FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Pokročilé metody paralelního řazení dat
Student:	Dominik Šíba
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte následující algoritmy řazení: Mergesort (Multiway-MergeSort, viz [1]) a Radixsort (Full Parallel Radix Right, viz [2])

2) Implementujte je v jazyce C++ a optimalizujte transformacemi kódu a paralelizací pomocí technologie OpenMP (vícevláknové nad sdílenou pamětí).

3) Změřte výkonnost těchto algoritmů na fakultním serveru Star pro různé vstupní posloupnosti a pro různé typy řazených objektů (32bitový int, 64bitový int, řetězce pevné délky, ...).

4) Porovnejte výkonnost algoritmů s existujícími implementacemi paralelního řazení zejména s [3] a [4]

Seznam odborné literatury

[1] William A. Greene, k-way merging and k-ary sorts, 1993, http://cs.uno.edu/people/faculty/bill/k-way-merge-n-sort-ACM-SE-Regl-1993.pdf
[2] Arne Maus a Stein Gjessing, Practical Parallel Programming – a plan for a B.S. course on how to design efficient parallel algorithms., 2014, http://heim.ifi.uio.no/ arnem/sorting/ParaRadix2014/MausGjessingFinal.pdf
[3] Arch D. Robison. A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP. Intel® Software. [online]. 11.4.2014 [cit. 2017-12-05]. Dostupné z: https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp
[4] Talácko, Rudolf. Pokročilé metody řazení ve vícevláknovém prostředí. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017

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

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

V Praze dne 30. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Pokročilé metody paralelního řazení dat

Dominik Šíba

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

15. května 2018

Poděkování

Rád bych poděkoval panu doc. Ing. Ivanu Šimečkovi, Ph.D. za ochotu, trpělivost a čas, který mi věnoval. Dále bych chtěl poděkovat rodině a přátelům za jejich podporu při psaní této práce.

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 15. května 2018 .....

Odkaz na tuto práci

Šíba, Dominik. *Pokročilé metody paralelního řazení dat.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato bakalářská práce se zabývá implementací řadících algoritmů MergeSort a RadixSort. Tyto algoritmy jsou implementovány v jazyce C++. Implementace jsou optimalizovány tak, aby byl časově efektivní.
V práci jsou algoritmy nejdříve analyzovány a dále jsou navrženy optimalizace, které by mohly zvýšit efektivitu algoritmů a způsoby paralelizace s využitím knihovny OpenMP, které vedou k nejlepšímu rozložení zátěže na jednotlivá vlákna.
Výsledkem práce jsou implementace výše zmíněných algoritmů, grafické zobrazení jejich výkonnosti a porovnání s již existujícími implementacemi.
Implementace algoritmu RadixSort řadí posloupnosti celých čísel v kratším čase než implementace algoritmu MergeSort obsažená ve standardní knihovně C++.

Klíčová slova paralelní řadící algoritmy, optimalizace a paralelizace řadících algoritmů, MergeSort, RadixSort, C++, OpenMP

Abstract

This bachelor thesis deals with the implementation of MergeSort and RadixSort algorithms. These algorithms are implemented in C ++. The implementations are optimized to be time-efficient.
In the thesis, these algorithms are first analyzed, and optimizations are designed to increase the efficiency of algorithms and parallelization methods using OpenMP that lead to the best distribution of the load on the individual threads.
The result of the thesis is implementation of the above-mentioned algorithms, graphical representation of their performance and comparison with already existing implementations.
The implementation of the RadixSort algorithm achieves better times when sorting integers than implementing the MergeSort algorithm contained in the standard C++ library.

Keywords parallel sorting algorithms, optimalization and parallelization of sorting algorithms, MergeSort, RadixSort, C++, OpenMP

Obsah

Seznam obrázků

Seznam tabulek

Úvod

S řadícími algoritmy se dnes setkáváme na každém kroku. Řadíme seznam kontaktů v telefonu podle jmen, příjmení či přezdívky, příchozí zprávy podle času nebo položky v e-shopu podle ceny či hodnocení. S řadícími algoritmy se však nesetkáváme pouze v takto čisté podobě. Jsou nedílnou součástí jiných algoritmů, které pro lepší efektivitu potřebují pracovat se seřazenými daty. Z toho důvodů je vývoj a optimalizace řadících algoritmů důležitá. Protože rychlejší a efektivnější řadící algoritmy nebudu mít vliv pouze na řazení dat, ale ovlivní i rychlosti všech ostatních algoritmů, které je využívají.
V dnešní době se rychlost jednotlivých jader procesoru příliš nezvyšuje, od dalšího zrychlování procesorů už nám začínají bránit zákony fyziky. Z toho důvodu se zvyšuje počet jader v procesoru. Samotné procesory jsou paralelně zapojovány ve velkém množství ve výpočetních serverech. Abychom tedy i nadále zvyšovali výkonnost programů, programy musí být psány tak, aby co nejlépe využila dostupná vlákna.
Těchto důvodů se autor práce rozhodl zabývat implementací řadících algoritmů MergeSort a RadixSort, které jsou pak optimalizovány a paralelizovány. Tyto algoritmy jsou v této práci používané zejména na řazení celých čísel a dále pak na řazení textových řetězců. Jejich optimalizované a paralelní implementace jsou testovány na výpočetním serveru ČVUT STAR. Výsledky tohoto testování jsou porovnány se stávajícími paralelními ale i sekvenčními implementacemi řadících algoritmů.

Kapitola 1

Cíl práce

Cílem rešeršní částí práce je nastudovat stabilní řadicí algoritmy MergeSort a RadixSort, analyzovat je a poté navrhnout optimalizace, které tyto algoritmy zefektivní.
Cílem praktické části je implementace těchto algoritmů v jazyce C++, na které se uplatní navržené optimalizace. Další část je paralelizace implementací za účelem dosažení nejrychlejšího času řazení a rozložení zátěže mezi jednotlivá vlákna procesoru při zachování stability řazení. K paralelizaci je použita knihovna OpenMP.
Implementace jsou testovány na výpočetním serveru STAR na posloupnostech různého uspořádání a různých datových typů. Výkonnost implementací je dále porovnávaná již s existujícími implementacemi stabilních řadících algoritmů.

Kapitola 2

Řadící algoritmy

Řadící algoritmy slouží k seřazení vstupních dat do správného pořadí. Vstupní data jsou nejčastěji řazena podle čísel nebo abecedně, ale mohou být řazena podle čehokoli. Řadící algoritmy jsou nedílnou součástí jiných algoritmů a některých datových struktur, proto je jejich optimalizace velmi důležitá.

2.1 Vlastnosti řadících algoritmů

Tato kapitola čerpá z [1, 2]. Každý algoritmus má různé vlastnosti. Mezi nejdůležitější vlastnosti se řadí tyto:

- časová složitost
- stabilita
- paměťová složitost
- citlivost

2.1.1 Časová složitost algoritmu

Časová složitost je důležitá vlastnost řadících algoritmů. Vyjadřuje závislost času na počtu řazených dat. Je vyjadřována jako počet nutných operací. Primitivní řadící algoritmy mají časovou složitost $O(n^2)$. Mezi tyto typy patří například InsertionSort, BubbleSort nebo SelectSort. Komplikovanější řadící algoritmy mají složitost $O(n \cdot \log n)$. Mezi tyto algoritmy patří například MergeSort, QuickSort a HeapSort. Je dokázáno, že algoritmy založené na binárním porovnávání použije v nejhorším případě $\Omega(n \cdot \log n$ porovnávání) [3].
Některé algoritmy nepoužívají k řazení porovnávání ale počítání. Mezi tyto algoritmy patří například BucketSort, RadixSort či CountSort.

2.1.2 Stabilita algoritmu

Řadící algoritmus je stabilní právě tehdy, když prvky, které mají stejnou hodnotu klíče, jsou ve výstupní posloupnosti ve vzájemně stejném pořadí, jako byly ve vstupní posloupnosti. Tato vlastnost se hodí například při řazení seznamu jmen lidí podle abecedy. Nejdříve se seřadí lidi podle křestních jmen a poté podle příjmení. Stabilní algoritmus zaručí, že lidi se stejným příjmením zůstanou seřazeni podle jmen. Mezi tyto algoritmy se řadí například InsertionSort, MergeSort nebo RadixSort.

2.1.3 Paměťová složitost

Paměťová složitost vyjadřuje závislost potřebné dodatečné paměti na velikosti vstupních dat. Rozlišujeme dva typy algoritmů „in-place“ a „out-of-place“. Algoritmy „in-place“ mají paměťovou složitost $O(1)$, zatímco algoritmu typu „out-of-place“ potřebují k řazení pomocné pole a většinou mají paměťovou složitost $O(n)$. Mezi „in-place“ algoritmy řadíme například BubbleSort a InsertionSort. Mezi „out-of-place“ patří naopak QuickSort a MergeSort.

2.1.4 Citlivost algoritmů

Algoritmus je datově citlivý pokud počet operací potřebných k řazení závisí na hodnotě dat. Příkladem citlivého algoritmu je InsertionSort.

2.2 InsertionSort

InsertionSort[4, 1] je řadící algoritmus, který je efektivní na řazení menšího počtu prvků. Algoritmus funguje na stejném principu, jako si většina karetních hráčů řadí karty v ruce. Začínají s prázdnou rukou a postupně sbírají karty z hromádky a vkládají je na správné místo v ruce. Jakmile je hromádka prázdná, karty jsou již ve správném pořadí.
Mějme vstupní posloupnosti n klíčů $(a_1, a_2, \ldots, a_n)$, která je potřeba seřadit. Algoritmus postupně prochází klíče posloupnosti. První klíč posloupnosti, který ještě nebyl seřazen, je porovnáván již se seřazenými klíči od největšího po nejmenší. Pokud je hodnota řazeného klíče menší než hodnota porovnávaného klíče, klíče se vymění a řazený klíč je dál porovnáván s dalším klíčem. Jakmile narazí na klíč s větší hodnotou, pokračuje se s řazením dalšího klíče.
Algoritmus má složitost $O(n^2)$, ale ve skutečnosti je daleko efektivnější. Například pro seřazenou posloupnost má časovou složitost $O(n)$. Algoritmus se řadí mezi „in-place“ algoritmy.

Algorithm 1 Insertion Sort

```
1: function InsertionSort((a_1, ..., a_n))
2:     for i ← 1 ... n − 1 do
3:         j ← i + 1
4:         key ← a_j
5:         while 1 < j and key < a_{j−1} do
6:             a_j ← a_{j−1}
7:             j ← j − 1
8:         a_j ← key
```

2.3 MergeSort

MergeSort[4, 5] je stabilní algoritmus, který využívá metodu „Rozděl a panuj“. Algoritmus je velmi jednoduchý, ale přesto efektivní.
Mějme vstupní posloupnost n klíčů $(a_1, a_2, \ldots, a_n)$. Posloupnost o velikosti jednoho klíče je seřazená. Vstupní posloupnost rozdělíme na dvě poloviny, které seřadíme rekurzivně. Tyto dvě již seřazené poloviny následně sloučíme do jedné seřazené.
Časová složitost algoritmu je $O(n \cdot \log n)$. Z důvodu potřeby pomocného pole ke sloučení posloupností je paměťová složitost algoritmu $O(n)$.

Algorithm 2 MergeSort

```
1: function MergeSort((a_1, ..., a_n))
2:     if n == 1 then
3:         return
4:     (x_1, ..., x_{n/2}) ← MergeSort ((a_1, ..., a_{n/2}))
5:     (y_1, ..., y_{n/2}) ← MergeSort ((a_{n/2+1}, ..., a_n))
6:     Merge((x_1, ..., x_{n/2}), (y_1, ..., y_{n/2}), (a_1, ..., a_n))
```

2.3.1 Merge

Tento algoritmus se používá ke slévání již seřazených polí. Mějme dvě seřazené vstupní pole a pole výstupní, do kterého budou data seřazeny. Dokud jedno ze vstupních polí není prázdné. Porovnávají se minima seřazených polí. Menší klíč se vyjme z pole a vloží se do výstupního pole. Jakmile je jedno pole prázdné, zbytek druhého pole se zkopíruje do výstupního pole.

2.4 K-way merge

K-way merge[6] s jehož podtypem, 2-way merge, jsme se seznámili v předchozí kapitole, slévá k seřazených polí do jednoho. Mějme n klíčů rozdělených do

Algorithm 3 2-way Merge

```
1: function MERGE((x_1, ..., x_n), (y_1, ..., y_m), (a_1, ..., a_{n+m}))
2:     i ← j ← k ← 1
3:     while i ≤ n and j ≤ m do
4:         if x_i ≤ y_j then
5:             a_k ← x_i
6:             i ← i + 1
7:         else
8:             a_k ← y_j
9:             j ← j + 1
10:        k ← k + 1
11:        if i ≤ n then
12:            (a_k, ..., a_{n+m}) ← (x_i, ..., x_n)        ▷ Zkopírování zbytku pole
13:        else
14:            (a_k, ..., a_{n+m}) ← (y_j, ..., y_m)
```

k seřazených polích. Cílem je sloučit všechna pole do výsledného pole. Ke spojení těchto polí se používají následující způsoby:

- Lineární vyhledávání minima
- Heap-merge
- Rozděl a panuj merge

Tyto způsoby jsou rozebrány níže:

2.4.1 Lineární vyhledávání minima

Lineární vyhledávání minima je nejpomalejší ze všech zmíněných způsobů slévání. Jak již jeho název naznačuje, algoritmus projde všechny nejmenší klíče ze vstupních polí a vybere z nich ten nejmenší, který odebere z pole a vloží ho do výstupního pole. Vyhledávání minima má složitost $O(k)$. Jelikož potřebuje najít každý klíč, celková časová složitost je $O(kn)$.

2.4.2 Heap-merge

Heap-merge k nalezení minima využívá binární haldu (heap). Z minimálních klíčů vstupních polí vytvoří haldu. Z haldy je odebraný nejmenší prvek, který se přidá do výstupního pole a v haldě je nahrazen minimem z pole, ze kterého byl původně odebrán. Jelikož odebrání minima a vložení prvku do haldy má složitost $O(\log k)$, celková časová složitost Heap-merge je $O(n \cdot \log k)$.

2.4.3 Rozděl a panuj merge

Rozděl a panuj merge využívá ke slévání klasický 2-way merge. Vstupních k polí rozdělí do $k/2$ párů, které jsou spojeny klasickým mergem. Tento algoritmus se volá rekurzivně, dokud nezůstane pouze jedno seřazené pole. Hloubka zanoření je $\log k$. Z toho důvodu je celková časová složitost algoritmu $O(n \cdot \log k)$.

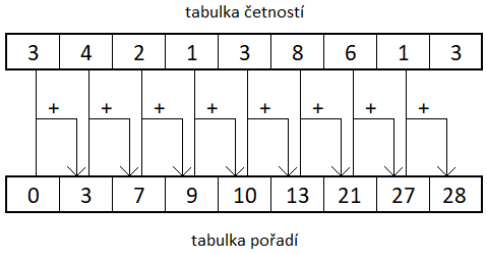
2.5 RadixSort

RadixSort[7] se od většiny ostatních řadících algoritmů výrazně liší. Tento způsob řazení se používal mnoho let předtím, než se začaly používat počítače. K řazení nepoužívá porovnávání klíčů, ale třídí jednotlivé číslice každého klíče. Z tohoto důvodu je tato metoda je obecně známa jako číslicové třídění (v angl. „digital sorting“ z anglického slova digit = číslice nebo cifra). Dále se pro nejmenší část klíče bude používat slovo *číslice*, přestože to může znamenat například jeden symbol z ASCII tabulky nebo řetěz těchto symbolů.
Řekněme, že chceme seřadit balíček karet primárně podle barev a sekundárně podle hodnoty karty. Prvním krokem bude rozdělit karty na hromádky podle hodnoty a tyto hromádky následně naskládat na sebe podle hodnoty tak, aby nejnižší hodnota byla navrchu. Druhým krokem bude rozdávat karty do hromádek podle barvy. Každá barva již bude seřazena podle velikosti. Když poté hromádky barev spojíme, budou seřazeny karty podle barvy a každá barva podle velikosti.
Tento způsob se dá jednoduše aplikovat na čísla v jakékoli soustavě či textové řetězce. U karet se používají hodnoty a barvy, u čísla jednotlivé číslice (v decimální soustavě jednotky, desítky, stovky, atd.) a slova podle písmen.
RadixSort je stabilní algoritmus. Časová složitost algoritmu je $O(kn)$, kde k je největší počet číslic obsažených v maximálním klíči ve vstupních datech. Tato složitost plyne z faktu, že algoritmus musí pro každou číslici rozdělit všechna data do hromádek. K řazení je použito pomocné pole, proto je paměťová složitost $O(n)$.
Mějme vstupní pole o velikosti n, které řadíme podle klíče K. Každý klíč K_i,

Algorithm 4 RadixSort

```
1: function RadixSort((a_1, ..., a_n))
2:     k ← max_lenght((a_1, ..., a_n))        ▷ Velikost nejdelšího klíče
3:     for i ← 0 ... k − 1 do
4:         CountSort((a_1, ..., a_n), i)
```

kde $0 < i < n$, je uspořádaná k_i-tice $d^k d^{k-1} \cdots d^1$, kde k_i je délka K_i.
Hlavní část algoritmu nejprve zjistí největší k. Následně vstupní data seřadí podle každé číslice. Algoritmus postupuje od nejméně významné číslice po



Obrázek 2.1: Princip výpočtu pozice uložení klíče

nejvíce významnou (0. až $k-1$.). Z tohoto důvodu se tento typ algoritmu RadixSort se označuje zkratkou LSD (least significant digit).

2.5.1 CountSort

CountSort nejdříve projde všechny klíče vstupních dat a pro každou číslici zjistí počet výskytů na předem určené pozici. Z těchto dat následně vytvoří tabulku pozic, na které se ve výstupním poli zařadí klíče s danou číslicí. Nejméně hodnotná číslice začíná na 0. pozici. Počáteční pozice ostatních číslic se rovnají součtu počtu všech méně hodnotných číslic. Ukázka výpočtu pozice je vyobrazena na obrázku 2.1.
V posledním kroku algoritmus prochází vstupní data a z tabulky pozic přečte, na jakou pozic má daný klíč zařadit. Tuto hodnotu v tabulce pozic následně zvýší o jedna.

Algorithm 5 CountSort

1: **function** CountSort($(a_1, \ldots, a_n), index$)
2: $(c_1, \ldots, c_k) \leftarrow (0, \ldots, 0)$ ▷ Inicializace pole četností
3: $(o_1, \ldots, o_k) \leftarrow (1, \ldots, 1)$ ▷ Inicializace pole pozic
4: **for** $i \leftarrow 1 \ldots n$ **do** ▷ Zjištění četností číslic
5: $digit \leftarrow a_i[index]$ ▷ Číslice na pozici
6: $c_{digit} \leftarrow c_{digit} + 1$
7: **for** $i \leftarrow 2 \ldots n$ **do** ▷ Výpočet pořadí
8: $o_i \leftarrow o_{i-1} + c_{i-1}$
9: **for** $i \leftarrow 1 \ldots n$ **do** ▷ Zjištění četností číslic
10: $digit \leftarrow a_i[index]$
11: $new\_index \leftarrow o_{digit}$ ▷ Zjištění pozice, na kterou se má klíč zařadit
12: $o_{digit} \leftarrow o_{digit} + 1$ ▷ Inkrementace pozice pro další klíče
13: $tmp_{new\_index} \leftarrow a_i$ ▷ Zkopírování klíče na správnou pozici do pomocného pole
14: $(a_1, \ldots, a_n) \leftarrow (tmp_1, \ldots, tmp_n)$

Kapitola 3

OpenMP

Tato kapitola čerpá z následujících zdrojů [8, 9, 10]. OpenMP je knihovna obsahující sadu OpenMP direktiv, které ovlivňují chod programu. Direktivy jsou navrženy tak, aby si program zachoval správné chování, i když je kompilátor nepodporuje. Direktivy, knihovní procedury a proměnné prostředí dovolují vytvořit paralelní programy. K využití tohoto standardu je potřeba do programu přidat direktivu **#include <omp.h>** a při kompilaci v kompilátorem GCC je třeba přidat přepínač **-fopenmp**.
Všechny OpenMP konstrukty v jazyku C a C++ jsou začínají **#pragma omp**, následují parametry a ukončeny jsou novým řádkem. Direktivy platí většinou pouze na příkaz, který následuje.

3.1 Direktiva parallel

Program celou dobu běží jednovláknově dokud nenarazí na tuto direktivu. Tato direktiva vytvoří paralelní blok, ve kterém je vytvořeno N vláken (N je určeno při běhu), z nichž každé z nich provede následující příkaz nebo blok příkazů, který je ohraničený složenými závorky. Po vykonání příkazů se vlákna opět spojí do jednoho.

3.1.1 Vlastnosti proměnných

Pomocí klausulí pro nastavení vlastností proměnných lze specifikovat, jaká data budou mezi jednotlivými vlákny sdílená a jaká ne. Pokud není nastaveno jinak, všechny proměnné kromě těch, které jsou definované v paralelním bloku, jsou nastaveny jako sdílené proměnné.

3.1.1.1 Klausule private a firstprivate

Pokud je proměnná určena jako **private**, každé vlákno si vytvoří neinicializovanou lokální proměnou tohoto typu a jména. Pokud je určena jako **firstprivate**, tak se proměnná inicializuje se stejnou hodnotou, kterou měla před vstupem do paralelního regionu.

3.1.1.2 Klausule lastprivate

Pokud má proměnná nastavenou vlastnost **lastprivate**, tak je vytvořena podobně jako proměnná s vlastností **private**, ale způsobuje, že hodnota z poslední dokončené úlohy je zkopírována do proměnné hlavního vlákna. Například pro konstrukt **for** to znamená, že v proměnné hlavního vlákna bude uložena hodnota proměnné vlákna, které dostalo přiřazenou a vykonalo poslední iteraci cyklu.

3.1.2 Konstrukt for

Direktiva **for** rozdělí následující cyklus mezi vlákna.

3.1.2.1 Rozdělování iterací

Počet přidělených iterací jednomu vláknu může být kontrolováno klausulí **schedule**, která má 5 typů a nepovinný parametr *velikost*. Typy jsou následující:

- **static**: Tato možnost je nastavena jako výchozí. Pokud je zvolen tento typ, každé vlákno dostane přiřazený stejný počet iterací jdoucí po sobě. Počet přidělených iterací je určen parametrem *velikost*. Pokud tento parametr není nastaven, iterace jsou mezi vlákna rozdělena rovnoměrně.
- **dynmaic**: Při volbě tohoto typu dostává každé vlákno počet iterací určený parametrem *velikost*. Pokud parametr není nastaven, každé vlákno dostává přidělenou 1 iteraci. V momentě kdy vlákno dokončí iteraci dostane přidělenou další, pokud ještě jsou nějaké k dispozici. Nelze předpovědět v jakém pořadí dostanou vlákna iterace přidělena.
- **guided**: Tento typ je podobný dynamickému přidělování. Počet přidělovaných iterací je zpočátku velký a postupem času se snižuje, aby se vyrovnala nerovnoměrná zátěž jednotlivých iterací. Parametr *velikost* určuje minimální přidělený počet iterací. Výchozí nastavení je podíl počtu iterací a počtu vláken.
- **auto**: Pokud je zvolen tento typ, o přidělování rozhoduje kompilátor.
- **runtime**: K výběru přidělování iterací je využita proměnná prostředí *OMP_schedule*.

3.1.2.2 Klausule ordered

Pořadí provádění iterací cyklů není specifikováno a záleží na běhových podmínkách. Pokud je potřeba, aby byly nějaké příkazy v cyklu provedeny ve správném pořadí (například výpis výsledků), do direktivy **for** je přidána klausule **ordered**. Ta zajistí aby všechny příkazy, které jsou v cyklu označeny pomocí direktivy **#pragma omp ordered**, byly vykonány ve správném pořadí.

3.1.2.3 Klausule collapse

Při použití této klausule se s vnitřními cykly bude jednat jako s jedním cyklem. Počet iterací se mezi cykly roznásobí a z tohoto počtu je pak spočítán počet iterací přiřazený jednomu vláknu. Počet vnitřních cyklů je určený parametrem této klausule.

3.1.2.4 Klausule reduction

Tato klausule zajistí, aby si všechna vlákna vytvořila vlastní instanci této proměnné a na konci použije zadaný operátor k zapsání výsledku do stejnojmenné proměnné hlavního vlákna

3.1.3 Konstrukt task

Konstrukt **task** představuje podporu pro složitější paralelismus. Kód i data jsou zapouzdřena. Je vhodný pro rekurzivní funkce. Direktiva **#pragma omp task** slouží k vytvoření nové úlohy, kterou začne vykonávat nové vlákno.

3.1.3.1 Klausule taskwait

Direktiva **#pragma omp taskwait** čeká na dokončení všech potomků této úlohy.

Kapitola 4

Analýza

V této kapitole jsou rozebrány jednotlivé algoritmy a uvedeny optimalizace, které by mohly zlepšit efektivitu řadících algoritmů.

4.1 Optimalizace algoritmu MergeSort

Klasický MergeSort má časovou složitost $O(n \cdot \log n)$, ale kvůli režii potřebné k rekurzivnímu volání dosahuje při řazení polí o menších velikostech horších výsledků. Přestože algoritmus *InsertionSort* má časovou složitost $O(n^2)$, pro posloupnosti o menší velikosti dosahuje lepších výsledků než MergeSort. Tato vlastnost je využita pro optimalizaci algoritmu MergeSort a do jeho implementace je přidán algoritmus InsertionSort, který řadí pole, jakmile jeho velikost klesne pod určitou hranici. Protože InsertionSort je datově citlivý algoritmus, tato optimalizace bude mít největší vliv zejména na vzestupně seřazené posloupnosti. Naopak nejmenší vliv bude mít na sestupně seřazené posloupnosti. Testováním bude určena optimální hranice pro toto přepnutí.
Další částí optimalizace je snížení zbytečného kopírování pole do pomocného na konci každého slučování seřazených polí. Kopírování je redukováno tak, že data zkopírujeme do pomocného pole již před řazením, aby nezáleželo z jakého pole jsou data řazena, ale pouze do kterého pole se seřadí. Tím dojde ke snížení počtu kopírování z $\log n$ (na každé hladině bylo potřeba zkopírovat všechny prvky) na pouhé jedno.
Při slučování polí se dá využít jejich seřazenosti. Je vybráno pole, jehož první klíč je větší než první klíč druhého pole. Pomocí metody půlení je zjištěno, kolik klíčů z druhého pole, je menší než vybraný klíč. Všechny vybrané prvky pak mohou být zkopírovány, což je rychlejší než klasický *merge*, kvůli práci s pamětí a možnosti použití vektorových instrukcí. Tato optimalizace se nejvíce projeví na seřazených posloupnostech, kdy bude možnost zkopírovat celé jedno pole a následně druhé. Na rozdíl od přidání algoritmu InsertionSort zde na směru seřazení záležet nebude.
Když algoritmus začíná se slučováním seřazených polovin polí, může (měl by)

mít v cache paměti uložený alespoň konec druhého pole. Tohoto se dá vyžít. Při každém řazení bude pole slučováno v opačném směru, čímž se využije zbytek dat uložených v paměti.

4.1.1 K-way MergeSort

MergeSort (tedy 2-way MergeSort) dělí pole na poloviny a hloubka zanoření je tedy $\log_2 n$. K-way MergeSort [6] dělí pole na části o velikosti $\frac{1}{k}$ původní velikosti pole a hloubka zanoření je $\log_k n$ a pro $k > 2$ platí, že $\log_2 n > \log_k n$, takže teoreticky je lepší využívat větší k. Problém ovšem nastává při slučování polí. Zatímco pro $k = 2$ slučování polí je lineární, protože minimum ze dvou čísel dokážu určit pomocí jednoho porovnání, určit minimum z k čísel, kde $k > 2$, je obtížnější. Pokud je minimum z k čísel hledáno lineárním prohledáváním pole, nalezení minima trvá k operací. Musí být nalezeno celkem n čísel, takže v nejhorším případě je složitost slučování k polí je $O(k \cdot n)$. Tento počet operací musím provést na každé hladině. Celková složitost algoritmu k-way MergeSort je tedy $O(n \cdot k \cdot \log_k n)$.
Jak bylo řečeno v kapitole o řadících algoritmech, existuje několik způsobů, které slučují pole se složitostí $\log k$. Toto slučování musí být provedeno na každé hladině a z toho důvodu je celková složitost algoritmu k-way MergeSort $O(\log_k n \cdot \log k \cdot n)$, což je ovšem rovno $O(n \cdot \log n)$, čímž se dostáváme na stejnou složitost jako má 2-way MergeSort. Vzhledem k tomuto faktu není očekáváno, že sekvenční k-way MergeSort bude rychlejší než sekvenční MergeSort.
Důležitá je také volba správného k, které bude vybráno za pomoci testování.

4.1.1.1 Rozděl a panuj merge vs Heap-merge

V kapitole o řadících algoritmech byly popsány dva algoritmy sloužící ke slučování k polí se složitostí $O(\log n)$. Jaký algoritmus tedy vybereme? Heap-merge má problém se stabilním řazení. Ve chvíli, kdy ja vytvořena halda z minimálních prvků, ztrácí se informace o tom, z jakého pole minimum pochází. V zájmu zachování stability proto při porovnání dvou minim nestačí porovnat jejich hodnotu. Pokud jsou si minima rovna, musí být porovnáno jejich původní pořadí, což je další porovnávání navíc. Další nevýhoda slučování pomocí haldy je v paralelizace, která není téměř možná. Z tohoto důvodu se tato práce způsobem slučování Heap-merge dále nezabývá. Ke slučování k polí je používán způsob využívající metodu „Rozděl a panuj“.

4.1.1.2 3-way MergeSort

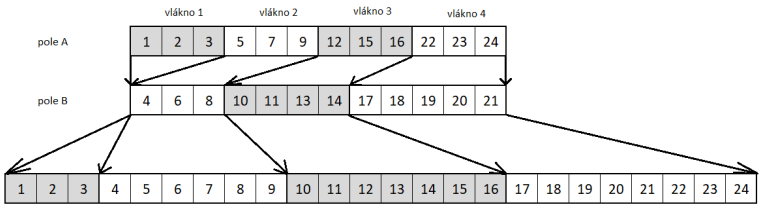
Výjimkou je algoritmus 3-way MergeSort, který při slučování nepotřebuje žádný složitý algoritmus. K určení minima ze tří čísel stačí pouze dvě porovnávání. Přestože platí $2 \cdot n \cdot \log_3 n > n \cdot \log_2 n$, při dobré implementaci je možné, že by 3-way MergeSort mohl dosahovat lepších výsledků než 2-way MergeSort.

4.2 Paralelizace algoritmu MergeSort

MergeSort dělí úlohu na mnoho menších úloh. Nabízí se tedy použít direktivu omp task, která každému vláknu přiřadí jednu úlohu. Toto řešení ovšem nemusí být tak efektivní, jak se na první pohled zdá. Před slučováním polí se musí čekat dokud obě předchozí řazení nebudou dokončena a samotné slučování provádí pouze jedno vlákno z předchozích vláken. Jelikož slučování polí je kritická část algoritmu, další kapitola je zaměřena na něj.

4.2.1 Paralelní merge

Mějme seřazená pole A o velikosti n a pole B o shodné velikosti n, které chceme sloučit do pole C za pomoci k vláken. Rozdělení pole A je jednoduché. Každé vlákno bude řadit stejnou část pole $A_{i\cdot(n/k)} \dots A_{(i+1)\cdot(n/k)}$ kde $0 \leq i < k-1$, dále jsou tyto části označovány $A_i \dots A_{i+1}$. Poslední vlákno bude mít k dispozici zbytek pole. Pole B však takto rozdělit nejde. Aby byl správně určen index B_{j+1}, tak pomocí metody půlení musí být v poli B nalezen prvek ležící na začátku části A_{i+1}. Takto je pole B rozděleno pro všechna vlákna opět s výjimkou posledního, které dostane zbytek pole B. Nakonec musí být rozděleno pole C. Každé vlákno dostane přidělený prostor v poli C, který odpovídá součtu velikostí částí polí A a B, které vlákno řadí.
Tento způsob není úplně optimální, protože pro určité typy posloupností, může být zátěž mezi vlákna rozdělena značně nerovnoměrně. Například pro seřazenou posloupnost, kde každý prvek v poli A je menší než nejmenší prvek v poli B, dostane každé vlákno přiřazenou svojí část A, ale z pole B nedostane nic. Poslední vlákno naopak dostane svojí část A a celé pole B. Tato paralelizace proto nejspíše bude způsobovat menší zrychlení pro seřazené posloupnosti než pro náhodně seřazenou posloupnost. Obrázek 4.1 ukazuje jak by si 4 vlákna rozdělila dvě pole o velikosti 12 a seřadilo ho do výsledného pole o velikosti 24. Na obrázku je vidět nevyváženost. Vlákno 1 řadí pouze tři prvky zatímco vlákno 4 řadí sedm prvků.
Předpokládejme, že paralelní merge budeme používat u algoritmu 2-way MergeSort, který využívá hranici přepnutí. Mějme p vláken a t hranicí přepnutí. Na poslední hladině jsou dvě pole o velikosti menší než t. Každé vlákno tedy dostane část o velikosti menší než t/p. Pokud tato velikost bude velmi malá,



Obrázek 4.1: Paralelní merge: ukázka paralelního merge

bude při zápisu do výstupního pole docházet k falešnému sdílení a paralelní verze by tedy mohla být dokonce pomalejší než sekvenční verze. Proto tuto hranici budeme muset upravit a otestovat její novou optimální hodnotu. Ovšem čím více bude zvyšována hranice přepnutí, tím větší části pole budou řazeny algoritmem InsertionSort, který má složitost $O(n^2)$ a není paralelizovaný.

4.2.2 Paralelní k-way MergeSort

K-way MergeSort jde paralelizovat stejnými dvěma způsoby jako klasický MergeSort. Pomocí direktivy task, která se aplikuje na na rekurzivní volání algoritmu MergeSort a na rekurzivní merge. Stejně jako u algoritmu MergeSort neočekáváme výrazné zrychlení. Druhý způsob je nahrazení použití paralelního slučování, který byl popsán v předchozí sekci. Tato paralelizace by měla být efektivnější než předchozí volba. Vzhledem k tomu, že k-way MergeSort má stejnou složitost jako má 2-way MergeSort, nepředpokládám, že paralelní k-way MergeSort bude rychlejší než paralelní 2-way MergeSort.

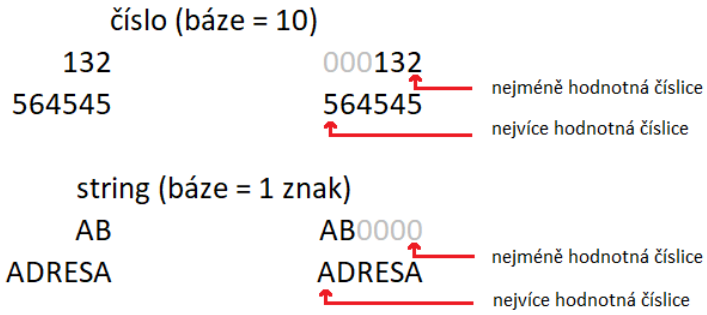
4.2.3 Paralelní k-way merge k-polí

V této části je konečně očekáváno efektivní využití paralelního slučování k polí. V předchozích částech bylo ukázáno několik způsobů, kterými lze paralelizovat algoritmus MergeSort. Paralelizována byla hlavní část algoritmu (slučování), ale aby toho bylo docíleno, musely být před samotnou paralelizací vypočítány počáteční indexy a tyto výpočty nejdou úplně paralelizovat. Samotný InsertionSort, který tvoří přibližně 5% algoritmu (lze jednoduše vypočítat), není paralelní. A nakonec režie spojená s rekurzivním voláním je vykonávaná pouze jedním vláknem. Tyto drobné nedostatky se pokusíme odstranit.
Pole hned na začátku je rozděleno na k částí. Hodnota k bude stejná jako počet dostupných vláken (pro sekvenční řazení bude hodnota $k = 2$). Každé vlákno dostane část pole, kterou seřadí sekvenčně. Jako řadící algoritmus, který bude řadit pole sekvenčně, bude vybrán ten, který dosáhl v sekvenčním běhu nejlepších výsledků.
Jakmile každé vlákno dokončí řazení, tak se výsledná pole paralelně sloučí.

Právě zde je využito paralelní slučování k polí, které bylo popsáno v předchozích kapitolách.
Tato implementace docílí toho, že na dokončení algoritmu InsertionSort nečeká žádné vlákno a každé vlákno si bude řešit vlastní režii spojenou s rekurzí.
Posloupnost, na které by tento algoritmus dosahoval nejhorších výsledků, by musela mít prvních $k - 1$ částí seřazené a poslední část by seřazená nebyla. V tom případě by $k - 1$ vláken dokončilo řazení dříve než poslední vlákno a musela by na něj čekat.

4.3 Optimalizace algoritmu RadixSort

Aby byla zvýšena efektivita algoritmu RadixSort, nabízí se několik možných optimalizací.
První z nich je stejně jako v algoritmu MergeSort odstranění zbytečného kopírování na konci algoritmu CountSort. Kopírování je odstraněno, takže po dokončení bude výstup uložen v pomocném poli. Při příštím zavoláním algoritmu CountSort je jako vstupní pole použito pomocné pole, v němž je uložen výsledek řazení z předchozího běhu. Toto prohazování je opakováno při každém volání. Pokud počet spuštění algoritmu CountSort bude lichý, výsledek posledního řazení bude uložen v pomocném poli a proto musí být zkopírován do správného pole. Toto je jediný případ, kdy dochází ke kopírování dat. V původní verzi dochází ke k kopírování.
Při každém spuštění algoritmu CountSort jsou počítány četnosti číslic, ze kterých je vypočítáváno pořadí pro zadanou pozici. Pro každé počítání číslic musí být klíč načten do paměti a zjištěna číslice na zadané pozici. Dále jsou data v paměti nevyužita. Aby byla práce s pamětí lépe využita, je tato činnost odebrána z algoritmu CountSort a je vložena do samostatné funkce, která spočítá četnosti číslic pro každou pozici předem a potom z nich vypočítá pořadí. Funkce tedy za účelem spočítání četnosti načte do paměti klíč pouze jednou a nikoli k-krát jako v původním případě.
Na výpočet četností a kopírování pole by mohla být aplikována optimalizační technika loop unrolling, aby byly více využity rychlé registry procesoru.
Řazení algoritmem RadixSort je také ovlivněno maximálním počtem hodnot, které mohou jednotlivé číslice nabývat. Tento počet bude dále označována jako báze. Čím větší je báze, tím více se zvyšuje velikost pole četnosti a také pole pořadí, které musí být vypočítáno a s velikostí pole se čas tohoto výpočtu prodlužuje. Na druhou stranu pokud je velikost báze vhodně určena, jsou sníženy velikosti klíčů. Jelikož složitost algoritmu RadixSort je přímo závislá na maximální velikosti klíče ($O(kn)$), snížením této velikosti je zrychleno řazení. Zvyšováním báze je také zvyšována paměťová složitost. Tento nárůst je ovšem minimální, takže může být zanedbán. Optimální bázi je nalezena testováním. Následuje rozbor vlastností, které by měla mít optimální báze a jak



Obrázek 4.2: Optimalizace RadixSortu: rozdíl mezi řazením čísel a textovými řetězci

správně určovat hodnotu číslice.
Začněme celočíselnými datovými typy. Pokud algoritmus RadixSort řadí posloupnost, která může obsahovat záporná čísla, musí být nějakým způsobem ošetřena. Kdyby znaménka nabývala pouze minimální hodnotu číslice, všechna záporná čísla by byla před kladnými čísly, ale byla by seřazena v opačném pořadí. Proto je před získáváním číslice z čísla, které může nabývat záporných hodnot, od něj odečtena minimální hodnota, kterou číslo může nabývat.
Jakých hodnot by měla báze nabývat? U čísla číslici na i-té pozici, kde nejméně významná číslice leží na indexu 0, lze získat následujícím vzorečkem: (číslo/-báze)%báze. Vzhledem k tomu, že operace dělení je poměrně náročná operace pro procesor, je nahrazena bitovým posunem. Z toho důvodu musí být báze rovna nějaké mocnině čísla 2. Operaci modulo lze nahradit bitovým operátorem &.
Druhý datový typ je *std::string* (dále jen string). String je realizovaný jako pole datového typu *char*, který může nabývat hodnot $(0 \ldots 255)$. Řazené textové řetězce ale mohou obsahovat například pouze malá písmena a-z a podle toho může být upravena báze. V tomto případě může být redukována na velikost 26. Báze může být rozšířena o druhé písmeno, aby algoritmus RadixSort řadil řetězce podle dvou písmen najednou, čímž se báze exponenciálně zvýší. Na rozdíl od čísel, které musí být zarovnané tak, aby byly nejméně významné číslice pod sebou a chybějící číslice jsou nahrazeny nejmenší hodnotnou, kterou číslice může nabývat, textové řetězce musí být zarovnány naopak. Všechny textové řetězce musí být zarovnané podle prvního znaku. Rozdíl mezi čísly a textovými řetězci je zobrazen na obrázku 4.2.

4.4 Paralelní RadixSort

Zátěž mezi více vláken u algoritmu RadixSort [11] je rozdělena tím způsobem, že každé vlákno dostane přiřazeno stejně velkou část vstupního pole, které

1) počáteční pole četností

	ČÍSLICE SOUSTAVY O ZÁKLADU 4			
vlákno	0	1	2	3
thr1	3	0	3	4
thr2	0	2	0	8
thr3	3	2	4	1
thr4	3	1	1	5

2) naplnění sloupců pole pozic

	ČÍSLICE SOUSTAVY O ZÁKLADU 4			
vlákno	0	1	2	3
thr1	0	0	0	0
thr2	3	0	3	4
thr3	3	2	3	12
thr4	6	4	7	13
total	9	5	8	18

3) celkový počet výskytů číslic

	ČÍSLICE SOUSTAVY O ZÁKLADU 4			
vlákno	0	1	2	3
thr1	0	0	0	0
thr2	3	0	3	4
thr3	3	2	3	12
thr4	6	4	7	13
total	9	14	22	40

4) výsledné pole pozic

	ČÍSLICE SOUSTAVY O ZÁKLADU 4			
vlákno	0	1	2	3
thr1	0	9	14	22
thr2	3	9	17	26
thr3	3	11	20	34
thr4	6	13	21	35
total	9	14	22	40

Obrázek 4.3: Paralelní RadixSort: ukázka výpočtu pozic

pak všechna vlákna řadí do jednotného výstupního pole. Problém nastává při výpočtu pozice, na kterou musí být klíč uložen. Jelikož se prvky v poli, které je přiřazeno vláknu, mění pro každou pozici. Nelze četnosti a pořadí vypočítat pro všechny pozice dopředu, jak bylo navrženo v předchozí optimalizaci. Z toho důvodu se při výpočtu vychází z původní verze. Mějme k dispozici k vláken. Pole je rozděleno do k částí: $P_1 \dots P_k$. Aby byla zachována stabilita algoritmu, musí pro nové pozice ind prvků se stejnou číslicí z různých částí pole platit následující: $x \in P_i, y \in P_j : ind_x < ind_y$, kde $1 \leq i < j \leq k$.
K vypočtení správného pořadí je použit následující algoritmus. K uložení četností a pořadí bude použito dvojrozměrné pole o rozměrech $k + 1$ řádcích a b sloupcích, kde b značí maximální počet hodnot, kterých může nabývat číslice. Pro práci s pamětí je vhodnější, aby hodnota b byl dělitelná velikostí cache line, aby v dalších částech algoritmu nedocházelo k falešnému sdílení. Nejdříve si každé vlákno spočítá četnosti číslic ve své části pole a uloží je na přidělený řádek pole. Prvnímu vláknu jsou nastaveny počáteční pozice všech číslic na hodnotu 0. Dále je vyplněn zbytek pole pořadí. Pozice, kam vlákno může uložit klíč s danou číslicí, je rovna součtu výskytů číslic ve všech předchozích vláknech. Do posledního řádku je uložen celkový počet výskytů číslic. Poslední řádek je upraven podobným způsobem. Hodnota prvku posledního řádku ve sloupci je přičtena k hodnotě prvku v následujícím sloupci. Nakonec je tato hodnota přičtena ke všem řádkům následujícího sloupce kromě posledního. Tím je vytvořeno pole pořadí. Obrázek 4.3 ukazuje příklad, jak by se měnily tabulky pro výpočet počátečních indexů 40 čísel, které jsou reprezentovány ve čtyřkové soustavě.
Samotné řazení probíhá stejně jako v sekvenční verzi s tím rozdílem, že ka-

ždé vlákno má uložené indexy v přiděleném řádku společné tabulky. Přestože vlákna ukládají data na stejný řádek, výskyt falešného sdílení bude velmi vzácný.

Kapitola 5

Implementace a testování

5.1 Parametry testování

V této části jsou popsána data, které algoritmy budou při testování řadit a výpočetní prostředky, na kterých budou testována. Všechny implementace jsou kompilovány následujícím příkazem:
g++ -std=c++11 -O3 -march=core-avx-i -fopenmp -ffast-math

5.1.1 Testovací server STAR

Výkonnost algoritmů je měřena na výpočetním serveru STAR. K měření jsou využívány jeho uzly *gpu-01* a *gpu-2*. Tyto uzly nejsou přímo dostupné a veškeré úlohy jsou spouštěny přes dávkový plánovač. Oba uzly mají stejné parametry. Využívají dva procesory *6core Xeon 2620 v2 @ 2.1Ghz* a paměť o velikosti 32GB. Jak název uvádí, oba procesory mají k dispozici 6 jader. Z toho důvodu jsou paralelní verze programů testovány pro běh s 12 vlákny.

5.1.2 Testovací data

Algoritmy řadí posloupnosti několika typů. Pokud nebude uvedeno jinak, algoritmy jsou testovány na datech o velikosti 2^{26}. Aby bylo porovnávání efektivity přesnější a všechny algoritmy řadily stejná data, všechny posloupnosti vychází ze stejného souboru, který obsahuje 2^{27} celých čísel, které nabývají hodnot v rozmezí $0 \ldots 2^{31} - 1$. Tyto čísla byla vygenerována programem, který používá funkci *rand* ze standardní knihovny. Typy posloupností jsou popsány níže.

- **Náhodná posloupnost**
  Tato posloupnost obsahuje neseřazená náhodná čísla datového typu *int*. Na optimalizaci řazení této posloupnosti je tato práce zaměřena.

- **Seřazená posloupnost**
  Posloupnost je již vzestupně seřazena posloupnost a měla by ukázat citlivost algoritmů. Algoritmus RadixSort by tuto posloupnost řadit přibližně ve stejné rychlosti, jako posloupnost prvního typu, zatímco některé verze algoritmu MergeSort by měl dosahovat lepších výsledků, jelikož využívají algoritmus InsertionSort, který je datově citlivý a seřazenou posloupnost řadí v čase $O(n)$.

- **Opačně seřazená posloupnost**
  Tato posloupnost je na rozdíl od předchozího typu seřazena sestupně. S těmito typy posloupnosti má naopak algoritmus InsetrionSort problém a řadí je v čase $O(n^2)$.

Následující typy posloupností se liší v datových typech. Při testování na těchto posloupnostech budeme pozorovat, jak různé algoritmy pracují s daty různých datových typů.

- **Posloupnost velkých náhodných čísel**
  Tento typ posloupnosti obsahuje náhodná čísla datového typu *uint64_t*. Čísla jsou upravená čísla z původní posloupnosti.

- **Posloupnost textových řetězců**
  Tento typ posloupnosti obsahuje náhodné textové řetězce o délce až 28 znaků. Tyto řetězce jsou vygenerované z čísel, které jsou obsaženy v posloupnosti velkých náhodných čísel. Řetězce obsahují pouze malá písmena od a do z.

5.2 Pomocné funkce

Soubor *function.h* obsahuje pomocné funkce, které přímo nesouvisí s řazením dat.

5.2.1 Funkce low_bound

Funkce *low_bound* má 4 vstupní parametry: první a poslední iterátor datové struktury, prvek, který musí být stejného datového typu jako prvky obsažené v datové struktuře a funktor, který slouží k porovnávání prvků. Funkce vrací iterátor na prvek, pro který platí, že předchozí prvek je menší a následující větší nebo roven. Tento iterátor je nalezen pomocí binárního půlení.

5.2.2 Funkce loadArray

Funkce *loadArray* k načtení dat ze souboru. Funkce dostane jako vstupní parametr ukazatel na prvek nebo vstupně výstupní vektor, do kterého jsou načteny data ze souboru. Počet dat i název souboru jsou další vstupní parametry.

5.2.3 Funkce checkArray

Funkce *checkArray* slouží ke zkontrolování pole, jestli je vzestupně seřazené. Prochází pole po prvcích a porovnává sousedy, jestli jsou ve správném pořadí.

5.3 Sekvenční MergeSort

Základní implementace algoritmu MergeSortu vychází z implementace [12]. Tato implementace je upravená, aby algoritmus mohl řadit data různých typů. K jejich řazení je potřeba dodat komparátor k tomuto datovému typu.
K testování je použita náhodná, vzestupně seřazená a sestupně seřazená posloupnost. Na tuto implementaci jsem aplikoval optimalizace, které byly uvedeny v kapitole 4.1 v následujícím pořadí.

1. odstranění zbytečného kopírování
2. přidání hranice na přepnutí na algoritmus InsertionSort [13]
3. vyhledávání minima pomocí metody půlení
4. změna směru slučování

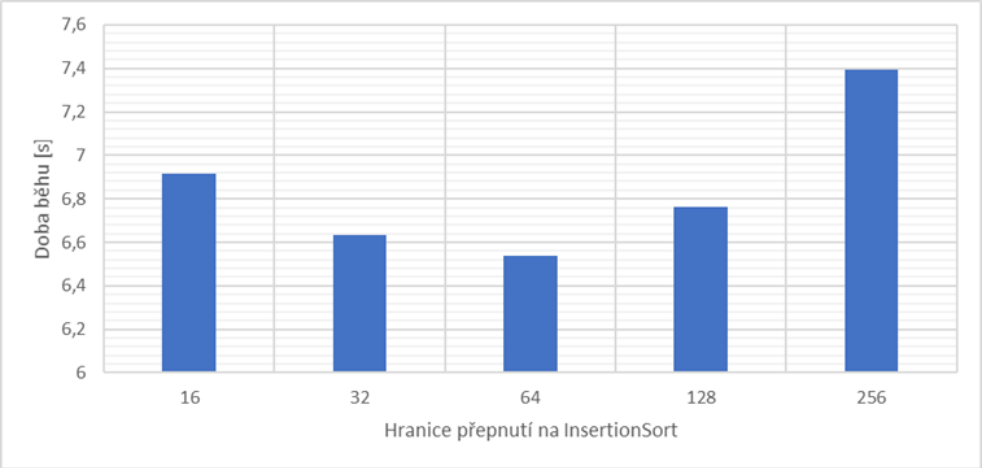
Na grafu 5.2 je zobrazen vliv jednotlivých optimalizací.

5.3.1 Odstranění zbytečného kopírování

Odstranění kopírování mělo velký pozitivní vliv na všechny typy posloupnosti. U náhodné posloupnosti jsem dosáhl zrychlení 1,5 a seřazené posloupnosti řadí algoritmus dokonce 2x rychleji.

5.3.2 Přidání hranice na přepnutí na InsertionSort

Hranice přepnutí na řazení algoritmem InsertionSort byla nastavena na 64 (tzn. pokud velikost řazeného pole bude menší než 64, pole bude seřazeno agloritmem InsertionSort). Tuto hranici jsem určil na základě testování. Výsledky testování jsou vyobrazeny v grafu 5.1. Tato optimalizace měla pozitivní vliv na náhodnou a seřazenou posloupnost. Ale jak jsem předpokládal, tato implementace algoritmu řadí posloupnost seřazenou v opačném směru pomaleji. Jelikož dosažené zrychlení řazení předchozích typů posloupností je vyšší než zpomalení a zpomalení nastává pouze ve vzácných případech. Optimalizaci budu nadále požívat.



Obrázek 5.1: MergeSort: testování hranice přepnutí

5.3.3 Vyhledávání minima pomocí metody půlení

Při slučování jsem využil funkci low_bound na vyhledání menšího minima v opačném poli, abych zjistil jaké prvky při slučování nemusí být porovnávány a stačí je pouze zkopírovat. Doba řazení náhodné posloupnosti se téměř nezmění. Ovšem velkého zrychlení jsem dosáhl u řazení seřazených posloupností. Vzestupně seřazenou posloupnost řadí implementace s touto optimalizací dokonce 3,7 krát rychleji než implementace bez optimalizace, proto je tato optimalizace ponechána.

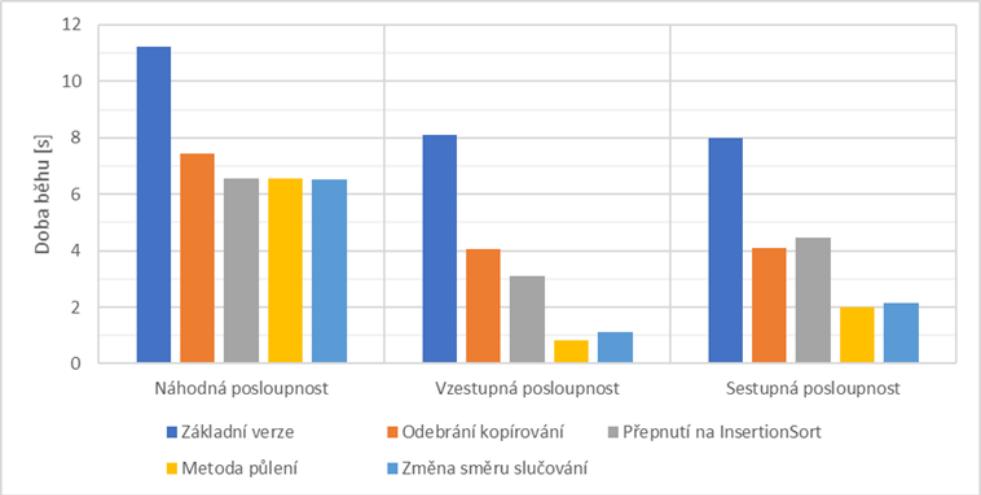
5.3.4 Změna směru slučování

Implementace změny směru slučování měla minimální vliv na řazení náhodné posloupnosti. Bohužel došlo ke zpomalení řazení již seřazených posloupností. Z tohoto důvodu tuto optimalizaci jsem z implementace vynechal.

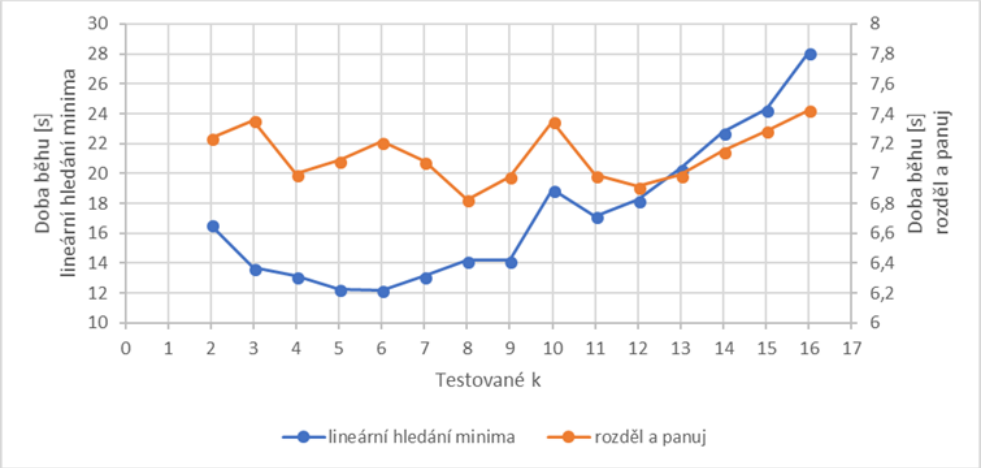
Optimalizace byly celkově velmi úspěšné. Řazení náhodné posloupnosti je přibližně 1,7 krát rychlejší. Ovšem nejvyššího zrychlení jsem dosáhl při řazení vzestupně seřazené posloupnosti, která je řazena téměř 10 krát rychleji.

5.3.5 Sekvenční k-way MergeSort

První implementace algoritmu k-way MergeSort, která je testována, slučuje pole pomocí lineárního hledání minima (dále bude používána zkratka LHM). Toto slučování je dále nahrazeno metodou rozděl a panuj (dále jen R&P). Vhodné k jsem zvolil na základě testování pro každou metodu zvlášť. Výsledky

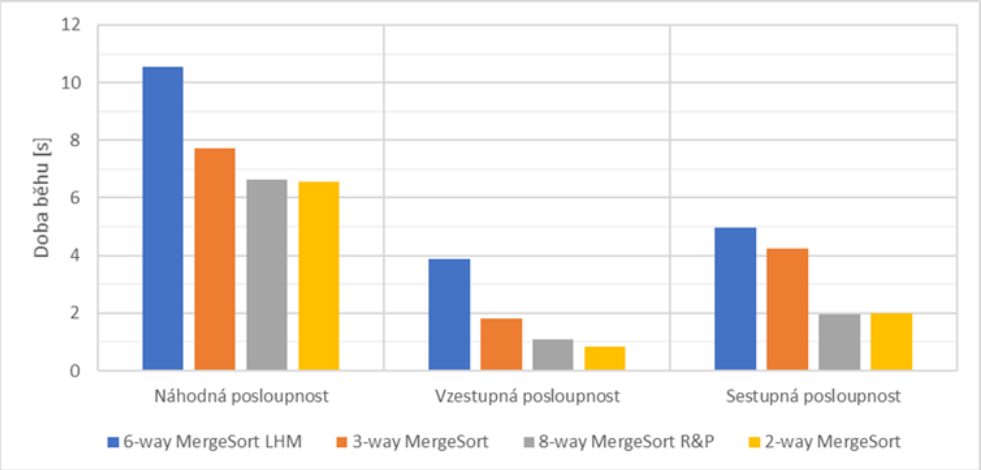


Obrázek 5.2: MergeSort: vliv optimalizací



Obrázek 5.3: K-way MergeSort: hledání vhodného k

jsou k vidění v grafu 5.3. Jelikož obě metody dosahují výrazně jiných časů, každá je vyobrazená pomocí jiné osy. Metoda LHM dosahuje nejlepších výsledků pro $k = 6$ a metoda R&P pro $k = 8$. Poslední testovanou implementací je 3-way MergeSort, který minimum vyhledává pouze pomocí 2 porovnávání.



Obrázek 5.4: MergeSort: porovnávání algoritmů

5.3.6 Nejrychlejší sekvenční algoritmus

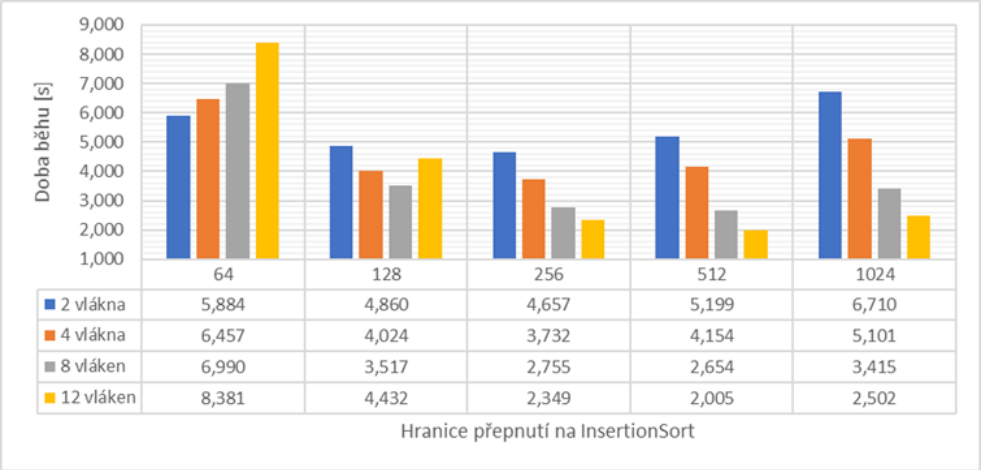
V této sekci jsou porovnávány sekvenční algoritmy. Výsledky algoritmů jsou vidět v grafu 5.4. Porovnávané algoritmy jsou následující:

- 6-way MergeSort využívající ke slučování metodu LHM (dále jen 6-way MergeSort LHM)
- 3-way MergeSort
- 8-way MergeSort využívající ke slučování metodu R&P (dále jen 8-way MergeSort R&P)
- 2-way MergeSort, který jsme v předchozí kapitole optimalizovali

Implementace využívající metodu LHM dosahuje podle očekávání nejhorších výsledků. Implementace je téměř 2x pomalejší než nejrychlejší verze. Implementace 3-way MergeSort dosahuje také špatných výsledků, proto se těmto dvěma verzím již dál nevěnuji. Implementace 8-way MergeSort R&P dosahuje téměř stejných výsledků jako 2-way MergeSort. Podle očekávání v kapitole 4.1.1 je o něco pomalejší.

5.4 Paralelní MergeSort

Nejprve jsem MergeSort paralelizoval s využitím direktivy omp task. Pro každé rekurzivní volání byl vytvořen omp task. Hranice přepnutí na algoritmus InsertionSort byla použita stejná jako v sekvenční verzi. Nicméně došlo ke



	64	128	256	512	1024
2 vlákna	5,884	4,860	4,657	5,199	6,710
4 vlákna	6,457	4,024	3,732	4,154	5,101
8 vláken	6,990	3,517	2,755	2,654	3,415
12 vláken	8,381	4,432	2,349	2,005	2,502

Obrázek 5.5: Paralelní MergeSort: testování hranice přepnutí

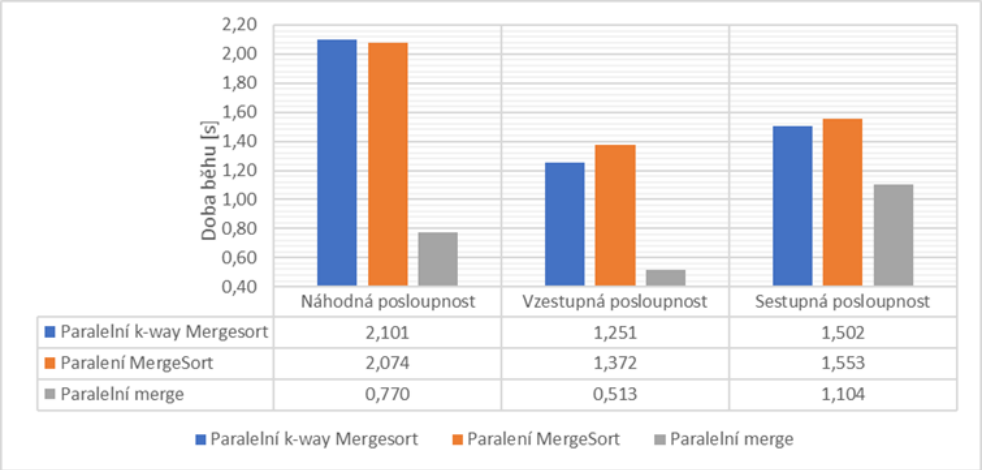
zpomalení a to pravděpodobně z důvodu falešného sdílení. Proto jsem pro paralelní verzi otestoval tuto hranici znovu. Výsledky jsou vidět v grafu 5.5. Implementace algoritmu seřadila posloupnost nejrychleji při použití hranice 512, ale při použití 12 vláken nedosahuje program ani 4 násobného zrychlení oproti sekvenční verzi. Za povšimnutí stojí, že při posledním navýšení hranice, program zpomalil pro všechna vlákna stejně, zatímco předtím docházelo ke zpomalení pouze při použití většího množství vláken. Je to způsobeno tím, že již při použití předchozí hranice, bylo zabráněno falešnému sdílení, takže si vlákna nepřekážela a vyšší použití algoritmu InsertionSort zpomalilo rovnoměrně všechna vlákna.

Dále jsem se pokusil aplikovat paralelní merge popsaný v sekci 4.2.1. Bohužel výsledky dopadly velmi špatně. S počtem použitých vláken nedocházelo ke zrychlení, doba trvání řazení se naopak s použitím většího množství vláken trochu zvyšovala. Pravděpodobně docházelo k falešnému sdílení při zápisu do výstupního pole. Při použití vyšší hranice přepnutí na algoritmus InsertionSort, se program začal zpomalovat kvůli pomalému algoritmu InsertionSort, jak bylo vidět v předchozím grafu. K-way MergeSort s využitím metody R&P slučuje pole o stejných velikostí jako 2-way MergeSort a proto tam paralelní merge nebudu testovat.

K-way MergeSort byl také paralelizován s využitím direktivy omp task. Hranici přepnutí jsem rovnou zvolili 512.

5.4.1 Parelelní slučování k-polí

Při posledním pokusu o paralelizaci jsem pole rozdělil podle počtu vláken a každé vlákno seřadilo svou část pole sekvenčně (vlákna pracoval paralelně).



Obrázek 5.6: Paralelní MergeSort: nejlepší paralelní implementace

Jako sekvenční algoritmus jsem vybral optimalizovaný 2-way MergeSort, protože dosahoval nejlepších výsledků. Při použití p vláken vzniklo p polí. Tyto pole jsou sloučeny využitím metody R&P, kde jsem k paralelizaci využil paralelního merge. Vzhledem k velikosti slučovaných polí, zde k falešnému sdílení již nedochází. Jelikož jsou jednotlivé částí pole řazeny sekvenčně, hranici přepnutí jsem nastavil na hodnotu 64.

5.4.2 Nejlepší paralelní MergeSort

Porovnány jsou následující tři paralelní implementace:

- Paralelní 2-way MergeSort s použitím omp task
- Paralelní k-way MergeSort s použitím omp task
- Paralelní merge p polí

Implementace jsou měřené pro výkon na 12 vláknech. Výsledky porovnání všech 3 paralelních algoritmů jsou vidět v grafu 5.6. Nejdříve zhodnotím implementace využívající direktivu omp task. Implementace dosahují přibližně stejných výsledků, ale k-way verze je více citlivější na vstupní data.
Z porovnávaných implementací je třetí absolutně nejlepší. Porovnání nejlepšího sekvenčního MergeSortu s nejlepším paralelním můžeme vidět v tabulce 5.1.
Paralelní implementace při použití 12 vláken je v řazení náhodné posloupnosti téměř 8,5x rychlejší než nejlepší sekvenční implementace. Nicméně při řazení již seřazených posloupností jsem takového zrychlení nedosáhl. Je to dáno tím,

implementace / posloupnost	náhodná	vzestupná	sestupná
sériový MergeSort	6,545	0,842	1,994
paralelní MergeSort	0,770	0,513	1,104
zrychlení	8,50	1,64	1,81

Tabulka 5.1: Porovnání sekvenční a paralelní implementace algoritmu MergeSort

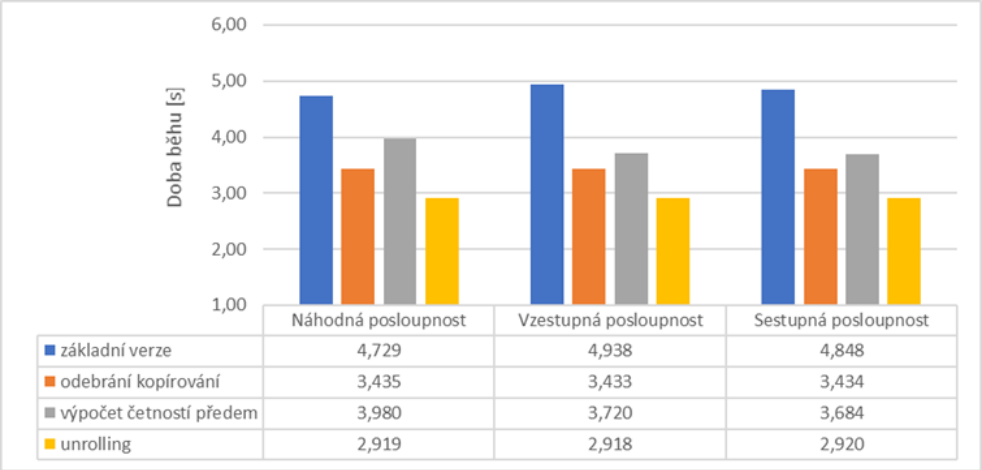
že sekvenční implementace je na tyto typy posloupnosti již velmi dobře optimalizována a ve většině případů dochází pouze ke kopírování. Jak je uvedeno v sekci 4.2.1, paralelní merge má problémy s rozdělením zátěže pro seřazené posloupnosti. Z těchto důvodů není zrychlení příliš velké.

5.5 Sekvenční RadixSort

Implementace algoritmu RadixSort vychází z implementace [14]. Implementaci jsem upravil, aby mohla řadit jakýkoli datový typ, pokud je k němu dodán funktor, který vrátí hodnotu číslice na určité pozici. Předán může být i funktor vracející délku klíče. Pokud není tento funktor dodán, je použit výchozí. Výchozí funktor sloužící k určení délky klíče může pro určité datové typy být pomalejší.
K prvnímu testování jsem použil náhodnou, vzestupně seřazenou a sestupně seřazenou posloupnost náhodných čísel. Použitá báze pro řazení celých čísel má hodnotu 16. Na tuto implementaci jsem aplikoval optimalizace popsané v kapitole 4.3 v následujícím pořadí.

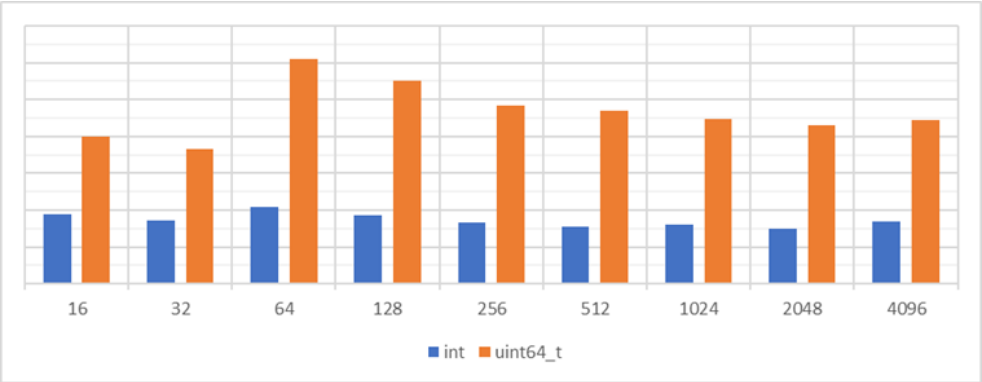
1. odstranění zbytečného kopírování
2. vypočítání četností předem
3. rozbalení cyklů

Vliv jednotlivých optimalizací je vyobrazen na grafu 5.7. Na grafu je na první pohled vidět, že RadixSort je datově necitlivý algoritmus, proto dále nebudu porovnávat tyto posloupnosti. Náhodná posloupnost čísel datového typu *int* (dále jen čísel) zůstává. Seřazené posloupnosti jsem nahradil posloupnostmi obsahující datové typy *std::string* (dále jen textový řetězec) a *uint64_t* (dále jen velká čísla). Jelikož posloupnost s textovými řetězci obsahuje řetězce délky až 28 znaků, tak vyžaduje větší nárok na paměť a celkově je řazení pomalé, proto bude tato posloupnost testována o velikosti pouze 2^{20} řetězců.
Stejně jako u algoritmu MergeSort odstranění kopírování mělo velký vliv na rychlost řazení. Naopak vypočítávání četností pro každou pozici předem řazení výrazně zpomalilo a z toho důvodu jsem tuto optimalizaci z implementace odstranil.



	Náhodná posloupnost	Vzestupná posloupnost	Sestupná posloupnost
základní verze	4,729	4,938	4,848
odebrání kopírování	3,435	3,433	3,434
výpočet četností předem	3,980	3,720	3,684
unrolling	2,919	2,918	2,920

Obrázek 5.7: RadixSort: vliv optimalizací

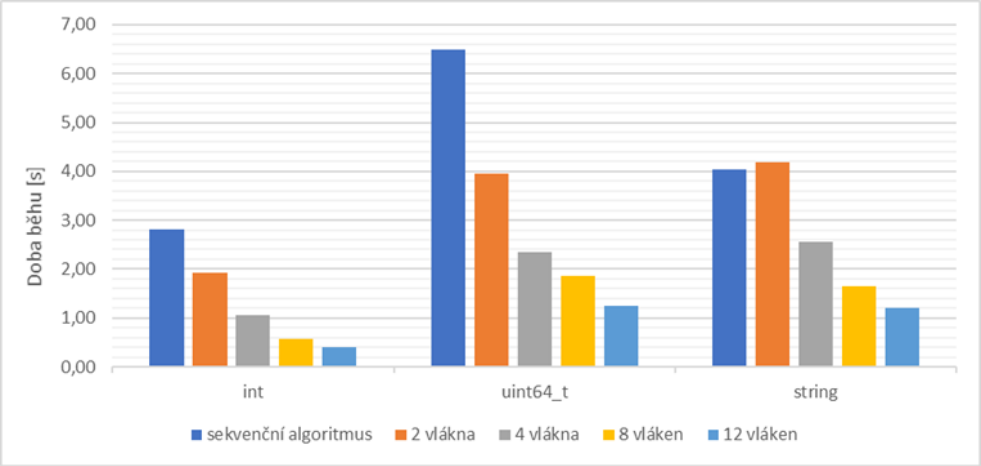


Obrázek 5.8: RadixSort: báze celočíselných typů

Cykly s počítáním četností a samotným řazením jsem rozbalil podle optimalizační techniky loop unrolling. Vyzkoušel jsem několik stupňů rozbalení. V grafu je zanesena pouze ta nejlepší.

5.5.1 Určení optimální báze

Optimální bázi jsem určil pomocí testování. V grafu 5.8 je vidět optimální báze celočíselných typů. Na určitých bázích lze vidět vetší rozdíly mezi rychlostmi. Je to dáno tím, že použitím právě této báze se nám sníží počet číslic v maximálním klíči a tedy i počet nutných průchodů řazení algoritmem CountSort. Zatímco řazení větších čísel dosahuje nejlepších výsledků s použitou bází o ve-



Obrázek 5.9: RadixSort: porovnání paralelní a sekvenční implementace

likosti 32, klasická čísla jsou nejrychleji řazena při použití báze s hodnotou 2048. Zásluhu na to má převod čísel na kladný datový typ *uint32_t*, který musí být proveden při každém zjišťování číslice. Velikost báze textových řetězců v grafu zanesená není. Báze jsem určil testováním na 26^4. RadixSort tedy bude řadit textové řetězce podle 4 znaků najednou.

5.6 Paralelní RadixSort

Paralelní RadixSort jsem implemtoval jak bylo popsáno v kapitole 4.4. Pomocí direktivy omp for jsem paralelizoval hlavní řadící cyklus, počítání četností, hledání nejdelšího klíče a část vypočítávání pořadí. Některé části mohou být prováděny pouze jedním vláknem z důvodu datové závislosti. Jelikož je k dispozici pouze jedna paralelní implementace, je rovnou porovnána s optimalizovanou sekvenční verzí. Výsledky jsou vidět v grafu 5.9 a zrychlení můžeme vidět v tabulce 5.2. Při použití 12 vláken dosahuje program při řazení posloup-

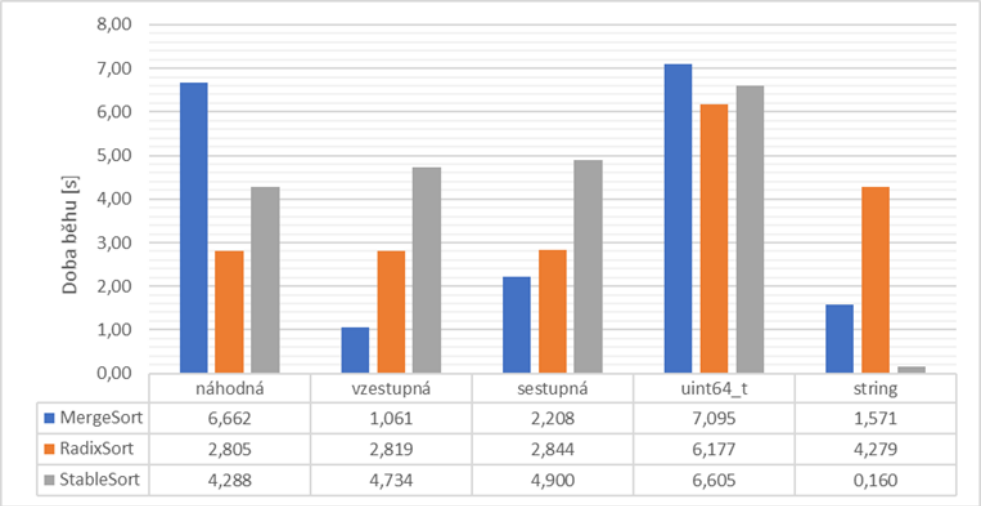
datový typ	int	uint64_t	string
sekvenční algoritmus	2,809s	6,484s	4,037s
2 vlákna	1,46	1,64	0,96
4 vlákna	2,66	2,76	1,58
8 vláken	5,00	3,48	2,44
12 vláken	7,15	5,16	3,32

Tabulka 5.2: RadixSort: zrychlení v závislosti na počtu vláken

nosti náhodných čísel zrychlení 7,15. Důvodem, proč zrychlení není vyšší, je výpočet pořadí, který již nelze paralelizovat. Řazení textových řetězců je pomalejší. Důvodem je velikost báze, na které je závislá velikost pole četností a pozic, které se musí vypočítat a tento výpočet je z velké části sekvenční. Když jsem zmenšil bázi (tedy počet znaků, podle kterých je řetězec řazen), tak se zvýšil počet průchodů a celý algoritmus se ještě více zpomalil. Proto se musíme smířit s faktem, že použití 2 vláken se v tomto případě nevyplatí.

5.7 Porovnání sekvenčních implementací

V této sekci porovnávám optimalizované sekvenční implementace algoritmů RadixSort a MergeSort, které byly popsány v předchozích kapitolách. Třetí porovnávaná implementace je *std::stable_sort* (dále jen stable_sort) z knihovny *alghoritm*, která implementuje algoritmus MergeSort v jazyce C++. K porovnávání poslouží posloupnosti náhodných čísel, vzestupně a sestupně seřazená posloupnost a posloupnosti textových řetězců a velkých čísel. Všechny posloupnosti obsahují 2^{26} prvků s výjimkou textových řetězců, kterých je pouze 2^{20} z důvodu uvedených v sekci 5.5.
Výsledky porovnávání jsou vidět v grafu 5.10.
Začnu s implementací algoritmu RadixSort. Tato implementace řadí náhodně seřazenou posloupnost téměř 2,5 krát rychleji než implementace algoritmu MergeSortu (implementací algoritmu MergeSort je vždy myšlena implementace popsána v této práci) a 1,5 krát rychleji než stable_sort. Za tento rozdíl může povaha algoritmu. Algoritmus je závislý na velikosti datového typu, která v případě čísel při použití báze 2048, je pouze 3, což je rovno počtu potřebných průchodu algoritmu CountSort. Zatímco algoritmus MergeSort je závislý na počtu řazených prvků. Trochu překvapivě implementace algoritmu RadixSort seřadila nejrychleji i posloupnost s velkými čísly, kterou sice řadí více než 2 krát pomaleji než posloupnost čísel, ale pořád rychleji něž obě implementace algoritmu MergeSort. Tento rozdíl je způsobený právě velikostí datových typů. Nejhoršího výsledku implementace algoritmu RadixSort dosáhla při řazení textových řetězců, které mají velkou délku.
Implementace MergeSort řadí náhodně seřazenou posloupnost asi o 42% pomaleji než stable_sort. Dalším velkým rozdílem mezi nimi je v citlivosti. Zatímco stable_sort dosahuje horších výsledků při řazení seřazených posloupností, implementace algoritmu MergeSort řadí již správně seřazenou posloupnost více jak 6 krát rychleji než neseřazenou. Oba algoritmy zpomalují při řazení větších kladných čísel a implementace algoritmu MergeSort se v rychlosti stable_sort téměř vyrovná. Je pomalejší přibližně o 7%. Toto zpomalení je u obou implementací pravděpodobně způsobeno kopírováním větších datových typů.
Implementace stable_sort dominuje v řazení textových řetězců. Pravděpodobně s textovými řetězci lépe pracuje ve vnitřních funkcích.

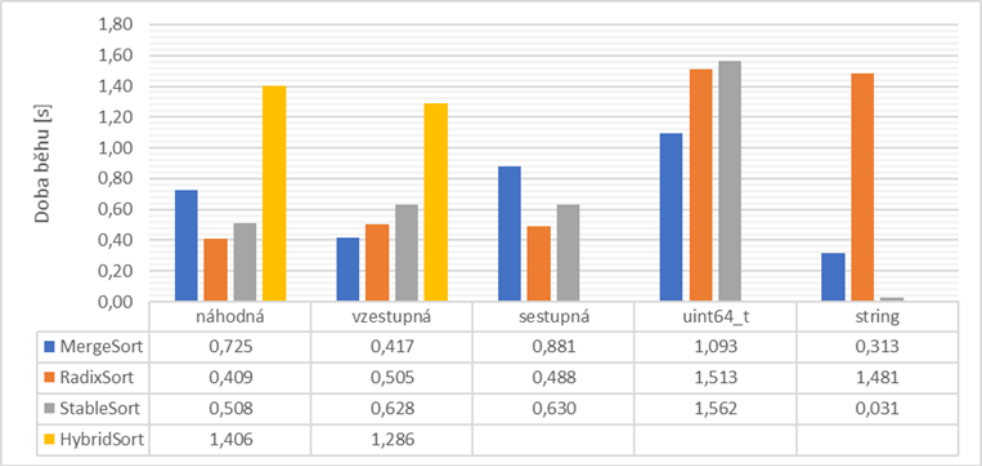


	náhodná	vzestupná	sestupná	uint64_t	string
MergeSort	6,662	1,061	2,208	7,095	1,571
RadixSort	2,805	2,819	2,844	6,177	4,279
StableSort	4,288	4,734	4,900	6,605	0,160

Obrázek 5.10: Porovnání sekvenčních implementací algoritmů

5.8 Porovnání paralelních implementací

Následuje porovnávání paralelních implementací algoritmů. Prvním testovaným algoritmem z této práce je algoritmus MergeSort. Vybraná byla nejrychlejší implementace a tedy paralelní slučování k-polí. Další porovnávanou implementací je paralelní implementace algoritmu RadixSort. Tyto implementace měly být porovnávané s implementací „A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP“ [15]. Při testování však tato implementace dosahovala velmi špatných výsledků, zejména při použití více jak 2 vláken, kdy se doba řazení exponenciálně zvyšovala. Z tohoto důvodu byla implementace nahrazen implementací stable_sort, který byla puštěna v paralelním režimu pomocí „The GNU libstdc++ parallel mode“ (dále jen paralelní stable_sort). Tato implementace pole rozělí na p částí o stejné velikosti, které paralelně seřadí a následně paralelně sloučí pomocí p-merge [16]. Poslední implementací je HybridSort [17], což je implementace kombinující algoritmy TimSort a MergeSort. Bohužel jsem neměl k dispozici tuto implementaci a vycházím tedy z dat, které jsou uvedeny v práci [17]. Implementace byla testovaná na stejném serveru na vzestupně seřazené posloupnosti a posloupnosti náhodných čísel o velikosti 2^{26} při 12 dostupných vláknech.
Měření jsem provedl s využitím 12 vláken na stejných posloupnostech jako v předchozí sekci.
Implementace HybridSort je ze všech nejpomalejší. Výsledky však mohou být nepřesné vzhledem k tomu, že se liší testovací data.
Další v pořadí je paralelní implementace algoritmu RadixSort, která opět ze všech implementací nejrychleji řadí náhodně seřazená čísla a nejlépe si vede



	náhodná	vzestupná	sestupná	uint64_t	string
MergeSort	0,725	0,417	0,881	1,093	0,313
RadixSort	0,409	0,505	0,488	1,513	1,481
StableSort	0,508	0,628	0,630	1,562	0,031
HybridSort	1,406	1,286			

Obrázek 5.11: Porovnání paralelních implementací algoritmů

i při řazení sestupně seřazené posloupnosti. Při řazeních dlouhých čísel si vede spíše průměrně a dosáhla zde pouze 4 násobného zrychlení, což při použití 12 vláken není příliš velké. Textové řetězce řadí dokonce 50 krát déle než paralelní stable_sort.
Paralelní implementace algoritmu MergeSort řadí celočíselné datové typy 9 krát rychleji než jeho sekvenční verze, což je největší dosažené zrychlení ze všech porovnávaných implementací. Při řazení seřazených posloupností už podobného zrychlení nedosahuje. Sestupně seřazenou posloupnost, kterou sekvenční verze řadí nejrychleji ze sekvenčních implementací, seřadí dokonce nejpomaleji z testovaných paralelních implementací. Vysokého zrychlení také dosahuje při řazení velkých čísel.
Paralelní stable_sort opět nejlépe řadí textové řetězce.

Závěr

Cílem práce byla analyzovat řadící algoritmy MergeSort a RadixSort, navrhnout optimalizace, tyto algoritmy efektivně implementovat a poté paralelizovat.
Algoritmy byly nejdříve analyzovány a poté optimalizovány pomocí transformací kódu a vektorizace. Většina optimalizací byla úspěšná a vedly k celkovému zrychlení.
Implementace dále byly paralelizovány pomocí knihovny OpenMP, což pomohlo k výraznému zrychlení u některých verzí algoritmů.
V poslední části byly výsledky změřeny a porovnány s již existujícími implementacemi stabilních řadících algoritmů, které pro některé vstupní posloupnosti dokonce předčily.

Literatura

[1] Malík, J.; Suchý, O. S.; Tvrdík, P.; aj.: Algoritmy řazení a binární halda. 2017, [cit. 2018-05-07]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-AG1/_media/lectures/bi-ag1-p4-handout.pdf

[2] Balík, M.; Vágner, L.; Vogel, J.: Algoritmy a složitost, vyhledávání a řazení [online]. 2017, [cit. 2018-05-14]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-PA1/_media/lectures/l09-cplx-cz.pdf

[3] Malík, J.; Suchý, O.; Tvrdík, P.; aj.: QuickSort, dolní odhad složitosti řazení, speciální algoritmy řazení [online]. 2017, [cit. 2018-05-14]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-AG1/_media/lectures/bi-ag1-p9-handout.pdf

[4] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to Algorithms*. Druhé vydání, ISBN 0-262-03293-7.

[5] Malík, J.; Suchý, O.; Tvrdík, P.; aj.: Rekurzivní algoritmy a metoda Rozděl-a-panuj [online]. 2017, [cit. 2018-05-07]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-AG1/_media/lectures/bi-ag1-p8.pdf

[6] Greene, W. A.: k-way merging and k-ary sorts [online]. 1993, [cit. 2018-05-07]. Dostupné z: http://cs.uno.edu/people/faculty/bill/k-way-merge-n-sort-ACM-SE-Regl-1993.pdf

[7] Knuth, D. E.: *The art of computer programming*. Addison-Wesley, druhé vydání, 1981, ISBN 0-201-89685-0.

[8] Yliluoma, J.: Guide into OpenMP: Easy multithreading programming for C++ [online]. 2016, [cit. 2018-05-15]. Dostupné z: https://bisqwit.iki.fi/story/howto/openmp

[9] Šimeček, I.: Technologie OpenMP [online]. 2017, [cit. 2018-05-15]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/omp.pdf

[10] OpenMP* Loop Scheduling [online]. 2014, [cit. 2018-05-15]. Dostupné z: https://software.intel.com/en-us/articles/openmp-loop-scheduling

[11] Maus, A.; Gjessing, S.: Practical Parallel Programming – a plan for a B.S. course on how to design efficient parallel algorithms. 2014, [cit. 2018-05-07]. Dostupné z: http://heim.ifi.uio.no/~arnem/sorting/ParaRadix2014/MausGjessingFinal.pdf

[12] Neckář, J.: Merge sort. 2016, [cit. 2018-05-07]. Dostupné z: https://algoritmy.net/article/13/Merge-sort

[13] Neckář, J.: Insertion sort. 2016, [cit. 2018-05-07]. Dostupné z: https://www.algoritmy.net/article/8/Insertion-sort

[14] Radix Sort [online]. [cit. 2018-05-07]. Dostupné z: https://www.geeksforgeeks.org/radix-sort/

[15] Robison, A. D.: A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP. 2014, [cit. 2018-05-07]. Dostupné z: https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp

[16] Singler, J.; Sanders, P.: The GNU libstdc++ parallel mode: Benefit from Multi-Core using the STL. [cit. 2018-05-07]. Dostupné z: http://ls11-www.cs.tu-dortmund.de/people/gutweng/AD08/VO11_parallel_mode_overview.pdf

[17] Talácko, R.: *Pokročilé metody řazení ve vícevláknovém prostředí.* Bakalářská práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Příloha **A**

Seznam použitých zkratek

OpenMP soustava direktiv pro překladač a knihovních procedur pro paralelní programování

LSD least significant digit

LHM lineární hledání minima

R&P rozděl a panuj

GNU svobodný operační systém

Příloha **B**

Obsah přiloženého flash disku

- `readme.txt` ..... stručný popis flash disku
- `src`
  - `sorts` ..... zdrojové kódy řadících algoritmů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `data.xlsx` ..... data používaná k tvorbě grafů
- `text` ..... texty práce
  - `BP_Siba_Dominik.pdf` ..... text práce ve formátu PDF
  - `BP_Siba_Dominik.ps` ..... text práce ve formátu PS