že pri menšom počte procesorov sa krivky držia istej lineárnej závislosti, avšak neskôr stúpajú pod väčším uhlom, čo môžeme pripísať väčšej réžii pri komunikácii medzi viacerými procesormi. Najviac sa to odrazilo práve pri spriemerovaní hodnôt z testovacej varianty č.1 (v grafe `TEST1`:priemer), kde sa meria väčší blok kódu s väčším množstvom medzi-procesorovej komunikácie. Medzi-procesorová komunikácia predstavuje väčšie množstvo práce a zaberá viac času v pomere k ostatnej práci, ktorá sa vykonáva paralelne, teda porovnanie dvoch prvkov.

Aj z tohto dôvodu vznikli ďalšie modifikácie tohoto algoritmu či už napríklad zvýšením počtu hodnôt na procesor ako *Merge-splitting sort* alebo zmenou topológie ako *Odd-even merge sort*. Komunikačný protokol vo forme sekvenčného diagramu (Obr. 2) sa nachádza na ďalšej strane spolu s použitou literatúrou.

Komunikačný protokol



Obr. 2: Sekvenčný diagram znázorňujúci komunikáciu medzi procesormi.

Literatúra

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