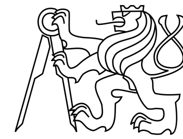
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Pokro ilé metody azení ve vícevláknovém prost edí
Student:	Rudolf Talácko
Vedoucí:	doc. Ing. Ivan Šime ek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte následující algoritmy azení: pomocí vícecestného slévání (k-way MergeSort, viz [1,2]) a TimSort (viz [3]).
2) Implementujte je v jazyce C++ a optimalizujte transformacemi kódu a paralelizací pomocí technologie OpenMP (vícevláknové nad sdílenou pam tí).
3) Navrhn te paralelní hybridní algoritmus využívající kombinaci algoritm z bodu 1) s ohledem na maximální výkonnost a množství dodate né pam ti.
4) Zm te výkonnost tohoto hybridního algoritmu na fakultním serveru Star pro r zné vstupní posloupnosti a pro r zné typy azených objekt (char, int, string, ...).
5) Porovnejte výkonnost hybridního algoritmu s existujícími implementacemi paralelního stabilního azení, zejména s [4].

Seznam odborné literatury

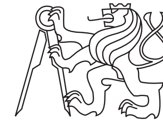
[1] Antonios Symvonis: Optimal Stable Merging, The Computer Journal,1993.
[2] Johannes Singler, Peter Sanders: The GNU libstdc++ parallel mode:Benefit from Multi-Core using the STL
[3] Petr McIlroy's: "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 467–474, January 1993
[4] Arch D. Robison. A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP. Intel® Software. [online]. 11.4.2014 [cit. 2016-11-18]. Dostupné z:
https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp

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

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 18. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Bakalářská práce

Pokročilé metody řazení ve vícevláknovém prostředí

Rudolf Talácko

Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

12. května 2017

Poděkování

Zde bych rád poděkoval panu doc. Ing. Ivanu Šimečkovi, Ph.D. za mnoho cenných rad při vedení této práce. Dále bych chtěl poděkovat zejména své rodině a přítelkyni za jejich podporu a trpělivost během celého 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 12. května 2017 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Rudolf Talácko. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

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

Abstrakt

Tato bakalářská práce se zabývá vybranými řadícími algoritmy. Konkrétně se jedná o algoritmy MergeSort a TimSort. Tyto algoritmy jsou implementovány v jazyce C++ a;pomocí metod transformací zdrojového kódu, efektivního využívání skrytých pamětí a paralelizací jsou upraveny tak, aby byly časově i paměťově efektivní. Dále je vytvořen autorem navržený hybridní algoritmus, využívající kombinaci předchozích algoritmů. Algoritmy jsou testovány na výpočetním serveru STAR, který umožňuje objektivní měření výkonnosti a efektivity algoritmů. Výsledkem práce je sada řadících algoritmů, grafické znázornění jejich výkonnosti a porovnání s již existujícími implementacemi paralelního stabilního řazení.

Klíčová slova Řadící algoritmy, MergeSort, TimSort, Optimalizace, Paralelizace, C++, OpenMP

Abstract

This bachelor thesis deals with sorting algorithms. Specifically, it is MergeSort and TimSort algorithms. These algorithms are implemented in C++, and by code transformation methods, efficient use of cache and parallelizations are adjusted to make their time and memory efficient. Furthermore, the author creates a hybrid algorithm using a combination of previous algorithms. Algorithms are tested on a STAR computing server that allows objective measurement of the performance and efficiency of algorithms. The result of this thesis is a set of sorting algorithms, graphical representation of their performance and comparison with already existing realizations of parallel stable sorting.

Keywords Sorting algorithms, MergeSort, TimSort, Optimization, Parallelization, C++, OpenMP

Obsah

Úvod 1

1 Cíl práce 3

2 Řadící algoritmy 5
- 2.1 Taxonomie řadících algoritmů 5
- 2.2 Složitost algoritmu 6
- 2.3 Přehled základních řadících algoritmů 8
- 2.4 InsertionSort 8
- 2.5 MergeSort 9
- 2.6 K-way MergeSort 10
- 2.7 TimSort 12

3 Optimalizační techniky 17
- 3.1 Kompilátorové optimalizece 17
- 3.2 Metody transformace zdrojového kódu 17
- 3.3 Paralelizace 19
- 3.4 Cache paměti 23

4 Testování a měření 25
- 4.1 Typy vstupních posloupností 25
- 4.2 Výpočetní server STAR 29

5 Implementace a dosažené výsledky 31
- 5.1 Pomocné funkce 31
- 5.2 TimSort 32
- 5.3 MergeSort 38
- 5.4 MergeSort s hranicí přepnutí 39
- 5.5 K-way MergeSort I. 43
- 5.6 K-way MergeSort II. 45

5.7 Hybridní algoritmus 51
5.8 Porovnání výkonnosti paralelních algoritmů 56

Závěr **61**

Literatura **63**

A Seznam použitých zkratek **67**

B Obsah přiloženého média **69**

Seznam obrázků

2.1 Princip řazení algoritmu k-way MergeSort 11
2.2 Princip efektivního slévání běhů algoritmu TimSort 14
2.3 Princip Galloping mode algoritmu TimSort 15

3.1 Amdahlův zákon - závislost zrychlení na počtu procesorů 20

4.1 Vstupní sekvence typu 1 26
4.2 Vstupní sekvence typu 2 26
4.3 Vstupní sekvence typu 3 pro malá N 27
4.4 Vstupní sekvence typu 3 pro velká N 27
4.5 Vstupní sekvence typu 3 pro velká N - detail bloku 28
4.6 Vstupní sekvence typu 3 pro velká N - základní prvek 28
4.7 Vstupní sekvence typu 4 29

5.1 TimSort - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti (porovnání metod volby MinRun) 34
5.2 Paralelní TimSort - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti 36
5.3 Paralelní TimSort - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken 37
5.4 Paralelní TimSort - závislost zrychlení na počtu řazených prvků a počtu paralelních vláken 37
5.5 MergeSort - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti 38
5.6 Paralelní MergeSort - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken 39
5.7 MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a hranici přepnutí 40
5.8 MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti 40

5.9 Paralelní MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a hranici přepnutí 41
5.10 Paralelní MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken 42
5.11 Paralelní K-way MergeSort I. - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti 44
5.12 K-way MergeSort II. - princip hledání hranic 46
5.13 K-way MergeSort II. - struktura uložení hraničních hodnot 46
5.14 K-way MergeSort II. - pomocná pole A-Count a A-Begin 47
5.15 Paralelní K-way MergeSort II. - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti 49
5.16 K-way MergeSort II. - ukázka špatné rozložení zátěže 50
5.17 Hybridní algoritmus - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti 52
5.18 Hybridní algoritmus - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken 53
5.19 Hybridní algoritmus - závislost zrychlení na počtu řazených prvků a počtu paralelních vláken (vstupní posloupnost 3) 54
5.20 Hybridní algoritmus - závislost zrychlení na počtu řazených prvků a počtu paralelních vláken (vstupní posloupnost 4) 54
5.21 Hybridní algoritmus - závislost doby běhu na typu vstupní posloupnosti a velikosti řazených prvků 56
5.22 Porovnání paralelních algoritmů - závislost doby běhu na typu vstupní posloupnosti 57
5.23 Porovnání hybridního algoritmu s existujícími implementacemi paralelního stabilního řazení 59

Seznam tabulek

2.1 Přehled řadících algoritmů a jejich vlastností 8

5.1 MergeSort - porovnání výsledků různých verzí algoritmu 42
5.2 K-way MergeSort - porovnání výsledků různých verzí 49

Úvod

Asi jen obtížně bychom hledali člověka, který by mohl říci, že se s žádnou formou řazení nikdy nesetkal. Řadící algoritmy jsou všude kolem nás a setkáváme se s nimi prakticky každý den. Řadit můžeme téměř cokoliv: jména a příjmení dle abecedního pořadí; předměty podle jejich velikosti, váhy nebo třeba ceny. Většina lidí provádí řazení jednodušších věcí tak nějak intuitivně a příliš se nezamýšlí nad nějakými pravidly nebo algoritmizaci. Pokud bychom ale měli za úkol seřadit velké množství prvků, jistě bychom chtěli nějaký čas ušetřit a řadit efektivně.

Řadící algoritmy jsou vyvíjeny a optimalizovány již několik desítek let. Za tu dobu stačila vzniknout celá řada algoritmů a nespočet jejich různých implementací. Mohlo by se zdát, že tato oblast je již podrobně prozkoumána a není třeba se jí dále zabývat. Opak je však pravdou.

Dnešní svět generuje obrovské množství dat, se kterými je nutné efektivně pracovat. Jsou vyvíjeny stále výkonnější počítače, které v některých směrech začínají narážet na své limity a při zvyšování výkonu se vydávají jinými směry, než tomu bylo doposud. Zvyšování frekvence taktu procesoru již není tak snadné, jako tomu bylo v minulosti. Při zvyšování výpočetního výkonu se tedy současná doba ubírá směrem přidávání dalších funkčních jednotek a vznikají tak velké paralelní systémy. Z tohoto důvodu je třeba se zaměřit na paralelizaci algoritmů, která by dokázala efektivně využít vše, co nám současný hardware nabízí. Proto se autor této práce rozhodl zabývat řadícími algoritmy z hlediska jejich paralelního programování a optimalizace pro více jádrové procesory.

Kapitola 1

Cíl práce

Cílem teoretické části práce je nastudovat řadící algoritmy MergeSort a TimSort. Dále nastudovat metody transformace zdrojového kódu a možnosti paralelizace nad sdílenou pamětí. Cílem praktické části je implementace těchto algoritmů a také návrh a implementace hybridního řadícího algoritmu, který využívá kombinaci předchozích algoritmů. V další části budou tyto algoritmy optimalizovány pomocí metod transformací zdrojového kódu a paralelizací s ohledem na maximální výkonnost a množství dodatečné paměti. Nakonec budou algoritmy otestovány na fakultním serveru STAR, kde bude změřena jejich výkonnost pro různé datové typy a velikosti vstupních dat. Algoritmy budou porovnány s již existujícími implementacemi a výsledky vyhodnoceny prostřednictvím tabulek a grafů.

Kapitola 2

Řadící algoritmy

Řadící algoritmy jsou velmi rozsáhlou a často používanou skupinou algoritmů. Jejich úkolem je seřadit vstupní data do požadovaného pořadí podle předem určeného klíče. Nejčastěji se můžeme setkat s řazením čísel podle numerické hodnoty nebo textu podle abecedního pořadí. Hodnoty mohou být dle typu nerovnosti seřazeny buď vzestupně nebo sestupně, my budeme dále uvažovat pouze vzestupné řazení.

Definice problému řazení:

- Vstup: libovolná posloupnost $A = \langle a_1, a_2, \ldots, a_n \rangle$,
- Výstup: taková permutace $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ vstupní posloupnosti A, že platí $a'_1 \leq a'_2 \leq \ldots, a'_n \rangle$,
- Příklad: $n = 6,\ A = \langle 23, 7, 19, 21, 5, 12 \rangle,\ A' = \langle 5, 7, 12, 19, 21, 23 \rangle$ [1].

2.1 Taxonomie řadících algoritmů

Řadící algoritmy můžeme rozlišovat podle několika parametrů, díky kterým lze určit vhodnost daného algoritmu pro konkrétní problém. V této kapitole bylo čerpáno z [1].

2.1.1 Časová složitost

Jedním z nejdůležitějších parametrů řadících algoritmů je jejich časová složitost. Tento údaj nám říká kolik času (operací) je třeba pro seřazení vstupních dat.

2.1.2 Paměťová náročnost

Algoritmy můžeme také dělit podle jejich paměťové složitosti, přesněji řečeno podle množství přídavné paměti potřebné pro běh algoritmu. Pokud je veli-

kost dodatečné paměti nezávislá na množství řazených dat, říkáme, že daný algoritmus je in-place. Pokud ovšem z nějakého důvodu potřebujeme další paměť (typicky násobek velikosti původních dat), označujeme algoritmus jako out-of-place.

2.1.3 Stabilita

Další důležitou vlastností řadících algoritmů je jejich stabilita. Pokud se v souboru dat, který chceme seřadit, nachází více než jeden prvek se stejným klíčem, může pro nás být důležité, zda po seřazení budou tyto prvky ve stejném pořadí jako před seřazením. Pokud řadící algoritmus zachovává původní pořadí, označíme ho jako stabilní, pokud ne, říkáme že je nestabilní. Stabilita není závislá pouze na řadícím algoritmu, ale na jeho konkrétní implementaci (může existovat algoritmus, který má stabilní i nestabilní verzi). Typickým příkladem může být například řazení dle jména a následně podle příjmení, kdy chceme, aby seřazenost křestních jmen u osob se stejným příjmením byla zachována.

2.1.4 Datová citlivost

Některé algoritmy mohou v závislosti na pořadí vstupních dat řadit různě rychle. Pokud například uvážíme jeden ze základních algoritmů - Bubble sort a necháme jej seřadit již seřazený soubor dat o velikosti n prvků, algoritmus provede pouze n porovnání a skončí. Pokud ovšem na vstup dáme posloupnost seřazenou opačně, bude nutné provést n^2 operací. Je to způsobeno tím, že množství seřazených prvků snižuje počet operací, které musí algoritmus udělat pro seřazení zbytku posloupnosti. Datově citlivý algoritmus dokáže některé typy vstupních posloupností (typicky již částečně seřazená data) seřadit rychleji než jiné. Naopak datově necitlivý algoritmus seřadí každou permutaci vstupních dat asymptoticky stejně rychle.

2.2 Složitost algoritmu

Text této kapitoly vychází z informací uvedených v [2]. Pro nějaký problém může existovat více algoritmů na jeho řešení. Každá taková implementace může být jinak efektivní (ať už se jedná o časovou nebo paměťovou složitost). Z tohoto důvodu se snažíme složitost jednotlivých implementací algoritmů analyzovat. Obvykle potřebujeme určit, jak se bude algoritmus chovat v závislosti na velikosti vstupních dat. Podle typu problému nás může zajímat:

- doba výpočtu
- počet provedených operací
- množství použité paměti

Jak již bylo řečeno v předchozí kapitole, někdy se může stát, že složitost algoritmu nezávisí pouze na velikosti vstupních dat, ale také na konkrétních hodnotách vstupní posloupnosti. Proto potřebujeme umět vyjádřit složitost:

- v nejlepším případě
- v průměrném případě
- v nejhorším případě

Určit složitost v průměrném případě je zvlášť složité, protože není zřejmé co znamenají vstupní data v průměrném případě. Někdy se může jednat o náhodně vygenerovaná data, ale vzhledem k povaze jednotlivých algoritmů to nemusí být vždy pravidlem.

2.2.1 Asymptotická složitost

Přesnou složitost lze určit jen u jednodušších algoritmů. Z praktického hlediska nás zajímá především, jak se bude algoritmus chovat pro velký počet řazených prvků ($N \to \infty$). Je to složitost vyjádřená pro instance problému dostatečně velké, aby se jasně projevil řád růstu složitosti v závislosti na velikosti vstupních dat. Typicky nám stačí zjistit řád růstu funkce vyjadřující složitost se zanedbáním příspěvků nižších řádů. Asymptoticky lepší algoritmus bude lepší pro všechny instance problému kromě konečného počtu menších instancí.

2.2.1.1 Asymptotická horní mez: O-notace

„*Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je nejvýše řádu $g(n)$, psáno $f(n) = O(g(n))$, jestliže*

$$\exists c \in R^{+} \exists n_0 \in N^{+} \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

O-notaci používáme pro vyjádření horní meze růstu funkce až na multiplikativní konstantu. Používáme ji pro odhady složitosti algoritmů v nejhorších případech.“[2]

2.2.1.2 Asymptotická dolní mez: Ω-notace

„*Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je nejméně řádu $g(n)$, psáno $f(n) = \Omega(g(n))$, jestliže*

$$\exists c \in R^{+} \exists n_0 \in N^{+} \forall n \geq n_0 : c \cdot f(n) \leq g(n)$$

Ω-notaci používáme pro vyjádření dolní meze růstu funkce až na multiplikativní konstantu. Používáme ji pro odhady složitosti algoritmů v nejlepších případech.“[2]

2.2.1.3 Asymptotická těsná mez: Θ-notace

„*Jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je téhož řádu jako $g(n)$, psáno $f(n) = \Theta(g(n))$, jestliže*

$$\exists c_1, c_2 \in R^+ \exists n_0 \in N^+ \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Θ-notaci používáme pro vyjádření faktu, že dvě funkce rostou asymptoticky stejně až na multiplikativní konstantu.“[2]

2.3 Přehled základních řadících algoritmů

Tabulka 2.1: Přehled řadících algoritmů a jejich vlastností

Algoritmus	Časová složitost	Přídavná paměť	Stabilita	Citlivost
BubbleSort	$\Omega(n),\ \Theta(n^2)$	$\Theta(1)$	Ano	Ano
ShakerSort	$\Omega(n),\ \Theta(n^2)$	$\Theta(1)$	Ano	Ano
SelectionSort	$\Theta(n^2)$	$\Theta(1)$	Ne	Ne
InsertionSort	$\Omega(n),\ \Theta(n^2)$	$\Theta(1)$	Ano	Ano
MergeSort	$\Theta(n \cdot log\ n)$	$\Theta(n)$	Ano	Ne
HeapSort	$O(n \cdot log\ n)$	$\Theta(1)$	Ano	Ne
QuickSort	$\Omega(n \cdot log\ n), O(n^2)$	$O(n)$	Ne	Ano

2.4 InsertionSort

InsertionSort[1, 3], neboli řazení vkládáním, je stabilní datově citlivý řadící algoritmus, který pro seřazení vstupní posloupnosti nevyžaduje dodatečnou paměť. Ačkoliv InsertionSort dokáže v nejlepším případě seřadit vstupní posloupnost v lineárním čase, většinou jsou výsledky mnohem horší a často se blíží až ke kvadratické složitosti. InsertionSort se proto nehodí k řazení velkých objemů dat, kde je mnohonásobně pomalejší než například MergeSort. Kde naopak InsertionSort vyniká, jsou částečně seřazená pole nebo velmi malá pole obsahující maximálně desítky prvku. V takovém případě se může jednat o jeden z nejrychlejších algoritmů vůbec. Díky těmto vlastnostem je InsertionSort často používán jako pomocná řadící funkce u složitějších algoritmů.

InsertionSort, jak jíž plyne z jeho názvu, funguje na principu vkládání. Nejprve rozdělí vstupní pole na seřazenou a neseřazenou část. Budeme předpokládat, že seřazená část je vždy vlevo. Jelikož můžeme říci, že jednoprvková množina je přirozeně seřazená, vkládání začíná až od prvku druhého. Vkládání obecně probíhá tak, že si do pomocné proměnné uložíme nejlevější prvek z neseřazené části pole a porovnáváme ho s prvkem o pozici nižším. Dokud je vkládaný prvek menší než ten předchozí a zároveň jsme ještě nenarazili na začátek pole, tak příslušné hodnoty vyměníme a porovnávání probíhá znovu.

Algorithm 1 InsertionSort

```
 1: procedure InsertionSort(Array, Begin, End)
 2:     for (i ← Begin to End) do
 3:         j = i + 1
 4:         Temp = Array[j]
 5:         while  (j > Begin ∩ Temp < Array[j − 1])  do
 6:             Array[j] = Array[j − 1]
 7:             j = j − 1
 8:         end while
 9:         Array[j] = Temp
10:     end for
11: end procedure
```

Když je prvek na správné pozici, snížíme velikost neseřazené části o jedna a pokračujeme vkládáním dalšího prvku dokud není neseřazená část prázdná.

2.5 MergeSort

Tato kapitola vychází z [1, 4, 5]. Řadící algoritmus MergeSort vytvořil v roce 1945 John von Neumann. MergeSort je typickým příkladem algoritmu, který k řešení problému využívá metody „rozděl a panuj“. Tato metoda se skládá z následujících tří části:

1. „Divide“ - Rozděl řešení problému na dvě nebo více řešení stejných podproblémů na podmnožinách vstupních dat.
2. „Conquer“ - Řeš podproblémy rekurzivně, dokud nejsou dostatečně malé, aby je bylo možné vyřešit přímou metodou.
3. „Combine“ - Zkombinuj výsledky řešení podproblému do celkového řešení.

Algorithm 2 MergeSort

```
1: procedure MergeSort(A, B, Low, High)
2:     if ((Low < High)) then
3:         Half ← ⌊(Low + High)/2⌋;                 ▷ Divide
4:         MegreSort(A, B, Low, Half);              ▷ Conquer
5:         MegreSort(A, B, Half + 1, High);         ▷ Conquer
6:         Merge(A, B, Low, Half, High);            ▷ Combine
7:     end if
8: end procedure
```

Procedura MergeSort přijímá jako parametr čtyři argumenty. Je to pole A o velikosti $High - Low + 1$ obsahující vstupní posloupnost. B je stejně velké

pomocné pole, které slouží pro slévání dílčích podposloupností. *Low* a *High* jsou přirozená čísla.

Pokud je splněna podmínka $Low < High$, postupně se provedou kroky „Divide“, „Conquer“ a „Combine“. Na řádce 3 je vzorec pro vypočtení dělící hranice pro rozdělení vstupního pole na dvě poloviny. Pokud má pole sudou délku, budou oba díly stejně velké. Pokud je délka pole lichá, bude druhá část o jeden prvek delší (případně obráceně pokud použijeme horní celou část místo spodní).

Na řádcích 4 a 5 se volá rekurzivně procedura MergeSort, které se parametrem předají vstupní pole, indexy začátku a konce příslušných polí. Takto se pole rekurzivně dělí dokud nevznikne jednoprvkové pole. Vzhledem k tomu, že pole obsahující pouze jeden prvek, je určitě seřazené, tak je možné přejít k dalšímu kroku, kterým je slévání seřazených podposloupností.

Slévání dvou seřazených podposloupností probíhá tak, že se v každém kroku porovnají první prvky obou posloupností a vybere se ten s nižší hodnotou. Tento prvek vložíme do pomocného pole a z původního odstraníme. Tento postup opakujeme dokud se jedno z polí nevyprázdní, potom už stačí zbytek druhého pole překopírovat bez dalšího porovnávání.

Algorithm 3 Merge

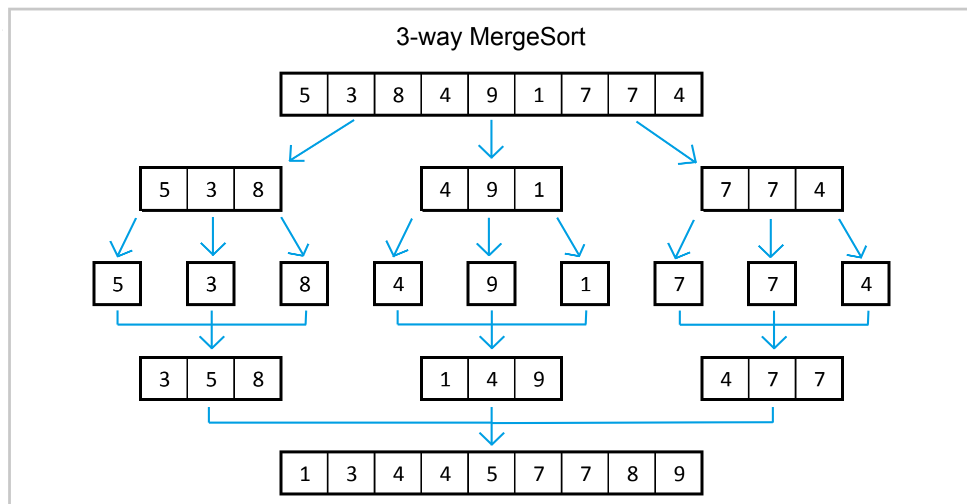
```
1: procedure Merge(A, B, Low, High)
2:     i1 ← Low; i2 ← Half + 1; j ← Low
3:     while ((i1 ≤ Half) ⋂ (i2 ≤ High)) do
4:         if (A[i1] ≤ A[i2]) then(B[j] ← A[i1]; i1 + +;)
5:         else(B[j] ← A[i2]; i2 + +;)
6:         end if
7:         j++;
8:     end while
9:     while (i1 ≤ Half) do
10:        B[j] ← A[i1]; i1 + +; j + +;
11:    end while
12:    while (i2 ≤ High) do
13:        B[j] ← A[i2]; i2 + +; j + +;
14:    end while
15:    A[Low : High] ← B[Low : High];
16: end procedure
```

2.6 K-way MergeSort

Tato kapitola vychází z informací uvedených v [6, 7]. Výše zmíněný algoritmus je někdy také nazýván jako 2-way MergeSort (2-cestný nebo také binární). Toto označení vyplývá z toho, že v každé úrovni rekurzivního volání se vstupní

pole rozdělí na dvě části. Sloučení pak probíhá pomocí dvou iterátorů, kdy se porovnávají nejmenší prvky obou polí.

Toto řešení lze zobecnit a místo dělení na dvě části se v každé úrovni rekurzivního volání vstupní pole rozdělí na částí K. Každá z těchto K částí je následně předána v parametru volání rekurzivní funkce. V tomto případě není vhodné dělení provádět až na pole o velikosti jednoho prvku. Lepším řešením je zvolení dělící hranice, od které se zbytek pole seřadí sekvenčním řadícím algoritmem. Po dokončení řady rekurzivních volání nastává fáze slučování, kdy místo dvou polí je třeba sloučit polí K. Hlavní rozdíl spočívá v tom, že místo dvou ukazatelů potřebujeme ukazatelů K a v každém kroku musíme najít nejmenší z K prvků.



Obrázek 2.1: Princip řazení algoritmu k-way MergeSort

Hledání nejmenšího prvku je možné provádět více způsoby. Základním postupem je najít minimum porovnáním všech nejmenších prvků K polí. Porovnání se provádí jednoduchým cyklem přes všechna pole, kdy je nutné si ukládat informaci o hodnotě minimálního prvku a poli, kterému tento prvek náleží. Jelikož je nutné v každém z N kroků nalézt minimum z K prvků lineárním způsobem, má tento způsob slučování časovou složitost $K * N$. Pokročilejším způsobem hledání minimálního prvku je pomocí binární haldy, díky které lze minimum najít v čase $N\ log\ K$. Implementace tohoto řešení je ale poměrně náročná a nenabízí téměř žádnou možnost paralelizace. Další strategií je postupné slučování dvou sousedních polí binárním způsobem, který byl popsán v minulé kapitole. Při každé iteraci se tak sníží počet polí na polovinu. Celkem se tedy provede $log\ K$ iterací a při každé z nich se iteruje přes N prvků. Sloučení tedy probíhá v čase $N\ log\ K$.

2.7 TimSort

TimSort[8] je hybridní řadící algoritmus využívající principy algoritmů MergeSort a InsertionSort. Pod vývojem algoritmu je podepsán Tim Peters, který TimSort v roce 2002 implementoval. TimSort je používán jako vnitřní řadící funkce jazyku Python a byl navržen s ohledem na mnoho druhů reálných dat. Funguje na principu hledání již seřazených částí, kterých využívá pro zrychlení celého řadícího procesu. Timsort je stabilní, datově citlivý řadící algoritmus, který dokáže seřadit vstupní data v nejlepším případě v lineárním čase, obecně však v čase $O(NlogN)$. Algoritmus pro řazení vyžaduje dodatečnou paměť o velikosti $O(N/2)$.

2.7.1 Run

TimSort iteruje přes vstupní sekvenci dat a hledá již seřazené bloky prvků. Tyto bloky budeme dále nazývat `run` (volně přeloženo jako běh). Každý run se nejdříve skládá z alespoň dvou prvků vstupního pole. Prvky v každém běhu mohou být seřazeny buď vzestupně nebo striktně sestupně, s tím, že sestupné sekvence budou následně transformovány do vzestupné podoby. Striktní sestupnost je vyžadována z důvodu udržení stability algoritmu, která by jinak byla při obracení posloupnosti porušena.

2.7.2 MinRun

Pokud přirozená sekvence uspořádaných hodnot v datech je kratší než prahová hodnota nazvaná $MinRun$, jsou zbývající prvky do daného běhu vkládány pomocí algoritmu InsertionSort. InstertionSort má sice velkou asymptotickou složitost, ale pro tak malý objem dat je to jeden z nejrychlejších způsobů řazení.

Vkládání hodnot začíná od prvku následujícímu tomu, kde skončila předcházející uspořádaná posloupnost a pokračuje dokud velikost běhu nedosáhne hodnoty $MinRun$. V okamžiku, kdy je minimální délka běhu splněna, jsou informace o jeho začátku a délce uloženy na zásobník a pokračuje se vytvářením dalšího běhu.

Pro výpočet hodnoty $MinRun$ mohou být použity různé metody. Základním pravidlem je, že pokud délka celé posloupnosti je menší než 64 prvků, je $MinRun$ zvolen právě na tuto hodnotu. To znamená, že celé vstupní pole je seřazeno v jediném běhu.

Autor uvádí, že z testování na náhodných datech vyplynulo, že optimální hodnota $MinRun$ se pohybuje v rozmezí 32 a 63 prvků. Pro ideální vyváženost následného slučovaní je důležité, aby celkový počet běhů byl roven nebo o něco nižší než mocnina dvou. To znamená, že hledáme číslo mezi 32 a 63 takové, aby velikost pole dělená hodnotou $MinRun$ byla menší nebo rovna mocnině dvou. Na první pohled se to může zdát obtížné, ale stačí si zobrazit délku

celého pole v binární podobě, vzít 6 nejméně významných bitů a 6. nejméně významný bit nastavit na hodnotu jedna. V programovacím jazyku `C++`, lze hodnotu vypočítat jako:

$$MinRun = ((Length \ \& \ 0x003F) \mid 0x0020)$$

2.7.3 Invariant

Po tom co je Run přidán na zásobník se nepokračuje přímo vytvářením dalšího běhu, jak je napsáno výše, ale ještě se zkontrolují podmínky, zda není třeba provést sloučení. Jelikož chceme sloučení provádět efektivně, musíme zajistit, aby se vždy slévaly přibližně stejně dlouhé běhy. Proto slévání neprovádíme ihned po každém vložení běhu na zásobník, ale snažíme se ho oddálit na co možná nejpozději. Sloučení by také nebylo vhodné provádět jednorázově po projití celého pole, kdy jsou všechny běhy uloženy na zásobníku, protože by to zvyšovalo paměťovou náročnost. Proto byl zvolen vhodný kompromis, díky kterému je sloučení provedeno za přesně stanovených podmínek, které jsou zároveň jednoduché na kontrolu. Tyto podmínky budeme nazývat invariant. Abychom invariant mohli ověřit, je nutné mít přístup k prvním třem prvkům na vrcholu zásobníku. Označme si tyto prvky jako A, B, C. Podmínky jsou definovány následovně:

- $A > B + C$
- $B > C$

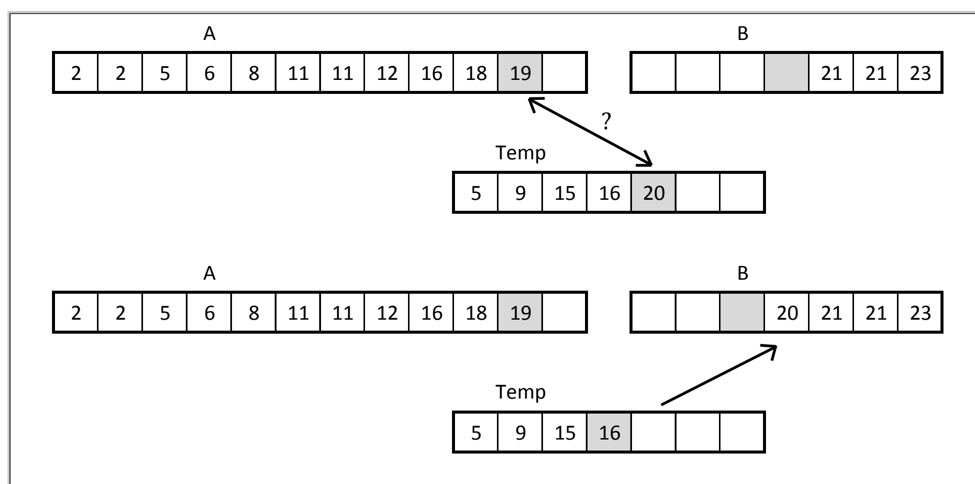
Pokud jsou obě podmínky po vložení nového běhu splněny, pokračuje se hledáním dalšího běhu. Pokud podmínky splněny nejsou, sloučíme Run označený jako B s menším z běhů A a C, zapíšeme informaci o nové délce do příslušného běhu a již nepotřebný běh ze zásobníku odstraníme. Vždy slučujeme Run B s jedním ze sousedních běhů, a to v pořadí, v jakém byly běhy vytvořeny ($C+B$ nebo $B+A$). Pokud by pořadí nebylo dodrženo, ohrozilo by to stabilitu řazení. Často se stává, že ani po sloučení dvou běhů invariant stále neplatí, proto se tento proces musí opakovat dokud nejsou obě podmínky splněny.

V okamžiku, kdy je na zásobník vložen poslední běh, je nutné provést sloučení všech běhů do jednoho, který bude obsahovat výsledná seřazená data. Slučování probíhá podobným způsobem jako během celého algoritmu. Vždy se slučuje běh B s menším z běhů A a C, s tím rozdílem, že se již nekontroluje invariant, ale slučování probíhá dokud nezbude pouze jediný běh.

2.7.4 Merge

Slučování dvou seřazených běhů se provádí stejným způsobem, jaký známe z algoritmu MergeSort. Překopírujeme prvky do pomocného pole a postupně porovnáváme nejmenší prvky z obou polí a menší z nich vložíme na správnou pozici zpět do původního pole.

TimSort zde přišel s vylepšením, díky kterému je možné sloučení provést s poloviční dodatečnou pamětí, a to se zachováním rychlosti slévání. Před každým sloučením porovnáme délky obou polí a vybereme z nich to s menším počtem prvků. Prvky menšího pole překopírujeme do pomocného pole o velikosti $N/2$, kde N je počet řazených prvků. Pokud je menší pole první v pořadí (vlevo), nastavíme pomocné ukazatele na nejmenší prvky obou polí a postupně inkrementujeme. Menší z porovnávaných prvků zapisujeme do výsledného pole od prvního indexu. Pokud je ovšem menší pole v pořadí až druhé (viz 2.2), je nutné provádět sloučení z pravé strany. To znamená nastavit pomocné ukazatele na poslední indexy polí A, B a $Temp$ a postupně je dekrementovat. Větší z porovnávaných prvků pole A a $Temp$ je překopírován na příslušný index do pole B.

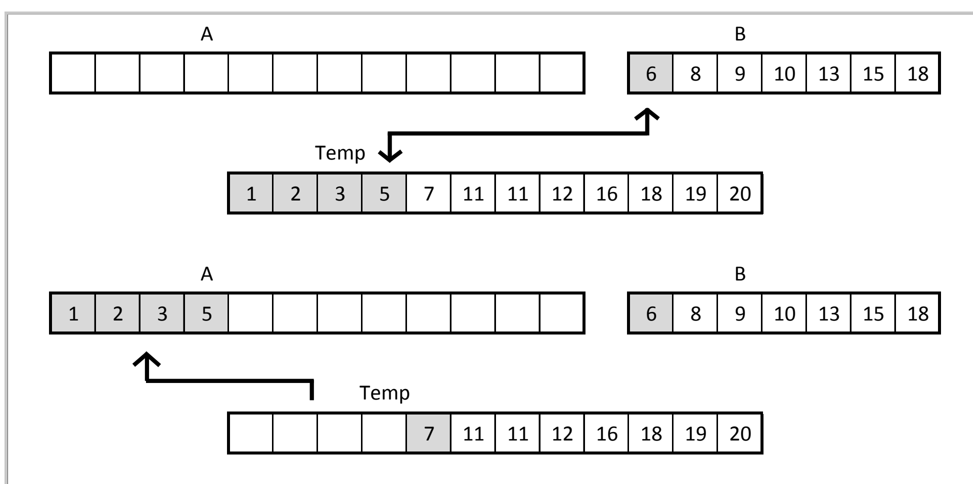


Obrázek 2.2: Princip efektivního slévání běhů algoritmu TimSort

Pokud se nad příkladem zamyslíme, zjistíme, že ke sloučení nám vlastně stačí správně umístit pouze prvky z pomocného pole $Temp$. Pokud se tomu tak stane, zbývá totiž správně umístit jen zbývající hodnoty pole A. Ale jelikož ukazatel indexu pozice na kterou se má vložit následující prvek je stejný jako pozice ukazatele dalšího vkládaného prvku, je manipulace s touto částí pole zbytečná, protože by se všechny tyto prvky kopírovaly na svou aktuální pozici. Tento princip slévání je podrobněji popsán v [5].

2.7.5 Galloping mode

Pokročilá veze TimSortu používá také takzvaný `Galloping mode`. Při slévání běhů je udržována hodnota počtu po sobě jdoucích prvků, které byly překopírovány z jednoho běhu. Pokud tato hodnota přesáhne mez označenou jako *min_gallop*, je aktivován Galloping mód. Tento mod funguje na principu vyhledávání indexu prvku v druhém slučovaném běhu, který je největším prvkem menším než nejmenší prvek v prvním běhu. Díky informaci o tomto indexu je možné zkopírovat všechny prvky až do tohoto indexu bez dalšího porovnávání s prvky prvního běhu.



Obrázek 2.3: Princip Galloping mode algoritmu TimSort

Tento index je hledán pomocí exponencálního vyhledávání[9], které funguje tak, že porovnává prvky na indexech druhých mocnin, dokud nenarazí na prvek, který je větší než náš hledaný prvek. Tento index bude představovat horní mez. Pokud horní mez vydělíme dvěma, získáme tak nejbližší nižší druhou mocninu u které víme, že index na tomto prvku je menší než hledaná hodnota. Tento index označíme jako spodní mez. Hledaný prvek se tedy nachází mezi dolní a horní hledanou mezí. Jelikož víme, že hodnoty v této části jsou seřazené, je možné využít binární vyhledávání pro nalezení konečné pozice hledaného prvku. Galloping mode ovšem nemusí být vždy efektivní. V nejhorším případě si může vyžádat až o 33 % více porovnání než klasické slévání vyžadující $m+n$ porovnání, kde m, n jsou délky jednotlivých běhů. Doporučená meze *min_gallop* je stanovena na hodnotu 7, kdy počet porovnání exponenciálního vyhledávání začíná být menší než pomocí lineárního procházení.

Kapitola 3

Optimalizační techniky

Mohlo by se zdát, že okamžikem, kdy je daný algoritmus úspěšně implementován a otestován, všechna práce končí. Opak je však pravdou - a to zejména, pokud chceme výkonnost dané implementace zvýšit na maximum. Existují totiž různé optimalizace a transformace zdrojového kódu, které sice nemají na asymptotickou hodnotu složitosti algoritmu žádný vliv, ale přesto v některých případech dokáží velmi ovlivnit dobu běhu daného algoritmu. Dále popisované optimalizační techniky vychází z [10, 11, 12]

3.1 Kompilátorové optimalizece

První možností optimalizace je použití různých přepínačů při kompilaci daného programu. Tyto přepínače se snaží zvýšit výkon výsledného kódu tím, že analyzují zdrojový kód a lépe tak naplánují provádění instrukcí. Jedním ze základních, je přepínač `-O3`, kde číslo označuje stupeň optimalizace. Tato optimalizace je zaměřena zejména na odhady počtu iterací jednotlivých cyklů a jejich rozbalovaní, také nabízí automatickou vektorizaci a další. Dále existuje celá řada přepínačů pro optimalizaci matematických výpočtů, nastavení různých instrukčních sad nebo cílové architektury, na které bude program spouštěn.

3.2 Metody transformace zdrojového kódu

3.2.1 Inline funkce

Při volání funkce je třeba do paměti uložit informace o hodnotách proměnných návratovou adresu funkce a podobně, což může při častém volání funkcí snižovat výkonnost celého programu. Tuto režii lze odstranit tím, že tělo dané funkce vložíme přímo na pozici volání této funkce do zdrojového kódu. Nevýhodou tohoto řešení je zvětšení délky zdrojového kódu a horší udržovatelnost. Toto vylepšení také nelze použít pro některé typy rekurzivních funkcí.

3.2.2 Rozbalování cyklů

Rozbalování cyklů je metoda transformace zdrojového kódu, která je založena na principu rozbalení těla cyklu několika iterací dopředu. Rozbalovaní probíhá tak, že se nejprve určí krok cyklu, to znamená kolik iterací původního cyklu bude uvnitř rozbaleného cyklu. O tento krok se bude zvyšovat hodnota proměnné, podle které cyklus iteruje. Následně se do těla cyklu zkopíruje tělo původního cyklu tolikrát, jaký byl zvolen krok a upraví se příslušné indexy. Pokud počet iterací není celočíselným násobkem kroku, je nutné přidat ještě jeden cyklus, který vykoná zbytek iterací klasickým způsobem.

Algorithm 4 Loop unrolling

```
 1: for (i = 0; i < N; i = i + 1) do                         ▷ before unroll
 2:     code(i);
 3: end for
 4:
 5: for (i = 0; i < N; i = i + step) do                      ▷ after unroll
 6:     code(i);
 7:     code(i + 1);
 8:     ⋮
 9:     code(i + step − 1);
10: end for
11: for (i = N − (N % step) + 1; i < N; i = i + 1) do
12:     code(i);
13: end for
```

Hlavní výhodou rozbalování cyklů je prodloužení těla cyklu, díky kterému je možné instrukce lépe naplánovat. Výhoda spočívá také ve snížení počtu iterací, z čehož plyne snížení počtu podmíněných skoků. Nevýhodou je výrazné prodloužení celého kódu, zejména pokud je stupeň rozbalení velký nebo tělo cyklu obsahuje větší množství instrukcí.

3.2.3 Struktura uložených dat

V programování je zvykem ukládat informace o vlastnostech nějakého prvku do struktury s příslušnými atributy. Pokud takových prvků potřebujeme uložit více, využijeme pole těchto struktur. Problém ovšem nastává, když potřebujeme pracovat se všemi prvky, ale při této práci využíváme jen některý z mnoha atributů daného prvku. V takovém případě dochází k neefektivnímu využívání skryté paměti, jelikož se do ní načítají i položky se kterými nepotřebujeme pracovat a tudíž je možné uložit méně prvků. Tento problém je možné vyřešit vytvořením struktury polí, kdy každá vlastnost bude uložena v odděleném poli a bude možné efektivně pracovat s atributy, které jsou právě potřeba.

3.3 Paralelizace

Frekvence procesorů se již několik let ustálila na podobných hodnotách a zvedá se jen velmi pomalu. Důvodem jsou fyzikální limity a velké zahřívání procesorů, které je obtížné chladit. Současný vývoj se tedy zaměřil na víceprocesorové systémy a snaží se zvýšit výkon paralelizací, kdy jsou výpočty rozdělené do několika nezávislých vláken. Abychom ovšem mohli program spustit na paralelním stroji, musíme ho nejdříve vhodně paralelizovat.

3.3.1 Amdahlův zákon

Pomocí Amdahlova zákona[13, 14] je možné vyjádřit předpokládané zrychlení systému poté, co je vylepšena některá jeho část. V našem případě nás bude zajímat, jakého zrychlení dosáhneme, pokud budeme některé výpočty programu provádět paralelně pomocí více vláken s lineárním zrychlením.

Označme si výkonnost programu před zlepšením jako P_{old} a výkonnost po zlepšení jako P_{new}. Hodnotu míry zlepšení výkonnosti označenou písmem S, která nám říká, kolikrát je vylepšený systém výkonnější než ten původní, vypočteme dle vzorce:

$$S = \frac{P_{new}}{P_{old}}$$

Dále označme původní dobu výpočtu zlepšované části úlohy jako $T_{p.old}$ a původní celkovou dobu výpočtu jako T_{old}. Potom procentuální podíl původní doby běhu zlepšované části F_p získáme jako:

$$F_p = \frac{T_{p.old}}{T_{old}}$$

Poslední hodnotu, kterou potřebujeme znát pro výpočet celkového zrychlení, je míra zrychlení vylepšované části úlohy S_p, kterou vypočítáme jako:

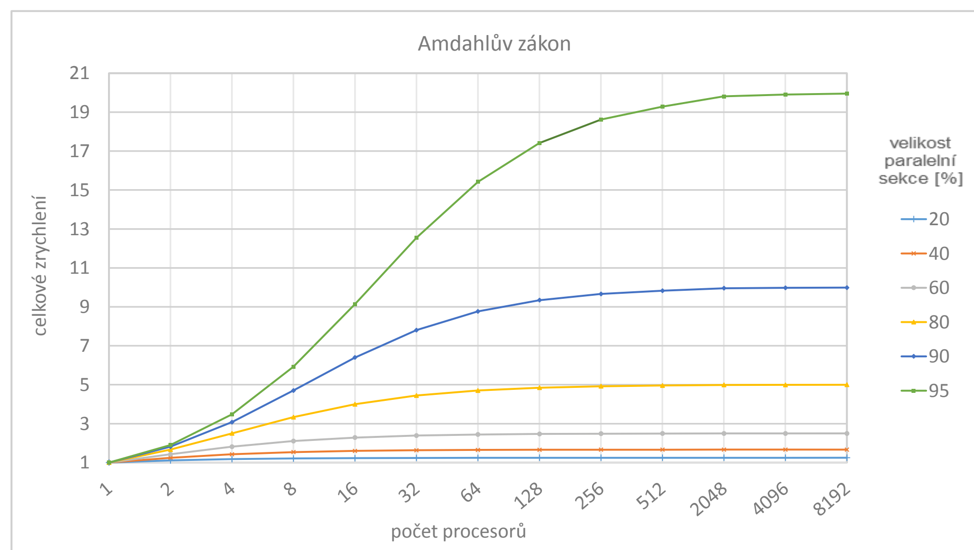
$$S_p = \frac{T_{p.old}}{T_{p.new}}$$

Celková oba výpočtu vylepšeného programu T_{new} se bude skládat z doby výpočtu části úlohy kterou nelze vylepšit $(1 - F_p) \cdot T_{old}$ a doby výpočtu vylepšené části úlohy $(F_p/S_p) \cdot T_{old}$. Dobu běhu vylepšeného programu tedy získáme vztahem:

$$T_{new} = T_{old} \cdot ((1 - F_p) + \frac{F_p}{S_p})$$

A celkové zrychlení S_o odpovídající danému vylepšení jako:

$$S_o = \frac{T_{old}}{T_{new}} = \frac{1}{(1 - F_p) + \frac{F_p}{S_p}}$$



Obrázek 3.1: Amdahlův zákon - závislost zrychlení na počtu procesorů

Z předchozích vzorců si můžeme všimnout, že celkové zrychlení závisí na velikosti vylepšované části vzhledem k celému programu a na koeficientu zrychlení zlepšované části samotné. Z toho plyne, že pokud chceme program efektivně paralelizovat, je nutné provést analýzu běhu programu, identifikovat výpočetně náročné části kódu a ty v co největším rozsahu paralelizovat. Míra zrychlení těchto částí by teoreticky měla být lineárně závislá na počtu vláken, která výpočet provádí. V praxi obvykle naměříme hodnoty o něco nižší. Zrychlení může záviset na mnoha okolnostech: na typu úlohy, hardwaru na kterém výpočet probíhá, efektivnosti využití cache pamětí, správné implementaci a tak dále.

3.3.2 OpenMP

Informace v této kapitole jsou čerpány ze zdrojů [15, 16, 17, 18]. OpenMP je soustava direktiv pro překladač a knihovních procedur pro paralelní programování. Jedná se o standard používaný pro paralelní programování nad sdílenou pamětí. OpenMP umožňuje vytváření vícevláknových programů v programovacích jazycích Fortran, C a C++.

3.3.2.1 Princip použití

Pokud chceme využívat možnosti OpenMP, je nutné do programu přidat direktivu `#include <omp.h>`. Dále je nutně v programu vyznačit bloky, které se mají provádět paralelně. Program je třeba zkompilovat s přepínači, díky kterým je zajištěno zpracování OpenMP direktiv (pro GCC je to volba `-fopenmp`).

Po puštění programu v procesu existuje vlákno, které je budeme označovat jako hlavní (master), které existuje po celou dobu provádění kódu. Na začátku paralelního bloku vytvoří hlavní vlákno pomocí join-fork mechanismu příslušný počet paralelních vláken. Vyznačený blok se provede paralelně a na konci bloku paralelní vlákna opět zaniknou. Z důvodu zmenšení režie procesu vytváření a zániku vláken je využíván thread pool.

3.3.2.2 Direktiva parallel

Jedná se o základní direktivu, která označuje začátek paralelního bloku. V místě programu, kde se objeví příkaz `#pragma~omp~parallel`, je vytvořen příslušný počet vláken. Počet vláken je možné ovlivnit několika způsoby. Mezi nejběžnější metody patří nastavení hodnoty `num_threads` nebo zavolání funkce `omp_set_num_threads(n)` (kde n označuje počet vláken). Vlákna jsou očíslována od 0 (pro hlavní vlákno), až po n-1 (pro poslední vytvořené vlákno). Po vytvoření vláken je kód paralelního bloku duplikován na všechna vlákna a začne se provádět. Na konci paralelního bloku je bariéra, kde jsou nově vzniklá vlákna ukončena a dále pokračuje pouze hlavní vlákno procesu. Pokud je z nějakého důvodu ukončeno některé z vláken uvnitř paralelního bloku, jsou ukončena i ostatní vlákna a tím i celý program.

3.3.2.3 Vlastnosti proměnných

Vlastnosti proměnných v paralelním bloku určující chování proměnných během paměťových operací:

- `shared` - proměnná bude sdílena mezi všemi vlákny
- `private` - každé vlákno bude mít nezávislou instanci této proměnné, na začátku je hodnota neinicializovaná
- `firstprivate` - stejné jako private, jen po vytvoření je inicializovaná na původní hodnotu
- `lastprivate` - stejné jako private, hodnota proměnné z poslední iterace bude překopírovaná do hlavního vlákna
- `default` - určuje jednu z předchozích možností, kterou budou mít defaultně všechny proměnné v paralelním bloku, pokud nebude uvedeno jinak

3.3.2.4 Direktiva for

Direktiva for se používá pro paralelizaci for-cyklů. Cyklus uvnitř paralelního bloku je rozdělen na nezávislé části, které se všemi vlákny postupně zpracovávají. Direktiva for má několik přepínačů, díky kterým lze průběh upravit:

- `nowait` - vlákna na konci paralelního bloku neprovádí bariéru
- `ordered` - pořadí operací je stejné jako při sekvenčním provádění
- `collapse` - slouží pro upřesnění u vnořených cyklů
- `schedule` - plánování rozdělení iterací mezi vlákna, tento parametr je možné dále specifikovat pomocí několika typů:
  - `static` - Iterace jsou rovnoměrně rozloženy mezi vlákna. Nevýhodou takového rozdělení je, že pokud složitost problému s každou iterací roste, mohou některá vlákna dlouho čekat na to poslední, které dostalo nejsložitější části problému. Tuto nevýhodu lze částečně odstranit číselným parametrem, který určuje maximální počet po sobě jdoucích iterací, které mohou být vláknu přiděleny.
  - `dynamic` - Iterace jsou po blocích přiřazována vláknům, která již dokončila svou předchozí práci. Velikost bloku lze stejně jako u volby `static` definovat parametrem.
  - `guided` - Každému vláknu je dynamicky přiděleno x iterací, kde x je počet neprovedených iterací děleno počtem vláken. Když vlákno dokončí svoji práci je přidělena další část podle stejného vzorce. Hodnotou v parametru je možné určit počet zbývajících iterací, které se již dále nedělí a jsou dokončeny jedním vláknem.
  - `auto` - plánování je ponecháno na kompilátoru a operačním systému.

3.3.2.5 Direktiva task

OpenMP je sice primárně zaměřena na datový paralelismus, ale obsahuje také podporu funkčního paralelismu.

- První mechanismus se nazývá `SECTIONS` a představuje podporu pro jednoduchý funkční paralelismus, ovšem za cenu složitější synchronizace.
- Druhým mechanismem je mechanismus `TASK`, který podporuje složitější funkční paralelismus a díky zapouzdření kódu i dat je vhodný pro rekurzivní funkce.

Direktiva `TASK` se musí nacházet uvnitř paralelního bloku, jinak se kód provádí sekvenčně. Direktiva `TASK` má poměrně velkou režii, proto je vhodné volání nějakým způsobem omezit. Tento problém řeší podmínka if, díky které je možné například v závislosti na hloubce zanoření rekurzivních volání `TASK` pozastavit a pokračovat sekvenčním způsobem. Pokud je podmínka splněna, tak je nový `TASK` vložen do *threadpool*, ze kterého si úlohy jednotlivá vlákna vyzvedávají k provádění.

3.4 Cache paměti

Kapitola zabývající se cache pamětmi vychází ze zdrojů [19, 20]. Cache paměť je hardwarová součást počítače, která slouží k vyrovnávání rychlostního rozdílu mezi rychlým procesorem a pomalou operační pamětí. Během provádění programu cache paměť obsahuje podmnožinu kódu a dat. V cache paměti jsou uložena nejčastěji nebo také naposledy používaná data. Ohled je brán zejména na lokalitu dat:

- `časová lokalita` - pokud procesor využívá nějakou položku v paměti, je vysoká pravděpodobnost, že ji bude používat znovu
- `prostorová lokalita` - pokud procesor pracuje s nějakou položkou v paměti, položky, které jsou umístěny v blízkosti s právě používanou položkou budou s vysokou pravděpodobností použity také

Současné moderní procesory mívají z pravidla víceúrovňovou cache pamět. Obvykle se setkáváme se třístupňovou cache pamětí, kde jednotlivé bloky jsou typicky označeny jako L1, L2 a L3. Obecně platí, že čím má blok menší číslo, tím je paměť menší a v paměťové hierarchii blíže k jádru procesoru.

3.4.1 Terminologie cache pamětí

- `cache hit` - případ, kdy jsou požadovaná data nalezena v cache paměti
- `hit rate` - poměr úspěšných přístupů k celkovému počtu přístupů do paměti
- `hit time` - vybavovací doba paměti při úspěšném čtení dat z cache paměti
- `cache miss` - požadovaná data nebyla nalezena v dané úrovni paměti
- `miss rate` - lze vyjádřit jako 1 - `hit rate`
- `miss penalty` - čas potřebný na získání dat z nižší úrovně paměti

3.4.2 Typy výpadků cache paměti

Výpadek paměti, neboli cache miss může mít několik příčin:

- `povinné výpadky` - první přístup k danému paměťovému bloku (spuštění počítače/programu)
- `kapacitní výpadky` - bloky ze skryté paměti jsou přepsány z kapacitních důvodů (paměť je zaplněná)
- `konfliktní výpadky` - 2 a více adres je mapováno na stejné místo v cache paměti (nízká asociativita, špatná nahrazovací strategie)

Kapitola **4**

Testování a měření

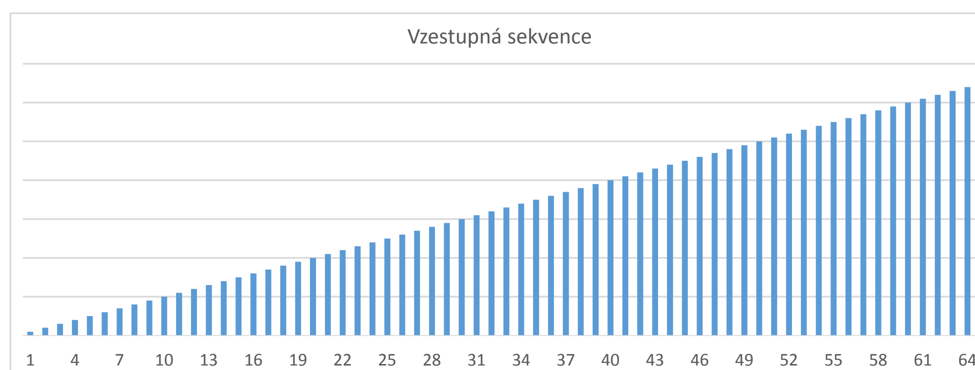
V této kapitole se seznámíme s generátorem vstupních dat a jednotlivými typy vstupních posloupností. Dále zde bude představen fakultní výpočetní server STAR, na kterém proběhne měření výkonnosti jednotlivých algoritmů.

4.1 Typy vstupních posloupností

Aby bylo možné jednotlivé algoritmy testovat a měřit jejich výkonnost, bylo třeba vytvořit generátor, který v závislosti na parametrech vytvoří vstupní sekvenci dat, kterou budou dané algoritmy řadit. Vzhledem k tomu, že se jedná o velké objemy dat, trvalo by načítání ze vstupního souboru velmi dlouho a proto bylo použito generování vstupních dat přímo za běhu programu. Při použití tohoto způsobu je ovšem nutné zajistit, aby generovaná posloupnost byla pro jednotlivé počty řazených prvků vždy stejná a následné porovnávání výkonnosti mohlo být objektivní. Jelikož jsou některé z testovaných algoritmů datově citlivé, bylo třeba vytvořit více typů vstupních posloupností, abychom mohli jednotlivé výsledky porovnávat a odhalit tak slabá místa testovaných algoritmů. Možností výběru bylo mnoho, ale vzhledem k počtu testovaných verzí algoritmů a časové náročnosti měření byli vybráni čtyři zástupci vstupních posloupností. Ve snaze bylo pokrýt co možná největší spektrum typů vstupních sekvencí. Proto se mezi kandidáty nachází seřazená, částečně seřazená i náhodná vstupní data. Pokud nebude uvedeno jinak, předpokládá se, že řazenými prvky jsou celá čísla datového typu `int`. Jelikož se tato práce zabývá efektivitou paralelního řazení a řeší problém časové náročnosti řazení velkých objemů dat, bylo by zbytečné algoritmy testovat pro krátké vstupní posloupnosti, které jsou seřazeny téměř okamžitě i méně efektivními řadícími algoritmy. Proto jsou pro měření použity delší sekvence o velikosti vstupních dat v rozmezí jednoho milionu až jedné miliardy řazených prvků.

4.1.1 Vzestupná sekvence

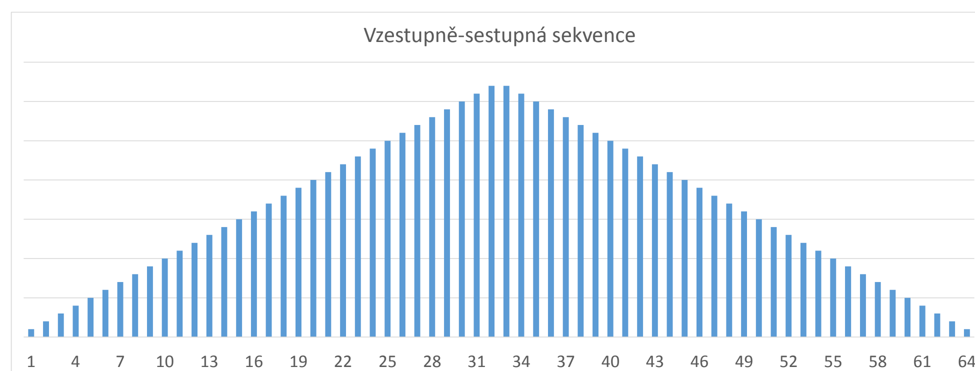
Vzestupná sekvence je jedním z nejjednodušších typů vstupních dat. Tato sekvence je již od samého začátku seřazená, tudíž velmi zvýhodňuje některé datově citlivé algoritmy, kterým je například dříve zmíněný InsertionSort, který takovou posloupnost zvládne seřadit v lineárním čase. Tato sekvence bude dále označována pořadovým číslem 1.



Obrázek 4.1: Graf znázorňující vstupní posloupnost typu 1

4.1.2 Vzestupně-sestupná sekvence

Další jednoduchá sekvence, která je ve své první polovině seřazena vzestupně a ve druhé polovině sestupně. Tato sekvence bude dále označována pořadovým číslem 2.

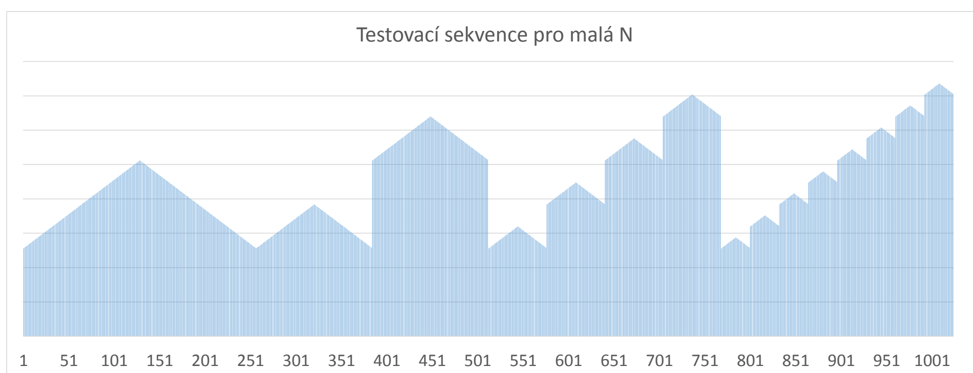


Obrázek 4.2: Graf znázorňující vstupní posloupnost typu 2

4.1.3 Testovací sekvence

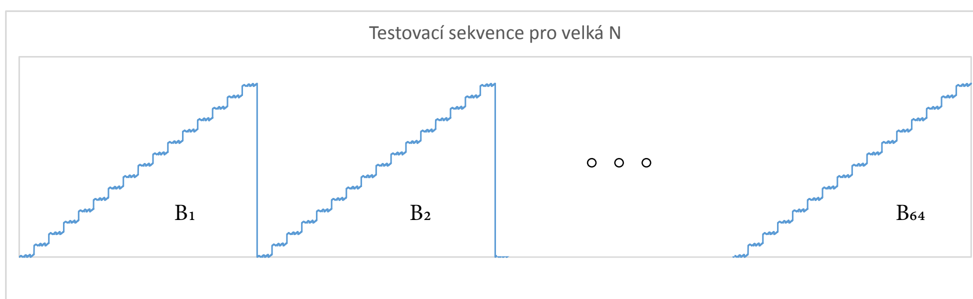
Tato sekvence vznikla za účelem vyplnění mezery mezi téměř seřazenými a náhodnými daty. Než tato sekvence získala svou konečnou podobu, prošla mnoha změnami. Základními stavebními kameny při tvorbě této sekvence jsou matematické operace násobení, dělení, odmocnina a modulo. Z důvodu ošetření dělení nulou, muselo být generování rozděleno do tří částí v závislosti na velikosti vstupu.

Pokud je velikost vstupu $N < 128$, vygeneruje se posloupnost typu 2 (vzestupně-sestupná sekvence). Pokud je velikost vstupu $N < 16384$, je výstupem sekvence znázorněná na obrázku 4.3.



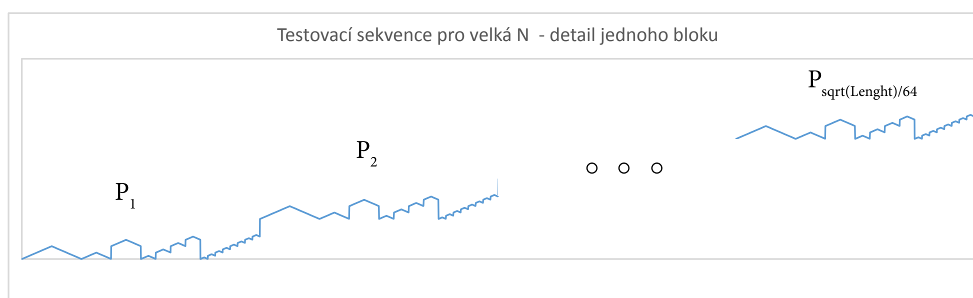
Obrázek 4.3: Graf znázorňující vstupní posloupnost typu 3 pro malá N

Pro počet řazených prvků 16384 a více, má posloupnost tvar jako na obrázku 4.4, kde je znázorněn její základní rys. Posloupnost ale nevypadá vždy stejně, protože její vzhled je závislý na konkrétním počtu generovaných prvků.



Obrázek 4.4: Graf znázorňující vstupní posloupnost typu 3 pro velká N

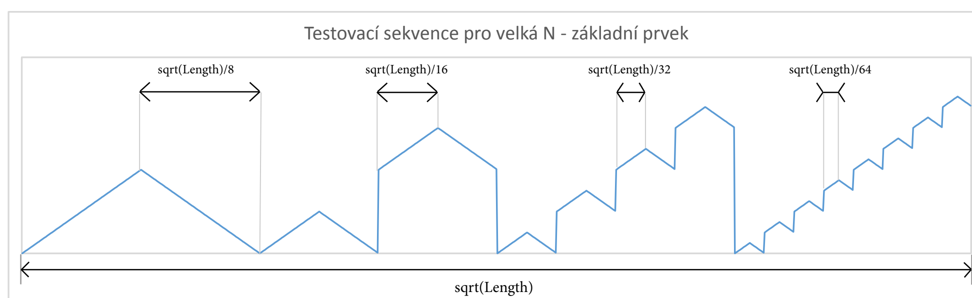
Aby se v datech neobjevovaly nežádoucí artefakty, je nutné, aby délka vstupu N byla druhou mocninou násobku čísla 128. Pokud je délka vstupu jiná, je třeba nalézt nejbližší nižší číslo splňující tuto podmínku. Rozdělme si tedy vstupní pole na hlavní část, jejíž délku označíme jako $Length$ a zbytek, který má délku $N - Length$. Hlavní část pole je rozdělena na 64 shodných bloků podle vzoru na obrázku 4.4. Na obrázku 4.5 vidíme detail jednoho bloku, kde každý z těchto 64 bloků je složen z $\sqrt{Length}/64$ základních prvků. Základní prvky v každém „trojúhelníku“ jsou od sebe výškově posunuty o hodnotu 1024. V celé posloupnosti je tedy $\sqrt{Length}$ základních bloků, které mají délku $\sqrt{Length}$.



Obrázek 4.5: Graf znázorňující vstupní posloupnost typu 3 - detail bloku

Základní blok 4.6 je navržen tak, aby měnil orientaci i délku přirozených běhů a simuloval tak částečně seřazená data. Velikost nejkratší a nejdelší seřazené části v každém bloku je možné vypočítat podle vztahů:

$$Min = \frac{\sqrt{Length}}{64} \quad a \quad Max = \frac{\sqrt{Length}}{8}$$

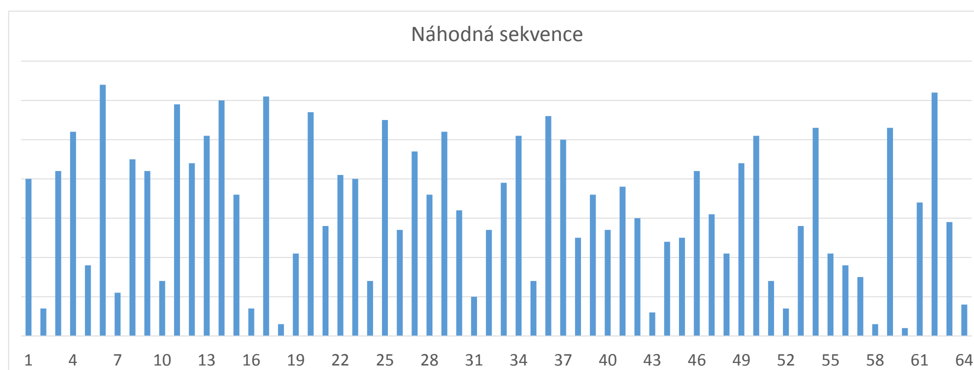


Obrázek 4.6: Graf znázorňující vstupní posloupnost typu 3 - základní prvek

Nakonec je naplněna zbývající část posloupnosti o délce $N - Length$. Do které je zkopírováno prvních $N - Length$ prvků. Tato sekvence bude dále označována pořadovým číslem 3.

4.1.4 Náhodná sekvence

Přestože v úvodu kapitoly bylo řečeno, že je třeba zachovat stejná vstupní data pro všechna měření, u náhodné posloupnosti musíme udělat výjimku. Tato sekvence bude dále označována pořadovým číslem 4.



Obrázek 4.7: Graf znázorňující vstupní posloupnost typu 4

4.2 Výpočetní server STAR

Server STAR [21] je výpočetní server Fakulty Informačních Technologií ČVUT v Praze. Server je využíván při výuce pro testování laboratorních úloh. Součástí svazku je centrální počítač, ke kterému je možné se přihlásit pomocí příkazu `ssh <username>@star.fit.cvut.cz`. Po přihlášení zde každý student nalezne jemu přidělený adresář, kde může spravovat své soubory. V tomto front-endu je dále možná kompilace úloh a jejich zařazení do fronty na některý z dostupných uzlů.

Server obsahuje tři hlavní výpočetní uzly nazvané jako gpu-01, gpu-02 a gpu-03 [22]. Třetí uzel slouží pouze pro testování úloh. Pro měření výkonnosti úloh v této práci byly použity uzly gpu-01 a gpu-02, které mají následující HW konfiguraci :

- CPU: 2ks 6core Xeon 2620 v2 @ 2.1Ghz
- 32 GB RAM

Kapitola 5

Implementace a dosažené výsledky

V této kapitole jsou blíže představeny implementované algoritmy a vyhodnoceny naměřené výsledky. Pokud nebude napsáno jinak, naměřené hodnoty odpovídají vstupní posloupnost typu 3 (testovací sekvence) a řazení celých čísel datového typu `int`. Uvedené grafy ve většině případů představují časovou závislost na počtu řazených prvků. Jelikož časová asymptotická složitost měřených algoritmů je rovna $O(N \: log \: N)$ a počty řazených prvků se zdvojnásobují, výsledná křivka přibližně odpovídá kvadratické funkci, což by způsobilo špatnou čitelnost průběhů menších vstupních polí. Z tohoto důvodu bylo nutné převést osu představující dobu běhu do logaritmického měřítka.

5.1 Pomocné funkce

V hlavičkovém souboru `Functions.h` jsou implementovány pomocné funkce využívané jednotlivými řadícími algoritmy.

5.1.1 Struktura Item

Struktura `Item` je využívána pro uložení krajních pozic částí pole a indexu sloužícího jako ukazatel na aktuální prvek. Struktura obsahuje tři atributy datového typu `int` a konstuktor, který slouží pro jejich inicializaci. Názvy atributů jsou m_Begin, m_End a $m_Pointer$.

5.1.2 Funkce isSorted()

Funkce `isSorted()` zkontroluje správnost seřazení pole mezi zadanými indexy. Pokud je požadovaná část seřazená, vrátí hodnotu true, jinak vrátí hodnotu false.

5.1.3 Funkce BinarySearch

Funkce `BinarySearch` je modifikací klasické vyhledávací funkce založené na binárním půlení. Tato funkce je upravena dle specifických potřeb hledání mezí pro vícecestné slučování. Funkce mezi zadanými mezemi pomocí binárního půlení vyhledá index na jehož pozici je nejvyšší hodnota menší než hledaný prvek. Pokud je hledaná hodnota prvním prvkem pole, bude návratová hodnota rovna -1. Funkce přijímá čtyři parametry: ukazatel na pole hodnot, dolní a horní mez a hledanou hodnotu.

5.1.4 Funkce FillArray

Funkce `FillArray` přijímá tři vstupní parametry: typ vstupní posloupnosti, ukazatel na datové pole a požadovaný počet prvků. Podle typu vstupní posloupnosti bude pole vyplněno příslušnou sekvencí dat. Jednotlivé typy posloupností byly představeny v předchozí kapitole.

5.2 TimSort

Implementace TimSortu vychází ze slovního popisu uvedeného v [8] a používá jeho základní principy řazení. Vzhledem k tomu, že tento popis nedefinuje přesné implementační konstrukce, bude se způsob implementace pravděpodobně do jisté míry lišit.

5.2.1 Třída RunList

Úkolem této třídy je vytváření a mazání jednotlivých běhů, kontrola platnosti invariantu a slučování běhů. Konstruktor `RunList` jako parametr přijímá ukazatel na pole hodnot a ukazatel na pomocné pole, které si uloží do svých atributů. V konstruktoru je dále hodnotou NULL inicializován poslední atribut, jímž je ukazatel na strukturu Run. Tento ukazatel je vstupním uzlem pro manipulaci s celým spojovým seznamem běhů.

```
class RunList
{
        RunList( int * Data, int * Temp );
        ~RunList( );
        void AddItem( int Begin, int Length );
        void CheckINV();
        void T_Merge();
        void MergeAll();
        Run * List;
        int * A_Data;
        int * A_Temp;
}
```

5.2.2 Struktura Run

Run je pomocná struktura jejíž instance reprezentuje jeden Run, který TimSort při svém běhu vytvoří. Konstruktor struktury `Run` přijímá tři parametry, jejichž hodnotami inicializuje příslušné atributy:

- `m_Begin` - index pozice prvního prvku běhu
- `m_Length` - délku celého běhu
- `m_Prev` - ukazatel na předchozí běh

Jednotlivé Runy tedy nejsou ukládány na zásobník v pravém slova smyslu, ale do jednosměrně zřetězeného spojového seznamu.

5.2.3 Funkce TimSort

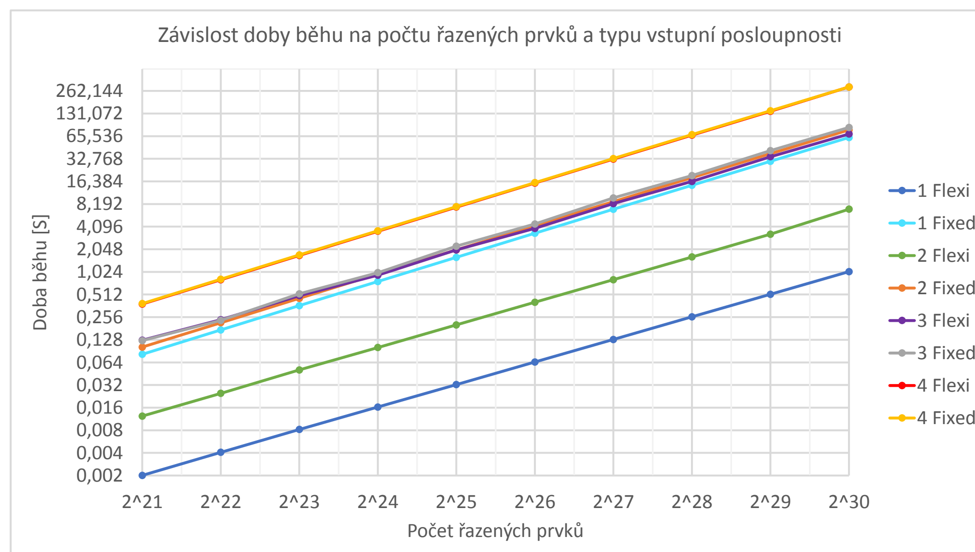
Funkce `TimSort` je vstupním bodem řadícího algoritmu. Funkce přijímá tři parametry, mezi které patří ukazatel na pole vstupních dat a indexy začátku a konce řazené části pole. Z těchto indexů je vypočítána délka řazené posloupnosti. Dále je třeba vypočítat hodnotu $MinRun$, kterou získáme vztahem:

$$MinRun = ((Length \ \& \ 0x003F) \mid 0x0020)$$

V dalším kroku již začíná hledání prvního běhu, které probíhá podle dříve popsaného postupu. V této implementaci je ovšem rozdíl v tom, že pokud je délka přirozeného běhu delší než hodnota $MinRun$, tak se běh uměle nezkracuje a nedělí se do více běhů. To má za následek mnohem rychlejší řazení některých typů vstupních posloupností. Nevýhodou tohoto řešení je pomalejší fáze slévání, kdy jednotlivé běhy mohou mít různé délky a slévání je tím pádem nevyvážené.

Porovnání obou těchto přístupů je zobrazeno na obrázku 5.1. Měření bylo provedeno pro všechny typy testovacích posloupností a pro různé velikosti vstupních dat. Křivky označené jako Fixed reprezentují verzi algoritmu s pevně danou velikostí hodnoty $MinRun$. Křivky s označením Flexi připouští i delší běhy než je zvolená hodnota.

Největší rozdíl je patrný u posloupností typu 1 a 2, tedy u téměř seřazených vstupů, kdy verze Flexi vytvoří mnohem méně běhů než verze Fixed. U náhodných dat (posloupnost 4), kde se očekávají jen velmi malé běhy a většinu práce zde odvede InsertionSort, jsou výsledky téměř stejné. Zde se také projevil důvod proč nebyla u verze Flexi zvolena konstantní základní velikost $MinRun$, ale je vypočítávána dle původního vzorce i přes to, že v některých případech může být délka běhu větší. Pokud jsou v posloupnosti dlouhé běhy, nevyváženost slučování se vykompenzuje menším počtem běhů. Pokud se v posloupnosti objevují pouze krátké běhy, tak velikost běhu málokdy přesáhne hodnotu MinRun a sloučení je tak velmi vyvážené díky tomu, že počet běhů se blíží druhé mocnině.



Obrázek 5.1: TimSort - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti (porovnání metod volby MinRun)

V případě, že je nalezen běh seřazený v opačném pořadí, je tato část nejdříve otočena pomocí jednoduchého `while` cyklu a dvou ukazatelů postupujících proti sobě. Teprve až po otočení posloupnosti probíhá doplnění na velikost $MinRun$. Když nový Run splňuje požadované podmínky, je zavolána metoda `AddItem`.

5.2.4 Metoda RunList::AddItem

Metoda `AddItem` s parametry *Begin* a *Length* vytvoří novou instanci struktury `Run`, jejíž ukazatel se uloží do atributu *List*. *List* tedy vždy ukazuje na poslední přidaný běh. Nakonec je zavolána metoda `CheckINV`.

```
void AddItem( int Begin, int Length )
{
        List = new Run( Begin, Length, List );
        CheckINV();
}
```

5.2.5 Metoda RunList::CheckINV

Tato metoda zkontroluje platnost invariantu a zařídí případné sloučení problémových běhů. Invariant se kontroluje tak dlouho, dokud není jeho platnost splněna. Aby bylo možné invariant zkontrolovat, vytvoříme si pro přehlednost tři proměnné typu ukazatel na `Run`, které nazveme A, B, C. Nyní C inicializujeme hodnotou $List$, B hodnotou $C \to m_Prev$ a A hodnotou $B \to m_Prev$. Zkontrolujeme následující pravidla:

- $A \to m_Length > B \to m_Length + C \to m_Length$
- $B \to m_Length > C \to m_Length$

Pokud jsou podmínky splněny, je zavolán `return` a pokračuje se hledáním dalších běhů. Pokud podmínky splněny nejsou. Je nalezen kratší z běhů A a C, který se následně sloučí s během B v metodě `Tim_Merge`. Důležité je pořadí sloučení. Jako první se vždy do této funkce musí předat běh vytvořený dříve. V našem případě má přednost A pak B a nakonec C. Při nedodržení tohoto pořadí by mohlo dojít k porušení stability řazení.

5.2.6 Metoda RunList::Tim_Merge

Úkolem metody `Tim_Merge` je sloučení dvou běhu. Sloučení probíhá tak, že se vybere kratší běh a ten se zkopíruje do pomocného pole A_Temp. Pokud je kratší pole první v pořadí (levé), probíhá slučování z levé strany od nejmenších prvků. Pokud je menší z polí to druhé (pravé), probíhá sloučení v pravé strany od největších prvků. Tímto způsobem je docíleno toho, že pro sloučení stačí pouze N/2 přídavné paměti při zachování původní rychlosti sloučení.

5.2.7 Metoda RunList::MergeAll

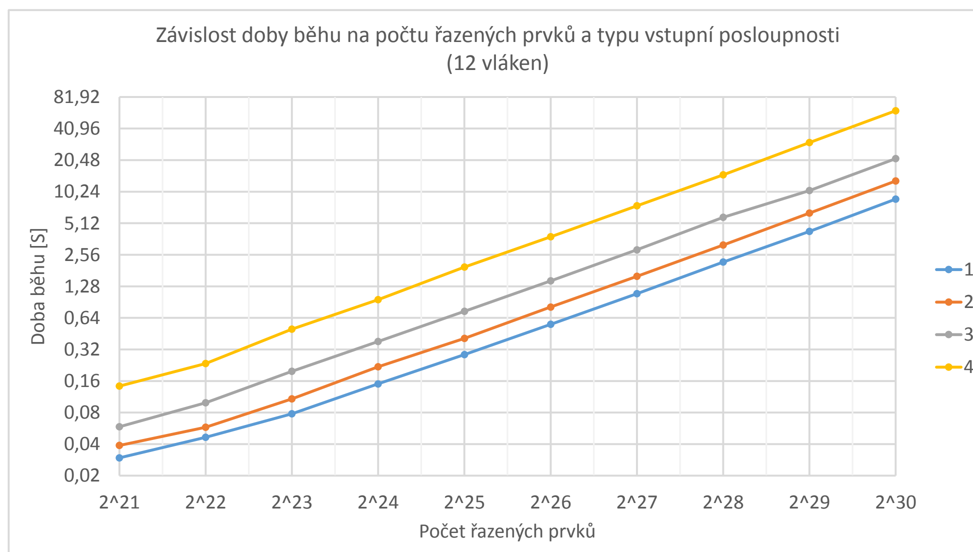
Metoda `MergeAll` je volána po tom, co hlavní část řadícího algoritmu dojde při hledání běhů na konec vstupního pole a na zásobník přidá poslední běh. V tento moment je třeba sloučit všechny zbývající běhy do jednoho běhu velikosti N. Sloučení probíhá podobně jako je tomu u metody `CheckINV` s tím rozdílem, že je vynechána kontrola invariantu a metoda `Tim_Merge` se volá dokud existuje více než jeden běh ve spojovém seznamu.

5.2.8 Paralelizace

Vzhledem k tomu, že TimSort sám o sobě není příliš známý algoritmus, o možnostech jeho paralelizace je možné se toho dozvědět ještě méně. Z dostupných zdrojů nebylo zjištěno nic podstatného, co by autorovi této práce mohlo při řešení paralelizace TimSortu pomoci a tak se pokusil vydat vlastní cestou.

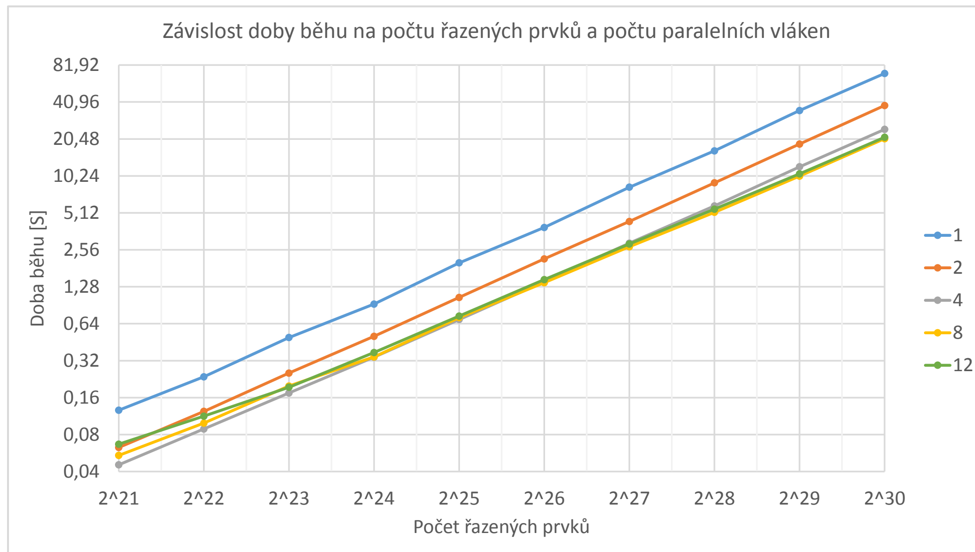
Po prostudování sekvenční implementace algoritmu se jako jediný efektivní způsob paralelizace jevilo rozdělení vstupní sekvence do K částí, které

budou paralelně seřazeny klasickým TimSortem a následně sloučeny do jedné sekvence. Výsledné slučování používá prvky ze samotného TimSortu. Z jednotlivých seřazených části jsou vytvořeny běhy, které se uloží do spojového seznamu a jsou slučovány stejným způsobem jako je tomu u klasického TimSortu. Tedy v pořadí druhý běh na vrcholu zásobníku je sloučen s menším z běhů jedna a tři. Slučování se opakuje dokud v zásobníku nezbude jediný běh.



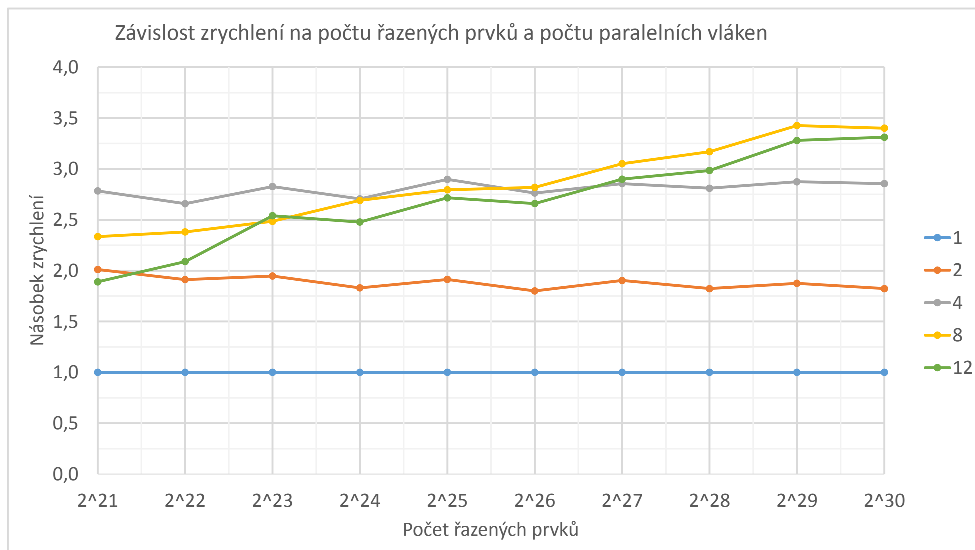
Obrázek 5.2: Paralelní TimSort - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti

Paralelní TimSort byl otestován na serveru STAR, kde bylo také naměřeno několik hodnot, které jsou vyneseny v následujících grafech. V grafu 5.2 je znázorněn průběh měření paralelního TimSortu v závislosti na typu vstupních sekvencí. Měření bylo provedeno s dvanácti paralelními vlákny a pro různý počet řazených prvků. Pokud hodnoty porovnáme se sekvenční verzí, jejíž výsledky jsou vyneseny v grafu 5.1, můžeme si všimnout zhoršení výkonnosti u vstupních posloupností typu 1 a 2. Zhoršení je způsobeno slučováním výsledných sekvencí z jednotlivých částí vstupního pole. U sekvenční verze byl totiž u posloupnosti prvního typu vytvořen pouze jediný běh, který byl rovnou výsledkem algoritmu. U paralelní verze je i již seřazená vstupní posloupnost rozdělena do K částí a následně sloučena zpět dohromady, což má za následek zmíněné zhoršení výkonnosti. U posloupností typu 3 a 4, které neobsahují tak dlouhé přirozené běhy, dochází k tvorbě několikanásobně většího počtu běhů, tudíž fáze TimSort trvá déle a konečné sloučení výsledný čas tolik neovlivní. Výsledkem je průměrně asi tří násobné zrychlení.



Obrázek 5.3: Paralelní TimSort - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken

Na obrázku 5.3 je vynesen graf doby běhu řadícího algoritmu v závislosti na počtu řazených prvků a počtu paralelních vláken. Hodnoty jsou měřeny pro 3. typ vstupní posloupnosti.



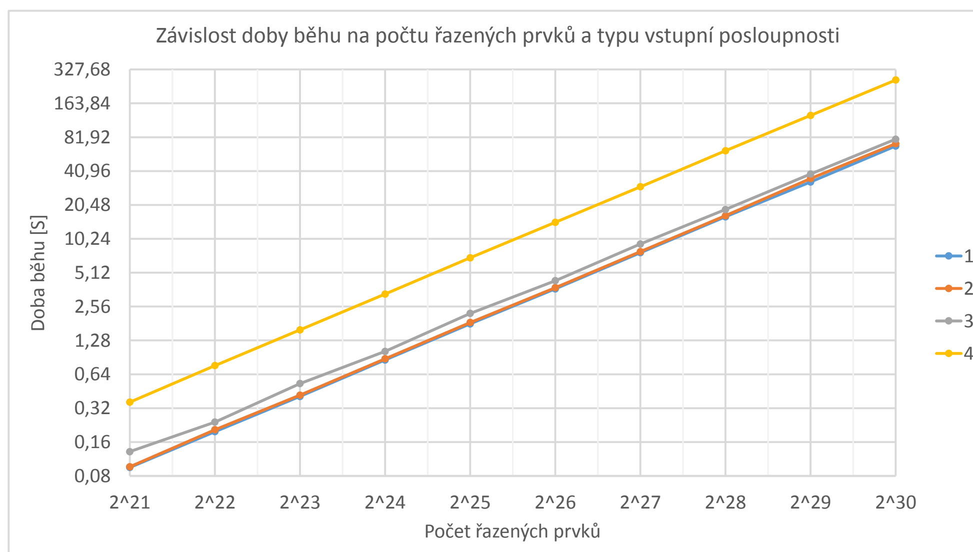
Obrázek 5.4: Paralelní TimSort - závislost zrychlení na počtu řazených prvků a počtu paralelních vláken

Posledním obrázkem 5.4 je graf znázorňující násobek zrychlení v závislosti na počtu paralelních vláken. Zde si můžeme všimnout, že pro dvě paralelní vlákna bylo zrychlení téměř lineární. Při použití čtyř vláken bylo zrychlení už jen tří násobné a stejně tak i pro osm a dvanáct vláken. Stagnace zrychlení vychází z Amdahlova zákona, ze kterého víme, že celková velikost zrychlení závisí na poměru zrychlované části vůči celému programu. V tomto případě bylo paralelizováno pouze řazení samotným TimSortem a konečné sloučení již probíhalo sériově. Dále si můžeme všimnout, že efektivita většího počtu vláken se začala projevovat až s větším počtem řazených prvků a pro menší vstupy způsobila složitější synchronizace horší výsledky než pro menší počet vláken.

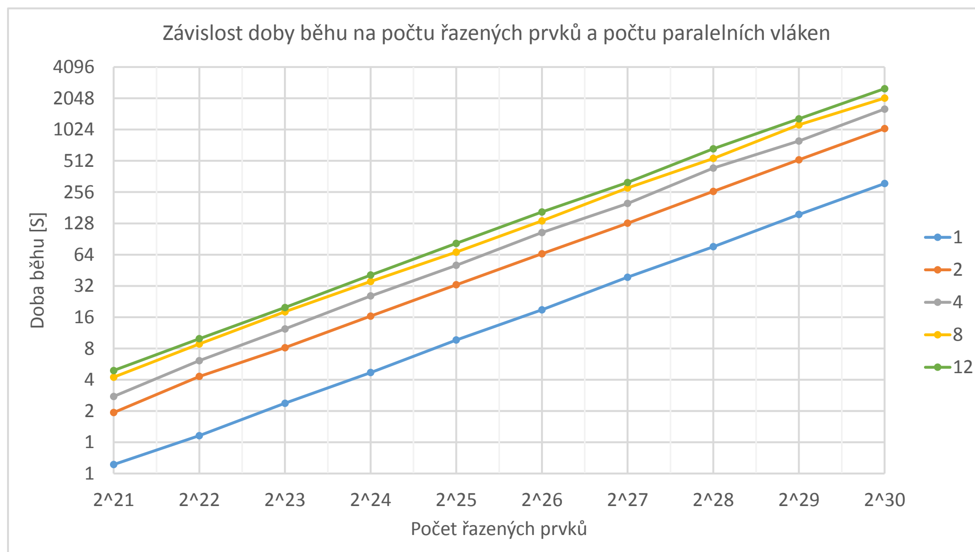
5.3 MergeSort

Tato implementace představuje klasickou verzi algoritmu MergeSortu popsanou v dřívější kapitole. Vstupní pole je rekurzivně děleno na poloviny. Sekvence rekurzivních volání se zastaví v okamžiku, kdy je velikost pole v dané větvi rovna jedné. Následně pokračuje slučování seřazených posloupností zpět, až do velikosti původního pole. Výsledky měření pro různé počty řazených prvků a různé vstupní posloupnosti jsou zobrazeny na obrázku 5.5.

Paralelizace tohoto algoritmu spočívá v rozdělení rekurzivních volání do OMP Tasků, které jsou vkládány do threadpoolu a zpracovávány dostupnými vlákny. Před následným sloučením je nutné vytvořit bariéru, aby bylo zajištěno, že sloučení bude probíhat na již setříděných polích.



Obrázek 5.5: MergeSort - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti



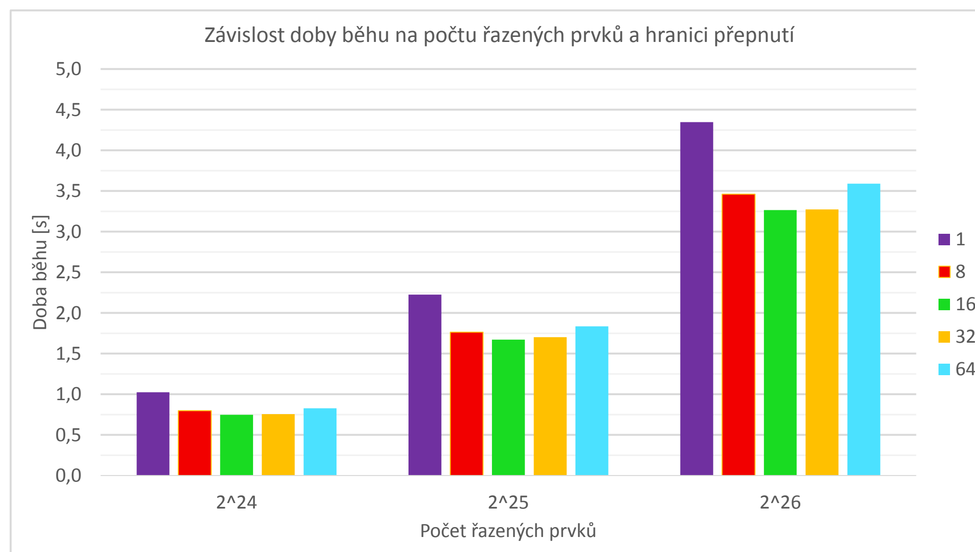
Obrázek 5.6: Paralelní MergeSort - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken

Jak můžeme vidět v grafu 5.6, tak toto přímočaré řešení nefunguje. Výsledkem je rapidní zhoršení s každým dalším paralelním vláknem. Důvodem je zřejmě falešné sdílení. Falešné sdílení je jev související s cache pamětí, kdy se snaží více vláken zapisovat do jednoho bloku v cache paměti. Při každém takovém přístupu musí být daný blok ostatním vláknům zneplatněn nebo aktualizován. Tento problém nastává, když jsou rekurzivní volání nejvíce zanořená a pracují s poli o velikosti jednotek prvků.

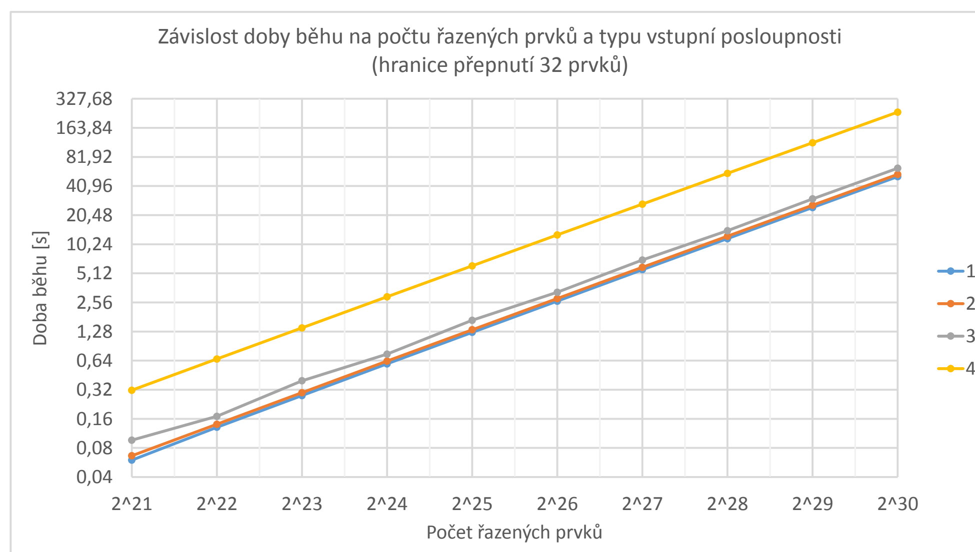
5.4 MergeSort s hranicí přepnutí

Aby se zabránilo falešnému sdílení, je třeba přidat dělící hranici přepnutí, od které bude řazení probíhat jiným způsobem. Jako pomocný řadící algoritmus byl vybrán InsertionSort, který je velmi efektivní pro malé objemy dat (max. desítky prvků). Implementačně tato změna spočívá pouze v přidání podmínky do těla funkce `t_MergeSort`, která zkontroluje délku pole. Pokud platí, že $End - Begin + 1 < Threshold$ je zavolána pomocná funkce `InsertionSort`. Pokud podmínka splněná není, program pokračuje klasický rozdělením pole na další dvě poloviny.

Před tím, než bude možné změřit výkonnost vylepšené verze algoritmu, je třeba najít optimální hranici přepnutí, která bude mít nejlepší výsledky. Naměřené hodnoty pro jednotlivé hranice můžeme vidět na obrázku 5.7. Z grafu je patrné, že nejlepších výsledků bylo dosaženo s hraniční hodnotou přepnutí 16 a 32 prvků nezávisle na velikosti vstupních dat.



Obrázek 5.7: MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a hranici přepnutí



Obrázek 5.8: MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti

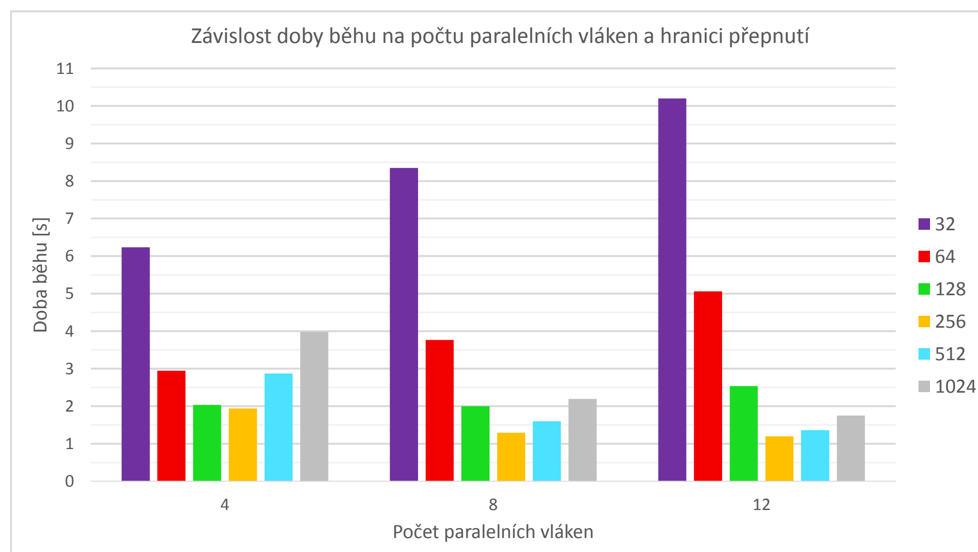
Jako optimální tedy zvolíme hodnotu 32 a proměříme všechny typy vstupních posloupností a různé počty řazených prvků. Výsledky měření jsou na

obrázku 5.8. Pokud porovnáme dosažené časy pro velikost vstupu 2^{30} prvků s klasickým MergeSortem, zjistíme, že došlo ke zvýšení výkonnosti o 10 až 30 % v závislosti na typu vstupní posloupnosti. Největší navýšení výkonu bylo dosaženo u vstupní posloupnosti typu 1, kdy se výsledný čas snížil z původních 68,3 vteřin na hodnotu 51,2 vteřin. Nejmenší nárůst výkonu byl zaznamenám u náhodné posloupnosti, kde není InsertionSort tolik efektivní.

5.4.1 Paralelizace

Paralelizace algoritmu MergeSort s hranicí přepnutí probíhá stejným způsobem jako tomu bylo u klasického MergeSortu. Tedy pomocí tvorby OMP tasků pro jednotlivé dělící větve. Jelikož řazení probíhá pomocí více vláken, budeme předpokládat zvýšení nároků na využívání cache pamětí, které se projeví zvětšením optimální hranice přepnutí oproti sekvenční verzi. Tento předpoklad si proto stejně jako v předchozím případě ověříme měřením pro různé velikosti hraničních hodnot a nově přidáme i různé počty vláken.

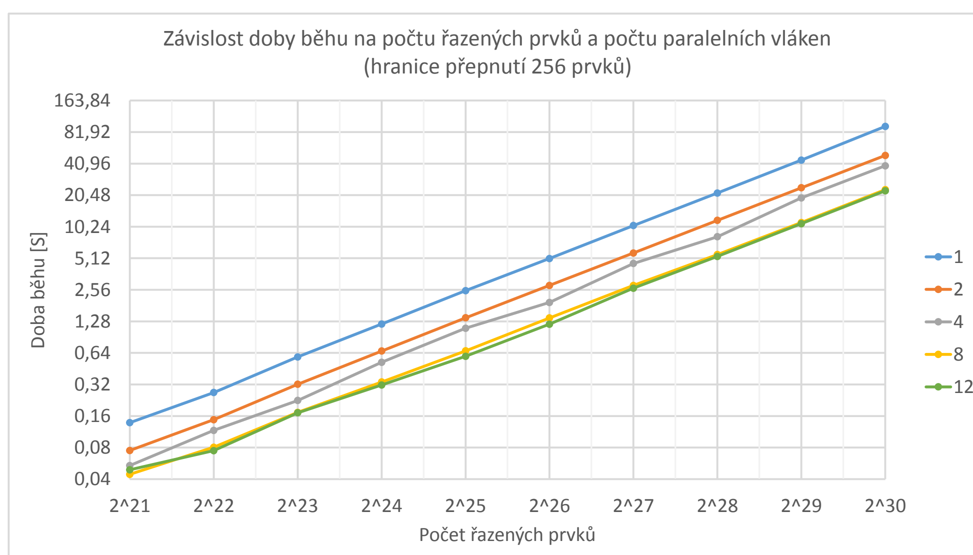
Z grafu 5.9 je zřejmé, že se náš předpoklad potvrdil a dělící hranice 32 prvků je pro paralelní verzi algoritmu naprosto nevyhovující. Pokud totiž hranici zvolíme 256 prvků, získáme tří násobné zrychlení pro 4 paralelní vlákna a až desetinásobné zrychlení pro 12 paralelních vláken než by tomu bylo s hraniční hodnotou 32 prvků jako u sekvenční verze algoritmu. V grafu si dále můžeme všimnout, že lepším řešením je volit spíše větší hranice než ty menší, se kterými je doba řazení výrazně pomalejší.



Obrázek 5.9: Paralelní MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a hranici přepnutí

Když už známe hraniční hodnotu pro paralelní verzi algoritmu, můžeme změřit podrobnější průběhy a zjistit nejlepší dosažené výsledky a také to, jak počet vláken ovlivňuje celkové zrychlení. Měření provedeme pro testovací posloupnost typu 3.

Výsledky měření paralelního MergeSortu s hranicí přepnutí 5.10 jsou mnohem pozitivnější než tomu bylo u paralelizace samotného MergeSortu 5.6, která měla velmi negativní efekt na dobu řadícího procesu.



Obrázek 5.10: Paralelní MergeSort s hranicí přepnutí - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken

Tabulka 5.1: MergeSort - porovnání výsledků různých verzí algoritmu

Algoritmus	2^{27}	2^{28}	2^{29}	2^{30}
MergeSort 1 serial	9,216	18,651	38,520	78,668
MergeSort 2 serial	7,031	14,124	30,166	62,599
MergeSort 2 parallel	2,653	5,328	10,933	22,559

Paralelní MergeSort s hranicí přepnutí sice dospěl ke zrychlení oproti sekvenční verzi, ovšem zrychlení je při využití dvanácti vláken maximálně čtyřnásobné, což je stále nedostatečný výsledek. Při podrobnější analýze průběhu algoritmu bylo zjištěno, že výpočetně náročnější částí je následné slučování, které probíhá stále sériově, proto se na jeho optimalizaci v další části zaměříme.

5.5 K-way MergeSort I.

Při snaze o další optimalizaci algoritmu MergeSort budeme vycházet z předchozí verze algoritmu s hranicí přepnutí na sekvenční řadící algoritmus. Optimalizaci rozdělíme na dvě části. V této částí bude probíhat úprava algoritmu z dvoucestného MergeSortu na K-cestný a vytvoření jeho paralelního řešení. Ve druhé části se zaměříme na hlavní kámen úrazu a pokusíme se optimalizovat fázi samotného sloučení.

5.5.1 K-array

Prvním krokem je rozdělení vstupního pole do K částí. Rozdělení provedeme tak, že si vypočteme délku vstupního pole pomocí vztahu $End - Begin + 1$. Když tuto délku známe, vydělíme jí počtem částí K, tak zjistíme délku prvních $K - 1$ částí. Tuto hodnotu si označíme jako $PartLen$. Meze prvních $K - 1$ částí vypočítáme pomocí for cyklu, kdy v každé iteraci podle i od 0 do $K - 1$ vypočítáme začátek $i - té$ části jako $Begin + i * PartLen$ a konec jako $Begin + (i + 1) * PartLen - 1$. Jelikož podíl $(End - Begin + 1)/K$ nemusí být vždy celočíselný, musíme délku poslední části navýšit o zbytek po dělení. Respektive nebudeme konec poslední části vypočítávat dle vzorce, ale nastavíme ho přímo na hodnotu End. Z tohoto důvodu je výpočet poslední části vyjmut z těla cyklu a vykonává se samostatně.

5.5.2 Rekurzivní volání

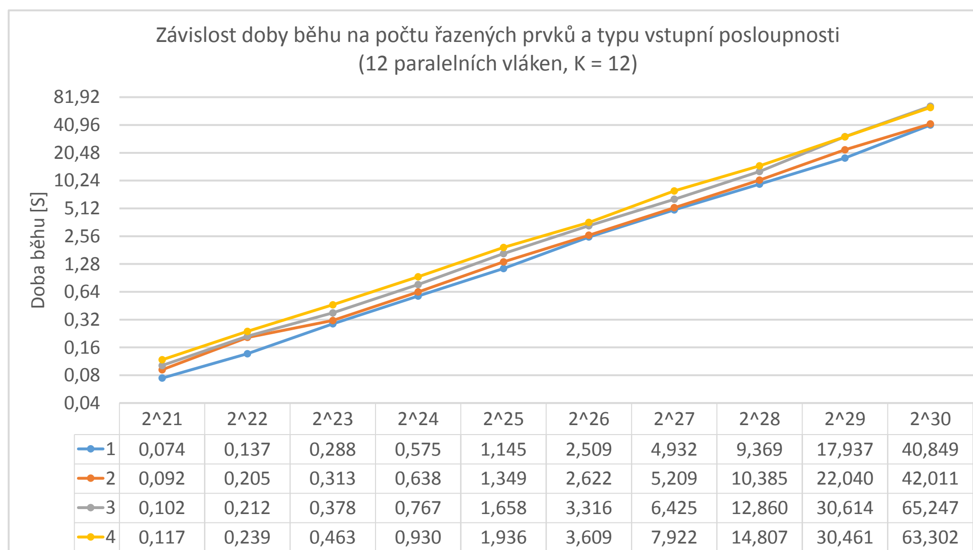
Po vypočítání a uložení příslušných mezí je možné pokračovat voláním rekurzivních funkcí pro každé z K polí. Z předchozí zkušenosti víme, že pokud rekurzivní dělení necháme dojít příliš hluboko, může to způsobit velmi negativní dopad na efektivitu algoritmu. Z tohoto důvodu opět využijeme dělící hranici přepnutí, která se v minulé verzi osvědčila. Dělící hranici ponecháme na hodnotě 256 prvků, která se v předchozích měřeních jevila jako optimální. Pokud bude velikost pole menší než tato hodnota, použijeme pro seřazení těchto částí řadící algoritmus InsertionSort.

5.5.3 Merge

Po zanoření rekurzivních volání do maximální hloubky $log_K N$ nastává zpětná fáze slučování. Oproti klasickému MergeSortu, kdy probíhalo sloučení dvou podpolí, je v tomto případě třeba sloučit polí K, do pole délky odpovídající součtu délek těchto K polí.

Pro sloučení potřebujeme K ukazatelů, kterými budeme v každém poli označovat aktuálně nejmenší prvek. Hledání nejmenšího prvku bude probíhat lineárním procházením všech polí. Nejmenší hodnota bude zkopírována z pomocného pole na výslednou pozici a ukazatel v poli obsahující tuto nejmenší hodnotu bude posunut o jednu pozici doprava. Při každém posunutí tohoto

ukazatele je třeba provést kontrolu, zda se ukazatel stále nachází v mezích krajních hodnot daného pole. V opačném případě dané pole odstraníme a pokračujeme slučováním $K-1$ polí a tak dále, dokud každý ukazatel nedosáhne své horní meze. Po úspěšném sloučení proběhne návrat z tohoto rekurzivního volání a sloučení se provádí opět o úroveň výše, dokud nedosáhne svého vrcholu.



	2^21	2^22	2^23	2^24	2^25	2^26	2^27	2^28	2^29	2^30
1	0,074	0,137	0,288	0,575	1,145	2,509	4,932	9,369	17,937	40,849
2	0,092	0,205	0,313	0,638	1,349	2,622	5,209	10,385	22,040	42,011
3	0,102	0,212	0,378	0,767	1,658	3,316	6,425	12,860	30,614	65,247
4	0,117	0,239	0,463	0,930	1,936	3,609	7,922	14,807	30,461	63,302

Obrázek 5.11: Paralelní K-way MergeSort I. - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti

5.5.4 Paralelizace

Paralelizace tohoto řešení spočívá v paralelním volání rekurzivních funkcí pro každou z K částí. Paralelně také probíhá kopírování hodnot v dané větvi do pomocného pole, které je v algoritmu zařazeno před výsledným sloučením.

Výsledky tohoto řešení nejsou dle předpokladů nijak převratné. Naměřené hodnoty pro všechny typy vstupních posloupností a různé počty řazených prvků jsou zobrazeny v grafu 5.11. Počet paralelních vláken a stupeň dělení K byl pro všechna měření shodný, a to s hodnotou 12. Jelikož se jedná pouze o mezikrok k finální podobě algoritmu, nebudeme se výsledky dále zabývat a přejdeme k optimalizaci slučovací fáze, která do této chvíle probíhala sekvenčně.

5.6 K-way MergeSort II.

Při optimalizaci fáze sloučení využijeme podobného principu z první části a pokusíme se slučovaná pole rozdělit do P částí tak, aby mohlo být sloučení provedeno paralelně. Nejprve si představíme základní myšlenku pro $P = 2$. Máme tedy dvě seřazená vstupní pole A a B o m a n prvcích, která chceme paralelně sloučit pomocí dvou vláken. V prvním kroku rozdělíme pole A pomocí jedné hraniční hodnoty do dvou částí. První Část na intervalu $A[0 \cdots (m/2)-1]$ a druhá část v mezích $A[m/2 \cdots m]$. Následně podle hodnoty na indexu $A[m/2]$ najdeme dělící hranici v druhém poli. V tomto momentě máme každé ze dvou vstupních polí rozdělené na dvě části, které můžeme nezávisle na sobě sloučit. První vlákno sloučí levou polovinu pole A s levou polovinou pole B a druhé vlákno obě pravé poloviny. Tento princip lze zobecnit tak, že každé z K vstupních polí rozdělíme pomocí $P - 1$ hraničních hodnot na P částí, které pak pomocí P vláken efektivně sloučíme dohromady.

Na první pohled docela jednoduchý princip si kvůli své obecnosti vyžádal poměrně složitou implementaci s několika režijními výpočty a ošetřením mnoha mezních případů. Pojďme se tedy na implementaci podívat blíže.

5.6.1 A_Parts

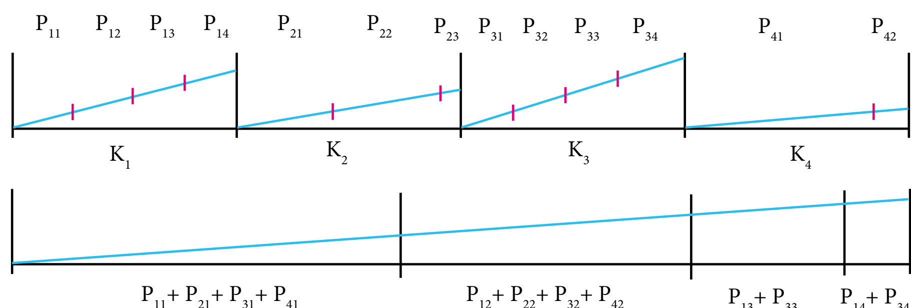
První část programu je s předchozí verzí téměř totožná. Hlavní změnou je pouze pomocné pole A_Parts, které slouží pro ukládání krajních pozic jednotlivých polí a jejich iterátorů. V části I. probíhalo sloučení pouze K polí, takže bylo třeba uložit K krajních mezí a iterátorů. Nyní ale slučujeme $P \times K$ polí, takže pole A_Parts musí být dvojrozměrné. Dvojrozměrné pole se v programovacím jazyce `C++` konstruuje obvykle dvěma hlavními způsoby. Prvním způsobem je jednorozměrné pole délky $P \times K$, kdy se na jednotlivé pozice musí přistupovat po násobcích P respektive K. Z praktického hlediska byla vybrána druhá možnost, kterou je pole polí, tedy opravdové 2D pole indexované pomocí dvou indexů. S tímto typem pole je snazší manipulace, lépe se upravuje délka jednotlivých polí a tak podobně.

```
Item ** A_Parts = new Item * [P];
for( int p = 0; p < P; p++ )
{
        A_Parts[p] = new Item[K];
}
```

Struktura uložení jednotlivých hodnot v poli je znázorněna na obrázku 5.13. Ve fázi rekurzivního dělení je ovšem naplněna pouze část pole A_Patrs, protože zbytek hodnot je možné dopočítat až poté, co budou seřazeny jednotlivé části vstupního pole. Způsob dělení a rekurzivní volání je popsáno již v první části, takže se rovnou přesuneme do fáze, kdy vstupní pole obsahuje K seřazených částí.

5.6.2 Hledání mezí

Rozdělení prvního pole probíhá pomocí dělení jeho délky číslem P, které označuje počet částí, do kolika má být každé z K podpolí rozděleno. Obvykle se hodnota P rovná hodnotě K, ale pro větší obecnost je umožněno v parametrech při spuštění programu zadat i hodnoty odlišné. První pole je tedy lineárně rozděleno do P stejných částí (opět krom té poslední, která může být delší). Další pole už takto jednoduše rozdělit nemůžeme, protože dělící hranice jsou závislé na hodnotách prvků prvního pole. Pro každou část P_i každého pole K_j musí platit, že všechny prvky v ní obsažené jsou menší, než je hodnota prvku, ležícího na pozici dolní meze P_{i+1} části pole K_0. Dělení ostatních částí se tedy provádí pomocí vyhledávání hraničních hodnot prvního, lineárně rozděleného pole.



Obrázek 5.12: K-way MergeSort II. - princip hledání hranic

Samotné hledání mezí probíhá na principu binárního vyhledávání, které má logaritmickou složitost. Aby bylo hledání ještě efektivnější, nevyhledává se vždy v celém podpoli K_j, ale pouze mezi již nalezenou horní hranicí předchozí části a koncem pole K_j. Binární vyhledávání je mírně modifikované pro naše potřeby. My totiž nepotřebujeme nalézt konkrétní prvek v daném poli (protože se v něm často ani nenachází), ale hledáme nejpravější pozici v poli, která obsahuje hodnotu menší než je zadaná hodnota.

A_Parts[P][K]		P											
		0			1			2			3		
		Begin	Pointer	End	Begin	Pointer	End	Begin	Pointer	End	Begin	Pointer	End
K	0	0	0	15	16	16	31	32	32	47	48	48	63
	1	64	64	90	91	91	117	118	118	127	-1	-1	-1
	2	128	128	145	146	146	160	161	161	175	176	176	191
	3	192	192	230	231	231	256	-1	-1	-1	-1	-1	-1

Obrázek 5.13: K-way MergeSort II. - struktura uložení hraničních hodnot

Zde bylo nutné ošetřit několik mezních podmínek. Může se totiž stát, že všechny prvky některé z K částí budou menší, nebo větší než hledaná hodnota a několik dalších nechtěných případů. Což má za následek, že ne vždy je možné každé pole K_j rozdělit pomocí $P - 1$ hraničních hodnot, do P částí. Pokud některé pole je rozděleno do méně než P částí, jsou zbývající bloky pole *A_Parts* vyplněny hodnotami -1 (viz obrázky 5.12 a 5.13).

5.6.3 Validace A_Parts

Po nalezení všech příslušných mezí se provádí kontrola tabulky *A_Parts*, kde všechny buňky s hodnotami -1 jsou přesunuty na konec pole (ve směru K) a je zjištěn skutečný stav počtu slučovaných polí v každé z P částí. V příkladu 5.13 budou slučovány 4, 4, 3 a 2 pole. Tyto hodnoty jsou uloženy do pomocného pole *A_Count*, které má délku P prvků (obrázek 5.14).

5.6.4 Výpočet počátečních indexů

Posledním přípravným krokem před samotným sloučením, je výpočet P počátečních indexů, od kterých budou prvky ve výsledném poli zapisovány. Výpočet se provádí opět z pole *A_Parts*, kdy pomocí dvou for cyklů postupně přičítáme počty prvků v každé části P_i do pomocné proměnné. Jelikož se řazený úsek může nacházet v libovolné části celého vstupního pole, je pomocná proměnná před začátkem cyklu nastavena na hodnotu *Begin*. Tyto pozice jsou uloženy do dalšího pomocného pole *A_Begin*, které má stejně jako pole *A_Count* délku P prvků. Obě pomocná pole i s příkladem vypočítaných hodnot z tabulky *A_Parts* můžeme vidět na obrázku 5.14.

A_Count[P]	P			
	0	1	2	3
	4	4	3	2

A_Begin[P]	P			
	0	1	2	3
	0	100	184	225

Obrázek 5.14: K-way MergeSort II. - pomocná pole A-Count a A-Begin

5.6.5 Merge

Když je vše připraveno, můžeme konečně přistoupit k závěrečnému slévání. Slévání se bude provádět P-krát, kde v každé části P_p budeme provádět sloučení $A_Count[p]$ polí a výsledek budeme zapisovat do původního pole *A_Data* od pozice $A_Begin[p]$.

Slévání provedeme v cyklu `for( p = 0; p < P; p++ )`. Kde do těla cyklu umístíme `while` cyklus s ukončovací podmínkou $A_Count[p]$. V každé iteraci cyklu `while` porovnáme hodnoty v polích na pozicích určených ukazateli v tabulce *A_Parts* a najdeme nejmenší prvek. Hledání provádíme opět lineárním

způsobem. Tuto hodnotu umístíme na správnou pozici do výsledného pole A_Data a zvětšíme příslušný ukazatel o hodnotu jedna. Dále zkontrolujeme jestli tento ukazatel nepřesáhl svou horní mez. Pokud ano, tak umístění informací o této části v tabulce A_Parts přesuneme na konec (ve směru K) a snížíme hodnotu $A_Count[p]$ o jedna. Tímto způsobem budeme mít vždy všechna ještě nesloučená pole na prvních pozicích tabulky A_Parts. Pokud jsou meze v pořádku, pokračujeme hledáním dalšího nejmenšího prvku. Tento proces opakujeme, dokud je splněna podmínka `while(A_Count[p])`. V momentě kdy $A_Count[p]$ dosáhne hodnoty 0 je cyklus ukončen, inkrementuje se proměnná ve vnějším for cyklu a pokračuje se sléváním $p + 1$. části. Po dokončení vnějšího cyklu je daná část pole seřazena a provede se návrat do vyšší úrovně rekurzivního volání, kde se celý proces opakuje dokud se nevynoříme do nejvyšší úrovně. V tento okamžik je seřazené celé vstupní pole o N prvcích.

5.6.6 Paralelizace

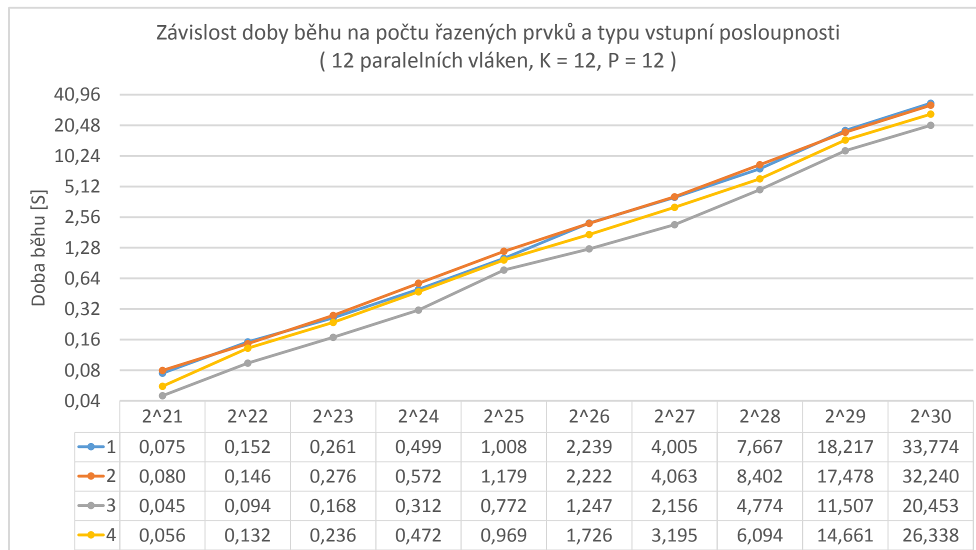
Paralelizace tohoto řešení je rozdělena do několika částí. Jelikož z Amdahlova zákona víme, že míra zrychlení je dána nejen násobkem zrychlení dané části, ale hlavně poměrem zrychlované části ku celému programu, pokusíme se paralelizovat co možná největší část. V programu se nachází několik for cyklů takže se zaměříme právě na ně. Paralelizaci budeme provádět pomocí direktivy `#pragma omp parallel for`.

První paralelní blok vytvoříme u for cyklu zajištujícího volání rekurzivních funkcí, stejně jako tomu bylo v první části. Druhý paralelní blok bude zajišťovat kopírování hodnot do pomocného pole. Dále by se nabízelo paralelní hledání mezí jednotlivých polí (plnění tabulky A_Parts), ale při proměření výkonnosti tohoto kroku bylo zjištěno zpomalení doby výpočtu. Zpomalení bylo zřejmě způsobeno častým přístupem do tabulky A_Parts různými vlákny a také výpadky cache paměti, jelikož pole jsou seskupena podle proměnné K a vnitřní cyklus iteruje podle proměnné P.

Řešením by bylo provádění cyklů otočit, ale paralelizace druhého cyklu není možná z důvodu datových závislostí, protože výpočty další iterace jsou závislé na výsledných hodnotách iterace předchozí. Tento blok tedy bude prováděn sériově. Paralelně nebude prováděn ani cyklus zajištující uspořádání mezí v tabulce A_Parts, protože doba provádění tohoto úkonu je více než zanedbatelná. A na závěr nejdůležitější část. Hlavním důvodem celé optimalizace bylo vytvoření paralelního p-cestného slévání, takže začátek paralelního bloku umístíme také právě před závěrečný for cyklus zajišťující p-way merge. Zde se objevil problém častých přístupů všech paralelních vláken do tabulky A_Parts, proto v každém paralelním bloku bylo vytvořeno pomocné pole, které obsahuje podmnožinu dat této tabulky, která je potřebná pouze pro sloučení dané části.

5.6.7 Naměřené hodnoty

Na obrázku 5.15 si můžeme prohlédnout naměřené hodnoty paralelního k-way MergeSortu s paralelním p-way sloučením. Měření probíhalo pro velikost vstupu 2^{30} prvků. Hodnoty K, P a počet vláken byl shodně nastaven na hodnotu 12. Naměřené časy řazení se pohybují v rozmezí 20 až 34 vteřin v závislosti na typu vstupní posloupnosti. Tyto výsledky jsou nejlepší dosažené výsledky ze všech předchozích algoritmů založených na principu řazení MergeSort.



Obrázek 5.15: Paralelní K-way MergeSort II. - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti

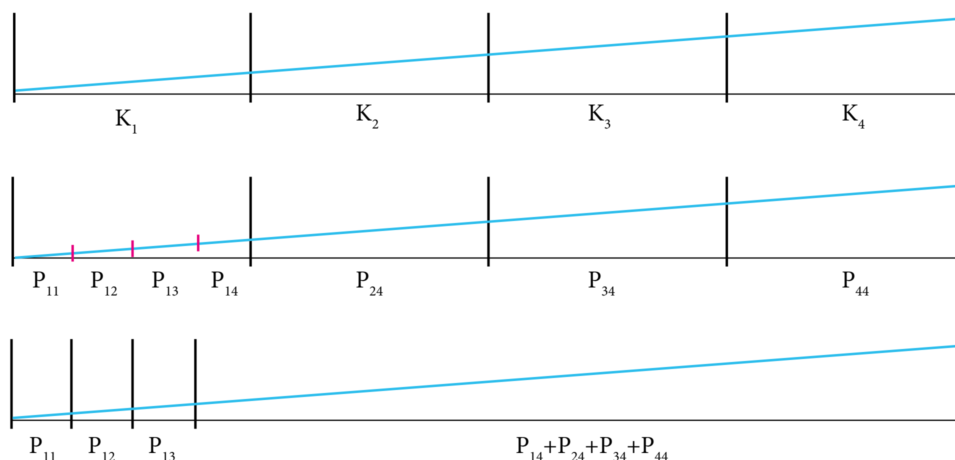
Tabulka 5.2: K-way MergeSort - porovnání výsledků různých verzí

Algoritmus/Vstup	1	2	3	4
Seriový K-way MS I.	222,365	211,529	256,282	340.943
Pralelní K-way MS I.	40,849	42,010	65,246	63,302
Pralelní K-way MS II.	33,774	32,240	20,452	26,338

Zde si můžeme všimnout, že mezi sériovou a paralelní verzí, kde paralelně probíhalo pouze rekurzivní volání, byl násobek zrychlení pro všechny typy posloupností zhruba stejný (asi pětinásobný). Pokud ovšem porovnáme obě paralelní verze, zjistíme, že u prvních dvou typů vstupních posloupností došlo pouze k minimálnímu zrychlení. Příčinu tohoto problému si vysvětlíme pro vstupní sekvenci typu jedna.

5.6.8 Problém s rozdělením zátěže

Celá situace je znázorněna na obrázku 5.16. První část představuje graficky znázorněnou posloupnost typu 1 (kompletně seřazená) rozdělenou do 4 částí (4-way MergeSort). V dalším kroku proběhne druhé dělení (příprava na paralelní 4-way Merge). Rozdělení hodnot v posloupnosti je pro naše potřeby ovšem zcela nevyhovující. Jak již bylo popsáno dříve, aby bylo sloučení provedeno správně, musí pro každou část P_i každého pole K_j platit, že všechny prvky v ní obsažené jsou menší, než je hodnota prvku, ležícího na pozici dolní meze P_{i+1}. části pole K_1. Což pro tento případ znamená, že pole K_2, K_3 a K_4 budou obsahovat pouze jedinou část. Tento problém se projeví ve třetí části, kde probíhá slévání.



Obrázek 5.16: K-way MergeSort II. - ukázka špatné rozložení zátěže

Jak vidíme na obrázku, tak rozložení zátěže mezi jednotlivá vlákna je velmi nevyrovnané. První 3 vlákna pouze překopírují prvky jediného bloku do výsledného pole a čekají na dokončení práce čtvrtého vlákna. Jelikož poslední vlákno vykonává více než tři čtvrtiny celkového objemu práce, je doba provedení tohoto úkonu několikanásobně delší než u předchozích vláken, tudíž se efektivita paralelního provádění nemůže projevit. V ideálním případě by se zátěž měla rovnoměrně rozložit mezi všechna vlákna a doba provádění každé větve by trvala stejně dlouho. V takovém případě by došlo k maximálnímu zrychlení. Tento problém by bylo možné částečně vyřešit zvolením pokročilejší strategie výběru hraničních hodnot, každopádně ani lepší strategií výběru nelze tento problém zcela odstranit.

U posloupností typu 3 a 4 došlo k podstatně lepšímu rozložení zátěže a zlepšení se tak projevilo ve větší míře. Dokonce natolik, že paralelní verze

s dvanácti vlákny je téměř 13× rychlejší než ta sériová! Mírnou nevýhodou tohoto řešení je, že se algoritmus stal datově citlivým, ale i přesto se jedná o zatím nejvýkonnější řadící algoritmus na bázi MergeSortu.

5.7 Hybridní algoritmus

Jedním z úkolů autora této práce je navrhnout paralelní hybridní algoritmus, který využívá principy algoritmu TimSort a K-way MergeSort. Jak již z předchozích kapitol víme, hybridním algoritmem je již TimSort sám o sobě. Pro seřazení vstupního pole totiž využívá metody algoritmů InsertionSort a MergeSort. Pro spojení výše zmíněných dvou algoritmů se nabízí více možností.

5.7.1 Návrh hybridního algoritmu

Prvním nápadem bylo upravení algoritmu TimSort, respektive jeho fázi slévaní běhů, která dosud probíhá dvoucestným slučováním na paralelní k-cestné slévání. Velikost běhů je ale pro toto použití příliš malá a vzhledem k náročnosti režijních operací spojených s vícecestným sléváním by řešení nebylo příliš efektivní. Navíc algoritmus TimSort je optimalizován pro částečně seřazená data, která řadí o dost rychleji než ostatní běžné algoritmy. O to více ztrácí na náhodných datech a vstupních sekvencích obsahujících jen minimum přirozených běhů.

Další možností je podívat se na problém z druhé strany, tedy ze strany k-way MergeSortu, který obsahuje dělící hranici, podle které rozhoduje, zda bude řazení probíhat obvyklým způsobem nebo zda je zbývající část už dostatečně malá na to, aby mohla být efektivně seřazena pomocným sekvenčním řadícím algoritmem. Dosud jsme pro tyto účely používali algoritmus InsertionSort, který je i přes svoji kvadratickou asymptotickou složitost velmi efektivní pro řazení malých polí. Dělící hranice byla v takovém případě 32 prvků pro sériovou verzi algoritmu a 256 prvků pro vícevláknovou paralelní verzi. Řadit takto malé vstupní sekvence by se ale určitě nevyplatilo ze stejného důvodu, jako v minulém odstavci použití k-way MergeSortu uvnitř řadícího algoritmu TimSort.

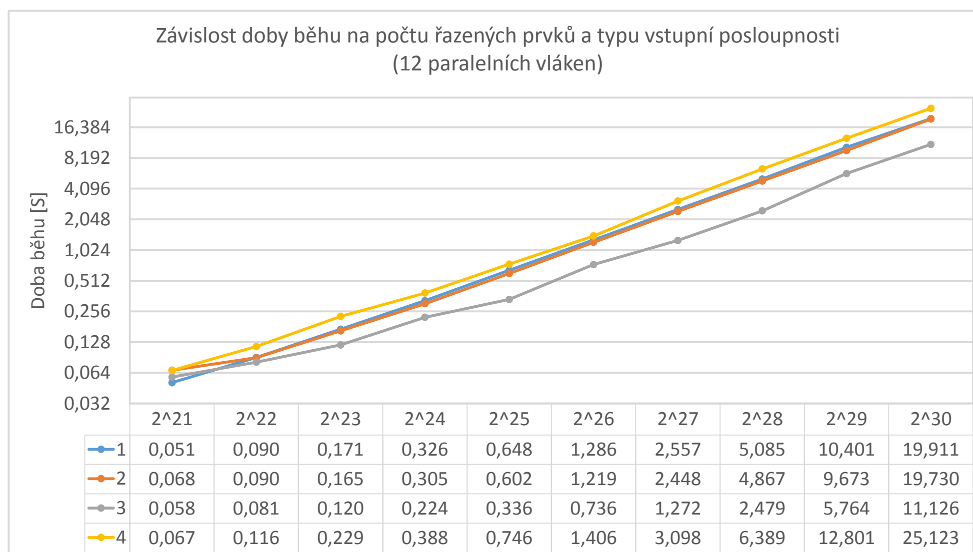
Co se ale stane pokud tuto hranici zvětšíme? Čím bude tato hranice nižší, tím méně efektivně bude algoritmus řadit úplně nebo částečně seřazená data, protože bude docházet k nerovnoměrnému rozložení zátěže mezi jednotlivá vlákna, stejně jako tomu bylo u paralelního k-way MergeSortu. Pokud bude hranice dělení příliš vysoko, bude se projevovat neefektivita TimSortu pro náhodná data.

Jelikož strana TimSortu je méně prozkoumanou oblastí, zkusíme se přiklonit spíše na jeho stranu a držet se ideologie částečně seřazených „reálných“ dat. Dělící hranici budeme tedy volit co možná největší. Otázkou je, zda se potom vyplatí režie přepínání kontextu rekurzivních volání. Co kdybychom se pokusili bez rekurze obejít úplně?

5.7.2 Výsledné řešení

Navrhneme tedy následující řešení. Vstupní pole rozdělíme do K částí, kdy každou část seřadíme algoritmem TimSort a následně tyto části sloučíme p-cestným sléváním. Na první pohled je toto řešení velmi podobné s naším návrhem paralelního algoritmu TimSort. Rozdíl se nachází pouze ve fázi slučování, kde paralelní TimSort používá principy vlastního algoritmu a z jednotlivých seřazených části jsou vytvořeny běhy, které se uloží do spojového seznamu a jsou slučovány stejným způsobem jako je tomu u klasického TimSortu. Slévání v tomto případě probíhalo sériově. V aktuální verzi bude toto řešení nahrazeno právě paralelním p-way slučováním. Výsledkem by tedy mohl být univerzální a poměrně rychlý paralelní řadící algoritmus. Pokusíme se ho tedy implementovat a změřit jeho výkonnost na výpočetním serveru STAR.

Se základními funkčními bloky použitými v implementaci tohoto algoritmu jsme se již seznámili dříve, proto je zde nebudeme znovu podrobně popisovat. Prvním krokem algoritmu je rozdělení vstupního pole do K částí a uložení jednotlivých mezí do pomocné tabulky A_Parts. Ve druhém kroku se nachází paralelní blok obsahující for cyklus, v jehož těle je volána pomocná funkce TimSort na příslušné části vstupního pole, která zajistí jejich seřazení. Po seřazení všech K částí následuje výpočet a hledání hraničních hodnot jednotlivých částí pole. Dále úprava tabulky A_Parts, zjištění skutečného počtu slévaných částí a vypočítání počátečních indexů pozic ve výsledném poli. Nakonec samotné paralelní p-cestné slévání.



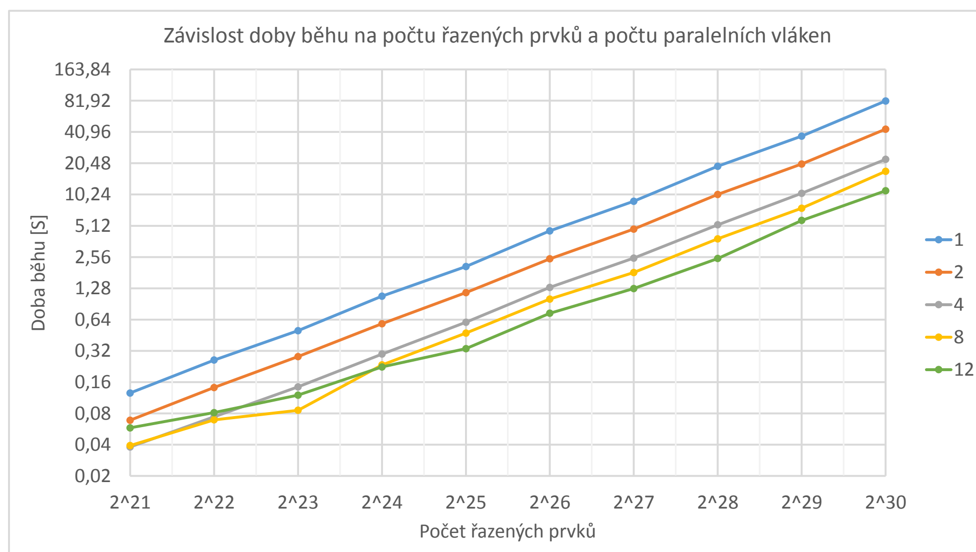
Obrázek 5.17: Hybridní algoritmus - závislost doby běhu na počtu řazených prvků a typu vstupní posloupnosti

5.7.3 Naměřené výsledky

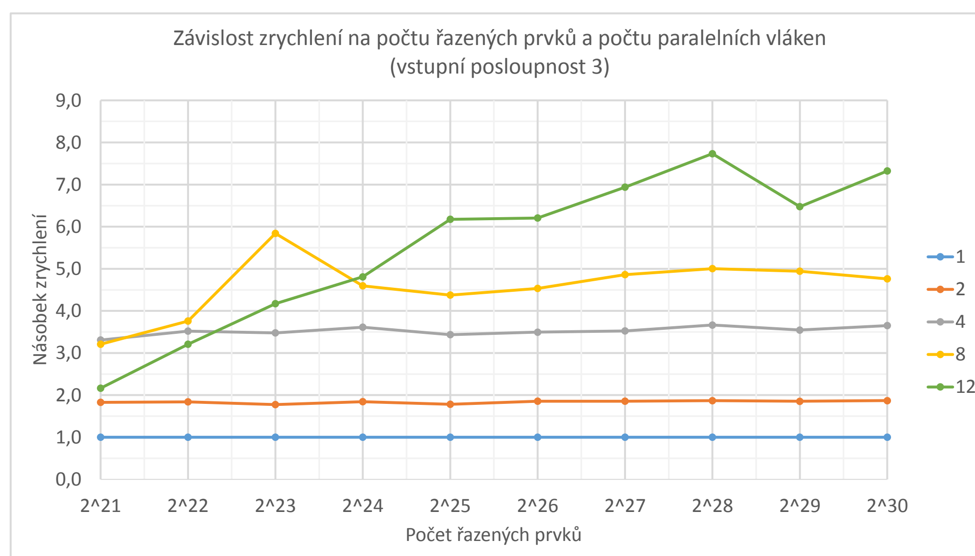
Nyní algoritmus spustíme na serveru STAR a podíváme se na první naměřené hodnoty. Na obrázku 5.17 je vynesen graf závislosti doby běhu na počtu řazených prvků pro různé typy vstupní posloupnosti. Řazení probíhalo pomocí dvanácti paralelních vláken a parametrů $K = P = 12$. Při vyhodnocování výsledků se opět zaměříme časy řazení vstupních polí o velikosti 2^{30} prvků.

Nejpomaleji seřazenou posloupností byla dle očekávání náhodná posloupnost (typ 4), která byla seřazena za 25,1 vteřiny, což je o 40 vteřin rychleji než paralelní TimSort, ze kterého tento hybridní algoritmus vychází. Překvapivějším výsledkem je, že hybridní algoritmus seřadil 2^{30} náhodných prvků rychleji než paralelní K-way MergeSort s paralelním sléváním, který tento úkol zvládl za 26,3 vteřiny. Lepšího času dosáhlo řazení posloupností typu 1 a 2, které byly seřazeny za necelých 20 vteřin. Nejrychleji se podařilo seřadit posloupnost typu 3, u které se pomyslné stopky zastavily na hodnotě 11,125 vteřiny. Což je bezkonkurenčně nejlepší výsledek ze všech měřených algoritmů.

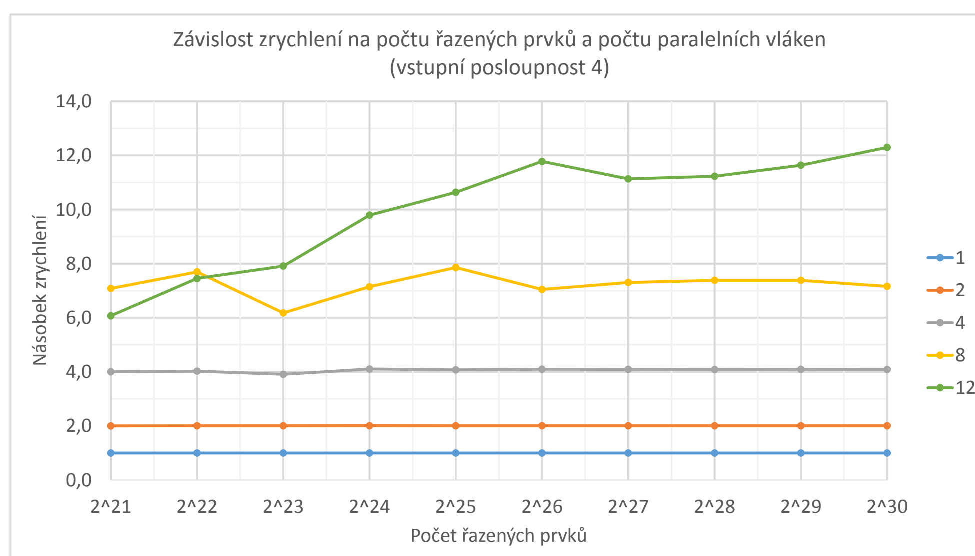
V případě vstupní posloupnosti typu 4 byl výpočet zpomalen delším řazením náhodných prvků algoritmem TimSort. Při řazení posloupností 1 a 2 se projevila nevyváženost slučovaných částí, kdy jedno z vláken muselo pracovat s nepřiměřeně velkým objemem dat. U posloupnosti typu 3, která má vyvážené obě části, se projevil výkon algoritmu nejvíce a výsledkem je poloviční doba běhu proti konkurenčním algoritmům.



Obrázek 5.18: Hybridní algoritmus - závislost doby běhu na počtu řazených prvků a počtu paralelních vláken



Obrázek 5.19: Hybridní algoritmus - závislost zrychlení na počtu řazených prvků a počtu paralelních vláken (vstupní posloupnost 3)



Obrázek 5.20: Hybridní algoritmus - závislost zrychlení na počtu řazených prvků a počtu paralelních vláken (vstupní posloupnost 4)

Nyní se blíže podíváme na vliv počtu paralelních vláken. Násobek zrychlení řazení v závislosti na počtu paralelních vláken můžeme vidět na obrázku 5.19. Řazení se efektivně zrychlovalo do hodnoty počtu vláken rovných 4, kde bylo dosaženo téměř lineárního zrychlení ($3,6\times$). Pro 8 a 12 vláken zrychlení nebylo tak výrazné a maximální hodnotu je $7,7\times$ menší čas oproti sériové verzi výpočtu.

Pro posloupnost typu 4 byly výsledky na obrázku 5.20 o něco pozitivnější. Zde bylo zaznamenáno lineární zrychlení téměř v celém měřeném rozsahu. Nejvyšší hodnotou bylo zrychlení pro 12 paralelních vláken, které dosahovalo hodnoty 12,3 násobku sériové verze algoritmu. Z grafu si ale můžeme všimnout, že pro větší počty paralelních vláken se zrychlení začalo v celé míře projevovat až pro delší vstupní posloupnosti.

5.7.4 Hybridní algoritmus a různé typy řazených prvků

Jedním z bodů zadání této práce bylo otestovat hybridní algoritmus pro různé typy řazených prvků. Do této chvíle jsme výkonnost algoritmů měřili pouze pro vstupní sekvence celých čísel datového typu `int`. Tento datový typ v paměti zabírá místo o velikosti 4 B. Nyní se pokusíme zjistit, jaký vliv má paměťová náročnost řazených prvků na celkovou dobu běhu řadícího algoritmu. Algoritmus by bylo možné otestovat pro několik běžných datových typů, ale s ohledem na objektivitu měření bylo nutné zachovat stejnou strukturu vstupních posloupností, jako tomu bylo v předchozích případech. Proto se autor práce rozhodl měření provádět trochu jiným způsobem.

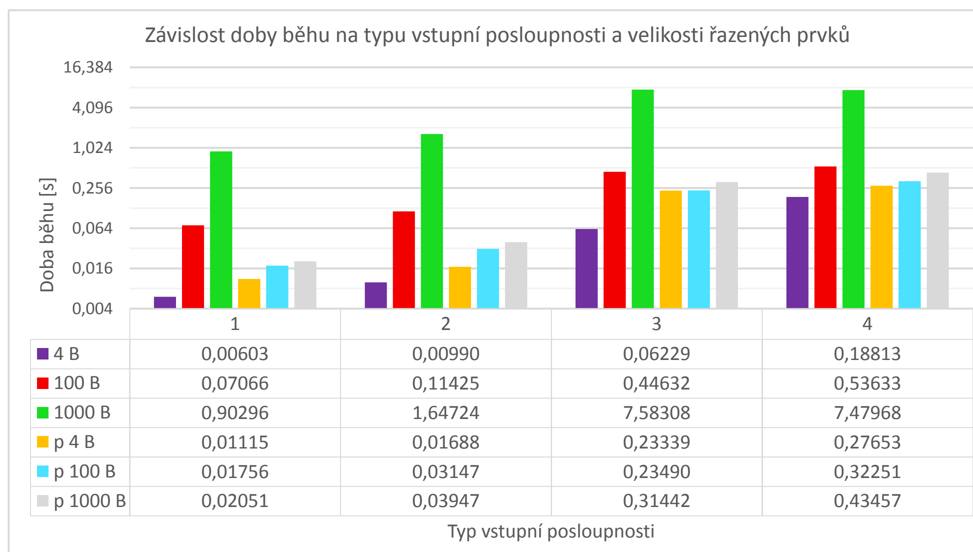
Řazeným prvkem byla zvolena struktura obsahující dva atributy. První atribut je datového typu int, který bude sloužit jako klíč a umožní vygenerovat příslušné typy vstupních posloupností. Druhým atributem je statické pole celých čísel. Toto pole je ve struktuře obsaženo z důvodu zvětšení paměťové náročnosti řazených prvků, které způsobí prodloužení času potřebného pro manipulaci s jednotlivými prvky během řazení. Tímto způsobem můžeme simulovat například řazení textových řetězců při zachování formátu měřených posloupností. Při testování byly zvoleny délky pole 24 a 249 prvků. Celková velikost struktury tedy byla 100 B a 1000 B.

Další způsob testování probíhal tak, že místo pole celých čísel, popřípadě výše zmíněných struktur, bylo řazeno pole ukazatelů na tyto objekty. Ukazatel v 64bitovém prostředí má velikost 8 B, takže složitost manipulace s ukazatelem je stále stejná, nezávisle na velikosti objektu na který ukazuje.

Z důvodu paměťové náročnosti byl řazen pouze 1 milion prvků. Naměřené hodnoty jsou znázorněny v grafu 5.21. Položky začínající na „p“ označují ukazatel na strukturu příslušné velikosti. Pokud analyzujeme naměřené výsledky, zjistíme, že nejrychleji řazeným prvkem je datový typ `int`. Nejpomaleji se podařilo seřadit pole struktur, kde jeden prvek měl velikost 100 a 1000 B.

Nejvýraznější zpomalení bylo zaznamenáno u 3. a 4. typu vstupní posloupnosti, kde řazení prvků velikosti 1000 B proběhlo za neuvěřitelných 7,5 vteřiny!

Přitom u posloupnosti typu jedna stačila k seřazení pouhá vteřina. Na takto špatný výsledek má největší vliv algoritmus InsertionSort, který je použit pro tvorbu jednotlivých běhů uvnitř algoritmu TimSort.



	1	2	3	4
4 B	0,00603	0,00990	0,06229	0,18813
100 B	0,07066	0,11425	0,44632	0,53633
1000 B	0,90296	1,64724	7,58308	7,47968
p 4 B	0,01115	0,01688	0,23339	0,27653
p 100 B	0,01756	0,03147	0,23490	0,32251
p 1000 B	0,02051	0,03947	0,31442	0,43457

Obrázek 5.21: Hybridní algoritmus - závislost doby běhu na typu vstupní posloupnosti a velikosti řazených prvků

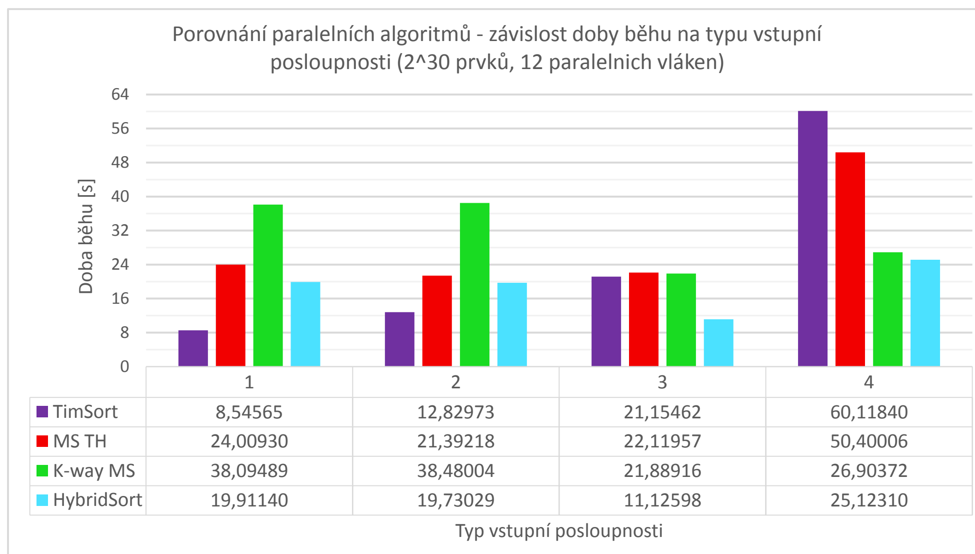
U prvních dvou typů vstupní posloupnosti byly běhy vytvořeny téměř bez použití pomocného vkládání, takže se výsledný čas tolik nezhoršil. Ovšem u částečně seřazené a náhodné posloupnosti bylo třeba vykonat obrovské množství přesunů jednotlivých prvků, což způsobilo tak výrazné zpomalení. Problém řazení paměťově náročných prvků lze vyřešit řazením pole ukazatelů na tyto prvky. Tento způsob dosahoval vyrovnaných výsledků pro všechny velikosti řazených prvků a byl více než 20× rychlejší než řazení pole struktur.

5.8 Porovnání výkonnosti paralelních algoritmů

Nyní si porovnáme dosažené výsledky všech námi implementovaných paralelních verzí řadících algoritmů. Dosažené časy představují dobu běhu řadící části algoritmu pro seřazeni 2^{30} vstupních prvků s využitím 12-ti paralelních vláken.

Algoritmus TimSort vynikal u prvních dvou typů vstupních posloupností, kdy dosáhl výsledných časů 8,5 a 12,8 vteřiny. Takto dobrých výsledků dosáhl zejména díky vnitřnímu sekvenčnímu řazení TimSort které probíhalo v téměř lineárním čase. Výsledné sloučení pomocí slévání dvojic běhů a průběžného vyvažování je i přes své sériové provedení velmi rychlé a dosahuje lepších vý-

sledků než sekvenční K-way merge. U posloupnosti typu 3 dosáhl paralelní TimSort srovnatelných hodnot s ostatními algoritmy, kdy vstupní posloupnost seřadil za 21 vteřin. Nejhorším výsledkem celého měření byl čas seřazení náhodné posloupnosti, na kterou TimSort potřeboval celých 60 vteřin!



	1	2	3	4
TimSort	8,54565	12,82973	21,15462	60,11840
MS TH	24,00930	21,39218	22,11957	50,40006
K-way MS	38,09489	38,48004	21,88916	26,90372
HybridSort	19,91140	19,73029	11,12598	25,12310

Obrázek 5.22: Porovnání paralelních algoritmů - závislost doby běhu na typu vstupní posloupnosti

Algoritmus MergeSort s hranicí přepnutí dosahoval poměrně stabilních výsledků kolem 23 vteřin. Stabilních výsledků bylo dosaženo díky tomu, že MergeSort s hranicí přepnutí je oproti ostatním algoritmům v největší míře datově necitlivý. Rychlost řazení v závislosti na vstupních datech ovlivňuje pouze řadící algoritmus InsertionSort, který je použit pro řazení polí menších, než je daná hraniční hodnota. Větším výkyvem bylo opět řazení náhodné posloupnosti, které si vyžádalo dvojnásobně delší čas.

Řazení pomocí algoritmu K-way MergeSort dosáhlo naopak nejhorších výsledků u vstupní posloupnosti typu 1 a 2. Dlouhá doba řazení byla způsobena již dříve popsanou nevyvážeností zátěže mezi jednotlivými vlákny u paralelního p-cestného slévání. Výsledkem jsou časy 38 vteřin, které znamenají nejhorší dosažené výsledky u těchto typů posloupností. K-way MergeSort si svou reputaci vylepšil při řazení částečně seřazených a náhodných dat, kde se plně projevila efektivita p-cestnéhlo slévání.

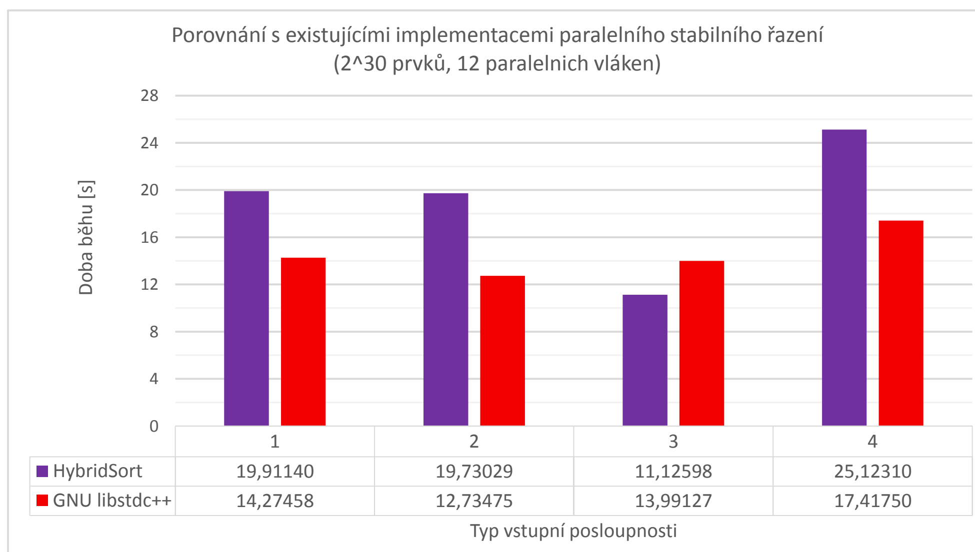
Posledním měřeným algoritmem byl hybridní řadící algoritmus navržený autorem této práce. Algoritmus představující sloučení metod algoritmů TimSort a K-way Merge dosáhl velmi dobrých výsledků u všech měřených typů vstupních posloupností. Posloupnosti typu 1 a 2 dokázal seřadit za necelých

20 vteřin, což ho řadí na druhé místo za paralelní algoritmus TimSort. Důvodem proč je hybridní algoritmus pomalejší než TimSort, který pracuje na velmi podobném principu je ten, že slučování K částí u paralelního TimSortu není datově citlivé. U hybridního algoritmu se stejně jako u K-way MergeSortu projevila nevyváženost zátěže mezi jednotlivými vlákny, která z paralelního slévání vytvořila téměř sériové. Posloupnost 3, tedy posloupnost obsahující přirozené běhy různých dálek, která představuje „reálná“ data, pro která byl algoritmus optimalizován, byla hybridním algoritmem seřazena za rekordních 11,126 vteřiny, což je poloviční čas ve srovnání s ostatními měřenými algoritmy. Tento typ posloupnosti umožnil hybridnímu algoritmu projevit potenciál TimSortu na částečně seřazených datech a výkon p-cestného sloučení na rovnoměrně rozdělené posloupnosti. Poslední, nejvíce obávanou posloupností je náhodná sekvence dat, u které hybridní algoritmus předčil veškerá očekávání a s časem 25,1 vteřiny se stal nejrychlejším algoritmem i pro tento typ vstupní posloupnosti.

5.8.1 Porovnání hybridního algoritmu s existujícími implementacemi paralelního stabilního řazení

Posledním úkolem této práce je porovnání výkonnosti hybridního algoritmu s existujícími implementacemi stabilního paralelního řazení. Jako první porovnávaný algoritmus byl zvolen vysoce optimalizovaný algoritmus stabilního řazení standardní knihovny `C++`[7, 23]. Tento algoritmus byl spuštěn v takzvaném „parallel_mode“ pomocí kompilátoru GNU libstdc++. Tento mód umožňuje jednoduché použití vybraných funkci z STL v paralelním režimu. Paralelního režimu lze dosáhnout buďto kompilací s příslušnými parametry nebo voláním přímo paralelní verze oné funkce. Paralelismus je založen na knihovně OpenMP, kterou pro paralelizaci využíváme i v námi implementovaných programech. Z celé škály paralelizovaných funkcí nás ovšem bude zajímat pouze funkce `std::stable_sort()`, která seřadí zadané vstupní pole a zajistí stabilitu řazení. Tato funkce ve svém jádru používá paralelní k-way MergeSort, s jehož implementací jsme se v této práci již setkali. Porovnejme tedy tento algoritmus s naším hybridním řadícím algoritmem pro 2^{30} řazených prvků, všechny typy vstupních posloupností a s využitím dvanácti paralelních vláken.

V grafu 5.23 můžeme vidět naměřené hodnoty jednotlivých vstupních sad. Knihovní verze stabilního řazení dosáhla velmi vyrovnaných výsledků pro všechny typy vstupních sekvencí a ve třech ze čtyř případů předstihla námi navržený hybridní algoritmus. Výkon hybridního algoritmu byl v nejhorším případě o 35 % nižší než u konkurenčního algoritmu. Naopak u vstupní sekvence typu 3, pro kterou byl hybridní algoritmus optimalizován, bylo dosaženo o 20% lepšího času než u algoritmu standardní knihovny. S přihlédnutím na úroveň optimalizace knihovní verze hodnotím tyto výsledky jako velmi zdařilé.



Obrázek 5.23: Porovnání hybridního algoritmu s existujícími implementacemi paralelního stabilního řazení

Další porovnávanou implementací měl být „Parallel Stable Sort Using C++11 for OpenMP“[24] od autora Arch D. Robisona. Tento algoritmus je taktéž založený na řadícím algoritmu MergeSort a používá některé prvky standardu C++11, zejména pak konstrukci `move`. Pro paralelizaci je opět použita knihovna OpenMP. Při měření tohoto algoritmu byly ovšem zjištěny výrazně pomalejší výsledky ve srovnání s předchozími dvěma algoritmy, které se žádným způsobem nepodařilo vylepšit. Autor bohužel neudává žádné údaje o výkonnosti této implementace a komentuje to slovy, že výsledky závisí na konkrétním hardwaru a datovém typu klíčového prvku a nechť si každý vyzkouší tuto implementaci pro svůj oblíbený data set. Takže nemůžeme posoudit, zda je chyba v nesprávném použití nebo v menší efektivitě této implementace.

Závěr

Cílem této práce bylo nastudovat řadící algoritmy MergeSort a TimSort. Tyto algoritmy implementovat v jazyce C++ a pomocí metod transformací zdrojového kódu a paralelizací zvýšit jejich výkonnost. Dále byl požadován návrh a implementace hybridního algoritmu, který by fungoval na principech řazení výše zmíněných algoritmů. Výkonnost algoritmů změřit na fakultním serveru STAR a porovnat s existujícími implementacemi paralelního stabilního řazení.

Cíle této práce byly splněny. V teoretické části byly představeny požadované algoritmy a principy jejich fungování. Dále zde byly popsány metody kompilátorových optimalizací, možnosti transformací zdrojových kódu a principy používání knihovny pro paralelní programování OpenMP. V praktické části byly tyto algoritmy implementovány a aplikovány na ně příslušné optimalizační techniky. V závislosti na dosažených výsledcích těchto algoritmů byl navrhnut hybridní řadící algoritmus kombinující jejich principy řazení. Hybridní algoritmus byl otestován pro různé typy vstupních posloupností i řazených objektů. Na závěr byla výkonnost algoritmu porovnána s existujícím řešením paralelního stabilního řazení standardní knihovny C++, kde byly u hybridního algoritmu zjištěny velmi vyrovnané výsledky ve srovnání s knihovní funkcí. Pro jeden ze čtyř typů testovaných vstupních posloupností bylo dokonce hybridní řazení rychlejší než výše zmíněný algoritmus.

Jelikož se jedná o velmi obsáhlé téma, nebylo možné v této práci podrobně prozkoumat všechny oblasti, kterých se problém týká. Tato práce ale představuje dobrý základ pro další pokračování a otevírá možnosti další optimalizace na základě již dosažených výsledků. Pokračování by se například mohlo ubírat směrem pokročilejších metod hledání dělících hranic paralelního slévání, díky kterým by bylo možné rovnoměrnějšího rozdělení objemu práce mezi jednotlivá paralelní vlákna. Dále by bylo možné zapracovat na efektivnějším hledání minimálních prvků u k-cestného slévání, které je v této práci řešeno lineárním vyhledáváním. V neposlední řadě se nabízí přidání pokročilých funkčních bloků algoritmu TimSort, jehož originální implementace je velmi rozsáhlá a komplikovaná a v této práci nebyla použita v plném rozsahu.

Literatura

[1] Tvrdík, P.: Algritmy řazení. [online], 2014, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EFA/_media/lectures/bi-efa2014prednaska3-sort-anim.pdf

[2] Kolář, J.: Odhady růstu funkcí. [online], 2015, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-ZDM/_media/lectures/zdm-p00bigo-h.pdf

[3] Neckář, J.: Insertion sort. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://www.algoritmy.net/article/8/Insertion-sort

[4] Neckář, J.: Merge sort. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://www.algoritmy.net/article/13/Merge-sort

[5] Symvonis, A.: Optimal Stable Merging. *The Computer Journal*, ročník 38, 1993: s. 681–690. Dostupné z: https://pdfs.semanticscholar.org/771e/a2251dde91415ea002b3697cdfde0117a81e.pdf

[6] Greene, W. A.: k-way merging and k-ary sorts. [online], 1993, [cit. 2017-05-10]. Dostupné z: http://cs.uno.edu/people/faculty/bill/k-way-merge-n-sort-ACM-SE-Regl-1993.pdf

[7] Singler, P., Johannes; Sanders: The GNU libstdc++ parallel mode: Benefit from Multi-Core using the STL. [online], [cit. 2017-05-10]. Dostupné z: http://ls11-www.cs.uni-dortmund.de/people/gutweng/AD08/VO11_parallel_mode_overview.pdf

[8] Peters, T.: TimSort. [online], 2002, [cit. 2017-05-10]. Dostupné z: http://svn.python.org/projects/python/trunk/Objects/listsort.txt

[9] McIlroy, P.: Optimistic Sorting and Information Theoretic Complexity. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, Philadelphia, PA, USA: Society for Industrial and

Applied Mathematics, 1993, ISBN 0-89871-313-7, s. 467–474. Dostupné z: http://dl.acm.org/citation.cfm?id=313559.313859

[10] Šimeček, I.: Pokročilá nastavení kompilátoru GCC. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/gcc.pdf

[11] Šimeček, I.: Kompilátorové optimalizace I: Metody transformací zdrojových kódů. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/kompilator.pdf

[12] Šimeček, I.: Volba optimálních datových struktur a jejich kombinací. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/volba.pdf

[13] Lórencz, H. J. Z. T., Róbert: Kvantitativní principy návrhu pocítačů. [online], [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-APS/_media/aps1.pdf

[14] Šimeček, I.: Úvod do paralelního počítání. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/parallel.pdf

[15] OpenMP API 4.5 C/C++. [online], 2015, [cit. 2017-05-10]. Dostupné z: http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

[16] Eijkhout, V.: Parallel Programming in MPI and OpenMP. [online], 2015, [cit. 2017-05-10]. Dostupné z: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/index.html

[17] Barney, B.: OpenMP. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://computing.llnl.gov/tutorials/openMP/

[18] Šimeček, I.: Technologie OpenMP. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/omp.pdf

[19] Šimeček, I.: Kompilátorové optimalizace I: Modely chování skryté (cache) paměti. [online], 2016, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/komp_cache.pdf

[20] Lórencz, J. Z. T., Róbert; Hlaváč: Pameťová hierarchie, návrh skryté paměti 1. [online], [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-APS/_media/aps_mem_1.pdf

[21] Šimeček, I.: Kompilace a spouštění úloh pod sdílenou pamětí na serveru STAR. [online], 2015, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/tutorials/star

[22] Šimeček, I.: Výpočetní prostředky. [online], 2015, [cit. 2017-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/tutorials/vyp_pr

[23] Singler, P., Johannes; Sanders: The GNU libstdc++ parallel mode: Software Engineering Considerations. [online], [cit. 2017-05-10]. Dostupné z: https://algo2.iti.kit.edu/singler/mcstl/parallelmode_se.pdf

[24] Robison, A. D.: A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP. [online], 2014, [cit. 2017-05-10]. Dostupné z: https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp

Příloha A

Seznam použitých zkratek

CPU - central processing unit

RAM - random-access memory

STL - standard template library

Příloha B

Obsah přiloženého média

```
readme.txt ................................ stručný popis obsahu média
src
├── sort .............................. zdrojové kódy řadících algoritmů
└── thesis ...................... zdrojová forma práce ve formátu LaTeX
text
└── BP_Talácko_Rudolf_2017.pdf ........... text práce ve formátu PDF
data
├── results ..................................... naměřené časy řazení
└── images ................................ grafy a další použité obrázky
```