1. projekt v PRL 2018/2019: Paralelný radiaci algoritmus *Bucket sort*
Meno a priezvisko: Dávid Bolvanský
Login: xbolva00

1 Úvod

Cieľom tohto projektu je implementovať paralelný radiaci algoritmus *Bucket sort*. Súčasťou je aj shell skript, ktorý vypočíta optimálny počet procesorov pre zadaný počet prvkov.

2 Popis algoritmu

Princíp algoritmu spočíva v radení prvkov pomocou stromu procesorov. Prvky zo vstupu sa rovnomerne rozdelia do košov (*buckets*) medzi listové procesory. Každý listový procesor zoradí svoj koša optimálnym sekvenčným algoritmom. Následne na každej nelistovej úrovni stromu smerom ku koreňu dochádza k tomu, že každý uzol na danej úrovni spojí postupnosti od svojich synov za použitia napr. *straight merge*. Výsledkom tohto postupu je, že v koreni sa bude nachádza zoradená postupnosť prvkov zo vstupu.

Optimálny počet procesorov je daný vzťahom $p(n) = 2 * log_2(n) - 1$. Algoritmus štandardne predpokladá, že počet prvkov na vstupe je mocnina 2. Ak teda počet prvkov na vstupe je $n = 2^m$, potrebný počet listových procesorov je m. Nech b je počet prvkov v *buckete* listového procesoru, potom platia vzťahy $b = n/log_2(n)$ a $b = n/m$.

3 Analýza algoritmu

1. Koreňový procesor rozdelí postupnosť prvkov do m košov o veľkosti b. Každý listový procesor následne prečíta svoj kôš. Zložitosť tohto kroku je $\mathcal{O}(b)$, čiže $\mathcal{O}(n/log_2(n))$.

2. Každý listový procesor zoradí svoj kôš pomocou optimálneho sekvenčného radiaceho algoritmu s $\mathcal{O}(n \cdot log(n))$ - zložitosť tohto kroku je $\mathcal{O}(b/log(b)) \rightarrow \mathcal{O}(n/log(n)/log(n/log(n))) \rightarrow \mathcal{O}(n))$. Implementované pomocou `std::sort`.

3. Každý nelistový procesor spojí postupnosť prvkov od svojich synov. Použitím *straight merge* na spájanie postupností každá iterácia na určitej úrovni stromu zaberie $k.n/2^i$ krokov (k je konštanta). Spájanie postupností v uzloch vo všetkých iteráciách nám určujú zložitosť celého tohto kroku. Zložitosť tohto kroku je $\mathcal{O}(n)$. Implementované pomocou `std::merge`.

Zistená časová zložitosť je $t(n) = \mathcal{O}(n)$. Potrebný počet procesorov je $p(n) = 2 * log_2(n) - 1$, po odstránení multiplikatívnej a adatívnej zložky platí, že $p(n) = \mathcal{O}(log(n))$. Na každej úrovni stromu je uložených n čísiel a počet týchto úrovní je $log_2(n)$, z čoho je možné odvodiť priestorovú zložitosť $s(n) = \mathcal{O}(n \cdot log_2(n)))$. Cena paralelného riešenia je všeobecne definovaná ako $c(n) = t(n) \cdot p(n)$, v našom prípade cena paralelného *Bucket sortu* je $c(n) = \mathcal{O}(n \cdot log(n))$. Algoritmus má optimálnu cenou v prípade, že platí $c(n)_{optim} = t_{seq(n)}$. Keďže časová zložitosť optimálneho sekvenčného radiaceho algoritmu a aj cena paralelného *Bucket sortu* je $\mathcal{O}(n \cdot log(n))$, paralelný radiaci algoritmus *Bucket sort* je optimálny.

4 Implementácia

Algoritmus je implementovaný za pomoci knižnice Open MPI. Jedná sa kód v jazyku C doplnený o použitie funkcií `std::sort` [1] a `std::merge` [2] z STL jazyka C++. V použitom štandarde C++11 je u `std::sort` garantovaná časová zložitosť $\mathcal{O}(n \cdot log(n))$ a to aj v najhoršom prípade. Pri `std::merge` je garantovaná lineárna časovú zložitosť. Funkcia `std::sort` implementuje *Introsort*, čo je hybridný radiaci algoritmus spájajúci *Quick sort* s *Heap sortom*. Keďže čísla na vstupe sú z rozsahu 0 až 255, v implementácii sa používa dátový typ `unsigned char`, u MPI je to typ `MPI_BYTE`.

Na samotnom začiatku sa spustí funkcia `MPI_Init` na inicializáciu MPI prostredia. Koreňový procesor načíta čísla zo súboru `numbers` do alokovaného poľa, zistí ich celkový počet. Radenie pre prípad $n = 1$ prvkov je vykonávané len koreňovým procesorom. Inak vypočíta, koľko prvkov má byť v jednom koši. Tieto dva čísla sa rozpošlú (*broadcast*) pomocou `MPI_Bcast` všetkým procesorom. Následne koreňový procesor rozdistribuuje pomocou `MPI_Send` daný počet prvkov v koši listovým procesorom. Listový procesor tieto postupnosti čísiel načíta

[1] `https://en.cppreference.com/w/cpp/algorithm/sort`
[2] `https://en.cppreference.com/w/cpp/algorithm/merge`

(veľkosť koša zistí z premennej, ktorá bola už predtým rozoslaná) a zoradí ich pomocou funkcie `std::sort`. Po zoradení svojho koša ho pošle ďalej svojmu otcovi v strome. Nelistové procesory prijímajú postupnosti čísiel od svojich synov v strome pomocou `MPI_Recv`. Pomocou informácie o aktuálnej hĺbke v strome a celkového počtu prvkov na vstupe je možné zistiť, koľko prvkov majú tieto nelistové procesory prijať. Prijaté postupnosti sa následne pomocou funkcie `std::merge`, ktorá realizuje *straight merge*, spoja do zoradenej postupnosti, ktorú ďalej pošlú svojmu otcovi. V prípade, že koreňový procesor prijal a spojil postupnosti od svojich synov, je táto postupnosť uložená a vypísaná na štandardný výstup. Po výpise zoradenej postupnosti čísiel sa uvoľní alokované pole pre vstupnú postupnosť a spustí sa funkcia MPI_Finalize pre ukončenie MPI prostredia.

Zaujímavosťou je, že algoritmus štandardne predpokladá, že zadaný počet čísiel bude mocnina 2. Pre zovšeobecnenie algoritmu na akékoľvek možný vstup bolo potrebné vykonať pár nasledovných úprav. Výpočet potrebných procesorov sa nachádza v shell skripte `test.sh`. Na úrovni tohto výpočtu je potrebné riešiť situáciu, keď $log_2(n)$ nie je celé číslo. Ak sa jedná o tento prípad, číslo $log_2(n)$ sa zaokrúhli na najbližšiu mocninu 2 (označme toto číslo ako m). Ďalej sa počet procesorov vypočíta klasicky cez vzťah $p(n) = 2 * m - 1$. Keďže práca s logaritmami a celkovo s desatinnými číslami nie je v shelli zrovna najlepšia a najpresnejšia, je výpočet potrebného počtu procesorov realizovaný zavolaním `Python` interpretu, ktorý vyčísli vyššie vysvetlený matematický výraz na získanie počtu procesorov. Avšak pre prípad $n = 1$ by výpočet počtu procesorov bol problémový, a preto je táto situácia riešená explicitným nastavením počtu procesorov na 1. Po získaní správneho počtu procesorov sa spustí paralelný *Bucket sort* pomocou príkazu `mpirun -np` ⟨`počet procesorov`⟩ `bks`.

Na úrovni samotného programu (`bks.cpp`) je taktiež potrebných pár úprav. Ak $n/log_2(n)$ vychádza ako necelé číslo, číslo $log_2(n)$ sa zaokrúhli na najbližšiu mocninu 2 (označme toto číslo ako n_m). Ak výsledok delenia celkového počtu prvkov a tohto zaokrúhleného čísla je zas necelé číslo, toto číslo je potrebné ďalej zaokrúhliť na najbližšie celé číslo (označme toto číslo ako $next\_power\_m$). Následne nový počet čísiel je výsledok násobenia $n\_m * next\_power\_m$. Rozdiel medzi novým a starým počtom čísiel udáva, koľko čísiel je potrebné doplniť do vstupnej postupnosti, aby binárny strom z tejto postupnosti bol úplný. Keďže radíme čísla z rozsahu 0 až 255, vhodným kandidátom na toto „dopĺňacie“ číslo je 0. Po doplnení postupnosti daným počtom 0 sa spustí samotný algoritmus, ktorý už vďaka tejto úprave má na vstupe úplný binárny strom, a je teda zaručená jeho správna činnosť. Pri výpise zoradenej postupnosti sa preskočia doplnené nuly. Nový počet prvkov, s ktorým algoritmus pracuje, už nemusí byť 2^m, kde m je počet listových procesorov. Ak by sa táto podmienka musela dodržať, počet doplnených čísiel by bol značne vyšší a taktiež nezanedbateľný v pomere ku počtu čísiel vo vstupnej postupnosti.

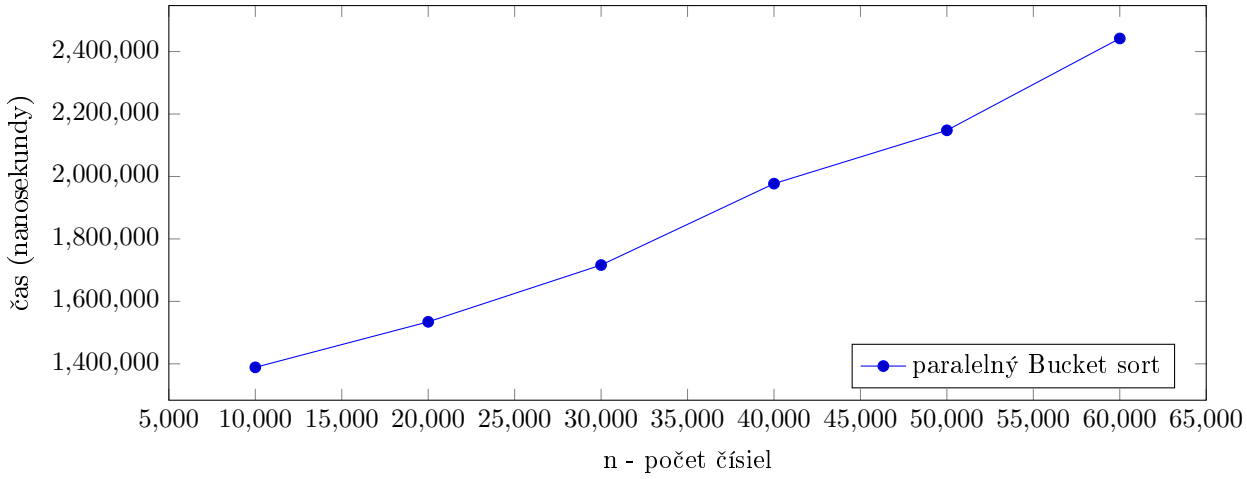
Príklad: Uvažujme $n = 17$. Číslo $log_2(17)$ je $4,08 \rightarrow$ po zaokrúhlení na mocninu $2 \rightarrow 8$. Číslo $17/8$ je $2,125$ a po zaokrúhlení na najbližšie celé číslo $\rightarrow 3$. Nový počet prvkov bude $3 * 8 \rightarrow 24$. Do postupnosti sa teda pridá $24 - 17$, čiže 7 núl. V prípade dodržania podmienky $n = 2^m$, by sa 17 zaokrúhlila na mocninu $2 \rightarrow 32$. Do postupnosti by sa teda muselo pridať až $32 - 17$, čiže 15 núl.

Implementáciu som priebežne testoval na svojom systéme Ubuntu 18.04 LTS a na školskom serveri *merlin* s CentOS 7.6. Na otestovanie implementácie a shell skriptu na výpočet procesorov som si napísal testovací skript, ktorý porovnával výstupy mojej implementácie paralelného *Bucket sortu* s výstupom referenčného programu na sekvenčné radenie čísiel, ktorý načítal súbor, zoradil ho pomocou `std::sort` a následne vypísal. Tento testovací skript je dostupný na `https://github.com/xbolva00/PRL1-Bucket-Sort-Tester`.

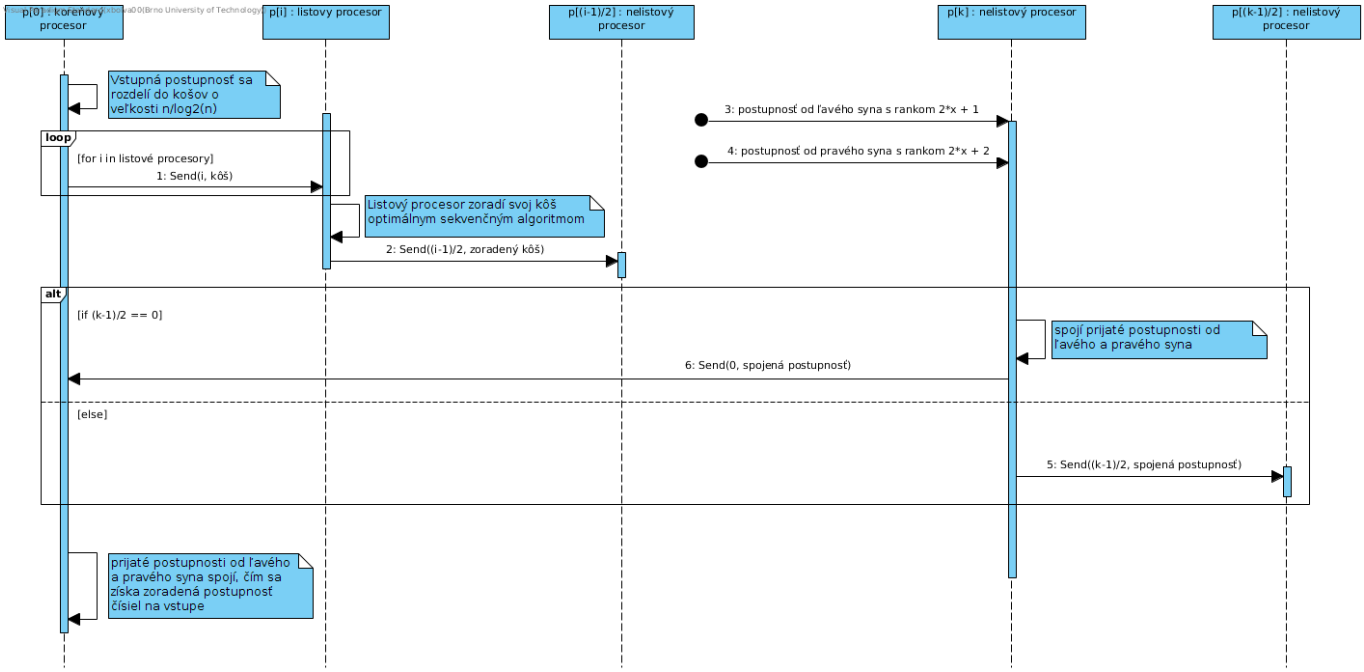
5 Experimenty

Cieľom experimentov bolo overiť časovú zložitosť paralelného *Bucket sortu*. Pre účely zistenia časovej zložitosti bolo potrebné eliminovať rušivé vplyvy, ktoré nesúvisia so samotným algoritmom. Vhodným vložením meracích miest do implementácie bolo možné správne zistiť časovú zložitosť - do výsledných časov sa nerátajú činnosti ako sú napr. načítanie čísiel zo súboru, výpis čísiel, alokácia a dealokácia vstupného pola. Pomocou funkcie `MPI_Wtime` som si zistil aktuálny čas pred spustením algoritmu (tj. predtým, ako koreňový procesor rozdistribuuje koše listovým procesorom) a čas po skončení algoritmu (tj. pred výpisom zoradenej postupnosti). Tieto zistené časy som od seba odčítal pre zistenie doby behu samotného algoritmu. Rozdiel časov som následne zo sekúnd previedol na nanosekundy. Pre každý zvolený počet čísiel n som vykonal 13 meraní na školskom serveri *merlin* - prvotný výsledok so šumom som zahodil a následne som taktiež zahodil najlepší a najhorší výsledok. Zo zostávajúcich 10 meraní sa spriemerovaním dosiahla finálna reprezentatívna hodnota pre daný počet čísiel. Režim merania doby behu algoritmu je možné zapnúť definovaním makra `MEASURE_TIME`. Vstupná postupnosť čísiel na experimenty bola získaná z `/dev/urandom`.

n	10000	20000	30000	40000	50000	60000
ns	1388773	1534433	1716377	1977012	2147857	2441973



6 Komunikačný protokol



Obr. 1: Sekvenčný diagram popisujúci komunikáciu n procesov u paralelného Bucket sortu

7 Vyhodnotenie experimentov a záver

Výsledky meraní a zhotovený graf v kapitole 5 poukazujú na fakt, že čas potrebný na radenie prvkov rastie lineárne voči počtu radených prvkov. Toto zistenie podložené vykonanými experimentami je v súlade s odvodenou teoretickou časovou zložitosťou v kapitole 3. Mierne výkyvy pri meraní možno pripísať rôznemu zaťaženiu školského servera v čase, kedy prebiehali experimenty.

8 Informačné zdroje

Distribuované a paralelní algoritmy a jejich složitost, algoritmy řazení (prednáška v predmete PRL)
`https://www.fit.vutbr.cz/study/courses/PDA/private/www/h003.pdf`