FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Předzpracování dat pro řazení algoritmem Sample sort
Student:	Pavel Erazím
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1) Nastudujte algoritmus Quicksort [1] a jeho zobecněnou verzi Sample sort [2],[3],[4].
2) Analyzujte vlastnosti těchto algoritmů co se týče časové složitosti, paměťové složitosti a stability.
3) Implementujte algoritmus Sample sort v jazyce C/C++. Použijte algoritmus Sample sort k seřazení dat. Dále použijte Sample sort pouze k předzpracování dat, jednotlivé předzpracované části následně seřaďte alespoň 2 jinými řadícími algoritmy.
4) Paralelizujte řešení z bodu 3) pomocí OpenMPI API.
5) Změřte výkonnost těchto jednotlivých implementací, porovnejte s implementací sort v GNU GCC.

Seznam odborné literatury

[1] https://academic.oup.com/comjnl/article/5/1/10/395338

[2] https://dl.acm.org/citation.cfm?id=321600

[3] http://parallelcomp.uw.hu/ch09lev1sec5.html

[4] https://www.researchgate.net/publication/234795819_Samplesort_A_Sampling_Approach_to_Minimal_Storage_Tree_Sorting

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 14. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Předřazení dat pro řazení algoritmem Sample sort

Pavel Erazím

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

16. května 2019

Poděkování

Rád bych poděkoval vedoucímu práce, doc. Ing. Ivanu Šimečkovi, Ph.D., za jeho ochotu a poskytnuté rady během psaní práce. Dále bych rád poděkoval své rodině za podporu během mého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 .....

Odkaz na tuto práci

Erazím, Pavel. *Předřazení dat pro řazení algoritmem Sample sort.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato bakalářská práce se zaměřuje na řadící algoritmus Samplesort.Cílem bylo algoritmus implementovat a paralelizovat. V této práci ukáži, že představená implementace pro CPU se podle výsledků měření ukazuje jako efektivnější než některé stávající implementace řadicích algoritmů.

Klíčová slova řadicí algoritmy, Quicksort, Samplesort, Flashsort, rozděl a panuj, předřazení, openMP, C++, paralelizace.

Abstract

This thesis focuses on sorting algorithm Samplesort. The main goal of this thesis was to implement and parallelize Samplesort. In this thesis I will show that the proposed CPU implementation proves to be even more effective than some existing sorting algorithms.

Keywords sorting algorithms, Quciskort, Samplesort, Flashsort, divide and conquer, pre-sorting, openMP, C++, paralelization.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Seřazení vstupních dat podle různých kritérií je úloha, jejíž vyřešení je velmi často požadováno nejen v informatických oborech. V současné době pracují počítačové programy se stále narůstajícím množstvím dat a informací. Tyto informace je potřeba rychle a efektivně seřadit.

Algoritmy, které takové úlohy řeší, nazýváme řadicí (nebo také třídicí). Známých řadicích algoritmů existuje poměrně velké množství. Liší se od sebe mimo jiné velikostí operační paměti, kterou potřebují pro svůj běh, stabilitou a časovou náročností.

Některé algoritmy, například Bucketsort ze skupiny třídicích algoritmů známé jako „Rozděl a panuj", pracují rychleji, pokud jsou vstupní data uniformně rozdělena, ale na jiných rozděleních jejich efektivita degraduje. Algoritmus Samplesort se tento problém snaží vyřešit.

Samplesort je zobecněná verze algoritmu Quicksort. Problém neuniformího rozdělení dat řeší tak, že vždy vybírá náhodný vzorek (angl. *sample*) prvků ze vstupních dat. Tento vzorek seřadí a následně určí intervaly, podle kterých data rozdělí do jednotlivých přihrádek. Tento krok může i několikrát opakovat a vytvářet tak mnohem menší přihrádky. Rozdělení dat v jednotlivých přihrádkách se tak blíží uniformnímu. Pokud dosáhne předem dané velikosti přihrádky, poté obsah všech jednotlivých přihrádek seřadí nějakým vhodným řadicím algoritmem. Efektivita takového algoritmu závisí na velikosti vzorku, způsobu jeho výběru a na použitém řadicím algoritmu.

V této práci se zabývám analýzou a implementací algoritmu Samplesort, nalezením vhodné hranice a vhodného řadicího algoritmu. Vycházím z původní práce, pospané v [1]. Věnuji se implementacia paralelizaci algoritmu pro použití na CPU, existují ovšem také práce, které se věnují paralelizaci na GPU (např. [2]).

V první části práce se věnuji analýze algoritmu Samplesort a obdobným algoritmům. Ve druhé, praktické části, se věnuji implementaci a paralelizaci svého algoritmu, změření doby běhu jednotlivých implementací a vzájemnému

porovnání efektivity. Na závěr provádím porovnání výsledků jednotlivých měření.

Kapitola 1

Cíl práce

1.1 Cíle práce

Cílem této práce je navrhnout vlastní algoritmus Samplesort a poté diskutovat smysl jeho použití k seřazení velkého množství dat. Algoritmus nejprve data vhodně předřadí, poté použije vhodný algoritmus k seřazení těchto předřazených částí.

Teoretická část je zaměřena na analýzu algoritmu Samplesort a jiných vhodných řadicích algoritmů. Dále popisuji návrh svého algoritmu. V této části vycházím z práce W. D. Frazera a A. C. McKellara [1].

V praktické části implementuji vlastní algoritmus Samplesort. Cílem praktické části je použít můj algoritmus pro předřazení dat a dále data seřadit pomocí vhodně zvoleného řadicího algoritmu. Dalším cílem praktické části je nejprve paralelizovat mé řešení a změřit efektivitu jednotlivých verzí algoritmu a poté tyto verze vzájemně porovnat.

1.2 Základní pojmy

Vstup (neboli *vstupní posloupnost*) je posloupnost dat, které algoritmus přijímá a na jejichž základě provádí svou činnost.

Výstup (neboli *výstupní posloupnost*) je posloupnost dat, kterou algoritmus po skončení svého běhu vrací a jež je výsledkem jeho činnosti.

Vzorek je množina náhodně vybraných prvků ze vstupní posloupnosti.

Přihrádka (angl. *bucket*) je podmnožina vstupních prvků se stejnou vlastností. V této práci jedna přihrádka obsahuje prvky ze stejného číselného intervalu.

Pivot je náhodně vybraný prvek, který rozděluje vstup na dvě (resp. tři) části: *levou* část tvoří prvky menší než pivot, *pravou* část prvky větší než

pivot (resp. *prostřední* část, kam patří prvky rovny pivotu). V případě řazení sestupně části analogicky otočíme.

Splitter je zobecněný pivot. Je to prvek vybraný ze *vzorku*. Množina po sobě jdoucích seřazených *splitterů* určuje intervaly, podle kterých vstup rozdělujeme do *přihrádek*.

Stabilní algoritmus zachovává při řazení takové pořadí ekvivalentních prvků, které zaujímaly na *vstupu*.

In-place algoritmus provádí řazení uvnitř vstupního pole.

Out-of-place algoritmus vytváří pro řazení pomocná pole, do kterých prvky ukládá.

Binární vyhledávání je algoritmus, který začíná vyhledávat prvek uprostřed seřazeného pole. Pokud je hledaný prvek menší než vyhledaný prvek, posune se vpravo (analogicky pro prvek větší se posune vpravo). Při každém posunutí se vyhledávání vždy posune v poli o poloviční vzdálenost než v předchozím kroku. Takovéto vyhledávání se podle [3] řídí složitostí $\mathcal{O}(\log n)$.

Offset označuje vzdálenost nějakého prvku od začátku pole, ve kterém je uložený.

Prefixový součet pro posloupnost $A = a_1, a_2, \ldots, a_n$ pro index $1 \leq i \leq n$ je součet všech předchozích prvků až po prvek a_i včetně .

Kapitola 2

Analýza algoritmů

Jak již bylo uvedeno, algoritmus Samplesort je zobecněný Quicksort. K pochopení Samplesortu si nejprve vysvětlíme princip Quicksortu.

2.1 Analýza algoritmu Quicksort

Quicksort je jeden z nejznámějších a nejoblíbenějších třídicích algoritmů ze skupiny "Rozděl a panuj". Pro tuto skupinu je význačné, že zadaný problém rozkládá na více menších podproblémů, dokud problém není rozložen na nějaký triviální případ, který umí snadně vyřešit [4]. Pro svou činnost využívají rekurze. Quicksort je jednoduchý na implementaci a dosahuje velmi dobrých výsledků pro reálná data. Existují však takové vstupy, kdy efektivita Quicksortu degraduje, protože algoritmus volí nevhodné pivoty (podrobněji popsáno v Sekci 2.1.1).

2.1.1 Princip algoritmu Quicksort

Činnost algoritmu Quicksort pro seřazení vstupu vzestupně lze popsat následovně (pro seřazení posloupnosti sestupně anologicky obrátíme postup ve fázi dvě):

Volba pivota

V první fázi dochází k volbě **pivota**. Tato část je pro algoritmus kritická, neboť volba nevhodného pivota[1] nám výrazně prodlouží dobu běhu algoritmu ve druhé a třetí fázi). Pro volbu pivota existuje několik strategií. Nejjednodušší z nich volí pivot náhodně, trochu pokročilejší pak jako pivot zvolí medián náhodného výběru několika prvků.

[1] Nevhodně zvolený pivot je například největší nebo nejmenší číslo na vstup [4, str. 10]

Rozdělení vstupu

Ve druhé fázi dochází k rozdělení vstupu na dvě části. Existují dvě verze algoritmu: *out-of-place* verze, která k rozdělení používá nových pomocných polí; a *in-place* verze, která prvky mezi sebou prohazuje pouze v rámci vstupního pole. Druhá varianta je popsána v původní práci [4].

1. **Out-of-place verze**
   Algoritmus zleva prochází každý prvek a porovnává jej s pivotem.

   - Zařazení prvků rovných pivotu záleží na konkrétní implementaci, častá volba je zavedení *prostřední části* (***M***). Toto také efektivně řeší problém, kdy vstup obsahuje jeden prvek několikrát a Quicksort by ho tak rozřazoval po několik iterací.
   - Je-li prvek menší než pivot, zařadí ho algoritmus do *levé části* (***L***).
   - Větší prvek zařadí do *pravé části* (***R***).
   - Každou část reprezentujeme jako nějaké nově vytvořené pomocné pole.

   Tato implementace zachovává stabilitu algoritmu (více v Sekci 2.1.3).

2. **In-place verze**

   Uvažujme nyní implementaci popsanou v [4, str. 10]. Algoritmus ve druhé fázi prochází pole z obou konců zároveň a snaží se najít hranici a uspořádat pole tak, aby prvky vlevo od této hranice byly menší nebo rovny pivotu a prvky vpravo od této hranice byly větší nebo rovny pivotu. Algoritmus se nejprve pohybuje zleva doprava a postupně porovnává každý prvek s pivotem.

   - Pokud je prvek menší než pivot, pokračuje následujícím prvkem. Narazí-li při tomto pohybu na nějaký prvek a_m větší než pivot, zapamatuje si pozici tohoto prvku a začne procházet pole zprava doleva.
   - Pokud algoritmus narazí na nějaký prvek a_n, který je menší než pivot, zastaví pohyb a prohodí tento prvek s a_m.
   - Poté se zleva i zprava posune o jeden prvek ve svém směru pohybu a pokračuje opět zleva doprava.
   - Takto algoritmus opakuje svou činnost, dokud pozice prvku zkoumaného zleva není napravo od prvku zkoumaného zprava, tedy dokud se jejich cesty z obou stran nepřekříží. V tom případě se zastaví a oba prvky určují *hranici*, která rozdělí pole na dvě hledané části.

Pokud je tato hranice určena mimo pole (například při nevhodném výběru pivota nebo pokud je hodnota všech prvků stejná), lze postupovat následovně:

a) pokud je pivot menší nebo roven všem ostatním prvkům, lze jej prohodit s prvním prvkem a snížit tím velikost pole o 1,

b) pokud je pivot vetší než nebo roven všem prvkům, analogicky jej lze prohodit s posledním prvkem a také snížit velikost pole o 1.

Toto nám zajistí, že se algoritmus skončí, neboť při každé iteraci snížíme velikost pole alespoň o 1 prvek.

Rekurzivní seřazení přihrádek

Ve třetí fázi Quicksort rekurzivně seřadí každou část z druhé fáze zvlášť. Takto seřazené je spojí za sebe v pořadí *levá*, *prostřední* a *pravá část* a vrátí jako výstup.

Algoritmus tyto kroky rekurzivně opakuje na zmenšující se pole, dokud problém nerozloží na triviální případ, v tomto případě pole o jednom prvku, které vrátí jako již seřazené. Protože všechny prvky v *levé části* jsou menší než všechny prvky v *pravé části*, je výstupem seřazená vstupní posloupnost.

2.1.2 Pseudokód

Předešlé procedury lze popsat následujícími algoritmy.

Algorithm 1 QuickSort - out-of-place verze

Vstup: Vstupní posloupnost čísel $A = (a_1, a_2, \ldots, a_n)$
Výstup: Seřazená vstupní posloupnost A

```
1: if n ≤ 1 then
2:     return A
3: end if
4: L = {}                                   ▷ Prázdné seznamy
5: M = {}
6: R = {}
7: for each a ∈ A do                        ▷ Od prvního prvku k poslednímu
8:     if a < p then
9:         L ← a                            ▷ Připoj a na konec seznamu L
10:    else if a > p then
11:        R ← a
12:    else
13:        M ← a
14:    end if
15: end for
16: return spoj(QuickSort(L), QuickSort(M), QuickSort(R))
```

Algorithm 2 QuickSort - in-place verze [4, 5]

Vstup: Vstupní posloupnost čísel $A = (a_1, a_2, \ldots, a_n)$
Výstup: Žádný, algoritmus seřadí posloupnost přímo ve vstupním poli.

```
1: if n ≤ 1 then
2:     return
3: end if
4: LP → begin(A)                          ▷ Pointer na začátek pole A
5: RP → end(A)                            ▷ Pointer na konec pole A
6: P ← random(A)                          ▷ Vyber náhodný pivot
7: while LP ≤ RP do  ▷ LP ukazuje na adresu nižší nebo stejnou jako RP
8:     if LP_value > P then
9:         while LP ≤ RP do
10:            if RP_value < P then
11:                swap(LP_value, RP_value)   ▷ Prohodí prvky, na které oba
   pointery odkazují
12:                LP ← LP + 1                ▷ Posune oba pointery o 1
13:                RP ← RP − 1
14:                break
15:            end if
16:            RP ← RP − 1
17:        end while
18:    else
19:        LP ← LP + 1
20:    end if
21: end while
```

2.1.3 Analýza algoritmu Quicksort

Analýzu uvádím pro originální *in-place* verzi algoritmu, jak jej popsal Hoare v [4]. Pro odvození časové náročnosti pro *out-of-place* verzi bychom postupovali analogicky – algoritmus ve fázi rozdělení pole navíc provede přidání prvku na konec pole/seznamu do části, kam prvek patří.

Časová složitost Quicksortu

Časovou analýzu Quicksortu uvádím tak, jak byla popsána v [5].

Z algoritmů popsaných výše lze nahlédnout, že většinu času běhu programu tvoří rozdělování pole, tedy porovnávání a prohazování prvků. Pro rozdělení vstupu o velikosti N musí provést N porovnání. V nejhorším případě (výběr nevhodného pivota) v každé iteraci algoritmu snížíme velikost vstupu o 1. Algoritmus v tomto případě musí provést N-i+1 iterací a na začátku i-té iterace ($1 \leq i \leq n$-1) je velikost pole N-i+1 a Quicksort musí provést právě tolik porovnání.

Nechť $\mathbf{W}(n)$ značí *časovou složitost* algoritmu *v nejhorším případě*. Počet porovnání ve všech iteracích lze tedy vyjádřit jako

$$n + (n-1) + (n-2) + \ldots 1 = \sum_{i=1}^{n-1}(n-i+1) = \frac{n(n-1)}{2} \tag{2.1}$$

a tedy

$$\mathbf{W}(n) = \frac{n(n-1)}{2} = \mathcal{O}(n^2) \tag{2.2}$$

Představme si nyní nejlepší případ, tedy pivot je vždy vybrán přesně jako medián posloupnosti, tedy rozděluje pole na dvě poloviny přibližně stejné délky. Buď $\mathbf{B}(n)$ *časová složitost* algoritmu *v nejlepším případě*. Snadno lze spočítat, že

$$\mathbf{B}(n) \approx n-1+2B(\frac{n}{2}) = (n+1)+2(\frac{n}{2}-1)+4B(\frac{n}{4}) = \ldots = \mathcal{O}(n \log n) \tag{2.3}$$

Spočtěme nyní $\mathbf{A}(n)$ časovou složitost *v průměrném případě*. Tu lze vyjádřit následující rekurentní rovnicí:

$$\mathbf{A}(n) = \begin{cases} 0 & n = 1 \\ n - 1 + \sum_{i=1}^{n} \frac{1}{n}[A(i-1) + A(n-i)] & n > 1 \end{cases} \tag{2.4}$$

Výraz lze upravit na

$$\mathbf{A}(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) = n - 1 + \frac{2}{n}[A(1) + A(2) + \ldots + A(n-2)] \tag{2.5}$$

tedy

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2n-1}{n(n+1)} \tag{2.6}$$

a protože $\mathbf{A}(0) = 0$, tak

$$\begin{aligned} \frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}\frac{A(n-2)}{n-1} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} = \\ = \ldots = \frac{2}{2\cdot 3} + \frac{4}{3\cdot 4} + \ldots + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} = \\ = 2\sum_{i=1}^{n}\frac{i-1}{i(i+1)} = 2\sum_{i=1}^{n}[\frac{1}{i} - \frac{2}{i(i+1)}]2\sum_{i=1}^{n}\frac{1}{i} - 4\sum_{i=1}^{n}\frac{1}{i(i+1)} = \\ = 2\sum_{i=1}^{n}\frac{1}{i} - 4\sum_{i=1}^{n}(\frac{1}{i} - \frac{1}{i+1}) = 2\sum_{i=1}^{n}\frac{1}{i} - 4(1 - \frac{1}{n+1}) = \\ = 2\sum_{i=1}^{n}\frac{1}{i} - \frac{4n}{n+1} \end{aligned} \tag{2.7}$$

Jak je uvedeno v [5]: pokud přijmeme přibližný odhad harmonické řady a předpokládáme, že n je dostatečně velké, potom

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + \gamma = \ln 2 \cdot \log_2 n = 0{,}693 \cdot \log_2 n + \gamma, \qquad \gamma \approx 0,577 \text{ je Eulerova konstanta} \tag{2.8}$$

tedy

$$\frac{A(n)}{n+1} \approx 2 \cdot (0{,}693 \cdot \log_2 n + 0{,}577) - \frac{4n}{n+1} \tag{2.9}$$

a tudíž časová složitost Quicksortu činí

$$\mathbf{A}(n) \approx 1{,}386 \cdot (n+1) \cdot \log_2 n - 2{,}846 \cdot n + 1{,}154 \approx \mathcal{O}(n \cdot \log n). \tag{2.10}$$

Paměťová složitost Quicksortu

Paměťovou složitost je nutné analyzovat pro in-place i out-of-place implementaci zvlášť. Nejprve uvádím in-place verzi.

In-place Quicksort

In-place verze algoritmu nepotřebuje žádné pomocné pole, prvky jsou řazeny přímo ve vstupním poli. K rozdělení pole v každé iteraci potřebuje konstantní množství dočasných proměnných - vybraný pivot, dva pointery (nebo indexy) ukazující na začátku na první a poslední prvek v poli a dočasnou proměnnou potřebnou k prohození dvou prvků. Paměťová složitost rozdělení pole je tedy vždy $\mathcal{O}(1)$.

Dále jako rekurzivní algoritmus využívá zásobník k ukládání adresy jednotlivých volání procedury. V nejhorším případě je těchto volání $\mathcal{O}(n^2)$, v nejlepším a průměrném případě $\mathcal{O}(\log n)$.

Out-of-place Quicksort

U *out-of-place* implementace navíc algoritmus vždy, když provádí porovnání zbytku pole s pivotem, uloží porovnávaný prvek do jednoho ze dvou (nebo tří) nově vytvořených polí. Je-li n velikost pole na začátku současné iterace, pak je součet velikostí všech dočasných polí roven právě n.

V **nejhorším případě** je v každé iteraci pole zmenšeno jen o 1 prvek. Označme *paměťovou složitost v nejhorším případě* jako $\mathbf{M}_{worst}(n)$. Velikost součtu $\mathbf{M}$ všech takto vytvořených polí do doby, než algoritmus narazí na triviální případ a začne se vynořovat z rekurze, je

$$\mathbf{M} = n + (n-1) + (n-2) + \ldots + 2 + 1, \tag{2.11}$$

což je aritmetická řada se součtem

$$\sum_{i=1}^{n} i = \frac{n(n-1)}{2} \tag{2.12}$$

a tedy

$$\mathbf{M}_{worst}(n) = \mathcal{O}(n^2). \tag{2.13}$$

V **ideálním případě** algoritmus dělí pole vždy na polovinu. Řazení obou polovin probíhá postupně, v i-té iteraci tedy algoritmus využívá pomocnou paměť $M(n, i)$ o velikosti

$$M(n,i) = \sum_{j=1}^{i} \frac{n}{2^{j-1}} = n + \frac{n}{2} + \frac{n}{4} + \ldots + \frac{n}{2_i}. \tag{2.14}$$

Buď $\mathbf{M}_{best}(n)$ *pamětová složitost* Quicksortu *v ideálním případě*. Než algoritmus narazí na triviální případ a začne se vracet z rekurze, provede právě $\log n$ zanoření a využívá

$$n + \sum_{k=1}^{\log_2 n} \frac{n}{2^k} = n + (\frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1) \tag{2.15}$$

dodatečné paměti. Odhad paměťové složitosti lze v tomto případě vyjádřit jako

$$\mathbf{M}_{best}(n) = \mathcal{O}(n) \tag{2.16}$$

Obdobně podle časové analýzy Quicksortu v Sekci 2.1.3 (a tedy i podle počtu jednotlivých porovnání) je *paměťová náročnost v průměrném případě*

$$\mathbf{M}_{average}(n) = \mathcal{O}(n) \tag{2.17}$$

Stabilita Quicksortu

Stabilita Quicksortu se liší podle implementace. Varianta *out-of-place* je **stabilní**, neboť ve fázi dvě, popsané v předchozí části, algoritmus postupně od prvního prvku rozřazuje vstup podle pivotu. Platilo-li pro nějaký prvek p_{k+l}, že se ve vstupní posloupnosti nacházel vpravo od nějakého prvku p_k, potom bude ve druhé fázi zpracován později, a tedy i později zařazen na konec pole. Tato verze algoritmu je tedy stabilní.

In-place varianta však může být **nestabilní**. Protože algoritmus v tomto případě prohazuje prvky z opačných konců pole, může nastat následující případ: Uvažujme *in-place* variantu popsanou v předchozí části. Určitě existuje nějaká vstupní posloupnost

$$a = a_1,a_2,\ldots,a_n,a_{n+1},\ldots,a_{n+m}$$

taková, že prvky a_n a a_{n+1} tvoří *hranici*, rozdělující pole na dvě části, a obsahuje prvky a_s, a_t, a_u a a_v takové, že

$$s < t < u < v \quad \text{a} \quad a_u = a_v \tag{2.18}$$

Nechť algoritmus vybral v první fázi nějaký pivot p, pro který platí

$$a_s, a_t > p \quad a \quad a_u, a_v < p. \tag{2.19}$$

Potom ve druhé fázi algoritmu může dojít k situaci, kdy budou nejprve prohozeny prvky a_s a a_v a později také a_t s a_u. Potom bude platit, že

$$v < u, \tag{2.20}$$

čímž dojde k porušení stability.

2.2 Analýza algoritmu Samplesort

Řadicí algoritmus Samplesort je zobecněnou verzí předešlého Quicksortu (popsaného v Sekci 2.1). Oproti Quicksortu však nevybírá pouze jeden *pivot*, ale několik prvků (tzv. ***splitterů***) z daného většího ***vzorku***, tj. prvků vybraných ze vstupní posloupnosti. Toto nám nabízí možnosti škálovatelnosti programu. Lze například na základě velikosti vstupu zvolit, jak velký *vzorek* vybereme a na kolik částí vstup rozdělíme.

Algoritmus data sám neřadí, pouze je rozděluje do přihrádek. K seřazení dat používá jiný řadicí algoritmus.

Svoji implementaci in-place sekvenčního Samplesortu podrobněji popisuji v Kapitole 3.

2.2.1 Princip algoritmu Samplesort

Průběh jedné iterace Samplesortu lze rozdělit do 4 navazujících fází:

Výběr vzorku

V první fázi dochází k výběru náhodného *vzorku* V ze vstupní posloupnosti. Vzorek následně seřadíme vhodným řadicím algoritmem. Dostatečnou velikostí vzorku lze získat poměrně dobrý odhad rozdělení pole do *přihrádek* a snížit tím tak pravděpodobnost, že vzájemný rozdíl mezi jejich velikostmi bude příliš velký [2, str. 6]. Toho lze dosáhnout například výběrem vzorku K-násobně většího, než je počet *splitterů*. Hledáním optimálního koeficientu $\mathbf{K}$, uvedeného v [2] jako „*oversampling*“, se zabývám v Kapitole 5.

Určení přihrádek

Vhodným způsobem ze vzorku **V** vybereme *množinu splitterů* $\mathbf{S} = (s_1, s_2, \ldots, s_{p-1})$. Vybíráme z již seřazeného vzorku, proto při postupném výběru zleva doprava vybereme již seřazenou posloupnost. Každé dva sousední *splittery* určují interval *přihrádek B* (angl. *buckets*), do kterých lze vstup roztřídit. První, respektive poslední *splitter* určuje interval $(-\infty, s_1)$, respektive $(s_{p-1}, +\infty)$.

Rozdělení vstupu

Vstup nyní rozdělíme do příslušných *přihrádek* určených ve druhé fázi. Zařazování prvků probíhá podobně jako u Quicksortu, pouze je určen index přihrádky, do které prvek patří. K tomu je vhodné použít například binární vyhledávání.

Out-of-place verze funguje podobně jako u Quicksortu. Vytváří pomocná pole, která pak seřadí a na konci iterace vrátí spojená za sebe. Navíc si zachovává stabiltu.

Také *in-place* implementace je velmi podobná jako u Quicksortu. Prvky jsou v poli prohazovány obdobným způsobem jako v případě Quicksortu, čímž může být porušena stabilita.

Rekurzivní seřazení přihrádek

Na závěr dochází k postupnému seřazení jednotlivých *přihrádek*. Na každou přihrádku je rekurzivně volán Samplesort. Přihrádky rozdělují vstup na po sobě jdoucí intervaly, lze je tedy postupně spojit a vrátit na výstup jako seřazené pole.

Takto Samplesort pokračuje až do chvíle, kdy velikost vstupního pole klesne pod předem určenou **hranici H**. Poté je na seřazení přihrádek použit jiný vhodný řadicí algoritmus. Například Frazer a McKellar ve své práci [1] použili Quicksort pro všechny další iterace.

2.2.2 Pseudokód pro algoritmus Samplesort

Následující pseudokód, vycházející z [2, str. 5], popisuje obecný Samplesort. Funkce *Sort* označuje libovolný alternativní řadicí algoritmus.
Funkce *Concat* spojí pole, daná jako argumenty, v uvedeném pořadí za sebe do jednoho pole.

Algorithm 3 SampleSort

Vstup: Vstupní posloupnost čísel $A = (a_1, a_2, \ldots, a_n)$, hranice pro změnu řazení H, počet splitterů SP, oversampling K.
Výstup: Seřazená vstupní posloupnost A.

1: **if** $n \leq H$ **then**
2: **return** $OtherSort(A)$ ▷ Přepne na nějaký jiný řadicí algoritmus
3: **end if**
4: $v = SP \cdot K$ ▷ Velikost vzorku
5: $m = SP + 1$ ▷ Počet přihrádek
6: $V = [v_1, v_2, \ldots v_v] \leftarrow SelectSample(A, v)$
7: $Sort(V)$
8: $S = [s_1, s_2, \ldots s_n)] \leftarrow SelectSplitters(V, SP)$
9: $B = [b_1, b_2, \ldots b_m] \leftarrow SelectBuckets(S, m)$
10: $PartitionArray(A, B)$
11: **return** spojené seřazené jednotlivé přihrádky

Níže uvádím pseudokód pro pomocné funkce Samplesortu (indexace pole začíná od 1):

Algorithm 4 SelectSample

Vstup: Vstupní posloupnost čísel $A = (a_1, a_2, \ldots, a_n)$, počet splitterů SP, oversampling K
Výstup: Množina vzorků V

1: $V = \{\}$
2: **for** v from 1 to $SP \cdot K$ **do**
3: $V \leftarrow A[random()]$
4: **end for**
5: **return** V

Algorithm 5 SelectSplitters

Vstup: Seřazená posloupnost vzorků V, počet splitterů SP, oversampling K
Výstup: Seřazená posloupnost splittterů S

1: $S = \{\}$
2: **for** s from 1 to SP **do**
3: $S \leftarrow V[s \cdot K]$
4: **end for**
5: **return** S

Následující pseudokód používá vlastní datový typ *Bucket*, který má tři vnitřní proměnné:

1. *from* - levý interval, na který rozdělí pole,

2. *to* - pravý interval,

3. *content* - přiřazené prvky.

Algorithm 6 SelectBuckets

Vstup: Seřazená posloupnost splitterů S, počet splitterů SP
Výstup: Seřazená posloupnost přihrádek B

```
1: B = {}
2: B[1].from ← T_MIN          ▷ Nejmenší prvek použitého datového typu
3: B[1].to ← S[1]
4: for s from 2 to SP do
5:     B[s].from ← S[s − 1]
6:     B[s].to ← S[s]
7: end for
8: B[SP + 1].from ← S[SP]
9: B[SP + 1].to ← T_MAX       ▷ Největší prvek použitého datového typu
10: return B
```

Metoda *PartitionArray* rozdělí posloupnost A do přihrádek B.

Algorithm 7 PartitionArray

Vstup: Vstupní posloupnost A délky n, množina přihrádek B

```
1: for k from 1 to n do
2:     i = index cílové přihrádky prvku A[k]
3:     B[i].content ← A[k]
4: end for
5: return S
```

2.2.3 Časová a paměťová analýza algoritmu Samplesort

Vzhledem k podobnosti Samplesortu s Quicksortem provedu analýzu Samplesortu na základě již provedené analýzy Quicksortu. Předpokládáme konstantní velikost výběru vzorku. Analýzu pro škálovatelnou velikost výběru provedu v Kapitole 3.

Časová složitost

Spočtěme časovou náročnost jednotlivých fází algoritmu.

- Ve fázi **výběru vzorku** vybíráme ze vstupních dat náhodně vzorek V o velikosti $|V| \cdot K$. Předpokládejme, že náhodný výběr nějakého prvku v poli trvá konstantní čas, tedy

$$T_{SelectSample}(n) = \mathcal{O}(|V| \cdot K). \tag{2.21}$$

Jelikož je velikost vzorku předem daná a v žádné iteraci se nemění, potom

$$T_{SelectSample}(n) = \mathcal{O}(1). \tag{2.22}$$

- **Seřazení vzorku** v případě konstantní velikosti trvá také vždy konstantně dlouho. Předpokládejme použití vhodného řadicího algoritmu, například Quicksort se složitostí $\mathcal{O}(n \cdot \log\ n)$ (analýza uvedena v Sekci 2.1.3).

$$T_{SortSample}(|V|) = \mathcal{O}(1). \tag{2.23}$$

- **Určení splitterů** opět trvá konstantně mnoho času, neboť vybíráme vždy konstantně velký počet splitterů SP. Tedy

$$T_{SelectSplitters}(SP) = \mathcal{O}(SP) = \mathcal{O}(1). \tag{2.24}$$

- Při **rozdělení vstupu do přihrádek** se časová složitost řídí rychlostí vyhledávání cílové přihrádky. V tomto případě je při konstantní velikosti splitterů SP rychlost vyhledávání vždy konstantní. V případě binárního vyhledávání v poli splitterů platí, že

$$T_{FindBucketIndex} = \mathcal{O}(\log SP) = \mathcal{O}(1). \tag{2.25}$$

  Vyhledávání provádíme pro každý prvek na vstupu zvlášť,proto

$$T_{parition}(n) = \mathcal{O}((\log SP) \cdot n) = \mathcal{O}(n). \tag{2.26}$$

Shrnutí časové složitosti

Analogicky podle analýzy Quicksortu v Sekci 2.1.3 dostáváme

$$T(n) = \begin{cases} \mathcal{O}(n \cdot \log n) & \text{v nejlepším a průměrném případě} \\ \mathcal{O}(n^2) & \text{v nejhorším případě} \end{cases} \tag{2.27}$$

Paměťová složitost

Opět spočteme složitost pro každou fázi zvlášť.

- Pro **uložení vzorku** o konstantní velikosti $|V|$ potřebujeme

$$M_{sample}(|V|) = \mathcal{O}(|V|) = \mathcal{O}(1) \tag{2.28}$$

  dodatečné paměti.

- Pro **uložení splitterů** S o konstantní velikosti SP je zapotřebí

$$M_{splitters}(SP) = \mathcal{O}(|S|) = \mathcal{O}(1) \tag{2.29}$$

  dodatečné paměti.

- Pro **uložení přihrádek** v **out-of-place** verzi není zapotřebí žádné dodatečné místo. Určení cílové přihrádky lze realizovat pouze binárním vyhledáváním v poli splitterů.

  In-place implementace potřebuje pro každý prvek uchovat nejprve index cílové přihrádky, tedy

  $$M_{bindex}(n) = \mathcal{O}(n). \tag{2.30}$$

  Můžeme pro usnadnění předpokládat, že algoritmus tuto paměť před rekurzivním zanořením uvolní. Poté pro každou přihrádku potřebuje uložit informaci, kolik prvků bude obsahovat a kolik prvků v něm je právě správné zařazeno.

  Přihrádek je vždy konstantní množství a i tento krok si vyžaduje konstantní paměť. Celková velikost přihrádek je označíme $|B|$ a tedy

  $$M_{bucket}(|B|) = \mathcal{O}(1). \tag{2.31}$$

- Při **rozdělování vstupu do přihrádek** nyní v případě **in-place** verze není zapotřebí žádné další dodatečné paměti, protože pracuje v rámci vstupního pole. Pouze je potřeba konstantní pamět pro uložení dočasných proměnných, potřebných pro prohazování prvků v poli.

  $$M_{partition}(n) = \mathcal{O}(1) \tag{2.32}$$

  Out-of-place verze, stejně jako v případě Quicksortu, každý prvek kopíruje do příslušného nového pole. Podobně jako u Quicksortu, v nejhorším případě rozdělí vstup pouze do jedné přihrádky a pole se tak sníží pouze o jeden prvek a paměťová náročnost je v tom případě

  $$M_{partition-worst}(n) = \mathcal{O}(n^2). \tag{2.33}$$

  V nejlepším případě je vstup rozdělený rovnoměrně do všech přihrádek. Potom analogicky jako u Quicksortu pro **nejlepší** i **nejhorší** případ:

  $$M_{partition}(n) = \mathcal{O}(n). \tag{2.34}$$

Shrnutí paměťové složitosti

Jako v případě Quicksortu, podle počtu iterací je paměťová složitost

$$\mathbf{M}_{out-of-place}(n) = \begin{cases} \mathcal{O}(n \cdot \log n) & \text{v nejlepším a průměrném případě} \\ \mathcal{O}(n^2) & \text{v nejhorším případě} \end{cases} \tag{2.35}$$

a pokud uvolňujeme pole indexů přihrádek, pak

$$\mathbf{M}_{in-place}(n) = \mathcal{O}(n) \tag{2.36}$$

Stabilita

Samplesort je, stejně jako Quicksort, v případě in-place verze **nestabilní**, ale při implementaci obdobné k out-of-place Quicksortu může být algoritmus **stabilní** (více v Sekci 2.1.3).

2.3 Analýza algoritmu Flashsort

Algoritmus Flashsort uvádím dle popisu v [6]. Algoritmus je představený jako in-place. Během své činnosti neporovnává prvky mezi sebou, ale podobně jako Samplesort odhaduje jejich finální umístění v posloupnosti. Nejprve nalezne minimální a maximální prvek a tím určí interval vstupní posloupnosti. Potom rozdělí pole do tzv. *tříd* (přihrádek), do kterých rozřadí vstup. Tyto třídy poté seřadí nějakým jiným algoritmem. Počet tříd se většinou určuje jako procentuální část z velikosti vstupu.

Pokud je vstup uniformně rozdělený, je časová složitost Flashsortu v nejlepším případě $\mathcal{O}(n)$, ale pokud je vstup rozdělený neuniformně, může časová složitost v závislosti na použitém řadicím algoritmu degradovat až na $\mathcal{O}(n^2)$.

2.3.1 Princip algoritmu Flashsort

Flashsort lze rozdělit do tří po sobě jdoucích fází: **klasifikace**, **permutace** a **seřazení**.

Klasifikace

Ve fázi klasifikace Flashsort předpokládá, že vstupní posloupnost $A(n)$ délky n je rozdělena uniformně. Poté lze pro každý prvek podle jeho hodnoty určit (klasifikovat) přibližné finální umístění, tzn. třídu.

K určení cílové třídy musí Flashsort nejprve nalézt nejmenší a největší prvek. Nalezení těchto prvků má časovou složitost $\mathcal{O}(n)$.

Označme nejmenší a největší prvek A_{min} a A_{max} a počet tříd $m = p \cdot n$, kde $p \in (0; 1)$ (v [6] je jako optimum uvedeno $m = 0{,}43 \cdot n$). Potom pro každý prvek $A(i)$ lze klasifikovat index třídy pomocí následující funkce:

$$K(A(i)) = 1 + \lfloor ((m-1) \cdot \tfrac{A_i - A_{min}}{A_{max} - A_{min}}) \rfloor$$

Klasifikací každého prvku lze zjistit konečnou velikost každé třídy, a tedy podobně jako u Samplesortu lze prefixovým součtem zjistit intervaly tříd v poli. Klasifikace každého prvku trvá $\mathcal{O}(n)$ a výpočet intervalů tříd se řídí časovou náročností $\mathcal{O}(m)$ a lze je uložit například jako vektor ukazatelů na adresu pravého kraje třídy (označme jej L).

Pokud je vstup rovnoměrně uniformně rozdělený, pak je v každé třídě přibližně $\frac{m}{n}$ prvků.

Permutace

Ve fázi permutace se prvky přesouvají do svých cílových tříd. Pro prvek se pomocí výše uvedené funkce spočítá index cílové třídy a $L[K(A(i))]$ určuje konečnou adresu v poli. Prvek na původní pozici je uložen do dočasné proměnné a opět zařazen do jeho korektní třídy. Vždy, když je jeden prvek zařazen, je adresa třídy v L dekrementována. Tento cyklus skončí, pokud je zaplněna první třída.

Nyní je potřeba zařadit zbývající prvky. Nalezneme tedy nejbližší prvek, který není v korektní třídě, a cyklus opakujeme, dokud se nenaplní tato třída. Takto pokračujeme, dokud všechny prvky nejsou správně zařazeny.

Seřazení

Nyní každou třídu seřadíme nějakým obyčejným řadicím algoritmem. Toto podle [6] (za předpokladu, že třídy mají přibližně stejnou velikost) má časovou složitost $\mathcal{O}((\frac{m}{n})^2)$ pro každou třídu. Ve zmíněné publikaci je navrhovaný Insertion sort.

Kapitola 3

Analýza a realizace vlastního algoritmu

K implementaci své verze Samplesortu jsem si vybral ***in-place*** verzi, která je sice nestabilní, potřebuje však pro svůj běh výrazně méně dodatečné paměti. V této kapitole se věnuji jejímu podrobnějšímu popisu. Naivní implementace sledovala původně popsaný algoritmus v [1]. Optimalizací několika částí došlo k celkovému zrychlení celého algoritmu. Algoritmus je škálovatelný, lze zvolit velikost vybíraného vzorku i velikost splitterů. Jako alternativní řadicí algoritmus používám existující implementaci *Flashsortu* z [7], který používá existující implementaci *Insertion sortu* z [8]. Tuto implementaci jsem ale upravil, protože neodpovídá algoritmu popsaném v [6]. Implementace v případě příliš velké velikosti třídy volala rekurzivně opět *Flashsort*, tuto část jsem tedy z kódu odstranil.

3.1 Implementace sekvenční verze

Dle zadání jsem algoritmus implementoval v jazyce C++. Implementace pracuje s celými čísly. Algoritmus je in-place a řadí přímo ve vstupním poli.

Jako alternativní algoritmus jsem zvolil *Flashsort* (popsaný v Sekci 2.3). Níže uvádím definici mého algoritmu:

```
void SampleSort (int *A, int from, int to,
                 const int THRESHOLD,
                 const int _K, const int SPLITSIZE );
```

Algoritmus tedy jako parametry přijímá ukazatel na začátek vstupního pole, interval (dva krajní indexy vstupního pole), na kterém bude řadit, hranici pro přepnutí algoritmu, oversampling koeficient a počet splitterů.

Jednotlivé fáze jsou implementovány následovně:

1. **Výběr vzorku** je prováděn náhodně a poté seřazen za použití *std::sort* z knihovny STL ve funkci *SelectSample*. Velikost vzorku V je určena jako

$$|V| = S \cdot K \cdot M,$$

kde

- S udává počet splitterů,
- K udává koeficient oversampling, který určuje, kolik vzorků je potřeba vybrat k určení jednoho splitteru[2],
- M je volitelná proměnná, obvykle volená jako procentuální část délky vstupního pole (toto zajistí dynamickou velikost vzorku v závislosti na velikosti vstupního pole).

Toto vykonává následující funkce:

```
void SelectSample ( int *arr, int from, int len,
                    int *sample, int ssize )
{
  for ( int i = 0; i < ssize; i++){
    sample[i] = arr[from+(rand()%len)];
  }
  std::sort(sample,sample+ssize);
}
```

Proměnná `from` určuje offset ve vstupním poli `arr`, proměnná `len` určuje celkovou velikost vstupního pole a proměnná `sample` určuje pole pro uložení vzorku o velikosti `ssize`.

2. **Určení splitterů** ze vzorku se řídí offsetem $O = K \cdot M$, tedy jako splitter je vybrán každý O-tý vzorek. Tuto fázi vykonává funkce *SelectSplitters*:

```
void SelectSplitters ( int *sample, int multiply, int
    SPLITSIZE )
{
  for ( int i = 0; i < SPLITSIZE; i++ )
    /* (multiply/2) zajisti, ze se splittery
    nebudou vybirat z~krajnich prvku      */
    sample[i] = sample[i*multiply+(multiply/2)];
}
```

Proměnná `multiply` udává hodnotu oversampling faktoru K, proměnná `SPLITSIZE` značí počet splitterů.

[2]Zvyšování hodnoty K podle [2, str. 5] sice zvyšuje dobu nutnou pro výběr a seřazení vzorku, zároveň ale také zvyšuje přesnost rozdělení pole do přihrádek.

3. **Určení přihrádek** je dáno jednotlivými splittery. Navíc jsem zavedl přihrádky pro prvky rovny splitterům, a tím zvýšil počet přihrádek p na

$$p = 2 \cdot SPL + 1.$$

Následně si na zásobník uložím pole pomocných struktur pro přihrádky P, které budu využívat v následující fázi. Přikládám implementaci pomocné struktury:

```
struct Bucket {
  Bucket(){this->size = 0; this->taken = 0;}
  int size;   // velikost prihradky
  int offset; // offset ve vstupnim poli
  int taken;  // zaplnenost korektnimi prvky
};
```

4. **Rozdělení pole** do přihrádek je implementováno následovně:

   - Nejprve je pro každý prvek určen index cílové přihrádky (implementováno jako binární vyhledávání v poli splitterů pomocí funkce *FindBucket*).
   - Pokaždé, když určím index přihrádky pro konkrétní prvek, inkrementuji velikost odpovídající přihrádky o 1.
   - Následně je pomocí prefixového součtu velikostí přihrádek spočítán jejich offset ve vstupním poli (první přihrádka má offset 0).
   - Jelikož je algoritmus in-place, je nutné prohazovat prvky přímo v poli. K tomu slouží funkce *AssignToBucket*. Je volána pro každý prvek zvlášť. Vždy za použití pomocného pole přihrádek P a offsetu jednotlivých přihrádek ověřím, zda je prvek umístěný v korektní přihrádce p_i. Pokud není, spočítám jeho konečný index v poli jako

     $$index = P[p_i].\text{offset} + P[p_i].\text{taken}$$

     a prohodím s prvkem na cílové adrese.
   - Tímto se ale tento prvek může opět nacházet v nesprávné přihrádce. Jelikož musí být zařazen korektně, předchozí proces opakuji do chvíle, dokud prvek není na začátku procesu zařazen korektně. Vždy při korektním zařazení nějakého prvku do přihrádky p_i inkrementuji $P[p_i].\text{taken}$ o 1. Jelikož jsou velikost přihrádek a index cílové přihrádky určeny předchozích krocích, je každý prvek takto zařazen nejvýše jednou, vždy do korektní přihrádky. Tato fáze tedy vždy skončí.

     Funkce *FindBucket* a *AssignToBucket* zde vzhledem k délce kódu neuvádím. Konkrétní implementace je k nalezení na přiloženém CD.

5. **Seřazení přihrádek** provádím rekurzivním voláním funkce *SampleSort* na bloky původního pole určené indexy

$$P[p_i].\text{offset} \quad \text{a} \quad P[p_i].\text{offset} + P[p_i].\text{size}.$$

Rekurzivní volání provádím pro každou lichou přihrádku; každá sudá přihrádka obsahuje prvky rovné splitterům a není tedy třeba je seřadit.

```
// seradi neprazdne prihradky
for ( int i = 0; i < bucket_count; i += 2){
  if ( B[i].size > 0 ){
    SampleSort( A, from+B[i].offset,
                (from+B[i].offset+B[i].size)-1,
                 AUX_BINDEX, AUX_SAMPLE, THRESHOLD, _K,
                     SPLITSIZE);
  }
}
```

Při dosažení konktétní hranice pro velikost pole `THRESHOLD` je místo volání *SampleSort* volán alternativní řadicí algoritmus. Protože jsou intervaly přihrádek seřazeny, je výsledkem seřazené vstupní pole.

3.2 Optimalizace sekvenční verze

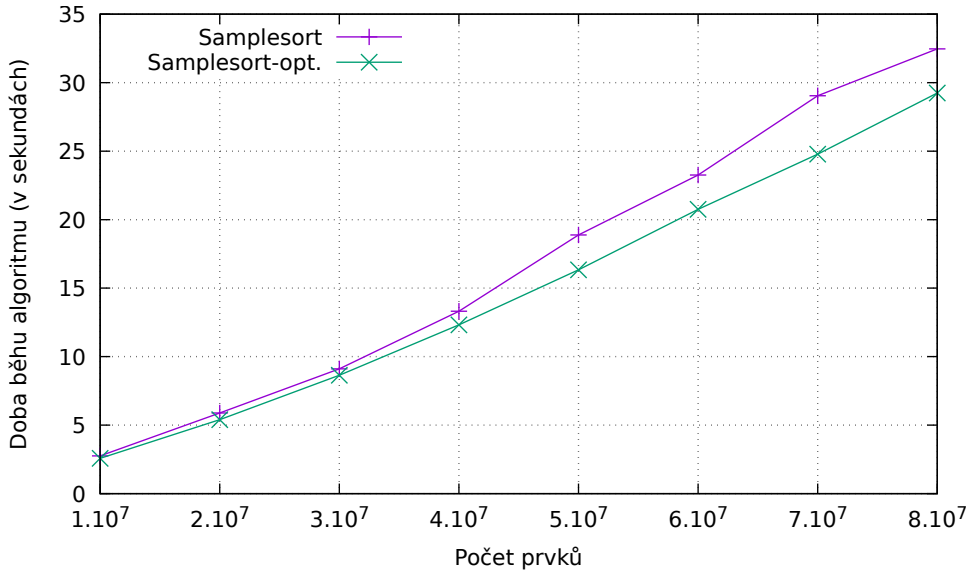
První implementace byla značně neefektivní, proto jsem svůj algoritmus optimalizoval. Jelikož funkce *FindBucket* se bude volat nejčastěji (pro každý prvek v každé iteraci nejméně jednou), věnoval jsem se její optimalizaci.

Původní funkce *FindBucket* pouze iterovala přes všechny splittery a při první shodě vracela index cílové přihrádky. Toto s narůstající velikostí vstupu a se zvětšujícím se počtem splitterů znamenalo znatelný nárůst doby běhu programu. Optimalizovaný *FindBucket* tedy využívá binárního vyhledávání, které se řídí časovou složitostí $\mathcal{O}(\log n)$ [3].

Na obrázku 3.1 je vidět graf doby běhu algoritmu před optimalizací a po optimalizaci. Data k seřazení [3] se řídí exponenciálním rozdělením, parametry k seřazení jsem náhodně zvolil:

- oversampling: 5
- počet splitterů: 50
- hranice `THRESHOLD`: $1{,}2 \cdot 10^6$

[3] Podrobnějšímu popisu vstupních dat a měření se věnuji v Kapitole 5.



Obrázek 3.1: Optimalizace algoritmu pomocí binárního vyhledávání.

Dále jsem provedl optimalizaci vícenásobným využitím alokované pomocné paměti. Označme

- $n = to - from + 1$ jako počet prvků, které algoritmus musí seřadit;
- s jako velikost vzorku.

Původní naivní verze vždy v každé iteraci alokovala pomocné pole k určení indexu přihrádky a k výběru vzorku a splitterů. Toto se však ukázalo jako časově náročné. S využitím znalosti, že pole pro ukládání vzorku a splitterů jsou potřebná pouze v prvních dvou fázích algoritmu (popsány v Sekci 2.2.1) a při rekurzivním volání již nejsou potřeba, lze opakovaně využít dvou dodatečných pomocných polí. Jelikož je počet splitterů vždy maximálně rovný počtu vybraných vzorků, lze navíc pro tyto dvě fáze využít pouze jednoho pole.

Funkce si tedy předem naalokuje dvě pomocná pole:

1. `AUX_SAMPLE` - pole typu *int* délky s. Toto pole je využíváno při výběru vzorku a určování splitterů.
2. `AUX_BUCKET_INDEX` - pole typu *unsigned char* délky n. Do tohoto pomocného pole algoritmus ukládá index přihrádky každého prvku. Index prvku v tomto poli odpovídá indexu prvku ve vstupním poli a jeho hodnota udává index přihrádky v aktuální iteraci. Datový typ *unsigned char* používám v rámci šetření místem. Toto podle limitní hodnoty *unsigned*

char (dané standardem C++) omezuje počet možných přihrádek na 256, lze však použít jiný datový typ.

Níže uvádím kód funkce Samplesort. Funkce nejprve alokuje potřebná pomocná pole a dále volá druhou funkci Samplesort, nyní již s naalokovanými prostředky.

```
void SampleSort (int *A, int from, int to, int THRESHOLD,
                 int _K, int SPLITSIZE )
{
  int len = to-from+1;
  int           *AUX_SAMPLE       = new int [SPLITSIZE * _K];
  unsigned char *AUX_BUCKET_INDEX = new unsigned char [len];

  SampleSort (A,from,to,AUX_BUCKET_INDEX, AUX_SAMPLE, THRESHOLD,
      _K, SPLITSIZE);

  delete [] AUX_BUCKET_INDEX;
  delete [] AUX_SAMPLE;
}
```

Tuto první funkci budu dále v práci nazývat **inicializační algoritmus**, nebo také **inicializační funkce**.

3.3 Analýza implementace

Časovou a paměťovou analýzu provádím pro každou část svého algoritmu zvlášť. Vstupní pole délky n značím $A(n)$.

3.3.1 Časová analýza

Časovou složitost uvádím pro

1. Výběr vzorku se řídí složitostí
   - $\mathcal{O}(1)$ pro konstantně danou velikost vzorku
   - $\mathcal{O}(SPL \cdot K \cdot M) = \mathcal{O}(|V|)$ pro dynamicky určenou velikost vzorku (V definováno v Sekci 3.1).
2. Seřazení vzorku závisí na použitém algoritmu, v případě *std::sort* v jazyce C++
   - $\mathcal{O}(|V| \cdot \log_2 |V|)$ podle standardu C++.
3. Výběr splitterů trvá
   - $\mathcal{O}(1)$ pro konstantně danou velikost vzorku;
   - $\mathcal{O}(\frac{|V|}{K}) = \mathcal{O}(|V|)$ pro dynamicky určenou velikost vzorku.

4. Určení indexu přihrádky, kde P je množina přihrádek, trvá
   - $\mathcal{O}(n \cdot \log_2 |P|)$, pokud je použito binární vyhledávání.
5. Prefixový součet jednotlivých přihrádek trvá
   - $\mathcal{O}(|P|)$.
6. Rozřazení pole v nejhorším případě nejprve volá *AssignToBucket* na první prvek a prohází v této funkci celé pole. Potom volá *AssignToBucket* na zbylých $n-1$ prvků, které jsou ale na správném místě, takže se funkce hned ukončí. V tom případě je časová složitost
   - $\mathcal{O}(n + n - 1) = \mathcal{O}(n)$.
7. Celková časová složitost v tomto okamžiku činí
   - $\mathcal{O}(n \cdot \log_2 n)$.
8. Seřazení přihrádek záleží na použitém řadicím algoritmu a dosažení hranice `THRESHOLD`. V ideálním případě je vždy v každé přihrádce stejný počet prvků. V nejhorším případě je vždy pouze v jedné přihrádce $n-1$ prvků a v ostatních přihrádkách dohromady 1 prvek. Celková časová složitost je tedy
   - $\frac{n}{\texttt{THRESHOLD}} \cdot \mathcal{O}(n \cdot \log_2 n) + \frac{n}{\texttt{THRESHOLD}} \cdot T_{alt}$ v nejlepším případě,
   - $(n - \texttt{THRESHOLD}) \cdot \mathcal{O}(n \cdot \log_2 n) + T_{alt}$ v nejhorší případě,

   kde T_{alt} značí časovou složitost použitého alternativního algoritmu.

3.3.2 Paměťová analýza

Algoritmus jako dodatečnou paměť pro svůj běh potřebuje:

- $\mathcal{O}(n)$ paměti[4] pro ukládání indexu cílové přihrádky ve funkcích *FindBucket* a *AssignToBucket*. V první iteraci Samplesort používá pro tyto funkce celé pole, v dalších iteracích pak jen části tohoto pole.
- $\mathcal{O}(|V|)$ paměti pro práci se vzorkem a splittery [5]. Vzorek a pole vybíráme vždy konstantní, proto lze opětovně využít jednoho pomocného pole.
- $\mathcal{O}(\log_2 n \cdot 3 \cdot |P|) = \mathcal{O}(\log_2 n \cdot |P|)$ paměti na zásobníku k uložení pomocných struktur pro přihrádky, neboť každá struktura *Bucket* obsahuje tři celočíselné proměnné.

[4] Pole `AUX_BUCKET_INDEX`, alokované na začátku běhu algoritmu.
[5] Pole `AUX_SAMPLE`, taktéž alokované na začátku běhu algoritmu.

Kapitola 4

Paralelizace algoritmu Samplesort

Paralelizaci sekvenčního algoritmu jsem provedl podle zadání pomocí knihovny OpenMP [9]. Hlavní myšlenka paralelizace mého algoritmu spočívá v paralelizaci seřazení přihrádek. Jak uvádím v Kapitole 3, tato část je volána rekurzivně pro každou přihrádku. Tento krok lze rozdělit mezi více vláken.

Před představením paralelizace Samplesortu nejprve stručně popíši knihovnu OpenMP.

4.1 Knihovna OpenMP

Informace v této sekci jsou čerpány výhradně z [10].

OpenMP je API, sloužící k programování paralelních aplikací. Používá sdílenou paměť. K výpočtu používá vlákna. Jak je uvedeno v [10]: *„Vlákno je nejmenší blok kódu, který lze plánovat a provádět na úrovni operačního systému a který sdílí kontext výpočtu procesu.“*

K paralelizaci kódu se v OpenMP používají tzv. *direktivy*. Pomocí direktiv se v kódu vytváří paralelní oblasti (regiony), které mohou být vykonávány paralelně běžícími vlákny nebo úlohami. OpenMP podporuje dva typy *paralelizmu*:

1. *datový paralelizmus* - direktiva `for`, která slouží k paralelizaci datově nezávislých iterací v cyklu;
2. *funkční paralelizmus* - direktiva `task`.

Během paralelizace jsem použil následující direktivy:

- `#pragma omp parallel` - tato direktiva označuje paralelní oblast. Kód v tomto bloku se vykonává paralelně. Pomocí klauzule `num_threads()` lze nastavit počet vláken, které této oblasti přidělíme. Pomocí klauzule

`shared()` lze nastavit sdílene proměnné. Pomocí klauzule `private()` lze nastavit lokální proměnné (každé vlákno má svou vlastní instanci těchto proměnných). Na konci oblasti jsou vlákna zrušena.

- `#pragma omp task` - tato direktiva vygeneruje novou úlohu (jednotku paralelního výpočtu), kterou vloží do *zásobárny úloh*, kde si ji vyzvedne volné vlákno.
- `#pragma omp single` - označuje část kódu, která bude provedena pouze jedním vláknem.
- `#pragma omp taskwait` - způsobí, že úloha v této části kódu čeká na ukončení všech úloh přímých potomků.
- `#pragma omp atomic` - zaručí atomické (tj. nepřerušitelné) provedení paměťové operace.

4.2 Paralelizace inicializační funkce

Inicializační algoritmus (níže uvedený jako *ssort*) jsem nejprve rozšířil o parametr `num_threads`, který určuje počet vláken, které programu přidělíme. Dále bylo potřeba vytvořit nové pole `AUXSAMPLE_PROCESSES` o velikosti `num_threads`. V tomto poli si každé vlákno bude uchovávat ukazatel na vlastní pomocné pole pro vybírání vzorku. Toto bylo nezbytné, neboť by si vlákna vzájemně přepisovala hodnoty vybraných vzorků či splitterů.

Níže uvádím upravenou verzi *inicializačního algoritmu*:

```
void ssort ( int *A, const int from, const int to,
             const unsigned char num_threads,
             const int THRESHOLD , const int _K,
             const int SPLITSIZE )
{
  const int len = to-from+1;
  unsigned char *AUX_BUCKET_INDEX = new unsigned char [len];
  int *AUXSAMPLE_PROCESSES[num_threads];
  omp_set_num_threads (num_threads);

#pragma omp parallel default (none) shared (A,AUX_BUCKET_INDEX,
                                            AUXSAMPLE_PROCESSES)
{

  AUXSAMPLE_PROCESSES[omp_get_thread_num()]
                                = new int [SPLITSIZE*_K];
  #pragma omp single
  ssort (A,from,to,AUX_BUCKET_INDEX, AUXSAMPLE_PROCESSES,
      THRESHOLD,_K,SPLITSIZE);
  delete [] AUXSAMPLE_PROCESSES[omp_get_thread_num()];
}
}
```

Direktiva `#pragma omp parallel` zahájí paralelní vykonávání programu. Sdílené proměnné jsou v tomto případě:

1. adresa vstupního pole A,
2. pomocné pole pro indexy přihrádek `AUX_BUCKET_INDEX`,
3. pole ukazatelů na pomocná pole `AUXSAMPLE_PROCESSES`.

V této paralelní části si každé vlákno alokuje vlastní pomocné pole `AUX_SAMPLE_PROCESSES`. Proměnná `AUX_BUCKET_INDEX` může být na rozdíl od `AUX_SAMPLE` sdílená, neboť jednotlivé procesy pracují s jinými bloky pole, určenými krajními indexy bloku. Bloky jsou odděleny splittery (nejméně tedy jedním prvkem) a tudíž žádné dva procesy nepracují ani se shodným krajním indexem.

Direktiva `#pragma omp single` způsobí, že volání druhé, řadicí funkce Samplesort vykoná pouze jeden proces, neboť chceme, aby zde funkci *ssort* začalo vykonávat pouze jedno vlákno; bez direktivy by funkci ssort začala provádět všechna vlákna (paralelní část nastává až při řazení přihrádek).

4.3 Paralelizace řadicí části

Nyní popíši paralelizaci druhé funkce Samplesort, která vykonává řazení. Jak jsem již uvedl na začátku této kapitoly, paralelizuji fázi rekurzivního řazení přihrádek. Níže uvedený kód zobrazuje tuto paralelizovanou část.

```
for ( int i = 0; i < bucket_count; i += 2){
  if ( B[i].size > 0 ){
    const int child_from = from+B[i].offset;
    const int child_to = (from+B[i].offset+B[i].size)-1;
    #pragma omp task default(none) shared(A,AUX,
        AUXSAMPLE_PROCESSES)
    ssort(A, child_from, child_to,AUX,AUXSAMPLE_PROCESSES,
        THRESHOLD);
  }
}
#pragma omp taskwait
```

Za povšimnutí zde stojí dvě nové proměnné *child_from* a *child_to* a direktiva `#pragma omp taskwait`. Proměnné *child_from* a *child_to* určují interval přihrádky ve vstupním poli A, kterou algoritmus předává volnému vláknu. V této části docházelo k pádu programu z důvodu neplatného přístupu k zásobníku. Toto jsem vyřešil přidáním direktivy `#pragma omp taskwait`, což ovšem může způsobovat celkové zpomalení programu, neboť tato direktiva způsobuje vznik bariéry v dané části programu, tudíž program čeká na dokončení všech vláken.

4.4 Paralelizace měření a přístupu ke globálním proměnným

K měření jsem použil globální proměnné. Jelikož budou vlákna k takovým proměnným přistupovat současně, realizuji přístup k nim pomocí direktivy `#pragma omp atomic`. Jako příklad uvádím počítání provedených iterací algoritmem *Samplesort*. Jako globální proměnnou jsem si deklaroval proměnnou `ITERATIONS`, kterou na začátku každého volání řadicí funkce *SampleSort* inkrementuji o 1. V kódu implementováno následovně:

```
#pragma omp atomic
ITERATIONS++;
```

Kapitola 5

Měření a porovnání výkonu jednotlivých implementací

Než uvedu výsledky měření, popíši nejprve použité technologie pro měření.

5.1 Použité prostředky pro měření

Měření provádím na virtuálním serveru, vytvořeném pomocí OpenStack platformy [11]. Dále virtuální server označuji zkráceně jako *VM* (angl. *virtual machine*).

5.1.1 Parametry virtuálního serveru

VM použité k měření má následující parametry:

- Velikost operační paměti: **16 GB**
- Počet virtuálních jader: **24**
- Operační systém: **Ubuntu 16.04**
- Verze g++: **5.4.0**
- Verze OpenMP: **4.0**

Parametry procesoru hypervizoru:

- Intel© Xeon© CPU E5-2660 v2, frekvence 2.20GHz, 8 fyzických jader s hyper-threadingem

5.1.2 Wolfram Mathematica

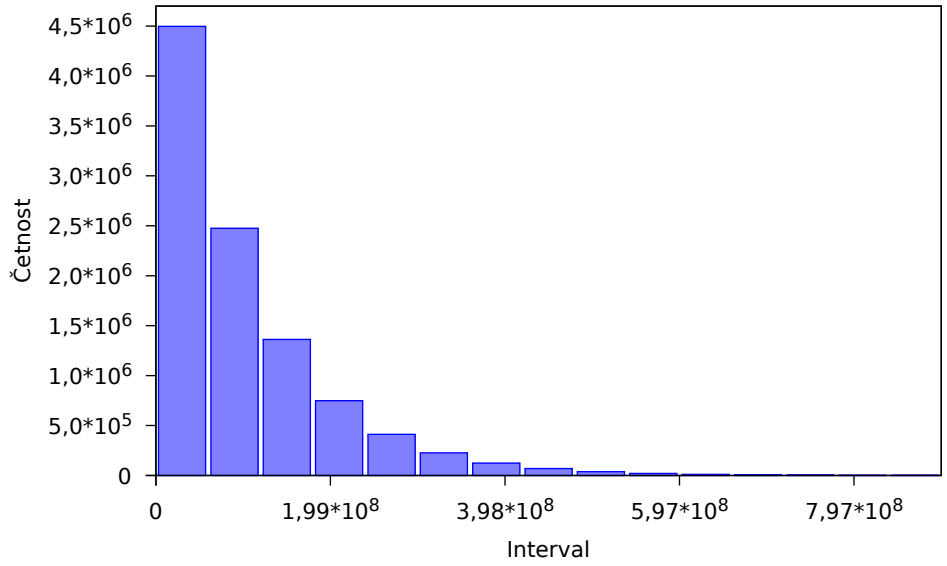
Ke generování náhodných čísel s konkrétním rozdělením jsem použil program *Wolfram Mathematica* (zkráceně *Mathematica*) [12], verze **11.3**. Pro generování 10^7 náhodných čísel jsem využil funkci *RandomVariate*. Náhodná data o velikosti $k \cdot 10^7$ byla vytvořena sloučením k náhodně vygenerovaných dat o velikosti 10^7, generovaných identickou náhodnou funkcí.

5.2 Generování náhodných testovacích dat

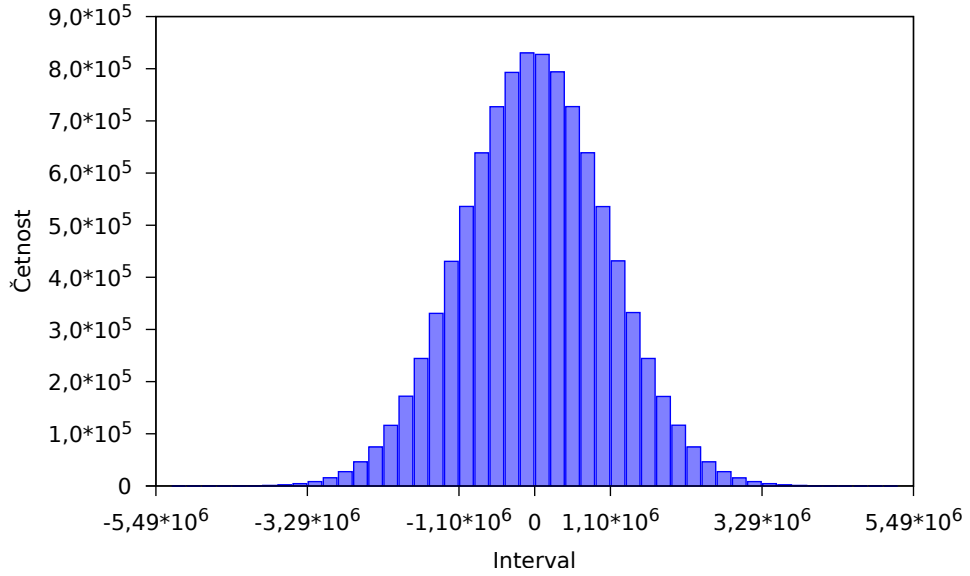
Data pro měření byla generována pomocí programu *Wolfram Mathematica* za použití funkce *RandomVariate* a následně exportována ve formátu CSV pomocí následujících příkazů (ukázka pro velikost dat 10^7):

```
dataexponential = Floor[
                    RandomVariate[
                     ExponentialDistribution[10^-8], 10^7] ]

datanormal = Floor[
               RandomVariate[
                NormalDistribution[0, 2^20], 10^7] ]

datauniform = Floor[
                RandomVariate[
                 UniformDistribution[{-2^20, 2^20}], 10^7] ]

export["normal-10e7.csv", datanormal, "CSV"]
export["exponential-10e7.csv", data, "CSV"]
export["uniform-10e7.csv", data, "CSV"]
```

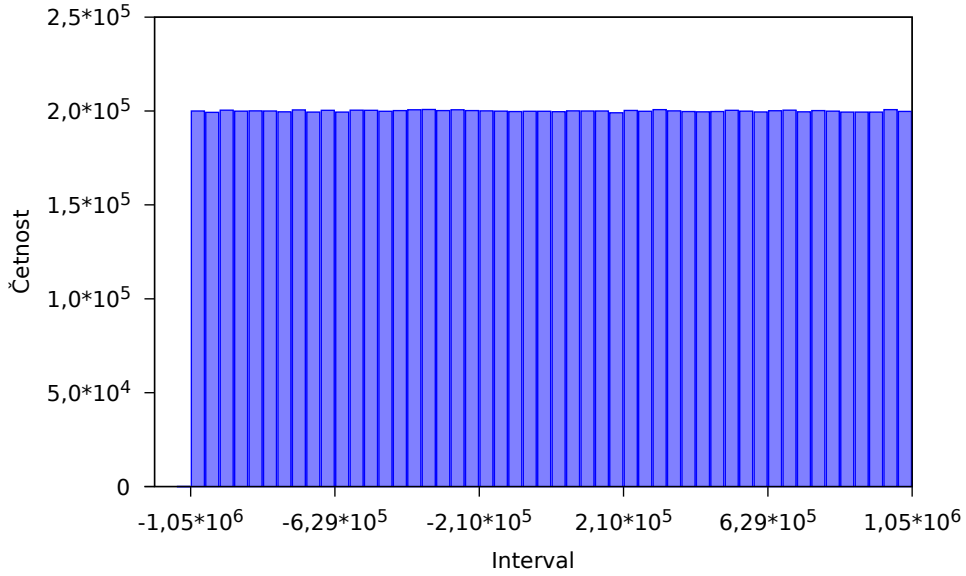
Obrázky 5.1, 5.2 a 5.3 ukazují histogramy těchto dat.



Obrázek 5.1: Histogram náhodně generovaných dat s exponenciálním rozdělením



Obrázek 5.2: Histogram náhodně generovaných dat s normálním rozdělením



Obrázek 5.3: Histogram náhodně generovaných dat s uniformním rozdělením

5.3 Hledání optimálních parametrů pro sekvenční Samplesort

Jelikož efektivita Samplesortu závisí na volbě vnitřních parametrů, provedl jsem nejprve hledání optimální hodnoty těchto parametů pro různě rozdělená data a různě velké vstupy. Měření jsem provedl na výše uvedeném virtuálním serveru, pro sekvenční Samplesort s použitím Flashsortu jako alternativního řadicího algoritmu. Kód byl zkompilován pomocí následujícího příkazu:

```
g++ -std=c++11 -fopenmp samplesort.cpp -o samplesort
```

Přepínač `-fopenmp` je nutný kvůli použití funkce *omp_get_wtime* pro měření doby běhu programu.

Měření provádím v závislosti na parametrech

- oversampling **K**,
- počet splitterů **S**,
- hranice pro přepnutí řadicího algoritmu **T**,
- velikost vstupní posloupnosti **n**.

Pro měření doby běhu algoritmu používám funkci *omp_get_wtime*, která je součástí OpenMP API. V níže uvedených tabulkách uvádím dobu běhu algoritmu v sekundách v závislosti na daných parametrech. Hodnoty byly

vybrány jako medián z naměřených hodnot pro deset identických měření. Kvůli velkému obsahu tabulek uvádím pouze tabulky pro $n = 5 \cdot 10^7$. Pro ostatní velikosti vstupu uvádím pouze optimální parametry v příslušných souhrnných tabulkách.

Dále uvádím grafy zobrazující porovnání časů běhu algoritmů Samplesort, Flashsort a *std::sort* při stejných vstupních datech.

5.3.1 Hledání optimálních parametrů pro data s exponenciálním rozdělením

Tabulka 5.1 ukazuje dobu běhu algoritmu v závislosti na daných parametrech pro náhodně generovaná data s exponenciálním rozdělením o velikosti $5 \cdot 10^7$ prvků. V tabulce je zvýrazněný nejmenší a největší naměřený čas. Z toho lze usoudit, že pro takováto data je optimální volba parametrů

$$T = 4 \cdot 10^6, \quad S = 25, \quad K = 10.$$

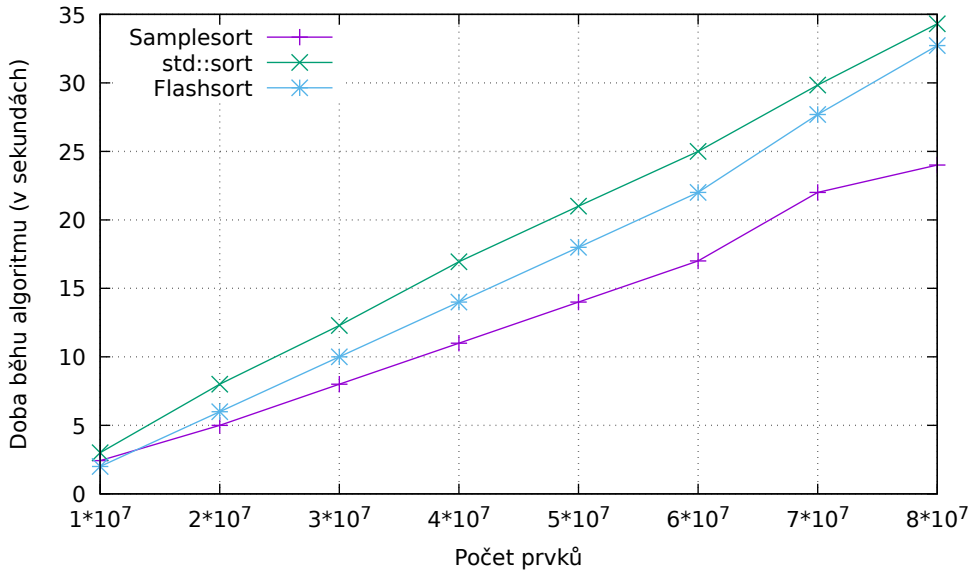
T	S	Doba běhu [s]						
		K=1	K=5	K=10	K=15	K=20	K=25	K=30
$2{,}5 \cdot 10^5$	5	24,6233	23,4639	22,852	22,9875	23,4811	23,4822	23,4940
	10	21,6653	20,7322	20,8043	21,0106	21,0288	21,0946	21,1149
	15	20,8881	19,9167	19,8373	19,8894	19,5086	19,5256	19,5956
	20	20,3498	18,944	18,2448	18,1832	18,2342	18,1568	18,056
	25	19,8126	18,4797	18,2184	18,3167	18,2713	18,2333	18,2596
$5 \cdot 10^5$	5	23,5312	21,4962	21,2665	21,0861	21,0943	21,2393	21,3323
	10	20,0755	19,7094	19,1154	19,0246	19,064	19,2463	19,2718
	15	19,5115	17,9567	17,9599	17,8865	17,798	18,0139	18,2054
	20	18,7568	17,9967	18,0633	18,1029	18,08	18,1615	18,1492
	25	18,6505	18,2253	18,2655	18,2953	18,2296	18,3447	18,6413
$1 \cdot 10^6$	5	21,6863	20,0847	20,0819	20,0723	19,9967	19,7478	19,8853
	10	18,9355	17,4913	17,7615	17,1567	17,1499	17,2377	17,2973
	15	18,4799	17,5671	17,6976	17,6775	17,8348	17,7844	17,7579
	20	17,9211	17,8227	18,115	18,148	18,112	18,1037	18,1524
	25	17,6594	17,9821	18,0176	18,2015	18,2889	18,1813	18,2345
$2 \cdot 10^6$	5	19,6262	18,9052	18,6159	18,6598	18,6368	18,5058	18,4297
	10	18,1143	17,1285	17,0789	17,1659	17,0937	17,1031	17,109
	15	17,4076	17,4194	17,3297	17,5795	17,6199	17,4261	17,5924
	20	17,2741	17,1361	17,1628	17,1985	17,3189	17,7341	17,5919
	25	16,967	16,7058	16,5716	16,5531	16,5649	16,0	16,3176
$4 \cdot 10^6$	5	18,503	17,6921	17,5798	17,4142	17,5917	17,4112	17,7338
	10	17,0443	16,5368	17,0591	16,7745	16,8305	16,7132	16,7383
	15	16,3773	15,7702	15,7804	15,4348	15,4482	15,7502	15,8708
	20	16,233	15,086	14,8417	14,8201	14,6061	14,6877	15,305
	25	15,802	14,5813	14,3754	14,3395	14,4501	14,3358	14,5774
$8 \cdot 10^6$	5	18,2121	17,077	17,2667	17,2295	17,2178	17,3275	17,1869
	10	17,2767	16,0153	15,8784	15,6464	15,6337	15,7734	15,8728
	15	15,7571	15,4783	15,0793	15,2819	15,0255	15,0981	15,0735
	20	15,8446	14,9799	14,8668	14,7205	14,6198	14,5989	14,5657
	25	15,4944	14,4771	14,8152	14,3691	14,5278	14,5185	14,39

Tabulka 5.1: Doba běhu Samplesortu pro $n = 5 \cdot 10^7$ - exponenciální rozdělení

Stejným měřením jsem určil i optimální parametry pro jiné velikosti vstupů, které včetně celkového počtu iterací Samplesortu uvádím v Tabulce 5.2. Obrázek 5.4 ukazuje potom porovnání doby běhu sekvenčních algoritmů *Flashsort*, *Samplesort* a *std::sort* na stejných testovacích datech.

n $[\cdot 10^6]$	T	S	K	Doba běhu $[s]$	Počet iterací
10	$1{,}6 \cdot 10^6$	10	5	2,43091	11
20	$3{,}2 \cdot 10^6$	20	15	5,37916	21
30	$4{,}8 \cdot 10^6$	25	5	8,12922	26
40	$6{,}4 \cdot 10^6$	25	25	11,1049	26
50	$4 \cdot 10^6$	25	10	14,3754	26
60	$4{,}8 \cdot 10^6$	25	15	17,7862	26
70	$11{,}2 \cdot 10^6$	20	15	22,2666	21
80	$6{,}4 \cdot 10^6$	25	20	24,4099	26

Tabulka 5.2: Souhrn optimálních parametrů Samplesortu - exponenciální rozdělení



Obrázek 5.4: Rychlost sekvenčních algoritmů - exponenciální rozdělení dat

5.3.2 Hledání optimálních parametrů pro data s nomálním rozdělením

Zde uvádím obdobonou Tabulku 5.3 pro $n = 5 \cdot 10^7$ a data s normálním rozdělením. Optimální volba parametrů pro toto rozdělení je

$$T = 4 \cdot 10^6, \quad S = 20, \quad K = 10.$$

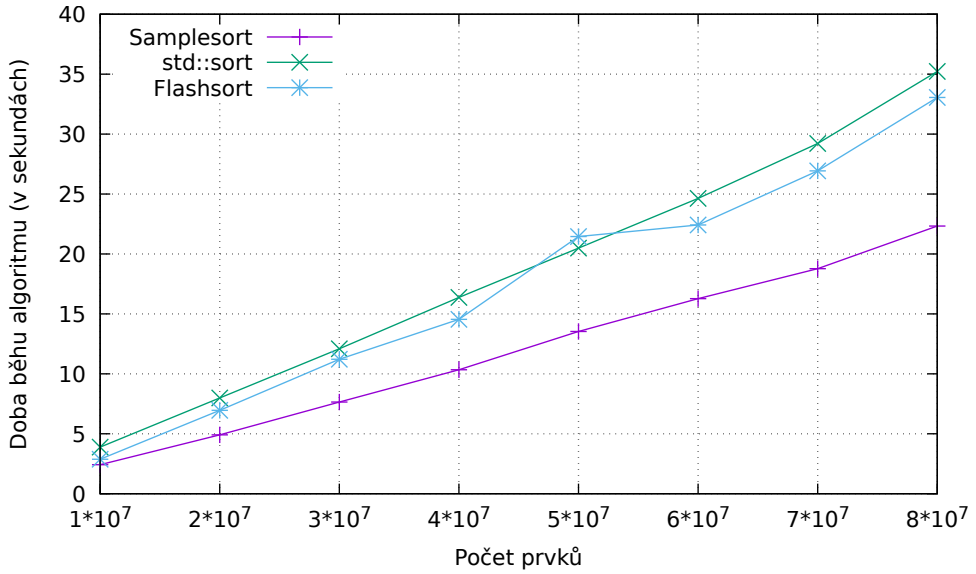
T	S	Doba běhu [s]						
		K=1	K=5	K=10	K=15	K=20	K=25	K=30
$2{,}5 \cdot 10^5$	5	24,1222	21,4615	21,9719	21,4365	21,7883	21,6376	22,0452
	10	20,9537	20,1472	20,1203	20,0772	20,1566	20,3454	20,3975
	15	20,3544	19,1049	18,6252	18,4465	18,7819	18,3854	18,2387
	20	19,2274	18,2739	17,7369	17,5094	17,4177	17,4103	17,3228
	25	18,3994	17,5151	17,3331	17,1653	17,2694	17,3352	17,2575
$5 \cdot 10^5$	5	21,2886	20,1821	20,031	19,908	19,9268	19,6622	19,506
	10	19,2747	18,5241	18,2485	18,4145	18,2561	18,4059	18,4996
	15	18,3801	17,0036	17,1824	16,8993	16,9936	16,8207	16,979
	20	18,3049	17,1416	17,336	17,2104	17,2142	17,2097	17,3065
	25	17,4477	17,3393	17,5338	17,3692	17,4328	17,5026	17,3241
$1 \cdot 10^6$	5	20,4247	18,6311	18,4195	18,7338	19,1233	18,4548	18,589
	10	18,3499	17,3202	16,4338	16,4595	16,3728	16,35	16,2714
	15	17,9086	16,9472	16,793	16,9267	16,861	17,0044	17,067
	20	17,158	16,7254	17,1909	17,2343	17,2533	7,2187	17,2209
	25	16,7778	17,1465	17,3057	17,3097	17,412	17,3507	17,3052
$2 \cdot 10^6$	5	18,0617	17,2386	17,0283	17,5279	17,6851	17,6812	17,862
	10	17,0648	16,3238	16,3382	16,2516	16,5038	16,3741	16,4566
	15	16,4616	16,5575	16,9127	16,8099	16,7475	16,5497	16,8308
	20	16,2976	16,0616	16,3075	16,4094	16,5434	16,7349	17,2039
	25	15,6051	15,4382	15,6831	15,4928	15,6329	15,9077	15,794
$4 \cdot 10^6$	5	18,2359	16,6692	16,7304	16,6421	16,5287	16,7503	16,6985
	10	16,0189	15,8523	15,6641	15,6966	16,0829	16,1964	16,2261
	15	15,8305	15,0582	15,0685	14,4812	14,7049	14,8022	15,0915
	20	15,2325	14,3174	13,539	14,164	13,9359	13,9775	14,1433
	25	15,1304	13,6288	13,6237	13,6686	13,7957	13,5894	13,6208
$8 \cdot 10^6$	5	18,5233	16,4854	16,6296	16,5155	16,5925	16,3909	16,382
	10	15,8996	15,1402	15,2192	15,1555	15,0968	15,2544	15,2897
	15	15,0633	14,4681	14,3106	14,2972	14,2797	14,1177	14,0867
	20	14,5472	14,3021	14,0411	13,9531	13,787	13,9477	13,9072
	25	14,1402	14,0025	13,6632	13,5775	13,6158	13,7643	13,5772

Tabulka 5.3: Doba běhu Samplesortu pro $n = 5 \cdot 10^7$ - normální rozdělení

Shrnutí optimálních parametrů pro ostatní velikosti vstupu opět uvádím v Tabulce 5.4 a porovnání doby běhu algoritmů v grafu na Obrázku 5.5

n $[\cdot 10^6]$	T	S	K	Doba běhu $[s]$	Počet iterací
10	$1{,}6 \cdot 10^6$	10	25	2,42778	22
20	$3{,}2 \cdot 10^6$	10	15	4,91009	11
30	$4{,}8 \cdot 10^6$	10	15	7,65308	11
40	$3{,}2 \cdot 10^6$	25	10	10,3387	26
50	$4 \cdot 10^6$	20	10	13,539	21
60	$4{,}8 \cdot 10^6$	20	20	16,2741	21
70	$5{,}6 \cdot 10^6$	25	25	18,7699	26
80	$12{,}8 \cdot 10^6$	20	5	22,3339	52

Tabulka 5.4: Souhrn optimálních parametrů Samplesortu - normální rozdělení



Obrázek 5.5: Rychlost sekvenčních algoritmů - data s normálním rozdělením

5.3.3 Hledání optimálních parametrů pro data s uniformním rozdělením

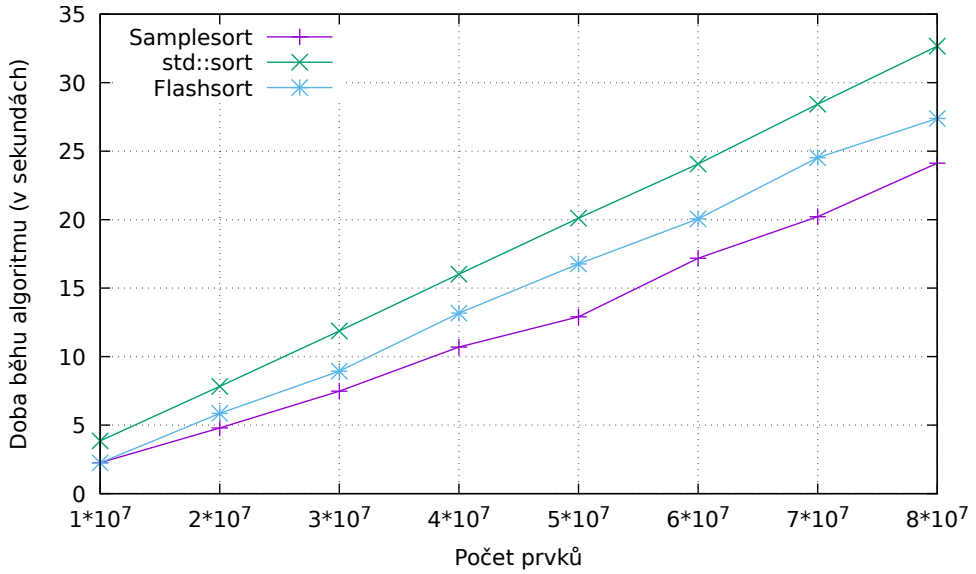
Tabulka 5.5 je obdobná jako tabulky v předchozích sekcích. Tentokrát obsahuje naměřenou dobu běhu Samplesortu pro data s uniformním rozdělením. Optimální volba parametrů pro toto rozdělení je

$$T = 4 \cdot 10^6, \quad S = 25, \quad K = 25.$$

T	S	Doba běhu [s]						
		K=1	K=5	K=10	K=15	K=20	K=25	K=30
$2{,}5 \cdot 10^5$	5	23,9635	22,17	22,4812	21,934	22,3347	22,0774	21,9487
	10	20,7757	19,9702	20,191	20,1011	20,1352	20,231	20,3152
	15	19,4927	19,2291	18,7207	18,2397	18,3234	18,4131	18,1042
	20	19,1518	18,0146	17,4856	17,2806	17,3067	17,3814	17,3707
	25	18,8052	17,5208	17,4164	17,4431	17,4552	17,4531	17,5104
$5 \cdot 10^5$	5	22,0425	20,8581	20,2695	20,3751	19,9137	20,2787	19,8197
	10	19,4681	18,5755	18,2434	18,057	18,2539	18,3355	18,4951
	15	18,4234	17,2522	16,919	16,9622	16,99	16,9482	16,8683
	20	18,0424	17,1795	17,2826	17,2294	17,2286	17,4229	17,5028
	25	17,6214	17,4279	17,371	17,3452	17,3587	17,4551	17,5529
$1 \cdot 10^6$	5	19,9938	18,8276	18,5984	19,1443	19,0559	19,0794	19,1458
	10	18,1948	16,8924	16,5886	16,4163	16,2696	16,3734	16,2975
	15	17,4691	16,7586	16,9863	16,8721	17,0391	17,0467	17,0687
	20	16,9824	17,0378	17,0869	17,2771	17,1761	17,2457	17,3345
	25	16,6724	17,1383	17,2738	17,313	17,3881	17,4037	17,559
$2 \cdot 10^6$	5	18,1398	17,0227	17,2086	17,3743	17,4001	17,2792	17,0792
	10	16,9883	16,1191	16,2753	16,4148	16,2095	16,1653	16,3766
	15	16,2935	16,6284	16,4457	16,4924	16,7682	16,5928	16,8856
	20	16,1094	15,9955	16,0103	16,2122	16,602	16,2711	16,4649
	25	15,7312	15,2428	15,2904	15,7014	15,5704	15,8256	16,1151
$4 \cdot 10^6$	5	17,5221	15,8685	16,0668	16,1829	15,9844	16,0806	16,2541
	10	16,0236	15,3041	15,683	15,6369	15,3289	15,9728	15,4987
	15	15,1626	14,5243	14,5088	14,5974	14,351	14,1986	14,3731
	20	15,2057	13,3803	13,2034	13,2216	13,3275	13,141	13,0646
	25	13,9124	13,1066	12,9818	13,0751	13,0933	12,9059	12,9252
$8 \cdot 10^6$	5	17,7095	16,1799	15,9564	16,2139	16,126	15,8597	15,6993
	10	15,3142	14,3833	14,3959	14,2869	14,0747	14,2933	14,491
	15	14,6171	13,8613	13,9286	13,5763	13,6414	13,7392	13,8662
	20	13,3369	13,0208	13,0517	13,0145	13,1096	13,0884	13,1684
	25	13,379	12,9645	12,9768	12,9475	13,0396	12,9394	12,976

Tabulka 5.5: Doba běhu Samplesortu pro $n = 5 \cdot 10^7$ - uniformní rozdělení

Souhrn optimálních parametrů Samplesortu pro řazení dat s uniformním rozdělením uvádím v Tabulce 5.6. Obrázek 5.6 ukazuje graf doby běhu jednotlivých algoritmů.



Obrázek 5.6: Rychlost sekvenčních algoritmů - data s normálním rozdělením

n $[\cdot 10^6]$	T	S	K	Doba běhu $[s]$	Počet iterací
10	$1{,}6 \cdot 10^6$	10	20	2,5771	11
20	$3{,}2 \cdot 10^6$	15	10	4,79263	16
30	$2{,}4 \cdot 10^6$	25	10	7,48127	26
40	$3{,}2 \cdot 10^6$	20	25	10,7013	21
50	$4 \cdot 10^6$	25	25	12,9095	26
60	$9{,}6 \cdot 10^6$	25	25	17,1825	23
70	$5{,}6 \cdot 10^6$	20	20	20,2099	21
80	$12{,}8 \cdot 10^6$	15	25	24,1249	16

Tabulka 5.6: Souhrn optimálních parametrů Samplesortu - uniformní rozdělení

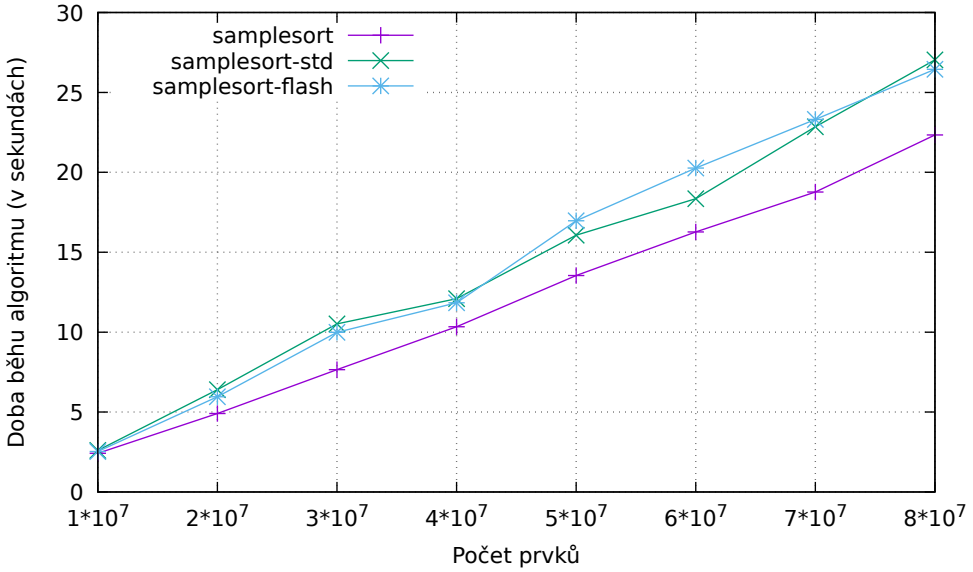
5.3.4 Diskuze výsledků hledání optimálních parametrů

Z výše uvedených Tabulek a grafů lze vyčíst, že optimální hranice `THRESHOLD` pro využití alternativního algoritmu se pro uvedená testovací data pohybuje nejčastěji okolo 15% velikosti vstupních dat. Dále můžeme pozorovat, že volba *oversampling* faktoru K v rozmezí 10 až 20 dosahuje obvykle nejlepších výsledků, tedy poměr času potřebného k seřazení vzorku a celkového zvýšení přesnosti výběru splitterů je tak optimální. Nakonec pozorujeme, že počet potřebných splitterů zpravidla stoupá s rostoucí velikostí vstupních dat. Toto je tedy potřeba při řazení algoritmem zohlednit.

5.4 Nerekurzivní algoritmus Samplesort

Jedním z cílů práce bylo použít Samplesort pouze k předzpracování dat, tedy volat Samplesort pouze jednou a dál řadit přihrádky pomocí jiných algoritmů místo rekurze. Toto řešení se však ukázalo jako neefektivní, jak ukazuje graf na Obrázku 5.7.

Jako jiné řadicí algoritmy jsem zvolil *std::sort* (v grafu uveden jako `samplesort-std`) a *Flashsort* (v grafu uveden jako `samplesort-flash`). Jako parametry pro první iteraci Samplesortu volím optimální parametry ze Sekce 5.3. Na grafu je vidět doba běhu těchto řešení na datech o velikosti $5 \cdot 10^7$ s normálním rozdělením. Tato data porovnávám s referenční dobou běhu Samplesortu (v grafu označen jako `samplesort`), naměřenou v sekci 5.3.2.



Obrázek 5.7: Doba běhu rekurzivního a nerekurzivního algoritmu Samplesort

5.5 Měření doby běhu paralelního Samplesortu

Měření běhu paralelního algoritmu probíhalo na uvedeném testovacím serveru. Program byl zkompilován následujícím příkazem

```
g++ -std=c++11 -fopenmp -Ofast samplesort-parallel.cpp -o
                        samplesort-parallel
```

s přepínačem `-fopenmp` pro překlad OpenMP direktiv a s optimalizačním přepínačem `-Ofast`. Tato optimalizace způsobila celkové zrychlení ostatních algoritmů, včetně algoritmů *std::sort* a *Flashsort*, proto uvádím i nově naměřená data pro tyto algoritmy.

Obrázek 5.8 zobrazuje rychlost běhu Samplesortu v závislosti na počtu přidělených vláken. Graf uvádím pro data o velikosti $8 \cdot 10^7$, s exponenciálním, normálním a uniformním rozdělením. Parametry pro každé uspořádání dat volím podle příslušných optimálních parametrů ze Sekce 5.3. Pro přesnost uvádím výsledky těchto měření v Tabulce 5.7.

# vláken	Doba běhu algoritmu [s] / rozdělení		
	exponenciální	normální	uniformní
1	18,9735	16,9738	17,1596
2	11,644	11,4654	11,873
4	8,53319	8,36165	8,5921
6	7,37341	7,41468	7,22961
8	6,70869	7,11787	6,78608
10	6,59559	6,7337	6,40242
12	6,2711	6,69703	6,39501
14	6,25955	6,36298	5,91312
16	6,03469	6,151	6,21061
18	5,98351	6,25568	5,8864
20	6,06789	6,51026	6,0661
22	5,93734	6,20092	6,10688
24	5,88966	6,05353	6,00276

Tabulka 5.7: Doba běhu paralelního algoritmu Samplesort pro velikost dat 8.10^7 - různá rozdělení

Tyto grafy lze porovnat s dobou běhu neparalelních algoritmů *std::sort* a *Flashsort* na stejných datech. Tato čísla (v sekundách) uvádím v referenční Tabulce 5.8.

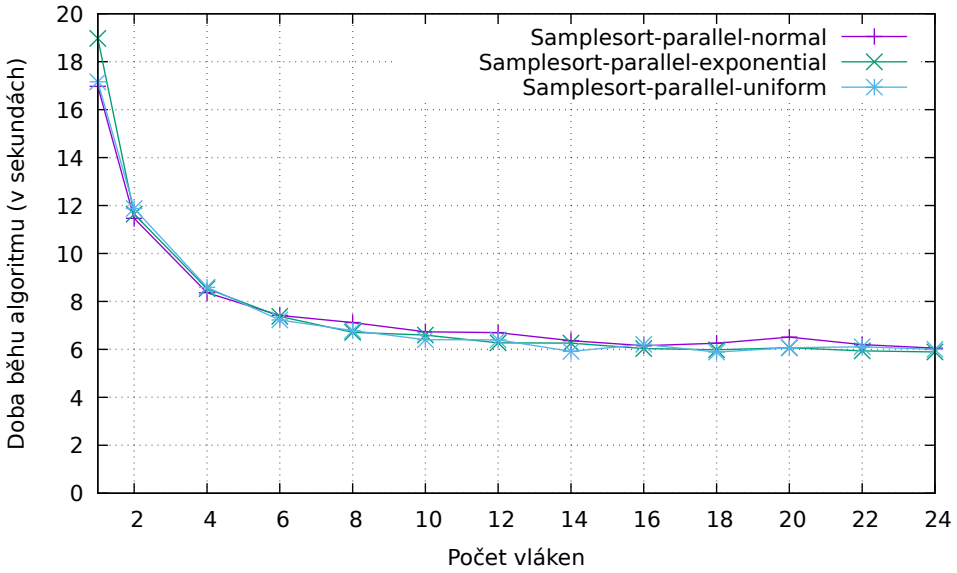
Rozdělení dat	std::sort	Flashsort
Exponenciální	9,40711	25,1293
Normální	8,66373	26,5975
Uniformní	8,23994	25,9196

Tabulka 5.8: Referenční doba běhu [s] algoritmů Samplesort a Flashsort pro data velikosti $5 \cdot 10^7$ - různá rozdělení

5.5.1 Diskuze výsledků měření doby běhu paralelního Samplesortu

Podle naměřených výsledků lze vyvodit, že algoritmus Samplesort byl úspěšně paralelizován. K největšímu paralelizmu dochází v případě, kdy je vstup optimálně rozdělen do přihrádek a každá přihrádka je seřazena vlastním vláknem, přičemž počet přihrádek se pohybuje okolo počtu dostupných vláken. Dále lze

podle Obrázku 5.8 usoudit, že paralelizace snižuje rozdíly v době běhu algoritmu pro data s jiným rozdělením.



Obrázek 5.8: Rychlost běhu paralelního Samplesortu v závislosti na počtu vláken - různá rozdělení

Závěr

Cílem práce bylo seznámit se s algoritmem Samplesort, implementovat vlastní verzi algoritmu a tuto verzi paralelizovat. Cílem práce bylo také vzájemně porovnat efektivitu jednotlivých implementací.

V první kapitole jsme si představili základní pojmy používané v této práci, nezbytné k pochopení obsahu, a uvedli cíle této práce.

Ve druhé kapitole jsme si přiblížili a analyzovali algoritmy Quicksort a Samplesort a představili algoritmus Flashsort. Tato kapitola byla převážně teoretická.

Ve třetí kapitole jsme se věnovali vlastní implementaci algoritmu Samplesort, včetně popisu optimalizací zvyšujících efektivitu tohoto algoritmu. Ke konci třetí kapitoly jsme provedli teoretickou analýzu navrhnuté implementace.

Čtvrtá kapitola sleduje průběh paralelizace implementace ze třetí kapitoly. Vysvětlili jsme si v ní hlavní myšlenku a jednotlivé kroky postupné paralelizace. Na úvod je zde navíc stručně popsána použitá knihovna OpenMP.

V páté a poslední kapitole jsme si definovali použité prostředky pro měření a způsob generování náhodných testovacích dat. Dále jsme se seznámili s reálnými naměřenými daty, která ukazují efektivitu jednotlivých implementací a nalezli optimální parametry pro algoritmus Samplesort. Zde jsme také zjistili, že uvedená implementace je efektivnější než některé existující řadící algoritmy (včetně *std::sort*). Nakonec jsme si ukázali efektivitu úspěšné paralelizace, která nám přinesla až několikanásobné zrychlení.

Na tuto práci lze v budoucnosti navázat. V práci jsme si představili implementaci pouze pro celočíselné datové typy a řazení algoritmem Flashsort. Je možné implementovat algoritmus pro další datové typy, použít jiný řadící algoritmus, provést hledání dalších optimálních parametrů, implementovat pokročilejší metody výběru vzorku či jinak algoritmus paralelizovat.

Bibliografie

1. DONALD FRAZER, W; C. MCKELLAR, A. Samplesort: A Sampling Approach to Minimal Storage Tree Sorting. *Journal of the ACM (JACM)*. 1970, roč. 17, s. 496–507. Dostupné z DOI: 10.1145/321592.321600.
2. LEISCHNER, N.; OSIPOV, V.; SANDERS, P. GPU sample sort. In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, s. 1–10. ISSN 1530-2075. Dostupné z DOI: 10.1109/IPDPS.2010.5470444.
3. BAJWA, M. S.; AGARWAL, A. P.; MANCHANDA, S. Ternary search algorithm: Improvement of binary search. In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, s. 1723–1725.
4. HOARE, C. A. R. Quicksort. *The Computer Journal*. 1962, roč. 5, č. 1, s. 10–16. ISSN 0010-4620. Dostupné z DOI: 10.1093/comjnl/5.1.10.
5. XIANG, Wang. Analysis of the Time Complexity of Quick Sort Algorithm. *International Conference on Information Management, Innovation Management and Industrial Engineering*. 2011, roč. 1, s. 408–410. ISBN 978-0-7695-4523-3. Dostupné z DOI: 10.1109/ICIII.2011.104.
6. *The Flashsort1 Algorithm* [online]. 1998 [cit. 2019-05-13]. Dostupné z: http://www.drdobbs.com/database/the-flashsort1-algorithm/184410496.
7. *Flashsort* [online] [cit. 2019-05-10]. Dostupné z: http://www.cs.utah.edu/~ccasper/cs/flashsort.cpp.
8. *C++ Example – Insertion Sort Algorithm* [online]. 2016 [cit. 2019-05-10]. Dostupné z: http://www.codebind.com/cpp-tutorial/cpp-example-insertion-sort-algorithm/.
9. DAGUM, Leonardo; MENON, Ramesh. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*. 1998, roč. 5, č. 1, s. 46–55.

10. LANGR, Daniel; SIMEČEK, Ivan; TVRDÍK, Pavel. *Paralelní a distribuované programování: Úvod do OpenMP* [online]. 2018 [cit. 2019-05-05]. Dostupné z: `https://courses.fit.cvut.cz/MI-PDP/media/lectures/MI-PDP-Prednaska02-OpenMP.pdf` [Soubor přístupný po přihlášení do sítě ČVUT.

11. SAHASRABUDHE, S. S.; SONAWANI, S. S. Comparing openstack and VMware. In: *2014 International Conference on Advances in Electronics Computers and Communications*. 2014, s. 1–4. Dostupné z DOI: `10.1109/ICAECC.2014.7002392`.

12. INC., Wolfram Research, *Mathematica, Version 11.3*. Champaign, IL, 2018.

Příloha A

Seznam použitých zkratek

CPU central processing unit

GPU graphical processing unit

CSV comma-separated values

API application programming interface

VM virtual machine

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `graphs` ..... grafy použité v práci
- `text` ..... text práce
  - `BP_Erazim_Pavel_2019.pdf` ..... text práce ve formátu PDF
  - `BP_Erazim_Pavel_2019_zadani.pdf` .... zadání práce ve formátu PDF