FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Vícevláknová metoda řazení Timsort
Student:	Daniel Blažek
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

1) Nastudujte [1,2] a implementujte sekvenční verzi algoritmu Timsort.
3) Diskutujte možnosti optimalizace a paralelizace tohoto algoritmu pomocí technologie OpenMP [3,4,5].
2) Implementujte vybrané optimalizace a i paralelní verzi algoritmu.
4) Porovnejte výkonnost jednotlivých verzí na školním serveru STAR a diskutujte dosažené výsledky.

[1] https://ericmervin.medium.com/what-is-timsort-76173b49bd16
[2] https://github.com/python/cpython/blob/bcb198385dee469d630a184182df9dc1463e2c47/Objects/listsort.txt
[3] https://github.com/rust-lang/rust/blob/5f60208ba11171c249284f8fe0ea6b3e9b63383c/src/liballoc/slice.rs#L841-L980
[4] https://saurabhsoodweb.wordpress.com/2017/04/18/parallelizing-timsort/
[5] https://mail.python.org/pipermail/python-dev/2002-July/026900.html

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 16. ledna 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Vícevláknová metoda řazení Timsort

Daniel Blažek

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

11. května 2023

Poděkování

Rád bych upřímně poděkoval svému vedoucímu bakalářské práce, doc. Ing Ivanu Šimečkovi, Ph.D., za jeho cenné rady, odborné vedení a trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při 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 předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, 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 zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). 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 zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 11. května 2023 . . . .

Odkaz na tuto práci

Blažek, Daniel. *Vícevláknová metoda řazení Timsort*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Cílem této práce je zabývat se sekvenčními optimalizacemi algoritmu Timsort a jeho paralelizací pomocí OpenMP. Nově vzniklé algoritmy jsou otestovány a porovnány se základní verzí Timsortu a dalšími vybranými řadícími algoritmy. Toto testování je prováděno na školním serveru STAR určenému k objektivnímu testování paralelních algoritmů. Veškerá implementace je v jazyce C++.

Klíčová slova timsort, c++, openmp, paralelizace, řadící algoritmus, mergesort, paralelní timsort, optimalizace, řadící algoritmy

Abstract

The aim of this thesis is to explore the sequential optimizations of the Timsort algorithm and its parallelization using OpenMP. Newly developed algorithms are tested and compared with the base version of Timsort and other selected sorting algorithms. This testing is performed on the school server STAR, designed for objective testing of parallel algorithms. All implementations are in the C++ language.

Keywords timsort, c++, openmp, parallelization, sorting algorithm, mergesort, paralel timsort, optimalization, sorting algorithms

Obsah

Úvod **1**

1 Cíl práce **3**
1.1 Sekvenční optimalizace algoritmu . . . 3
1.2 Paralelizace algoritmu . . . 3
1.3 Testování . . . 3

2 Analýza a návrh **5**
2.1 Popis algoritmu Timsort . . . 5
2.1.1 Slučovací strategie . . . 6
2.1.2 Slučovací algoritmus . . . 7
2.1.3 Slučovací paměť . . . 7
2.1.4 Galloping . . . 7
2.2 Návrh optimalizací . . . 9
2.2.1 Slučování více runů . . . 10
2.2.2 Timsort s detekcí runů od konce . . . 11
2.3 Paralelizace Timsortu . . . 11
2.3.1 První návrh . . . 11
2.3.2 Druhý návrh . . . 11
2.3.3 Třetí návrh . . . 11

3 Realizace **13**
3.1 Realizace klasického Timsortu . . . 13
3.1.1 Pomocná struktura Slice . . . 14
3.1.2 Třída TimSort . . . 14
3.1.3 Hledání runů . . . 16
3.1.4 Slučování runů . . . 16
3.1.5 Galloping . . . 17
3.2 Slučování více runů najednou (k-way merge) . . . 19

3.3 Timsort s hledáním runů od konce 24
3.4 Paralelní Timsort podle prvního návrhu 27
3.5 Paralelní Timsort podle třetího návrhu 28
3.6 Paralelní Mergesort pomocí OpenMP 30

4 Testování **33**
4.1 Server STAR 33
4.2 Generování testů 34
4.2.1 Náhodná data 34
4.2.2 Seřazená data 35
4.2.3 Porovnávací funkce 36
4.3 Spouštění testů 37
4.4 Testování základní implementace Timsortu 37
4.4.1 Shrnutí testování základní verze Timsortu 45
4.4.2 Poznámka k testování stringů 45
4.5 Testování optimalizací 45
4.5.1 Hledání runu od konce 45
4.5.2 Mergování více runů 46
4.6 Testování paralelních algoritmů 50

Závěr **67**

Bibliografie **69**

A Seznam použitých zkratek **73**

B Obsah přiloženého CD **75**

Seznam obrázků

4.1 Graf porovnávající náhodná data třídy LargeTestClass pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 38
4.2 Graf porovnávající náhodná data typu integer pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 38
4.3 Graf porovnávající náhodná data typu string pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 39
4.4 Graf porovnávající náhodná data stringů s různou délkou pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 39
4.5 Graf porovnávající data s hodně duplikáty pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 40
4.6 Graf porovnávající data s obsahující několik seřazených částí pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 41
4.7 Graf porovnávající data ve tvaru pyramidy pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 41
4.8 Graf porovnávající data seřazená v opačném pořadí pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 42
4.9 Graf porovnávající seřazená data pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 42
4.10 Graf porovnávající náhodná data řazena funkcí less pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 43
4.11 Graf porovnávající náhodná data řazena funkcí my_compare pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 43
4.12 Graf porovnávající náhodná data řazena funkcí slow_compare pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 44
4.13 Graf zobrazující počet porovnání náhodných dat pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort 44
4.14 Graf porovnávající algoritmy gfx::timsort, std::sort, std::stable_sort a timsort na různých datech o stejné velikosti 46

4.15 Graf porovnávající algoritmy timsort_rev, timsort, it_merge_2_sort_rev a it_merge_2_sort na náhodných datech . 47
4.16 Graf porovnávající algoritmy timsort_rev, timsort, it_merge_2_sort_rev a it_merge_2_sort na několika seřazených částech 48
4.17 Porovnání algoritmů it_merge_4b_sort, it_merge_4a_sort, it_merge_3_sort a it_merge_3_sort na náhodných datech typu LargeTestClass 49
4.18 Porovnání algoritmů it_merge_4b_sort, it_merge_4a_sort, it_merge_3_sort a it_merge_3_sort na náhodných datech typu integer 49
4.19 Porovnání algoritmů it_merge_4b_sort, it_merge_4a_sort, it_merge_3_sort a it_merge_3_sort na náhodných datech typu integer pomocí slow_compare 50
4.20 Zrychlení merge_sort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodná čísla 53
4.21 Zrychlení merge_sort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodná čísla 54
4.22 Zrychlení timsort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodná čísla 55
4.23 Zrychlení timsort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodná čísla 56
4.24 Zrychlení merge_sort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodné stringy 57
4.25 Zrychlení merge_sort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodné stringy 58
4.26 Zrychlení timsort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodné stringy 59
4.27 Zrychlení timsort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodné stringy 60
4.28 Porovnání doby běhu algoritmů pro 2 vlákna a různá data o stejné velikosti 61
4.29 Porovnání doby běhu algoritmů pro 4 vlákna a různá data o stejné velikosti 62
4.30 Porovnání doby běhu algoritmů pro 8 vláken a různá data o stejné velikosti 63
4.31 Porovnání doby běhu algoritmů pro 16 vláken a různá data o stejné velikosti 64
4.32 Porovnání doby běhu algoritmů pro 20 vláken a různá data o stejné velikosti 65

Seznam tabulek

2.1 Porovnání počtu porovnání galloping a lineárního prohledávání . . 9

4.1 Porovnání funkce less a my_compare u stringů 47
4.2 Porovnání počtu porovnání algoritmů it_merge_2_sort, it_merge_3_sort, it_merge_4a_sort a it_merge_4b_sort 48
4.3 Poměr doby běhu porovnávacích funkcí less a my_compare 51
4.4 Porovnání paralelních algoritmů pro pole obsahující několik seřazených posloupností 52

Úvod

Řadící algoritmy jsou všude kolem nás. Umožňují nám nejen ukazovat v jakém vztahu jsou různé objekty, ale pomáhají nám v nich i hledat. Ať už to je srovnávání cen v e-shopech a nebo řazení šanonů podle jmen. Zdá se, že řadící algoritmy potřebujeme čím dál tím více, abychom se v tomto komplikovaném světě mohli orientovat.

Nacházení nových optimalizací je proto velmi důležité pro vytváření rychlejších, paměťově efektivnějších a všestrannějších řadících algoritmů. Zároveň je důležité již nalezené optimalizace maximálně zjednodušit. Usnadní se tím implementace, sníží počet možných chyb a je jednodušší provádět důkaz korektnosti optimalizovaného algoritmu.

Tato práce se zabývá možnostmi sekvenčních optimalizací a paralelizací řadícího algoritmu Timsort. Tento algoritmus je založený na algoritmu Mergesort s několika optimalizacemi. Ty například zajišťují, že je Timsort adaptivní řadící algoritmus a zároveň vyžaduje méně paměti než klasický Mergesort. Proto se algoritmy založeny na principech Timsortu využívají například v jazyce Python, Java, Rust a Swift.

V této práci se snažím najít nějaké další potencionální optimalizace. Také zkouším jak by mohla fungovat paralelní verze tohoto algoritmu. Správně implementovaná paralelizace nám umožní dále zrychlit algoritmus za využití více vláken.

Při výběru tématu a studiu Timsortu a dalších řadících algoritmů, které se reálně používají v různých jazycích a knihovnách mě zaujalo, jak jsou promyšlené do nejmenšího detailu a že i malá změna algoritmu může udělat obrovský rozdíl pro určitý typ dat.

V následující sekci jsou popsány cíle této práce. Dále je vysvětleno, jak algoritmus Timsort funguje a jaké optimalizace se běžně používají. Poté navrhuji, jaké optimalizace lze přidat a nakonec se věnuji jejich implementaci a testování. Na závěr jsou zhodnoceny výsledky testování.

Kapitola 1

Cíl práce

1.1 Sekvenční optimalizace algoritmu

V rámci vylepšení algoritmu je důležitá jejich optimalizace. U řadících algoritmů bývá nejdůležitější jejich rychlost. Dále je důležitá i jejich paměťová složitost. Můžeme optimalizovat obecně celý algoritmus pro všechny vstupy a nebo se zaměřit pouze na vstupy, které dopadají špatně. Také můžeme optimalizovat tím, že zrychlíme některé operace – například přístup do paměti pomocí lepšího využití cache paměti.

Timsort je navržen tak, aby využíval co nejvíce struktur, které se v datech přirozeně vyskytují. Je tedy velmi těžké naleznout vylepšení samotného algoritmu, které by jej urychlilo. V mé práci se proto snažím upravit algoritmus tak, aby lépe využíval cache paměť.

1.2 Paralelizace algoritmu

Dalším způsobem jak je možné zrychlit algoritmus je jeho paralelizace. Při paralelizaci se program spustí ve více vláknech. Pokud je dobře navržena dojde tím k jeho zrychlení, protože se může vykonávat více částí programu najednou. Timsort vynucuje sekvenční chování díky svým invariantům. Cílem je vymyslet různé způsoby jak lze přes tento problém algoritmus paralelizovat.

1.3 Testování

Posledním cílem této práce je otestovat mnou navržené verze algoritmu. Pro zajištění co nejpřesnějších měření bude testování prováděno na školním serveru STAR. Tento server je přímo určen k objektivnímu měření paralelních programů.

Kapitola 2

Analýza a návrh

2.1 Popis algoritmu Timsort

Algoritmus Timsort je hybridní, adaptivní a stabilní řadící algoritmus se složitostí $\mathcal{O}(n \log n)$ odvozený z Mergesortu a Insertion sortu.[1] Funguje velmi dobře na datech reálného světa a proto se jeho principy používají v řadících algoritmech pro Pythonu[2], Rustu[3], Javě[4], V8 (Javascript engine)[5] a Swiftu[6]. Pro své dobré používání cache paměti a stabilitu se hodí především na neprimitivní typy[7]. Zbytek této sekce je založen především na detailním popisu Timsortu jeho tvůrcem Timem Petersem, včetně odůvodnění některých rozhodnutí a příkladů.[8, 9, 10, 11]

V první části algoritmu se hledají takzvané „natural runs", což jsou již seřazené části pole. Run může být vzestupný $a_0 \leq a_1 \leq a_2$ nebo ostře klesající $a_0 > a_1 > a_2$. Ostře klesající musí být, aby algoritmus zůstal stabilní, neboť se na tento run provede naivní in-place otočení posloupnosti. Stejné hodnoty by se jinak prohodily.

Dále existuje hodnota `minRun` určující minimální velikost pro run. Pokud jí run nedosáhne je doplněn dalšími prvky pomocí Binary insertion sortu. V poli s náhodnými daty to tedy znamená, že runy mají stejné délky při slučování, čímž získáme nejmenší nutný počet slučování. Navíc využijeme faktu, že pro malá pole je výhodnější použít Insertion sort než Mergesort, kvůli vyšší rychlosti. Výpočet hodnoty `minRun` pro $N < 64$ je roven N. Pokud je N násobkem dvěma lze použít hodnoty $16, 32, 64, 128$, které jsou zhruba ekvivalentní. Vyšší hodnoty zpomalují Insertion sort a nižší zpomaluje počet volání funkce slučování. Použití násobku dvěma je důležité, aby slučování runů bylo vyrovnané. Pokud ale N není násobkem dvěma, je potřeba se zamyslet nad tím, aby se zbytečné neslučovaly velká pole s malými. Proto se jako `minRun` používá hodnota taková, aby $\frac{N}{\texttt{minRun}} \in 32, ..., 65$ a zároveň aby $\frac{N}{\texttt{minRun}}$ byla mocnina dvojky, anebo ostře menší než mocnina dvojky, pokud nemůže být přesně mocnina dvojky.

Představme si, že bychom takto nevypočítávali minimální délku runu a použili fixní hodnotu 32. Uvažujme pole o délce 2112 prvků. Pokud jsou data náhodná, dostaneme velmi pravděpodobně 66 runů o délce 32. Sloučení prvních 64 runů bude dokonale vyvážené, avšak poté nastane situace, kdy budeme chtít sloučit runy o velikosti 2048 a 64. To je velmi neefektivní – je potřeba daleko více porovnání a kopírování dat.

Nyní uvažujme výpočet minimální délky runu, tak je popsán výše. Jako hodnota `minRun` nám vyjde číslo 33. S náhodnými daty opět velmi pravděpodobně získáme runy právě s minimální délkou - tím pádem jich bude 64. Jelikož je počet runů násobkem dvou, dostaneme jejich vyvážené slučování.

2.1.1 Slučovací strategie

Během hledání runů se může stát, že délky runů mohou být velmi rozdílné. Proto je potřeba organizovat slučování runů. Abychom zachovali stabilitu algoritmu můžeme slučovat pouze runy, které jsou vedle sebe. Uvažujme tedy, že máme 3 runy A, B, C, které chceme sloučit. Máme na výběr $(A + B) + C$ nebo $A + (B + C)$. Během slučování je potřeba najít kompromis, kdy ho provést, protože chceme využít dvou protichůdných vlastností. Jednak chceme, aby slučování bylo provedeno co možná nejpozději, abychom mohli využít struktury dat, které mohou přijít později. Zároveň chceme slučování provést co nejdříve, abychom využili dobře cache paměť, protože data byla nedávno použita. Udržování informací o runech také spotřebovává paměť navíc.

Jako dobrý kompromis se ukázalo udržovat dva invarianty na poslední 3 runy na zásobníku. Pro jejich délky A, B, C by pak mělo platit:

1. $A > B + C$
2. $B > C$

První invariant zajišťuje, že délky runů rostou alespoň tak rychle, jako Fibonacciho čísla. Proto nám stačí malý zásobník i pro velmi velká pole. Druhý invariant zajišťuje, že runy jsou seřazeny podle délek od největší po nejmenší. Tím získáváme dobré slučování, protože slučujeme runy s nejvíce podobnou délkou. Pokud je některý invariant nesplněn, po prvním sloučení se může pořád stát, že invarianty pořád nejsou splněny – viz příklad. Proto je potřeba toto provádět, dokud invarianty neplatí.

Pokud je $A \geq B+C$, tak menší z A a C je sloučen s B. Pokud je $A = C$, tak je preferováno C, z důvodu nedávného použití a tím větší pravděpodobnosti, že je pořád v cache paměti. Pokud tedy jsou poslední 3 záznamy na zásobníku následující: $A = 30$, $B = 20$ a $C = 10$, potom je B sloučeno s C a výsledek na zásobníku vypadá takto: $A = 30$, $BC = 30$.

Jak si můžeme všimnout, tak po sloučení v příkladu pořád neplatí invariant číslo dvě a musíme tedy slučovat znovu, dokud nejsou oba invarianty platné.

2.1.2 Slučovací algoritmus

Na slučování se používají dvě metody, které jsou si podobné. Jedna se stará o situaci $A \leq B$ a druhá o $A > B$. Tyto metody neví jaká je struktura dat, avšak při sloučení se dá zjistit, když jedna slučovaná strana vyhrává častěji.

Nejprve se porovná první pár prvků. Při každém porovnání se počítá, který run vyhrál kolikrát v řadě. Pokud tento počet dosáhne hodnoty `minGallop`, změní se klasické slučování na galloping mód. Při gallopingu módu se hledá v B, kam patří `A[0]` a překopírují se všechny prvky před tímto bodem. Poté se hledá v A, kam patří `B[0]` a přesune se zase celý blok prvků. Takto se pokračuje a střídá, dokud nalezené shluky jsou větší než hodnota `minGallop`. Poté se zas vracíme ke klasickému slučování po jednom páru.

Jedním z vylepšení přechodu do galloping módu, které Timsort využívá, je nepoužívat pořád stálý `minGallop` při vstupu do slučovacích funkcí, ale upravovat jeho hodnotu na základě předchozích dat. Funkce `merge_lo` a `merge_hi` tedy `minGallop` zvětšují či zmenšují v závislosti na tom, jestli se vyplácí nebo ne.

2.1.3 Slučovací paměť

Slučovací algoritmus je navržen tak, aby potřeboval co nejméně paměti a byl co nejrychlejší. Byla proto zvolena varianta, kde je potřeba $\min(A, B)$ paměti. Přestože existují in-place slučovací algoritmy, nebyly vybrány, protože jsou příliš složité a pomalé pro praktické využití.

Pokud je $A < B$ (funkce `merge_lo`), zkopíruje se A do pomocného pole a začne se slučovat s B na původní místa A. Na konci případně dokopírujeme zbytek pomocného pole či B jako v běžném Mergesortu. Pokud je $A > B$ (funkce `merge_hi`) algoritmus funguje obdobně, se změnou, že do pomocného pole zkopírujeme B a slučujeme z prava doleva –-na původní místo B. Pro $A = B$ lze využít obě funkce.

Abychom ještě snížili velikost pomocného pole, vyhledá se, kde první prvek B skončí v A a kde poslední prvek A skončí v B. Stačí nám pak sloučit prvky mezi těmito body. Ostatní už jsou na svém místě. Toto o trochu něco zpomalí algoritmus pro náhodná data, avšak může velmi zrychlit pokud náhodná nejsou.

2.1.4 Galloping

Galloping neboli exponenciální vyhledávání je technika podobná binárnímu vyhledávání. Je v Timsortu využívána, protože může ve vybraných případech překonat binární vyhledávání, když je hledaný prvek blízko začátku pole. Exponenciální vyhledávání pracuje v časové složitosti $\mathcal{O}(\log i)$, kde i je index prvku, zatímco binární vyhledávání je v $\mathcal{O}(\log n)$, kde n je počet prvků v poli.[12]

Předpokládejme bez újmy na obecnosti, že run A je kratší než run B. V galloping módu porovnáváme prvek `A[0]` s prvky `B[0], B[1], B[3], B[7], ..., B[`$2^j - 1$`]`, dokud nenajdeme hodnotu k takovou, že `B[`$2^{k-1} - 1$`]` $\leq$ `B[`$2^k - 1$`]`. Toto vyžaduje $\log B$ porovnání.

Po nalezení takového k se region nejistoty zredukuje na $2^{k-1} - 1$ za sebou jdoucích prvků a binární vyhledávání potřebuje přesně $k - 1$ porovnání pro nalezení správné pozice. Poté zkopírujeme všechny prvky z B do toho bodu a bod `A[0]`. Nezáleží na tom, kde `A[0]` patří v B, kombinace gallopingu a binárního vyhledávání toto místo najde za méně než $2 * \log B$ porovnání.

Při použití binárního vyhledávání bychom našli správnou pozici pro `A[0]` v B nejpozději za ceiling$(\log{(B + 1)})$ porovnání. Na rozdíl od toho galloping mód může najít pozici rychleji, zejména pokud je prvek blíže začátku pole.

Pokud jsou data náhodná a runy mají stejnou délku, pak `A[0]` patří do `B[0]` v 50% případů, do `B[1]` v 25% případů atd. Vyhrávající subrun o délce B má tedy pravděpodobnost pouze $\frac{1}{2^{k+1}}$. Delší vyhrávající subruny jsou tedy extrémně nepravděpodobné.

Pokud ovšem data mají nějakou strukturu nebo obsahují mnoho duplikátů, mají dlouhé vyhrávající subruny daleko větší pravděpodobnost. Snížení počtu porovnání z $\mathcal{O}(B)$ na $\mathcal{O}(logB)$ je v těchto případech velmi výhodné.

Galloping mód v Timsortu je kompromis, který spočívá v tom, že pokud není dlouhý vyhrávající subrun, tak rychle ukončí a přepne na slučovací mód. Pokud je dlouhý vyhrávající subrun, tak je poté galloping mód velmi efektivní. Tímto způsobem se snaží optimalizovat proces slučování a snížit počet nutných porovnání.

Nicméně galloping má i své nevýhody. Zvyšuje například režii na porovnání, což může ve výsledku zpomalit celý algoritmus, pokud není dobře vyvážen s běžným slučováním. Galloping může také potřebovat více porovnání než obyčejné lineární prohledávání. Pokud `A[0]` patří před `B[0]`, pak galloping i lineární prohledávání potřebují 1 porovnání. Galloping ovšem vyžaduju navíc i volání funkce. Pokud `A[0]` patří přímo před `B[1]`, pak galloping i lineární prohledávání vyžadují 2 porovnání. Třetí porovnání galloping je s prvkem `B[3]` a pokud `A[0]` patří před něj, pak musí galloping ještě zjistit jestli patří na `B[2]` nebo `B[3]`. Tím pádem potřebuje 4 porovnání. Pokud ale prvek patřil na `B[2]`, pak lineární prohledávání potřebuje pouze 3 porovnání. Galloping tedy potřeboval o 33% více porovnání a to je obrovský rozdíl, zejména pokud je porovnávací funkce pomalá. Toto můžeme vidět v tabulce 2.1.

Obecně potřebuje lineární prohledávání $i + 1$ porovnání a galloping potřebuje $2 *$ floor$(\log i) + 2$ porovnání. Galloping nevyhraje dokud je $i = 6$ a dokonce prohraje pokud je $i = 2$ a $i = 4$. Jelikož v náhodných datech očekáváme malou hodnotu i, mohlo by toto Timsort velmi zpomalit. Proto je jako počáteční hodnota `minGallop` zvolena právě hodnota 7. I přestože se v náhodných datech může přepnout do galloping módu, je velmi pravděpodobné, že se z něj zase velmi rychle přepne pryč. Pokud však jsou data gallopingu

Tabulka 2.1: Porovnání počtu porovnání galloping a lineárního prohledávání

index v B kam patří `A[0]`	počet porovnání lineárního prohledávání	počet porovnání gallopingu	počet porovnání binárního vyhledávání	celkem porovnání gallopingu
0	1	1	0	1
1	2	2	0	2
2	3	3	1	4
3	4	3	1	4
4	5	4	2	6
5	6	4	2	6
6	7	4	2	6
7	8	4	2	6
8	9	5	3	8
9	10	5	3	8
10	11	5	3	8
11	12	5	3	8
...				

nakloněna, je hodnota 7 zase zbytečně velká a proto si Timsort tuto hodnotu mění na základě již seřazených dat.

Takto funguje galloping mód ve funkci `merge_lo`. Obdobně funguje i ve funkci `merge_hi` s tím rozdílem, že se hledá od konce. Tím ušetříme spoustu porovnání neboť chceme začínat co nejblíže hledanému prvku. Proto mají tyto funkce parametr hint, který určuje, kde se má začít.

Ve výsledku galloping může zrychlit algoritmus zejména v případech, kdy můžeme využít kopírování větších částí pole. V jiných případech však může způsobit zpomalení, pokud se často přechází mezi módy slučování. Implementace galloping módu také zvyšuje složitost kódu Timsortu, což může vést ke stížení pochopení a k vyšší pravděpodobnosti chyb v implementaci. I přes tyto potenciální komplikace je galloping v Timsortu často užitečný pro zrychlení algoritmu v určitých situacích. Je tedy důležité uvažovat jaká data budeme chtít řadit a podle toho zvážit jestli se galloping vyplatí použít, nebo ne.

2.2 Návrh optimalizací

Algoritmus Timsort obsahuje následující části – hledání runů a slučování s gallopingem. U hledání runů pravděpodobně optimalizace moc neobjevíme – samotné procházení je $\mathcal{O}(n)$, pokud je run o délce r ostře klesající, otočíme ho za $\frac{r}{2}$, což rychleji také nepůjde. Máme tedy na výběr jestli run budeme procházet zleva do prava nebo z prava do leva. Další šance na zrychlení u hledání runu by mohla být například změna řadícího algoritmu pro doplnění minimální velikosti runu.

U slučování je prostor pro optimalizace největší. Hodlám se zajímat i o optimalizace, které algoritmus zrychlí i za cenu vyšší potřeby paměti.

2.2.1 Slučování více runů

Místo toho, abychom slučovali pouze dva sousední runy, můžeme zkusit optimalizovat Timsort tak, aby slučoval více runů najednou. Tato optimalizace by mohla zvýšit rychlost algoritmu, protože by mohla snížit počet nutných sloučení a využít efektivněji cache procesoru.[13] Zároveň hrozí, že algoritmu bude pomalejší, kvůli většímu počtu porovnání.[14, 15] Nevýhodou je potřeba větší paměti pro ukládání runů, protože nové invarianty by byly komplikovanější a měly by tím daleko větší režii. Další nevýhodou je ztráta gallopingu, který se u jednoduchého rozšíření popsaného níže nevyplatí.

Algoritmus slučování dvou runů můžeme jednoduše rozšířit na slučování tří runů tak, že porovnávat tři hodnoty mezi sebou a nejmenší uložíme do výsledku. Jakmile jeden run dojde na konec, použijeme na zbytek obyčejné slučování dvou runů. Obdobně lze rozšířit pro slučování ještě více runů.

Runy, jejichž počet je mocninou čtyř, můžeme slučovat ještě jiným způsobem. Například pro slučování čtyř runů lze použít sloučení dvakrát dvou runů do pomocného pole a poté sloučení těchto dvou runů zpět do původního pole. Tím ušetříme kopírování výsledků z dočasného pole do původního. Navíc protože ve skutečnosti používáme pouze slučování dvou runů, můžeme využít implementace z Timsortu, která obsahuje i galloping.

Sloučením více runů najednou můžeme teoreticky získat zrychlení.[15] Řadící algoritmy používající k slučování polí najednou nazýváme k-way mergesort. Ačkoliv asymptotická složitost je $\mathcal{O}(n * \log n)$, při podrobnější analýze k-way mergesortů[16] zjistíme následující:

$$\text{Pro } k = 2\text{: } \tfrac{n}{2} * 2 + \tfrac{n}{4} * 4 + ... + 1 * n = n * \log_2 n,$$
$$\text{pro } k = 3\text{: } 2 * \tfrac{n}{3} * 3 + 2 * \tfrac{n}{9} * 9 + ... + 2 * 1 * n = 2 * n * \log_3 n.$$

Násobení číslem 2 u $k = 3$ je způsobeno potřebným počtem porovnání pro zjištění minima. Obecně tedy lze říct, že pro k-way mergesort platí: počet_porovnání_k_prvků $* \, n * \log_k n$. Můžeme najít naivně minimum ve všech k polích. Tím by bylo porovnání k prvků za $k - 1$. Můžeme však využít haldu a nebo loser tree získat tak porovnání prvků za $\mathcal{O}(\log k)$.[17]

K-way merge se využívá u tzv. external sortů, kde se celá data nevejdou do RAM paměti počítače. Při slučování více polí najednou a tím omezíme přístup do pomalejší externí paměti jako jsou pevné disky.[18] Stejnou výhodu bychom mohli získat i u využívání cache paměti.

2.2.2 Timsort s detekcí runů od konce

Klasický Timsort prochází pole od začátku a hledá runy. Následně je postupně slučuje do jednoho seřazeného pole. Jedna z možných optimalizací je hledat runy procházením pole od zadu. Tento způsob procházení je implementován v jazyce Rust s odůvodněním, že častěji nastává slučování zleva doprava. Při srovnávání slučování právě toto údajně vyšlo lépe a proto je výhodnější hledá runy od konce pole.[3] Je možné, že pokud ke zrychlení dojde, tak může záležet také na vstupních datech.

2.3 Paralelizace Timsortu

2.3.1 První návrh

Paralelní hledání runů lze provést tak, že rozdělíme pole na menší části, které budou zpracovávány různými vlákny. Každé vlákno pak bude zodpovědné za seřazení své části pole. Hlavní vlákno nakonec provede ještě jednou Timsort a tím bude pole seřazené. Jelikož je Timsort adaptivní a identifikuje runy, tak poslední řazení bude ve výsledku pouze sloučení seřazených částí. Protože spouštění vláken má svoji režii, je potřeba rozdělit pole na tak velké části, aby se vyplatilo jednotlivá vlákna spouštět. V případě malých polí by jinak mohla být paralelní verze výrazně pomalejší. Dále je potřeba brát v potaz kolik vláken může procesor zpracovávat. Velkou výhodou je velmi jednoduchá implementace pomocí OpenMP.[19]

2.3.2 Druhý návrh

Druhou možností paralelizace lze provést tak, že každé vlákno sloučí dvojici runů a výsledné sekvence se následně sloučí ve vyšších úrovních. Velkou nevýhodou je potenciální nerovnoměrné zatížení vláken, pokud délky runů nejsou vyvážené. Navíc je třeba zajistit správnou synchronizaci a koordinaci mezi vlákny. Tato možnost paralelizace je proto pravděpodobně horší a navíc komplikovanější.

2.3.3 Třetí návrh

Třetí možností paralelizace je inspirována implementací paralelního Timsortu od Saurabha Sooda[20] a je velmi podobná druhé možnosti paralelizace. Na rozdíl od předchozího návrhu budeme během slučování runů udržovat pole obsahující informace o všech runech. Toto pole bude pro všechny spuštěné funkce vypadat stejně, ale prvky se v runech mohou prohazovat. Jako první krok je potřeba nalézt všechny runy. Poté uvažujeme pole s runy jako pole prvků, které chceme seřadit pomocí obyčejného Mergesortu. Rozdělujeme a rekurzivně voláme funkci, dokud nemáme pouze jeden run. O něm už víme, že je seřazený. Následně pak slučujeme dvojice těchto runů. Výhodou je poměrně jednoduchá implementace velmi podobná Mergesortu s možností jednoduché paralelizace volaných rekurzivních funkcí pomocí direktivy task z OpenMP.

Kapitola 3

Realizace

Veškerý kód je napsán v jazyce C++. Funkce se volají stejně jako řadící funkce ve standardní C++ knihovně – pomocí iterátorů a nepovinné porovnávací funkce. Jako výchozí porovnávací funkce je použita `std::less<>()`.

```cpp
// Příklad volání Timsortu
std::vector<int> numbers = {5, 3, 6, 8, 1, 2, 3};
timsort(numbers.begin(), numbers.end());
timsort(numbers.begin(), numbers.end(), std::greater<>());
```

Realizoval jsem tyto algoritmy:

- Klasický Timsort
- Timsort s vyhledáváním runů od konce
- Timsort s 2-way, 3-way a 4-way merge funkcí bez invariantů slučující iterativně a rekurzivně
- Dva paralelní Timsorty pomocí OpenMP
- Dva paralelní Mergesorty pomocí OpenMP s různými slučovacími funkcemi

3.1 Realizace klasického Timsortu

Algoritmus jsem se pokusil co nejvíce přiblížit původní implementaci pro Python. Ta využívá galloping a šetří maximum paměti při slučování. Jako základ posloužila implementace v Pythonu[2] a v Javě[4]. Tato verze Timsortu je k nalezení v souboru `timsort.hpp`.

3.1.1 Pomocná struktura Slice

Tato struktura ukládá informace o runu. První proměnná index obsahuje iterátor ukazující na první prvek runu. Proměnná length pak udržuje informaci o délce runu.

```
// Struktura Slice
template <class It>
struct Slice {
    It index;
    int length;
};
```

3.1.2 Třída TimSort

Tato třída udržuje všechny potřebné proměnné k tomu, aby Timsort mohl fungovat. Má také na starosti hledání runů a jejich následné slučování. Požaduje 2 template argumenty – první `It` je typ iterátoru a druhý `Compare` je typ porovnávací třídy. Dále obsahuje následující proměnné:

runs pole používané jako zásobník obsahující informace o runech

runsLen aktuální velikost zásobníku **runs**

tmp_ vektor sloužící jako dočasné pole při slučování

index iterátor ukazující na první prvek aktuálně zpracovávaného runu

first iterátor ukazující na první prvek řazeného pole

last iterátor ukazující na poslední prvek řazeného pole

comp porovnávací funkce

length celková délka řazeného pole

minGallop hodnota podle, které se určuje přechod mezi gallopingem a běžným slučováním, na začátku nastavena na 7, poté se upravuje podle dat

minRun minimální velikost runu

A následující metody:

gallop_left hledá zleva kam patří hledaný prvek a vrátí jeho index

gallop_right hledá zprava kam patří hledaný prvek a vrátí jeho index

merge_lo slučuje prvky obyčejným slučování a případně přejde do gallopingu, paměťově efektivní pokud je první run kratší

merge_hi slučuje prvky obyčejným slučování a případně přejde do gallopingu, paměťově efektivní pokud je druhý run kratší

merge_at sloučí run i a $i+1$, rozhodne kolik prvků je už na správném místě a zvolí lepší variantu z `merge_lo` a `merge_hi`

binary_insertion_sort doplní chybějí část run pomocí Binary insertion sortu

compute_min_run vypočítá `minRun` aby počet runů byl ostře nižší nebo rovnu mocnině dvojky v rozmezí 32 až 64, případně vrátí délku pole pokud je menší než 64

```
// výpočet proměnné minRun
int compute_min_run(int n) {
    int r = 0;          /* becomes 1 if any 1 bits are shifted
    ↪  off */
    while (n >= 64) {
        r |= n & 1;
        n >>= 1;
    }
    return n + r;
}
```

find_run nalezne další run, pokud je seřazen opačně, tak ho otočí a pokud je krátký, tak ho doplní na velikost `minRun`

sort jediná veřejně přístupná metoda třídy, započne seřazení zadaného pole, kontroluje invarianty

Dále jsou mimo třídu dvě obalovací funkce, které zařizují stejné volání jako má `std::sort`.

```
// obalovací funkce Timsortu s porovnávací funkcí
template <class It, class Compare>
void timsort(It first, It last, Compare comp) {
    TimSort<It, Compare> tim(first, last, comp);
    tim.sort();
```

```
}

//obalovací funkce Timsortu bez porovnávací funkce
template <class It>
void timsort(It first, It last) {
    timsort<It, decltype(std::less<>())>(first, last, std::less<>());
}
```

3.1.3 Hledání runů

O hledání runů se stará metoda `find_run`. Ta porovná první a druhý prvek aktuálního runu a rozhodne jestli jde o klesající run nebo neklesající. Jakmile porovná s prvkem, který poruší sekvenci, tak klesající run naivně otočí a v případě, že délka runu není dostatečná, doplní ho na velikost `minRun` pomocí metody `binary_insertion_sort`. Nakonec run uloží na stack `runs`.

3.1.4 Slučování runů

Slučování začíná v metodě `sort`. Ta nejprve najde run pomocí `find_run` a poté kontroluje stav invariantů. V případě, že je některý z invariantů porušen, zavolá `merge_at` a příslušné runy spojí. Pokud invarianty nejsou porušeny hledá další run, dokud nedojde na konec pole. Až se nalezne poslední run, začnou se všechny runy spojovat do jednoho. Tím nám vznikne seřazené pole.

```
// metoda sort ve třídě TimSort
void sort() {
    while (index != last) {
        find_run();
        while (runsLen > 1) {
            int n = runsLen - 2; //check invariants
            if ((n > 0 && runs[n - 1].length <= runs[n].length + runs[n + 1].length) ||
                (n > 1 && runs[n - 2].length <= runs[n - 1].length + runs[n].length)) {
                if (runs[n - 1].length < runs[n + 1].length)
                    --n;
            }
            else if (n < 0 || runs[n].length > runs[n + 1].length) {
                break;
            }
            merge_at(n);
        }
```

```
    }

    // merge remaining
    while (runsLen > 1) {
        int n = runsLen - 2;
        if (n > 0 && runs[n - 1].length < runs[n + 1].length)
            --n;
        merge_at(n);
    }
}
```

Slučování tedy postupuje dál do metody `merge_at`. Ta vytvoří nový run ze zadaného a z runu co následuje. Pokud byl zadán třetí run na stacku, je potřeba první posunout na místo druhého, aby nevznikla mezera. Poté najde kolik prvků z prvního runu je už na svém místě a kolik prvků z druhého runu je na svém místě a ignoruje je. Na zbytek zavolá `merge_lo`, pokud je délka prvního runu menší nebo rovna délce druhého runu, jinak zavolá `merge_hi`.

Pokračujeme do metod `merge_lo` a `merge_hi`. Ty jsou si velice podobné. `Merge_lo` překopíruje první run do pomocného pole a poté začne porovnávat druhý run s pomocným polem a vkládat výsledné prvky zleva na místo prvního runu. `Merge_hi` překopíruje druhý run a vkládá výsledné prvky od konce. Metody fungují i pokud jsou délky opačně než jsem specifikoval na konci minulého odstavce. Přijdeme tím ale o jistotu, že na dočasné pole je potřeba pouze $\min(a, b)$ paměti, kde a je délka prvního runu a b druhého.

V těchto metodách probíhá slučování klasickou metodou, kde porovnáme dva prvky a menší z nich uložíme do výsledku. Pokud ovšem jeden vyhraje víckrát než je hodnota `minGallop`, přepne se do galloping módu, kde kopírujeme celý blok dat na správné místo. Jakmile je velikost tohoto bloku menší než `minGallop`, přepne se do klasického slučování. Přechody mezi těmito módy tedy určuje proměnná `minGallop`. Tyto funkce také tuto proměnnou zvyšují, či snižují v závislosti na tom jak často se módy přepínají.

3.1.5 Galloping

O galloping se starají dvě metody – `gallop_left` a `gallop_right`. Jak již název napovídá, tak `gallop_left` začíná zleva a `gallop_right` zprava. Jinak jsou téměř identické. Popíši proto pouze metodu `gallop_left`.

Jak můžeme vidět v ukázce jejího kódu, tak přijímá parametr `_first`, odkazující na první iterátor pole, ve které chceme galloping použít. Dále zde najdeme parametr `key` určující prvek, kterému chceme nalézt místo v poli. Třetí parametr je `len` určující délku prohledávaného pole a poslední parametr `hint` určuje na jakém indexu má galloping začít.

V samotné metodě nalezneme proměnné `ofs` a `lastOfs`, které určují region ve kterém se následně má provést binární vyhledávání. Prvním krokem galloping algoritmu je rozhodnout jestli je hledaný prvek vlevo nebo vpravo od počátečního místa určené parametrem `hint`. Poté stačí najít region `[lastOfs, ofs)` ve kterém se hledaný prvek nachází. V tomto regionu na konci vyhledáme prvek pomocí binárního vyhledávání.

```
//metoda gallop_left ve třídě TimSort
int gallop_left(const It & _first, T key, int len, int hint) {
    int lastOfs = 0;
    int ofs = 1;
    if (comp(*(_first + hint), key)) { // key > a[base + hint]
        // Gallop right until a[base+hint+lastOfs] < key <= a[base+hint+ofs]
        int maxOfs = len - hint;
        while (ofs < maxOfs && comp(*(_first + (hint + ofs)), key)) {
            lastOfs = ofs;
            ofs = (ofs << 1) + 1;
            if (ofs <= 0)   // int overflow
                ofs = maxOfs;
        }
        if (ofs > maxOfs)
            ofs = maxOfs;
        lastOfs += hint;
        ofs += hint;
    } else { // key <= a[base + hint]
        // Gallop left until a[base+hint-ofs] < key <= a[base+hint-lastOfs]
        int maxOfs = hint + 1;
        while (ofs < maxOfs && !comp(*(_first + (hint - ofs)), key)) {
            lastOfs = ofs;
            ofs = (ofs << 1) + 1;
            if (ofs <= 0)   // int overflow
                ofs = maxOfs;
        }
        if (ofs > maxOfs)
            ofs = maxOfs;
        int tmp = lastOfs;
        lastOfs = hint - ofs;
        ofs = hint - tmp;
    }
    /*
     * We now know that a[b + lastOfs] < key <= a[b + ofs] is true. Do a binary
```

```
     * search, to find exact position.
     */
    lastOfs++;
    // binary search
    It pos = std::lower_bound(_first + lastOfs, _first + ofs,
    ↪  key, comp);
    return pos - _first;
}
```

3.2 Slučování více runů najednou (k-way merge)

Všechny zde popsané algoritmy jsou k nalezení v souboru `merging.hpp`. Nejprve jsem naimplementoval vylepšný rekurzivní Mergesort (`merge_2_sort`). Vylepšení spočívá v použití Insertion sortu pro $n < 32$, podobně jako u Timsortu. Poté jsem se implementoval rekurzivní 3-way (`merge_3_sort`) a 4-way (`merge_4b_sort`) mergesort. 2-way mergesort jednoduše rozšíříme na 3-way mergesort tím, že přidáme přídáme hledání minima třetího pole a jakmile jedno pole celé projdeme, tak se problém redukuje na 2-way slučování u kterého stačí zavolat správné parametry. Obdobně lze rozšířit 3-way mergesort na 4-way. U mých k-way mergesortů jsem zůstal u naivního hledání minima ve všech k polích. Pro větší k toto však nelze použít z důvodu dlouhého a nepřehlednému kódu a také proto, že velký počet potřebných porovnání začne algoritmus velmi zpomalovat způsobem popsaným v analýze. V následující ukázce si můžeme povšimnout ještě funkce `check_small_and_sort`, která pouze kontroluje délku pole a pokud je dostatečně malé, tak ho seřadí pomocí Binary insertion sortu.

```
//slučovací funkce 2-way merge
template <class It, class Compare>
void merge_2(It first1, It last1, It first2, It last2, It dest,
↪  Compare comp) {
    while (first1 != last1 && first2 != last2) {
        if (comp(*first2, *first1))
            *(dest++) = *(first2++);
        else
            *(dest++) = *(first1++);
    }
    // Copy any remaining elements of the first run
    while (first1 != last1)
        *(dest++) = *(first1++);
    // Copy any remaining elements of the second run
    while (first2 != last2)
        *(dest++) = *(first2++);
}
```

```
// rekurzivní 2-way mergesort
template <class It, class Compare>
void merge_2_sort(It first, It last, Compare comp) {
    int64_t size = std::distance(first, last);
    if (check_small_and_sort(first, last, comp, size, 32))
        return;
    It mid = std::next(first, size / 2);
    merge_2_sort(first, mid, comp);
    merge_2_sort(mid, last, comp);
    std::vector<typename It::value_type> tmp(std::distance(first, last));
    merge_2(first, mid, mid, last, tmp.begin(), comp);

    std::move(tmp.begin(), tmp.end(), first);
}
```

```
//slučovací funkce 3-way merge
template <class It, class Compare>
void merge_3(It first1, It last1, It first2, It last2, It first3, It last3, It dest, Compare comp) {
    // Merge the three runs into the temporary array
    while (first1 != last1 && first2 != last2 && first3 != last3) {
        if (comp(*first2, *first1)) {
            if (comp(*first3, *first2))
                *(dest++) = *(first3++);
            else
                *(dest++) = *(first2++);
        } else {
            if (comp(*first3, *first1))
                *(dest++) = *(first3++);
            else
                *(dest++) = *(first1++);
        }
    }

    if (first1 == last1)
        merge_2(first2, last2, first3, last3, dest, comp);

    if (first2 == last2)
        merge_2(first1, last1, first3, last3, dest, comp);

    if (first3 == last3)
```

```
        merge_2(first1, last1, first2, last2, dest, comp);
}

//rekurzivní 3-way mergesort
template <class It, class Compare>
void merge_3_sort(It first, It last, Compare comp) {
    int64_t size = std::distance(first, last);
    if (check_small_and_sort(first, last, comp, size, 32))
        return;

    It mid1 = std::next(first, size / 3);
    It mid2 = std::next(first, 2*(size / 3));
    merge_3_sort(first, mid1, comp);
    merge_3_sort(mid1, mid2, comp);
    merge_3_sort(mid2, last, comp);
    std::vector<typename It::value_type> tmp(std::distance(first, last));
    merge_3(first, mid1, mid1, mid2, mid2, last, tmp.begin(), comp);
    std::move(tmp.begin(), tmp.end(), first);
}

```

U 4-way mergesortu můžeme použít ještě jiné vylepšení. To spočívá v tom, že pokud chceme spojit pole `A`, `B`, `C` a `D`, můžeme nejdřív spojit `A` s `B` pomocí 2-way merge a `C` s `D` do pomocného pole. Poté sloučíme `AB` s `CD` do výsledného pole. Tím ušetříme zbytečné kopírování z pomocného pole do výsledného oproti použití 2-way mergesortu. Jméno funkce toho 4-way mergesortu je `merge_4a_sort` a implementaci její slučovací funkce můžeme vidět na ukázce níže.

```
// ukázka slučovací funkce pro merge_4a_sort
template <class It, class Compare>
void merge_4a(It first1, It last1, It first2, It last2, It first3, It last3, It first4, It last4, Compare comp) {
    std::vector<typename It::value_type> tmp(std::distance(first1, last4));
    It first = tmp.begin();
    It half = std::next(tmp.begin(), std::distance(first1, first3));
    It dest = tmp.begin();
    It finalDest = first1;
    It mid = half;
    It last = tmp.end();
```

```
    while (first1 != last1 && first2 != last2) {
        if (comp(*first2, *first1))
            *(dest++) = *(first2++);
        else
            *(dest++) = *(first1++);
    }
    // Copy any remaining elements of the first run
    while (first1 != last1)
        *(dest++) = *(first1++);
    // Copy any remaining elements of the second run
    while (first2 != last2)
        *(dest++) = *(first2++);
    while (first3 != last3 && first4 != last4) {
        if (comp(*first4, *first3))
            *(dest++) = *(first4++);
        else
            *(dest++) = *(first3++);
    }
    // Copy any remaining elements of the first run
    while (first3 != last3)
        *(dest++) = *(first3++);
    // Copy any remaining elements of the second run
    while (first4 != last4)
        *(dest++) = *(first4++);
    while (first != mid && half != last) {
        if (comp(*half, *first))
            *(finalDest++) = *(half++);
        else
            *(finalDest++) = *(first++);
    }
    // Copy any remaining elements of the first run
    while (first != mid)
        *(finalDest++) = *(first++);
    // Copy any remaining elements of the second run
    while (half != last)
        *(finalDest++) = *(half++);
}

```

Rekurzivní Mergesort ovšem nemůže být adaptivní, což je jedna z velkých výhod Timsortu. Vytvořil jsem proto i iterativní verze těchto algoritmů. Ty jsou inspirovány Timsortem a nejprve naleznou všechny runy pomocí obdoby funkce `find_run`. Rozdíl je, že nyní neudržujeme invarianty a nalezneme proto všechny runy najednou. Teprve pak dojde ke slučování.

Aby slučování bylo co nejefektivnější, je potřeba slučovat runy s co nejpodobnější délkou. Pokud jsou data náhodná a runy mají délku `minRun`, pak je pro nás výhodné slučovat první run s druhým, třetí se čtvrtým apod. Tím získáme vyvážené slučování. Pokud se objeví delší run, může se slučování sice zpomalit, ale tento run jsme získali tím, že data již byla seřazena. Ve výsledku je tedy algoritmus stejně o něco rychlejší. K samotnému slučování jsou pak použity stejné funkce jako v rekurzivních verzích algoritmu. Všechny tyto funkce mají jako jméno předponu `it_` a jméno jejich rekurzivní verze.

V následující ukázce si můžeme všimnout funkce `merge_runs_final`, která zajišťuje iterativní slučování dvou runů, tak jak je popsáno v předchozím odstavci. Tato funkce je pak využita i u iterativních k-way mergesortů, aby sloučila runy jakmile je počet_runů $< k$.

Tyto iterativní Mergesorty používají k ukládání runů narozdíl od mé implementace Timsortu strukturu `MySlice`. Ta se liší v tom, že místo délky runu ukládá jeho poslední iterátor.

```
//struktura MySlice
template <class It>
struct MySlice {
    It start;
    It end;
    MySlice() = default;
    MySlice(It start, It end) : start(start), end(end) {}
};
```

```
//iterativní sloučení runů, první s druhým, třetí se čtvrtým
↪  apod., dokud nezbyde pouze jeden výsledný run
template <typename It, typename Compare>
void merge_runs_final(std::deque<MySlice<It>> & runs, Compare
↪  comp) {
    while (runs.size() > 1) {
        int remainder = runs.size() % 2;
        int size = runs.size() / 2;
        for (int i = 0; i < size; ++i) {
            MySlice x = runs.front();
            runs.pop_front();
            MySlice xx = runs.front();
            runs.pop_front();
            It a = xx.start;
            It b = xx.end;
            It c = x.end;
            std::vector<typename It::value_type>
            ↪  tmp(std::distance(a, c));
            merge_2(a, b, b, c, tmp.begin(), comp);
```

```
            x.start = xx.start;
            std::move(tmp.begin(), tmp.end(), x.start);
            runs.push_back(x);
        }

        for (int i = 0; i < remainder; ++i) {
            runs.push_back(runs.front());
            runs.pop_front();
        }
    }
}

//iterativní 2-way mergesort
template <class It, class Compare>
void it_merge_2_sort(It first, It last, Compare comp) {
    std::deque<MySlice<It>> runs = find_run(first, last, comp);

    merge_runs_final(runs, comp);
}
```

3.3 Timsort s hledáním runů od konce

Algoritmus Timsort s vyhledáváním runů (třída ReverseTimSort) je téměř identický jako ten běžný. Nalezneme ho v souboru `timsort_reverse.hpp` a zavoláme jako `timsort_rev`. Drobně se liší ve 3 metodách, kterým jsem přidal koncovku `_rev`. Dále se liší v inicializaci třídy, kde jako proměnnou `index` na začátku nastavíme nyní poslední iterátor místo prvního.

V metodě `find_run_rev` nyní procházíme pole odzadu. Abychom mohli doplnit run na velikost `minRun`, máme použije nyní metodu `binary_insertion_sort_end`, která má seřazenou část na konci místo na začátku. Poslední úprava proběhla ve funkci `merge_at_rev`, kde bylo potřeba prohodit indexy polí, aby se mohly správně sloučit. Původní kód před úpravou lze vidět v komentářích ukázek níže:

```
//funkce merge_at při hledání runů od konce
void merge_at_rev(uint32_t i) {
    //It base1 = runs[i].index;
    //int len1 = runs[i].length;
    //It base2 = runs[i + 1].index;
    //int len2 = runs[i + 1].length;

    It base1 = runs[i + 1].index;
    int len1 = runs[i + 1].length;
    It base2 = runs[i].index;
```

```
    int len2 = runs[i].length;

    runs[i].length = len1 + len2;
    //přidán řádek
    runs[i].index = runs[i + 1].index;

    if (i == runsLen - 3) {
        runs[i + 1] = runs[i + 2];
    }
    --runsLen;
    int k = gallop_right(base1, *base2, len1, 0);
    base1 += k;
    len1 -= k;
    if (len1 == 0)
        return;
    len2 = gallop_left(base2, *(base1 + (len1 - 1)),len2, len2 -
    ↪  1);
    if (len2 == 0)
        return;
    if (len1 <= len2)
        merge_lo(base1, len1, base2, len2);
    else
        merge_hi(base1, len1, base2, len2);
}
```

```
//binary_insertion_sort určený pro řazení při hledání runů od
↪  konce
void binary_insertion_sort_end(It lo, It hi, It sorted) {
    if (sorted == hi) {
        --sorted;
    }
    for (; lo < sorted; --sorted) {
        T pivot = std::move(*(sorted - 1));
        It const pos = std::lower_bound(sorted, hi, pivot,
        ↪  comp);
        for (It p = sorted - 1; p < pos - 1; ++p) {
            *p = std::move(*std::next(p));
        }
        *(pos - 1) = std::move(pivot);
    }
}
```

```
//původní binary_insertion_sort
void binary_insertion_sort(It lo, It hi, It start) {
    if (start == lo) {
        ++start;
    }
    for (; start < hi; ++start) {
        T pivot = std::move(*start);
        It const pos = std::upper_bound(lo, start, pivot, comp);
        for (It p = start; p > pos; --p) {
            *p = std::move(*std::prev(p));
        }
        *pos = std::move(pivot);
    }
}
```

```
//funkce pro hledání runů od konce
void find_run_rev() {
 ^^I//It idx = index + 1;
    It idx = index - 1;
    if (comp(*idx, *(idx - 1))) { // strictly descending run
        do {
            idx--;
        //} while (idx != last && comp(*idx, *(idx - 1)));
        } while (idx != first && comp(*idx, *(idx - 1)));
        // make ascending
        int halfLen = std::distance(idx, index) / 2;
        for (int j = 0; j < halfLen; ++j) {
            T tmp = std::move(*(idx + j));
            *(idx + j) = std::move(*(index - (j + 1)));
            *(index - (j + 1)) = std::move(tmp);
        }
    } else { // ascending run
        do {
            idx--;
        //} while (idx != last && comp(*(idx - 1), *idx));
        } while (idx != first && comp(*(idx - 1), *idx));
    }
    //int force = std::distance(index, idx);
    int force = std::distance(idx, index);
    if (force < minRun) {
        //It hi = std::distance(idx, last) < minRun ? last : index + minRun;
        //binary_insertion_sort(index, hi, idx);
        //idx = hi;
```

```
        It lo = std::distance(first, index) < minRun ? first :
        ↪  index - minRun;
        binary_insertion_sort_end(lo, index, idx);
        idx = lo;
    }
    //runs[runsLen++]= { index, (int)std::distance(index, idx)
    ↪  };
    runs[runsLen++]= { idx, (int)std::distance(idx, index) };
    index = idx;
}

```

Pro další porovnání jsem vytvořil ještě verzi algoritmu `it_merge_2_sort` s nacházením runů od konce pojmenovanou `it_merge_2_sort_rev`. Ta má místo funkce `find_run` funkci `find_run_rev` se stejnými úpravami popsanými výše.

```
//it_merge_2_sort s vyhledáváním runů od konce
template <class It, class Compare>
void it_merge_2_sort_rev(It first, It last, Compare comp) {
    std::deque<MySlice<It>> runs = find_run_rev(first, last,
    ↪  comp);
    merge_runs_final(runs, comp);
}

//původní it_merge_2_sort
template <class It, class Compare>
void it_merge_2_sort(It first, It last, Compare comp) {
    std::deque<MySlice<It>> runs = find_run(first, last, comp);
    merge_runs_final(runs, comp);
}
```

3.4 Paralelní Timsort podle prvního návrhu

Paralelní algoritmus jsem implementoval podle prvního návrhu. Rozdělím řazené pole na n částí a ty ve vlastním vlákně spustím mnou vytvořený Timsort. Až vlákna doběhnou dostaneme n seřazených částí nad kterými opět spustím Timsort a výsledkem je seřazené pole.

Důležité jsou dva parametry – počet vláken a režie vytváření vlákna. Počet použitých vláken je získán jako počet vláken $= \frac{\text{délka pole}}{a}$. Proměnná a se nastaví tak, aby paralelizace dosáhla většího zrychlení než je režie vytváření vlákna. V mé implementaci je $a = 2^{12}$. Zespoda je počet vláken omezen číslem 1 a v takovém případě samozřejmě paralelizaci vůbec neprovedeme. Ze shora je počet vláken omezen hodnotou `omp_get_max_thread()`.

Použitím OpenMP získáme velmi jednoduchý a přehledný kód. Stačí použít direktivu `parallel for` nad sdíleným sdíleným polem. Ve for cyklu pak jednotlivým vláknům předáme Timsortu iterátory tak, aby se nepřekrývaly a spustíme ve řazení. Direktiva sama zařídí, že na konci je bariéra a počká až doběhnou všechny vlákna. Nakonec se spustí poslední řazení v hlavním vlákně.

```
// kód paralelního Timsortu pomocí OpenMP
template <class It, class Compare>
void timsort_parallel_a(It first, It last, Compare comp) {
    int64_t length = std::distance(first, last);
    uint32_t threads = length >> 12;
    uint32_t maxThreads = omp_get_max_threads();
    threads = threads > maxThreads ? maxThreads : threads;
    if (threads > 1) {
        int64_t step = length / threads;
        //split array into multiple parts and sort parts
        #pragma omp parallel for
        for (uint32_t i = 0; i < threads; i++) {
            It first1 = first+(i*step);
            It last1 = i == threads - 1 ? last : first1+step-1;
            TimSort<It, Compare> tim(first1, last1, comp);
            tim.sort();
        }
    }
    timsort(first, last, comp);
}
```

3.5 Paralelní Timsort podle třetího návrhu

Tato verze paralelního Timsortu (`timsort_parallel_b`) je inspirována paralelním Mergesortem v další sekci a paralelizací Timsortu podle Saurabha Sooda. Nejprve najde všechny runy pomocí funkce `find_run`. Poté rekurzivně volá funkci `merge_runs_parallel` s parametry `lo` a `hi`. Tyto parametry odkazují na první a poslední zpracovávaný index pole runů. Pokud se $lo = hi$ máme nejmenší seřazenou jednotku – v tomto případě run (v případě mergesortu by to byl jeden prvek). Pokud ne, tak rozdělíme aktuálně zpracovávané runy na půlku a zavoláme `merge_runs_parallel` s novými parametry. Jakmile se vrátíme z těchto funkcí, tak víme, že v aktuálně zpracovávané části pole máme dvě seřazené části. Ty sloučíme pomocí funkce slučovací funkcí z Timsortu `gfx::timmerge`. Zvolil jsem funkci z knihovny `gfx`, protože na rozdíl od mé implementace je přístupná samostatně mimo třídu TimSort. Vzhledem k tomu, že slučovací funkce mojí implementace a knihovny `gfx` je téměř stejná, neměl by být žádný rozdíl ve výsledku.

Výhodou této verze je adaptivita před započetím paralelního kódu díky tomu, že se jako první naleznou runy. Další výhodou je jednoduchá paralelizace pomocí OpenMP na volanou rekurzivní funkci. Poslední výhodou oproti mé první paralelní verzi Timsortu je, že není potřeba na konci volat znovu algoritmus Timsort aby seřadil poslední části pole.

```cpp
//funkce merge_runs_parallel
template <class It, class Compare>
void merge_runs_parallel(std::deque<MySlice<It>> & runs, uint64_t lo, uint64_t hi, Compare comp) {
    if (lo == hi)
        return;
    uint64_t dist = std::distance(runs[hi].start, runs[lo].end);

    uint64_t m = (lo + hi) / 2;
#pragma omp taskgroup
    {
#pragma omp task shared(runs) untied if (dist >= (1<<14))
        merge_runs_parallel(runs, lo, m, comp);
#pragma omp task shared(runs) untied if (dist >= (1<<14))
        merge_runs_parallel(runs, m + 1, hi, comp);
#pragma omp taskyield
    }
    gfx::timmerge(runs[hi].start, runs[m].start, runs[lo].end, comp);
}

//paralelní timsort podle 3. návrhu
template <class It, class Compare>
void timsort_parallel_b(It first, It last, Compare comp) {
#pragma omp parallel
#pragma omp single
    {
        uint64_t size = std::distance(first, last);
        if (size <= 32){
            binary_insertion_sort(first, last, first, comp);
        } else {
            auto runs = find_run(first, last, comp);
            merge_runs_parallel(runs, 0, runs.size() - 1, comp);
        }
    }
}
```

3.6 Paralelní Mergesort pomocí OpenMP

Tento algoritmus jsem převzal z přednášky předmětu Paralelní algoritmy na Fakultě eletrotechnické. [21] Upravil jsem ho tak, aby používal C++ iterátory. Během této úpravy mě napadlo velmi jednoduché vylepšení. To spočívá v nahrazení standardního slučovací algoritmu slučovacím algoritmem z Timsortu. Tím získáme galloping a tím i možnost významného zrychlení v některých případech. Obě tyto implementace nalezneme v souboru `mergesort_parallel.hpp`.

Původní algoritmus je schován pod funkcí `merge_sort_parallel_a`. Ta pouze obaluje funkci `merge_sort_parallel_a_rec`, kde se provádí veškeré řazení. Nejdříve zjistíme jestli mezi iterátory platí nerovnost $left < right$. Pokud by neplatila znamenalo by to chybně volanou funkci. Pokud platí, tak zkontrolujeme jejich vzdálenost a pokud je menší než 32 použijeme obyčejný `binary_insertion_sort`. Jinak jako v běžném mergesortu, rozdělíme na dvě půlky a opět voláme funkci `merge_sort_parallel_a_rec`. Tato volání můžeme opět jednoduše paralelizovat pomocí tasků z OpenMP. Nakonec proběhne sloučení těchto dvou částí.

```
//funkce merge_sort_parallel_a_rec
template <typename It, typename Compare>
void merge_sort_parallel_a_rec(It left, It right, Compare comp)
↪ {
    if (left < right) {
        if (right - left >= 32) {
            It mid = left + std::distance(left, right) / 2;
#pragma omp taskgroup
            {
#pragma omp task untied if(right - left >= (1<<14))
                merge_sort_parallel_a_rec(left, mid, comp);
#pragma omp task untied if(right - left >= (1<<14))
                merge_sort_parallel_a_rec(mid, right, comp);
#pragma omp taskyield
            }
            std::inplace_merge(left, mid, right, comp);
        } else {
            binary_insertion_sort(left, right, left, comp);
        }
    }
}

//funkce začínající řazení paralelním mergesortem
template <typename It, typename Compare>
void merge_sort_parallel_a(It first, It last, Compare comp) {
#pragma omp parallel
```

```
26 #pragma omp single
27     merge_sort_parallel_a_rec(first, last, comp);
28 }
29
30
```

Funkce `merge_sort_parallel_b` funguje obdobně, pouze volá funkci `merge_sort_parallel_b_rec`, kde místo slučování dvou částí pomocí `std::inplace_merge` na řádku 15 je použita funkce `gfx::timmerge`[22] z Timsortu tak, jak je popsáno v prvním odstavci této sekce.

Kapitola 4

Testování

Testování proběhlo na školním serveru STAR. K vytváření testů jsem vytvořil speciální program, který vygeneruje několik různých typů dat o různých velikostech. Poté jsem ještě vytvořil program k samotnému testování. Ten načte vygenerovaná data a měří zadané funkce s přednastavenými porovnávacími funkcemi. Poté zkontroluje, zda jsou všechna data seřazená a jestli je algoritmus stabilní. Nakonec spočítá speciální porovnávací funkcí, kolik je potřeba porovnání k seřazení dat daným algoritmem.

Samotné testování je pak rozděleno na tři části. V první části se testuje základní verze Timsortu s řadícími algoritmy ze standardní knihovny a porovnám ji s existující implementací Timsortu z knihovny gfx[22]. Ve druhé části se porovnává moje základní implementaci Timsortu a verze s potencionálními optimalizacemi. V poslední části budu testovat paralelní verze Timsortu a Mergesortu.

4.1 Server STAR

Výpočetní klastr STAR je školní server určen k testování paralelních algoritmů vytvořených za pomoci MPI a OpenMP. Je složen z front-end uzlu, na který jsou připojeny další výpočetní svazky. Front-end uzel není určen ke spouštění programů, je určen hlavně k zadání úloh vypočetním uzlům. To se dělá za pomoci dávkového plánovače Sun Grid Engine. Nejprve se musí vytvořit SGE skript, který je poté předán skriptu `qrun` s vybranou frontou a počtem procesorů. [23]

Vzhledem k tomu, že je na serveru STAR nastaveno omezení doby běhu programu na 10 minut, musel jsem testovat po jednotlivých řadících funkcích. Všechny SGE skripty jsou k dispozici ve složce `STAR` a všechny jsou zařazeny do fronty ke zpracování pomocí příkazu `make star`.

Každý výpočetní uzel je vybaven dvěma procesory Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz s 10 jádry a 64 GB RAM.[24] Pomocí fronty určené pro OpenMP může můj program být spuštěn nejvíce na jednom tomto uzlu a bude mít k dispozici tedy 20 jader. Ke měření výsledného času jsem využil knihovnu `chrono`.[25]

4.2 Generování testů

Soubor ve kterém je k nalezení implementace generování testů se nazývá `generate_tests.cpp`. Ten generuje testy na základě zadání v `main` funkci programu. Tyto testy jsou uloženy do složky `test`. Umí vygenerovat následující typy dat:

4.2.1 Náhodná data

Náhodná data jsou tvořeny pomocí pseudonáhodné funkce `WELLRNG512`[26]. Ta využívá stavy, které je potřeba nejdříve inicializovat. Můžeme generovat 3 různé typy náhodných dat – integery, stringy a speciální LargeTestClass. Ta má simulovat větší třídu s komplikovanějším porovnáním několika hodnot. Máme také dva druhy náhodných stringů – jedny s fixní délkou 5 a další s náhodnou délkou mezi 2 a 6. Dalším druhem náhodných dat jsou data s mnoha duplikáty. Ty mohou být typu string a integer a nabývají maximálně 256 různých hodnot. Poslední druh jsou data třídy StableTestClass, která jsou určena pro test stability algoritmu.

```
//funkce WELLRNG512
unsigned long WELLRNG512() {
    unsigned long a, b, c, d;
    a = state[index];
    c = state[(index+13)&15];
    b = a^c^(a<<16)^(c<<15);
    c = state[(index+9)&15];
    c ^= (c>>11);
    a = state[index] = b^c;
    d = a^((a<<5)&0xDA442D24UL);
    index = (index + 15)&15;
    a = state[index];
    state[index] = a^b^d^(a<<2)^(b<<18)^(c<<28);
    return state[index];
}
```

```
//třída LargeTestClass
struct LargeTestClass {
    //random variables
    uint64_t name = 5022;
    int y = 3;
```

```
    double z = 50.8;

    //variables used for sorting
    StableTestClass a;
    StableTestClass b;
    StableTestClass c;
    int d;
    bool operator<(const LargeTestClass& y) const {
        if (a == y.a) {
            if (b == y.b) {
                if (c == y.c)
                    return d < y.d;
                else
                    return c < y.c;
            } else
                return b < y.b;
        } else
            return a < y.a;
    }
};
```

```
//třída StableTestClass
struct StableTestClass {
    int a;
    int b;
    int c;
    bool operator==(const StableTestClass& y) const {
        return a == y.a && b == y.b && c == y.c;
    }
    bool operator<(const StableTestClass& y) const {
        if (a == y.a) {
            if (b == y.b) {
                return c < y.c;
            } else
                return b < y.b;
        } else
            return a < y.a;
    }
};
```

4.2.2 Seřazená data

Seřazená data mohou mít typ string nebo integer. Toto jsou tvary, kterých mohou nabývat:

1. vzestupně seřazená data
2. sestupně seřazená data
3. několik vzestupně seřazených částí – pro mé testování využito 8 částí
4. tvar pyramidy – první půlka řazena vzestupně a druhá sestupně

4.2.3 Porovnávací funkce

Testování je prováděno se 4 různými porovnávacími funkcemi. První je funkce `std::less` ze standardní knihovny C++. Druhá je funkce `my_compare`, která pouze využívá k porovnání operátor <. Třetí funkcí je funkce `slow_compare` simulující pomalejší porovnávání. K tomu je použit cyklus s volatile proměnnou, kterou kompilátor nezoptimalizuje. Poslední funkce `count_compare` počítá počet provedených porovnání.

```
//funkce my_compare
template <class T>
bool my_compare(const T & x, const T & y) {
    return x < y;
}
```

```
//funkce slow_compare
template <typename T>
bool slow_compare(const T & x, const T & y) {
    volatile int z = 0;
    while (true) {
        if (++z == 10)
            break;

    }
    return (x < y);
}
```

```
//funkce count_compare
std::atomic<uint64_t> compareCount;

template <class T>
bool count_compare(const T & x, const T & y) {
    compareCount++;
    return x < y;
}
```

4.3 Spouštění testů

Implementaci programu pro testování nalezeneme v souboru `tester.cpp` a slouží k načtení testovacích dat a spuštění na všech testovaných algoritmech. Měří čas algoritmů na různých vstupech, kontroluje jestli jsou opravdu seřazeny a jestli je algoritmus stabilní. Tento program můžeme spustit s následujícími přepínači:

`-a` připíše výsledek testu na konec souboru, místo aby soubor přepsal

`-c N` `N` je maximální počet vláken, který mohou paralelní algoritmy využít

`-n N` `N` je počet opakování testů pro lepší výsledek měření, do výsledného souboru se pak zapisuje průměrná hodnota testů

`-o V` zapíše výsledky do souboru `V`

`-t A` bude testovat pouze algoritmus `A`

`-h` zobrazí nápovědu spuštění programu

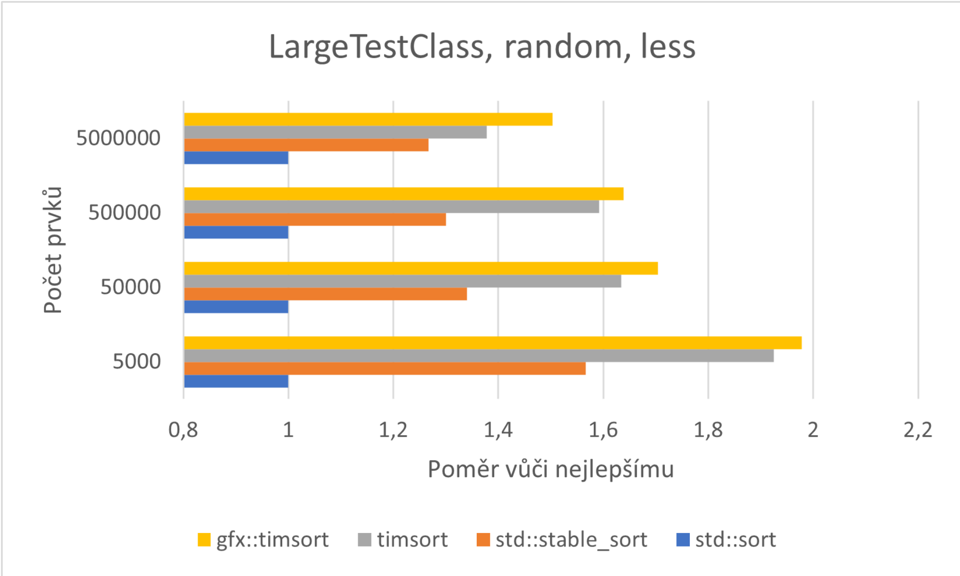
Pro větší přesnost výsledků testů byly všechny testy provedeny 5-krát a ve výsledcích v následujících sekcích je jejich průměrná hodnota. Veškeré naměřené časy jsou v mikrosekundách.

4.4 Testování základní implementace Timsortu

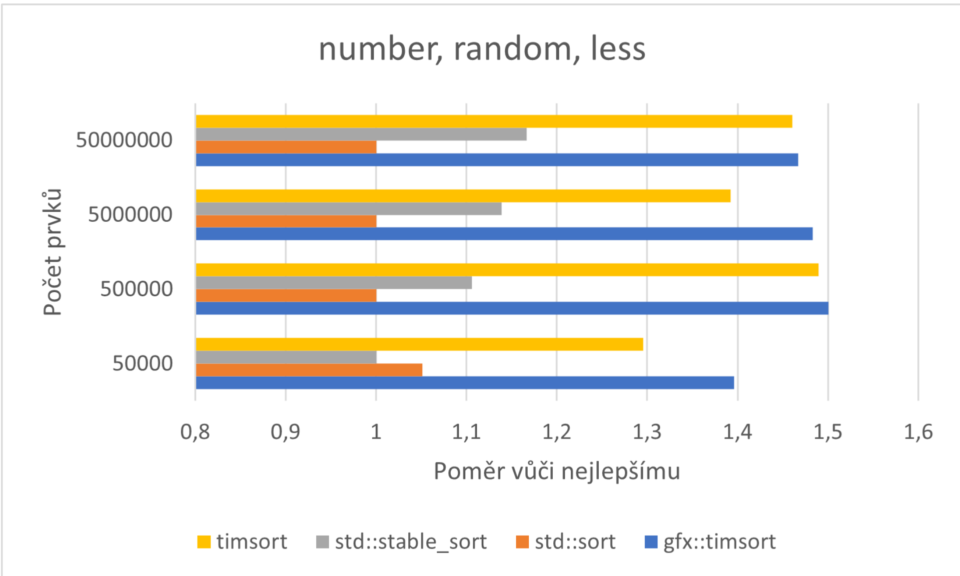
V této části porovnám moji implementaci Timsortu s algoritmy `gfx::timsort`, `std::sort` a `std::stable_sort`. Je nutno podotknout, že algoritmus `std::sort` není na rozdíl od ostatních stabilní.

Při porovnání různých algoritmů nás pravděpodobně zajímá jaký poměr rychlostí mezi nimi je. Každý graf se tedy zaměřuje na určitý typ dat a porovnává vůči nejlepšímu algoritmu pro daný typ dat a počet prvků. To znamená, že pro různý počet prvků se může nejlepší algoritmus změnit. Ve skutečnosti nastalo jen zřídka a když algoritmy byly skoro stejně rychlé.

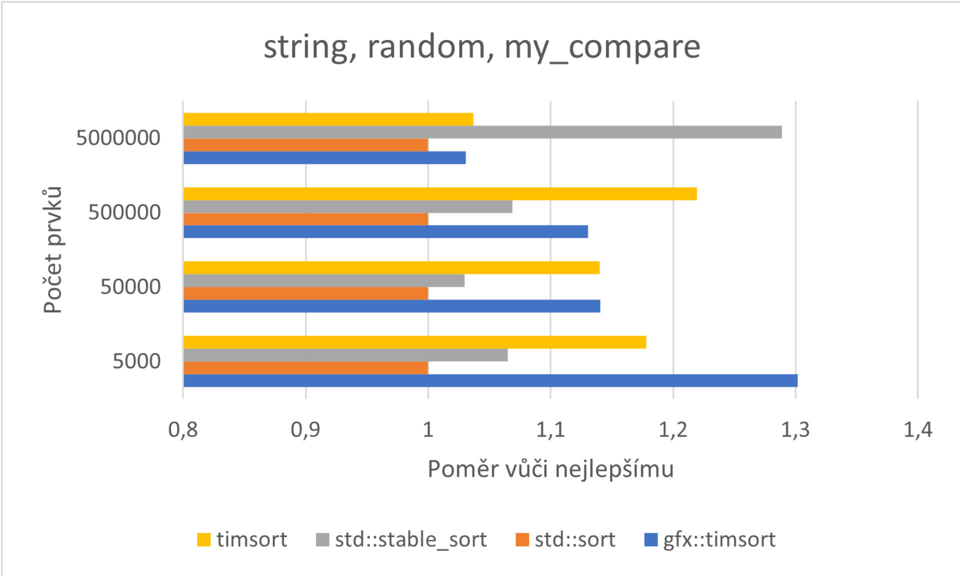
V prvních třech grafech (4.1, 4.2, 4.3) můžeme vidět porovnání náhodných dat pro třídu LargeTestClass, integery a stringy. Ve všech dominuje `std::sort`, jako druhý je nejčastěji `std::stable_sort` a na konci se střídají oba Timsorty. Nesmíme však zapomenout na to, že není stabilní, což může být pro někoho nedostačující. Nejlepší pro náhodná data z této čtveřice může být i `std::stable_sort` pokud je vyžadována stabilita. Poukázal bych také na fakt, že Timsort je se svým nejhorším výsledkem pouze 2-krát pomalejší. U stringů s rozdílnou délkou je dokonce pouze 1,1-krát pomalejší.



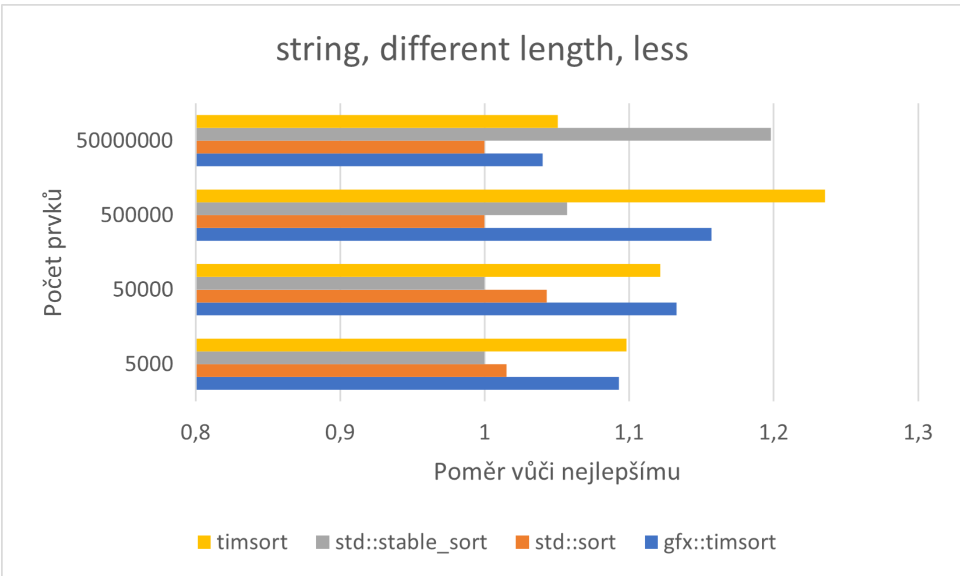
Obrázek 4.1: Graf porovnávající náhodná data třídy LargeTestClass pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



Obrázek 4.2: Graf porovnávající náhodná data typu integer pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



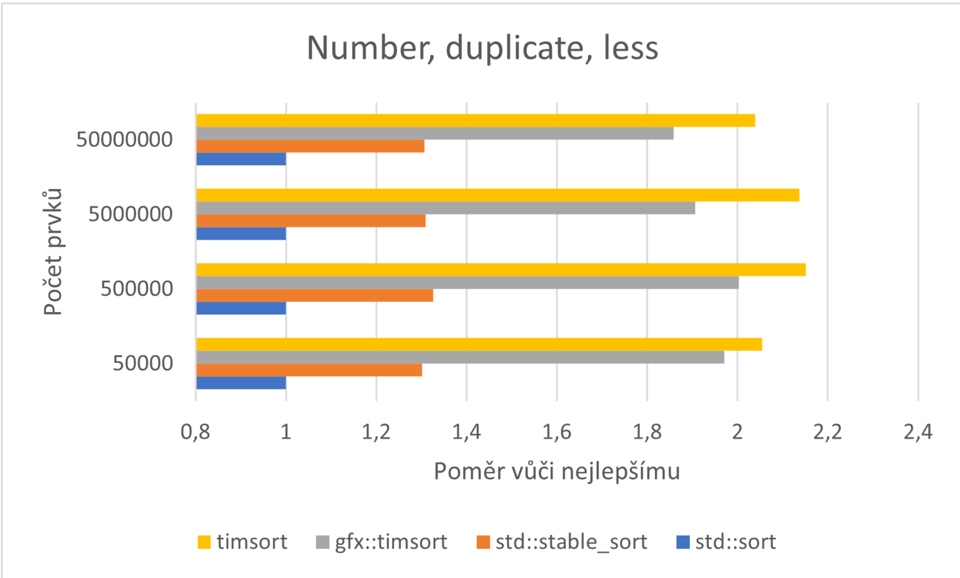
Obrázek 4.3: Graf porovnávající náhodná data typu string pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



Obrázek 4.4: Graf porovnávající náhodná data stringů s různou délkou pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort

V dalších grafech (4.5, 4.6, 4.7, 4.8, 4.9), kde data obsahují nějaké struktury už můžeme jasně vidět, proč se Timsort využívá přes to, že je pomalejší u náhodných dat. Čím víc jsou data seřazena, tím rychlejší je. Můžeme si také všimnout, že čím víc máme dat, tím víc Timsort vyhrává. Jediná data, která z této části dělají problém, jsou data obsahující mnoho duplikátů. Ovšem i zde je Timsort pouze zhruba 2-krát pomalejší. Porovnáme-li to se seřazenými daty, kde je při 50000000 prvcích zhruba 40-krát rychlejší než `std::sort` a `std::stable_sort` můžeme jasně vidět, proč je tak oblíbený.

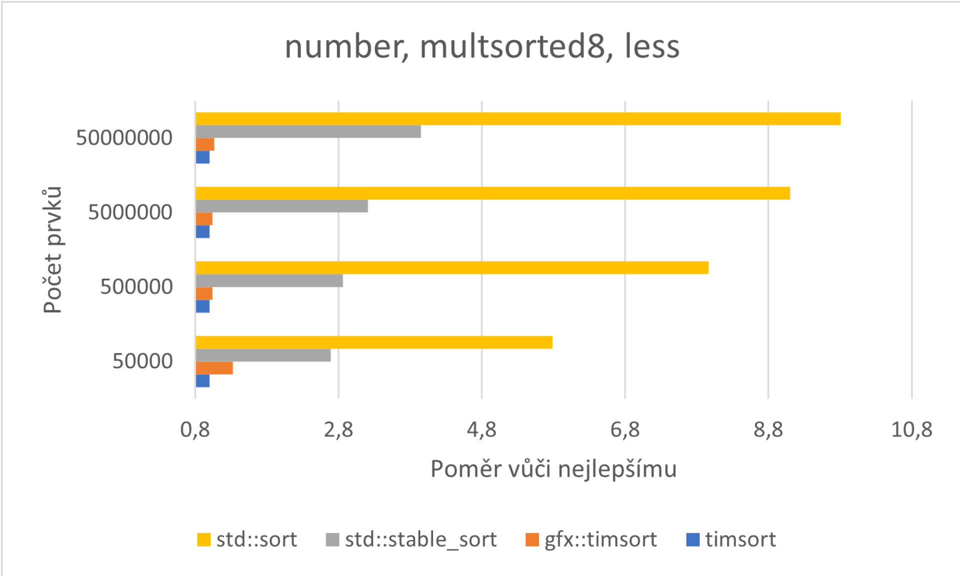
V běžném světě totiž často neřadíme úplně náhodná data, ale například přidáme nějaká data na konec už seřazeného pole. A v tomto případě bude mít Timsort obrovskou výhodu.



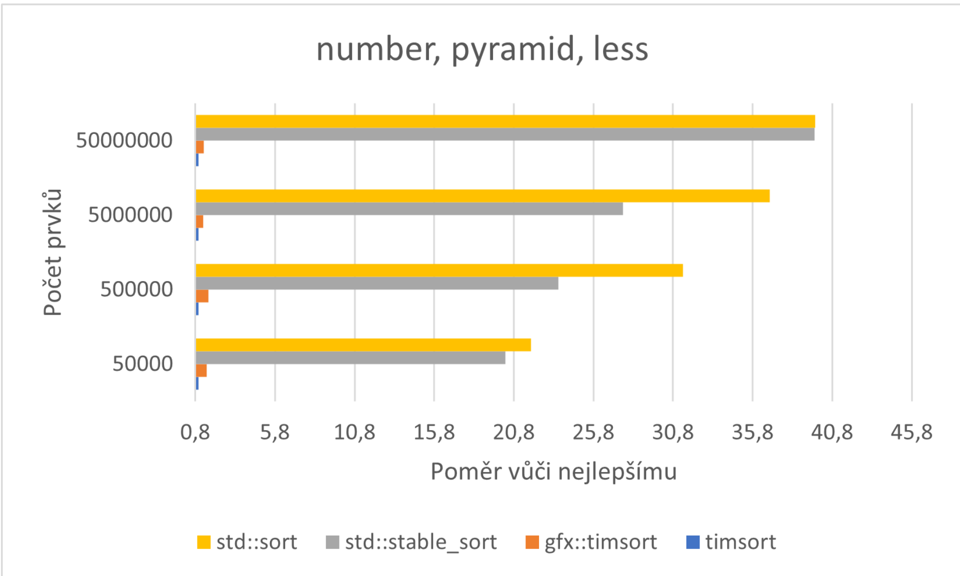
Obrázek 4.5: Graf porovnávající data s hodně duplikáty pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort

Dalším porovnávacím kritériem, které jsem zvolil je použití různých porovnávacích funkcí. Jak můžeme vidět na následujích grafech (4.10, 4.11, 4.12), použití jiné funkce může mít poměrně dramatický vliv. Přestože by funkce `less` a `my_compare` měly fungovat stejně, můžeme si všimnout velkého rozdílu. Ten bude pravděpodobně způsoben lepší optimalizací funkce `less` kompilátorem. Až na výjimku u stringů popsanou v poznámce níže a pár ojedinělých případů, však byla funkce `less` vždy rychlejší nebo stejně rychlá jako `my_compare`. Jeden z těchto případů je popsán v sekci Testování optimalizací v tabulce 4.3.

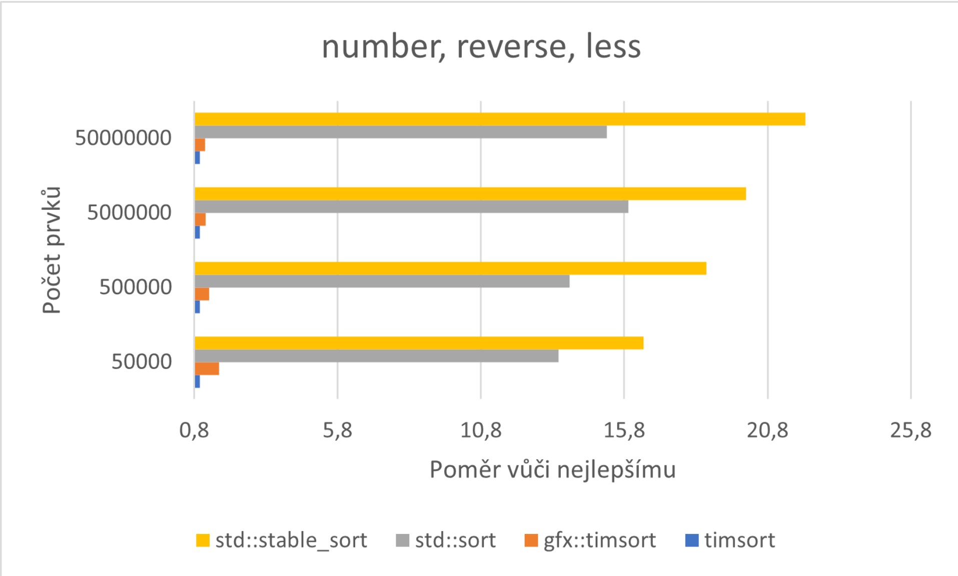
Na posledním grafu (4.13) je také zobrazen relativní počet potřebných porovnání řadících algoritmů. Můžeme vidět, že Timsorty vyžadují nejméně porovnání. To vysvětluje proč jsou s řadící funkcí `slow_compare` Timsorty skoro



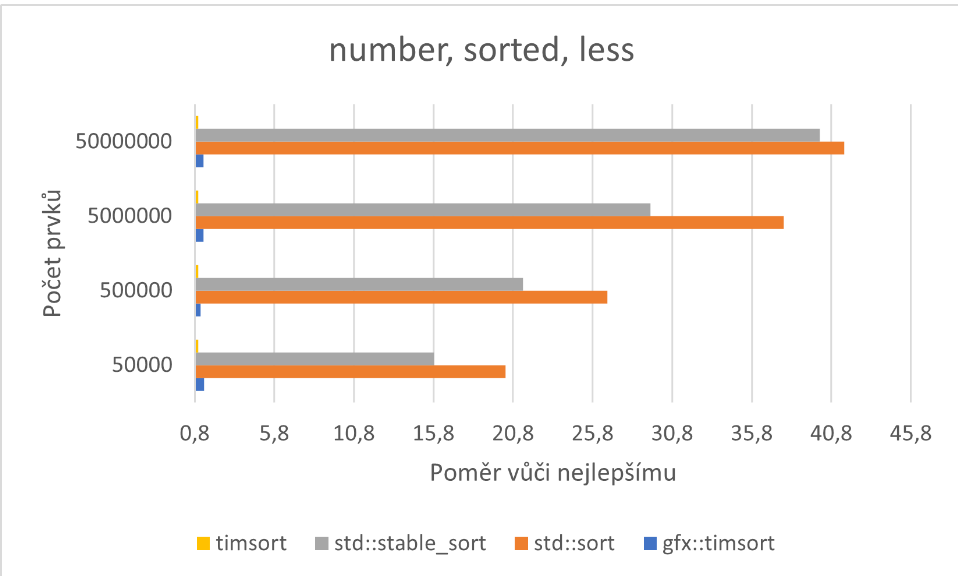
Obrázek 4.6: Graf porovnávající data s obsahující několik seřazených částí pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



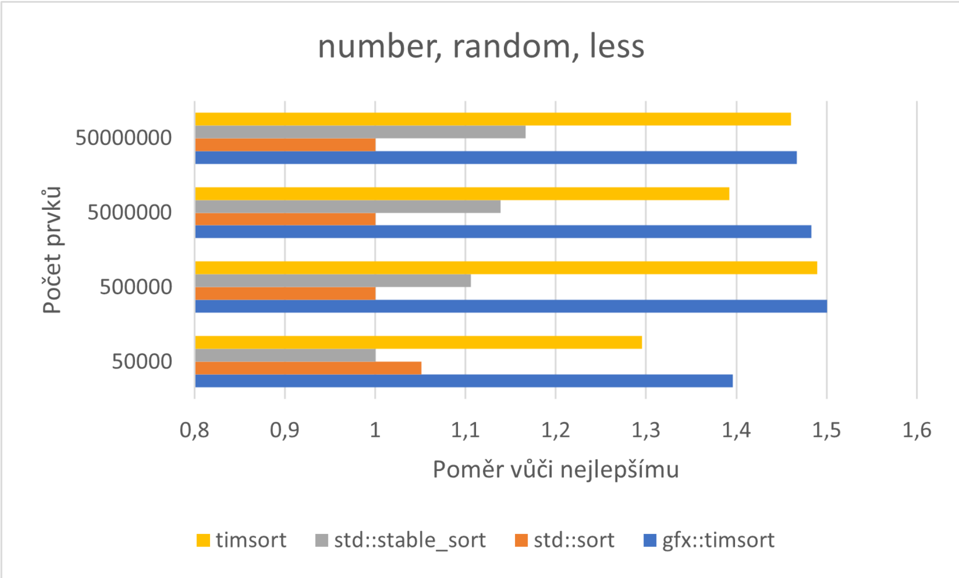
Obrázek 4.7: Graf porovnávající data ve tvaru pyramidy pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



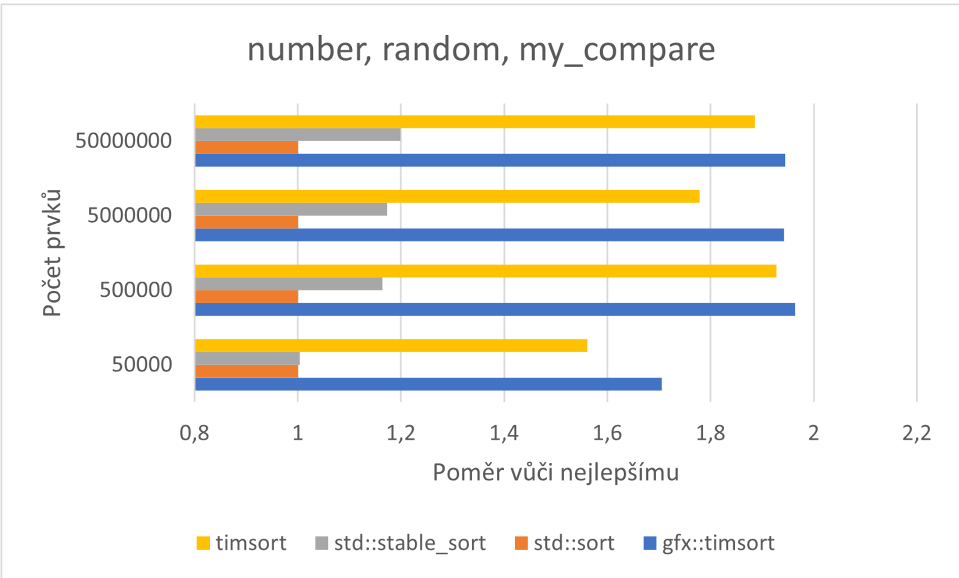
Obrázek 4.8: Graf porovnávající data seřazená v opačném pořadí pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



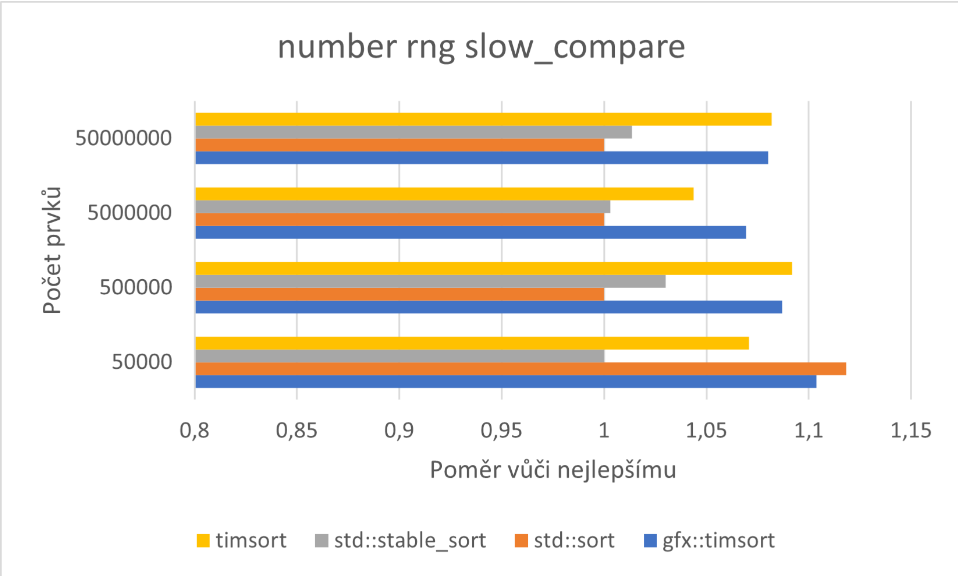
Obrázek 4.9: Graf porovnávající seřazená data pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



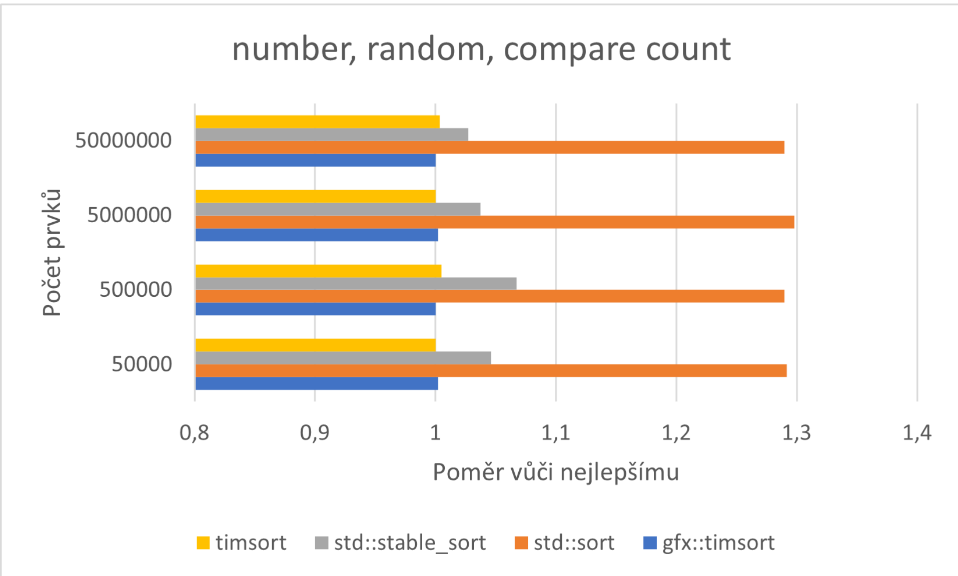
Obrázek 4.10: Graf porovnávající náhodná data řazena funkcí less pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



Obrázek 4.11: Graf porovnávající náhodná data řazena funkcí my_compare pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



Obrázek 4.12: Graf porovnávající náhodná data řazena funkcí slow_compare pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort



Obrázek 4.13: Graf zobrazující počet porovnání náhodných dat pro algoritmy gfx::timsort, timsort, std::stable_sort a std::sort

stejně rychlé jako nejrychlejší `std::sort`. Kdyby byla porovnávací funkce ještě o něco pomalejší, tak by Timsorty začaly být rychlejší. To je také ten důvod, proč se Timsort používá především pro neprimitivní typy, které mohou mít složité porovnávání.

4.4.1 Shrnutí testování základní verze Timsortu

V této sekci jsme porovnávali mou implementaci Timsortu. Ta obstála v porovnáním s Timsortem z knihovny `gfx` a chovala se přesně tak, jak bychom od Timsortu očekávali. Zajímavostí je, že se obě implementace drobně liší a v závislosti na datech mezi nimi jsou drobné rozdíly v rychlosti.

Dále máme porovnání s dalšími řadícími algoritmy a to konkrétně `std::sort` založený na Introsortu a `std::stable_sort` založený na Mergesortu. Můžeme si všimnout, že ačkoliv jsou tyto algoritmy rychlejší pro náhodná data, tak data obsahující struktury jim v porovnání s Timsortem dělají problém.

Výsledné porovnání algoritmů pro 50000000 integerů s různými strukturami (4.14). Vzhledem k logaritmickému měřítku doby běhu nemusí být na první pohled jasné obrovské zrychlení pro různě seřazená pole. Ještě jednou si můžeme na tomto grafu povšimnout, že se oba Timsorty opravdu chovají identicky.

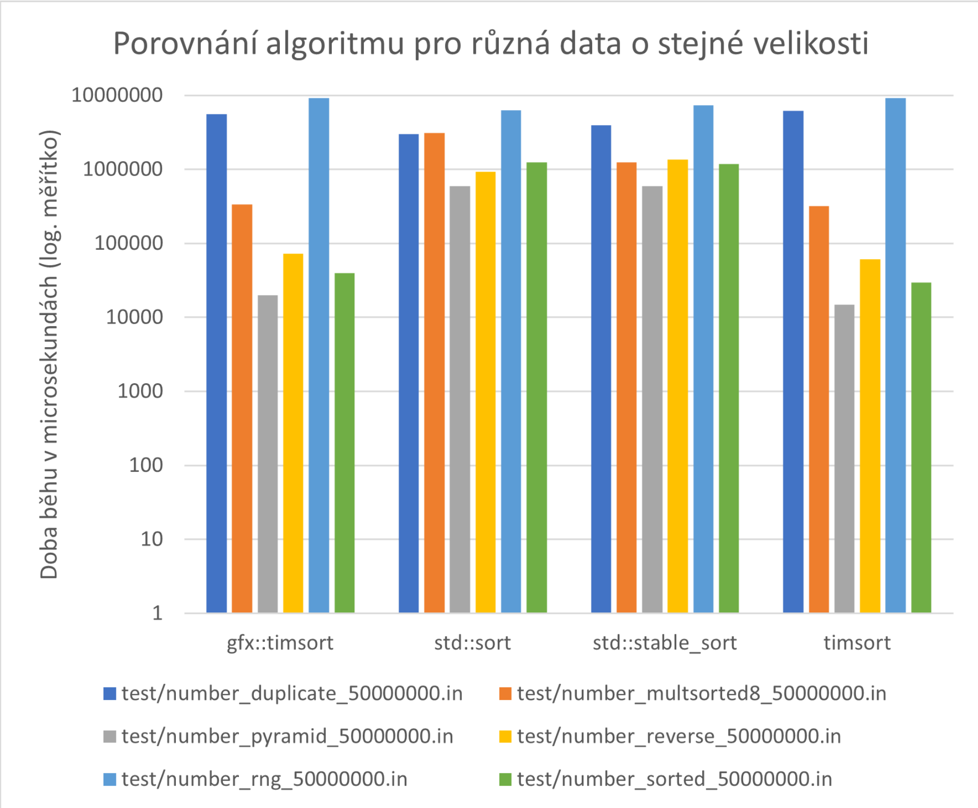
4.4.2 Poznámka k testování stringů

Během testování stringů se objevil problém s porovnávací funkcí `less` a některými algoritmy (tabulka 4.1). Zatímco algoritmus `std::sort` vracel očekávané hodnoty vždy, tak pro ostatní algoritmy byl problém s menším počtem prvků. Ve výsledku to vypadalo tak, že pro 5000 prvků byl algoritmus výrazně pomalejší. Dokonce byl pomalejší než ten samý algoritmus, také s porovnávací funkcí `less` pro 50000 prvků. Proto prezentuji výsledky funkce `my_compare`, kde už všechny hodnoty odpovídají očekávání.

4.5 Testování optimalizací

4.5.1 Hledání runu od konce

Tohoto porovnání se zúčastní 4 algoritmy – `timsort`, `timsort_rev`, `it_merge_2_sort` a `it_merge_2_sort_rev`. Při prozkoumaní grafů níže (4.15, 4.16) zjistíme, že "reverzní" algoritmy opravdu občas vyhrají – přesně tak jak bylo napsáno v komentáři u implementace řadící algoritmu v Rustu. Při podrobnějším zkoumání však zjistíme, že se nedá říci, který algoritmus je lepší. V jednom testu vyjdou nejlépe reverzní verze a v jiném zase normální verze. Při některých testech se dokonce stalo, že jedna reverzní verze byla nejrychlejší a druhá nejpomalejší. Proto jsou zde pouze dva grafy na ukázku, že se



Obrázek 4.14: Graf porovnávající algoritmy gfx::timsort, std::sort, std::stable_sort a timsort na různých datech o stejné velikosti

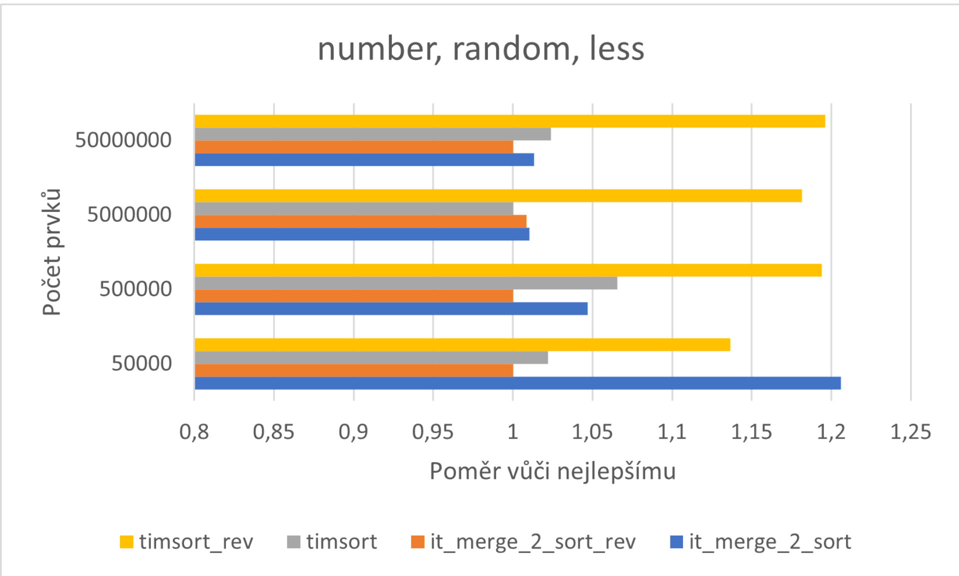
reverzní algoritmy s normálními střídají. Ačkoliv zrychlení mohlo dosáhnout i 20% bude pravděpodobně záležet na konkrétních datech. Abychom toto vyvrátili, bylo by potřeba testy opakovat s obrovským množstvím různých dat. Mým odhadem je, že to záleží na využívání dat v cache paměti a v některých případech používáme častěji ty samá data. Tím bychom tedy získali zrychlení. Bohužel se v takovém případě nedá mluvit o žádné struktuře dat, kterou bychom mohli předem znát.

4.5.2 Mergování více runů

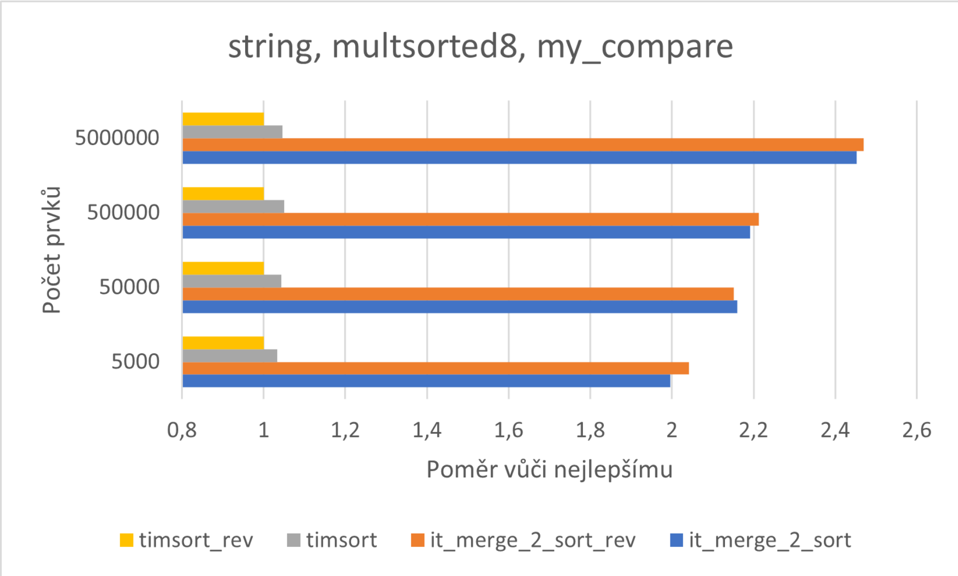
Tohoto porovnání se účastnili algoritmy `it_merge_2_sort`, `it_merge_3_sort`, `it_merge_4a_sort` a `it_merge_4b_sort` a jejich rekurzivní verze. Rekurzivní verze ovšem vynechám ve výsledcích zde, protože bychom přišli o adaptivitu Timsortu, pokud bychom je využili. Platilo pravidlo, že rekurzivní algoritmy byly ve stejném pořadí jako iterativní a byly oproti iterativním o něco málo rychlejší. Z grafů 4.17, 4.18 a 4.19 lze vyvodit následující:

Tabulka 4.1: Porovnání funkce less a my_compare u stringů

Počet prvků	algoritmus	less	my_compare
5000	gfx::timsort	17492	1584
5000	std::sort	1307	1217
5000	std::stable_sort	15585	1296
5000	timsort	15642	1434
50000	gfx::timsort	19712	19839
50000	std::sort	16746	17396
50000	std::stable_sort	16229	17917
50000	timsort	18521	19833
500000	gfx::timsort	247781	260819
500000	std::sort	220812	230737
500000	std::stable_sort	219269	246606
500000	timsort	262599	281412
5000000	gfx::timsort	3482919	3658339
5000000	std::sort	3435478	3549239
5000000	std::stable_sort	4039618	4574116
5000000	timsort	3434988	3680149



Obrázek 4.15: Graf porovnávající algoritmy timsort_rev, timsort, it_merge_2_sort_rev a it_merge_2_sort na náhodných datech



Obrázek 4.16: Graf porovnávající algoritmy timsort_rev, timsort, it_merge_2_sort_rev a it_merge_2_sort na několika seřazených částech

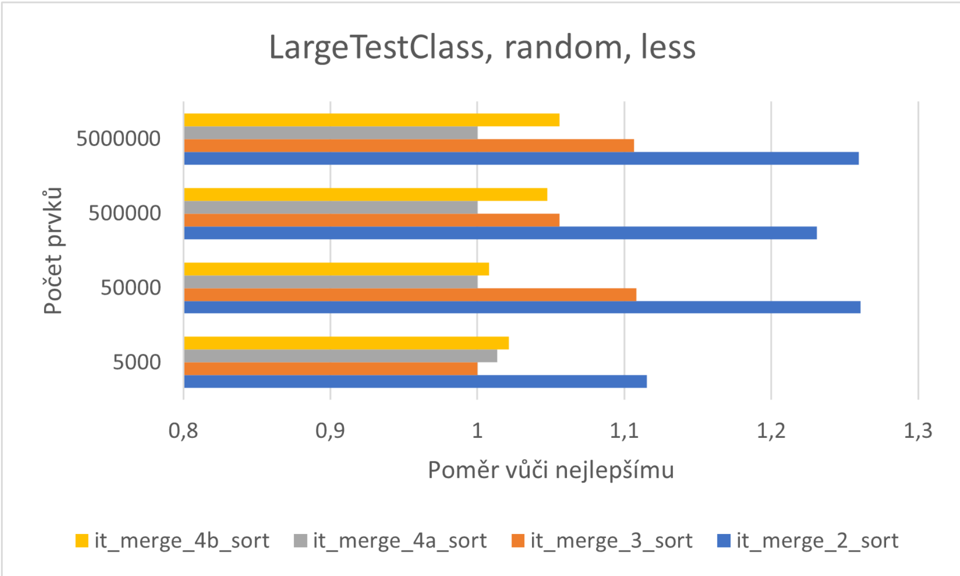
Algoritmus `it_merge_4a_sort` a algoritmus `it_merge_2_sort` dopadly většinou podobně. Rozdíl byl pokud jsme řadili větší objekt. Pak byl rychlejší algoritmus `it_merge_4a_sort`, tak jak bychom očekávali, díky ušetřenému kopírování.

Algoritmus `it_merge_4b_sort` byl nejrychlejší pokud byla porovnávací funkce rychlá. Jakmile však byla porovnávací funkce pomalá, byl nejpomalejší. Zpomalení způsobeno tím, že oproti ostatním zmíněným algoritmům potřebuje mnohem více porovnání.

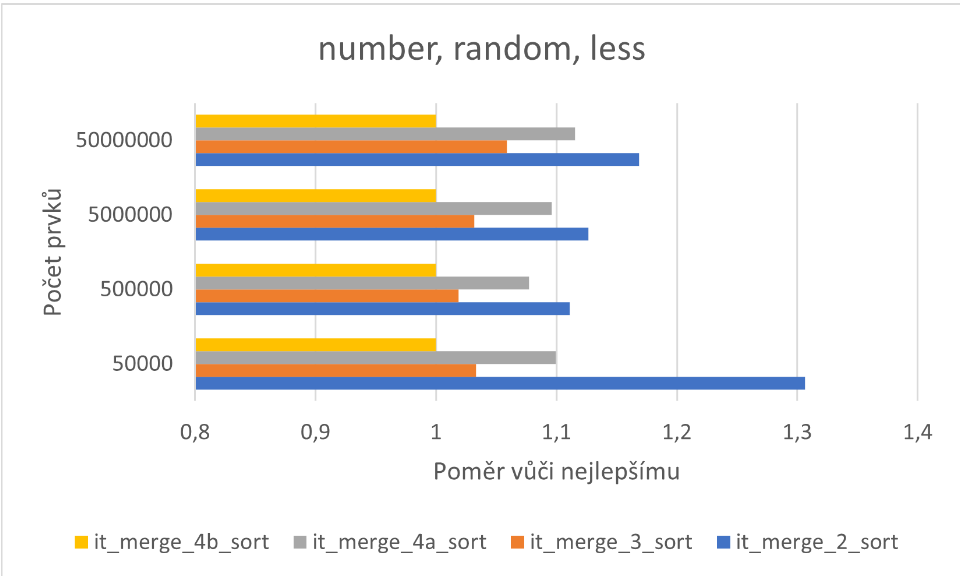
Poslední algoritmus `it_merge_3_sort` byl ze zmíněných asi nejvíce všestranný. Využil zrychlení stejně jako v případě `it_merge_4b_sortu`, ale nepotřebuje tolik porovnání (tabulka 4.2). Přesto jich potřebuje více než `it_merge_4a_sort` a `it_merge_2_sort`.

Tabulka 4.2: Porovnání počtu porovnání algoritmů it_merge_2_sort, it_merge_3_sort, it_merge_4a_sort a it_merge_4b_sort

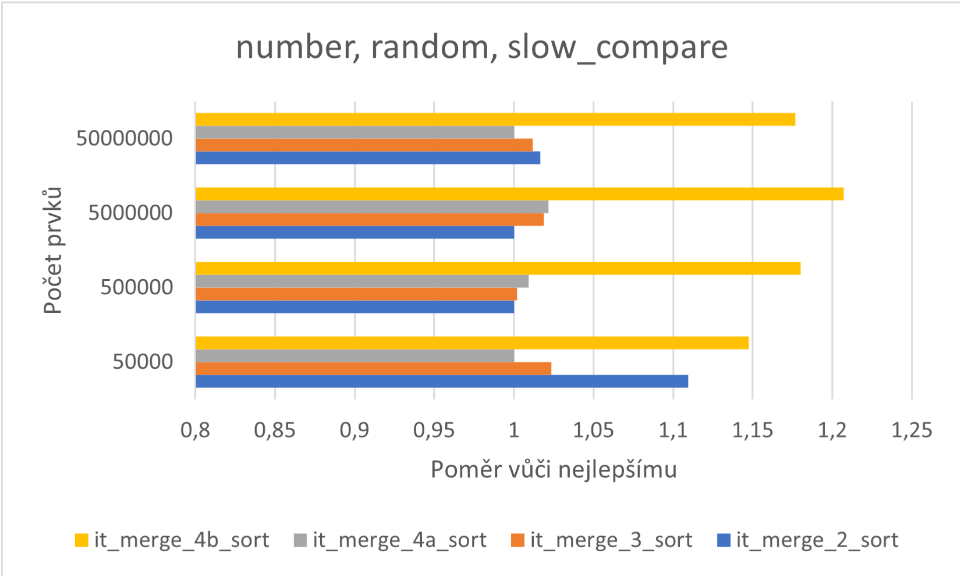
	prvků			
algoritmus	50000	500000	5000000	50000000
it_merge_2_sort	714211	8807412	104751146	1212802980
it_merge_3_sort	879242	11093638	132730581	1504735394
it_merge_4a_sort	714211	9275307	109670298	1212814976
it_merge_4b_sort	962129	12742307	154269783	1710706481



Obrázek 4.17: Porovnání algoritmů it_merge_4b_sort, it_merge_4a_sort, it_merge_3_sort a it_merge_3_sort na náhodných datech typu LargeTestClass



Obrázek 4.18: Porovnání algoritmů it_merge_4b_sort, it_merge_4a_sort, it_merge_3_sort a it_merge_3_sort na náhodných datech typu integer



Obrázek 4.19: Porovnání algoritmů it_merge_4b_sort, it_merge_4a_sort, it_merge_3_sort a it_merge_3_sort na náhodných datech typu integer pomocí slow_compare

Vzhledem k využití Timsortu k řazení převážně neprimitivních typů by bylo nejlepší použít algoritmus z `it_merge_4a_sortu`. Došlo by tím pravděpodobně k malému zrychlení. Pokud bychom chtěli zrychlit primitivní typy, pak se nabízí využití `it_merge_3_sort` nebo `it_merge_4b_sort`.

Ať už bychom si vybrali kterýkoliv z těchto algoritmů, přišli bychom tím o možnost využívat invariantů k ušetření paměti. Všechny runy by se pravděpodobně musely najít najednou, protože režie kontroly nových invariantů by mohla být moc velká.

Při porovnávání různých řadících funkcí těchto algoritmů jsem si všiml, že funkce `less` je pomalejší než `my_compare` pro LargeTestClass náhodná data (tabulka 4.3), kromě testu s 5000000 prvky. Mohlo by tedy být zajímavé zkoumat jaký vliv má funkce `less` na rychlost porovnávání dat různých typů.

4.6 Testování paralelních algoritmů

Všechny 4 paralelní algoritmy byly testovány s využitím 1, 2, 4, 8, 16 a 20 vláken a porovnány se základním Timsortem. V prvních čtyřech grafech je znázorněno zrychlení algoritmu při náhodných datech v závislosti na počtu vláken a počtu prvků. Můžeme vidět výrazné velmi výrazné zrychlení a to až 6,8-krát při plném využití 20 jader procesoru pro algoritmy `merge_sort_parallel_a` (graf 4.20) a `merge_sort_parallel_b` (graf 4.21). Algoritmus `timsort_parallel_a` (graf 4.22) dosáhl až pětinásobného zrychlení při 20

Tabulka 4.3: Poměr doby běhu porovnávacích funkcí less a my_compare

prvků	algoritmus	less	my_compare	less/my_compare
5000	it_merge_2_sort	2513	2204	1,1401
5000	it_merge_3_sort	2253	2064	1,0915
5000	it_merge_4a_sort	2283	1937	1,1786
5000	it_merge_4b_sort	2301	2060	1,1169
50000	it_merge_2_sort	29663	21660	1,3694
50000	it_merge_3_sort	26073	20523	1,2704
50000	it_merge_4a_sort	23530	18610	1,2643
50000	it_merge_4b_sort	23713	19589	1,2105
500000	it_merge_2_sort	272373	245590	1,1090
500000	it_merge_3_sort	233615	229381	1,0184
500000	it_merge_4a_sort	221258	210027	1,0534
500000	it_merge_4b_sort	231815	230400	1,0061
5000000	it_merge_2_sort	3174058	3390263	0,9362
5000000	it_merge_3_sort	2788695	2995951	0,9308
5000000	it_merge_4a_sort	2520025	2610436	0,9653
5000000	it_merge_4b_sort	2660849	2771709	0,9600

vláknech a poslední `timsort_parallel_b` (graf 4.23) je s 20 vlákny 2,5-krát rychlejší než běžný Timsort.

Při porovnávání náhodných stringů už je situace jiná. Nejhůře dopadl `timsort_parallel_b` (graf 4.27), kde i s 20 vlákny je pomalejší než základní Timsort. Ostatní paralelní algoritmy pak byly při 5 milionech prvků a 20 vláknech 2-krát až 3-krát rychlejší než sekvenční Timsort (grafy 4.24, 4.25, 4.26). S alespoň čtyřmi vlákny pak jsou tyto algoritmy stejně rychlé nebo rychlejší než obyčejný Timsort. S jedním vláknem očekáváme, že paralelní algoritmus může být pomalejší, kvůli větší režie okolo samotného řazení. U dvou vláken si můžeme všimnout, že pro více prvků už také dosahujeme zrychlení.

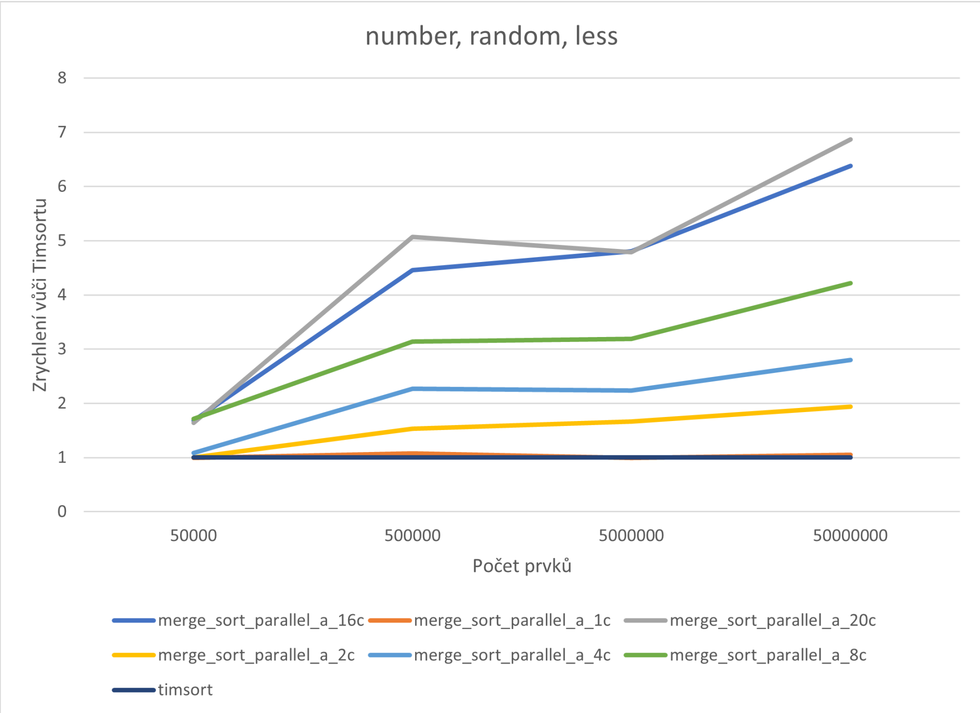
Dalším zajímavým testem je pole obsahující několik seřazených posloupností. V tomto testu je totiž `timsort_parallel_b` nejrychlejší, jak můžeme vidět v tabulce 4.4. Spoustu paralelních algoritmů dopadlo v tomto testu a dalších testech, kde jsou data již seřazená hůře než obyčejný Timsort. To je ovšem očekávané vzhledem k tomu, že paralelní algoritmy předem neví, že data jsou seřazená a zbytečně je rozdělují mezi vlákna. A proto je právě `timsort_parallel_b` v tomto testu nejrychlejší, protože napřed zjistí všechny runy a teprve potom je rozděluje mezi vlákna.

Také bych zmínil, že vylepšení paralelního mergesortu pomocí slučovací funkce z Timsortu opravdu zafungovalo. Při náhodných datech je zpomalení minimální a při seřazených datech je algoritmus znatelně rychlejší. To ostatně můžeme vidět v grafech 4.28, 4.29, 4.30, 4.31, 4.32, kdy při porovnávání výsledků těchto algoritmů lze vidět jasné zrychlení. Mezi těmito grafy lze také

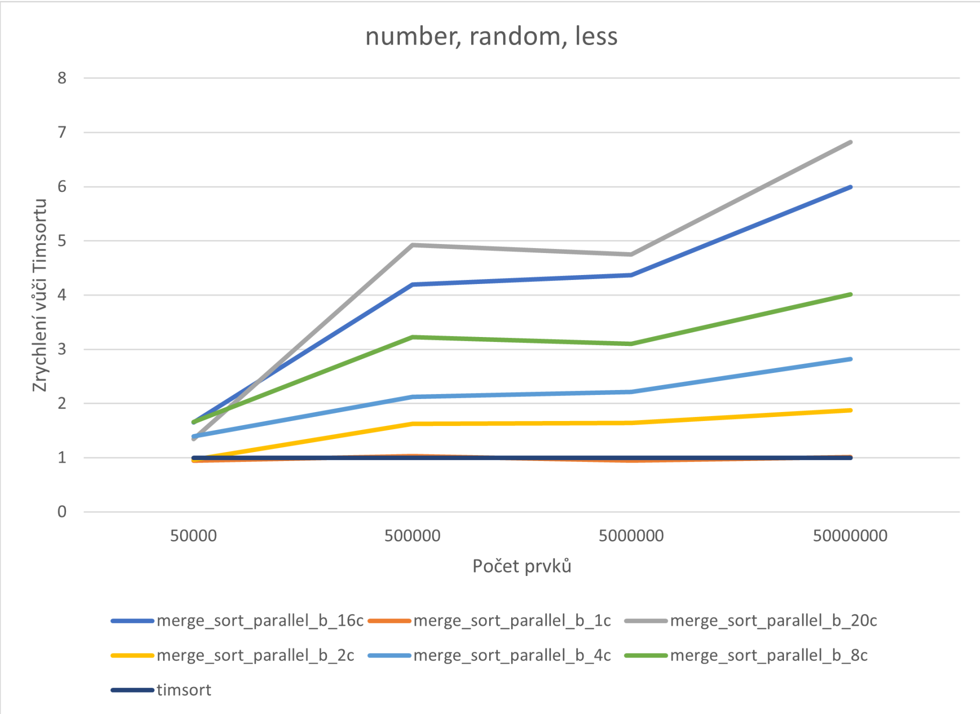
Tabulka 4.4: Porovnání paralelních algoritmů pro pole obsahující několik seřazených posloupností

Algoritmus	zrychlení vůči Timsortu
merge_sort_parallel_a_16c	0,7932
merge_sort_parallel_a_1c	0,1958
merge_sort_parallel_a_20c	0,8578
merge_sort_parallel_a_2c	0,2764
merge_sort_parallel_a_4c	0,3928
merge_sort_parallel_a_8c	0,5654
merge_sort_parallel_b_16c	1,0822
merge_sort_parallel_b_1c	0,3845
merge_sort_parallel_b_20c	1,0725
merge_sort_parallel_b_2c	0,5079
merge_sort_parallel_b_4c	0,6301
merge_sort_parallel_b_8c	0,8173
timsort	1
timsort_parallel_a_16c	0,9348
timsort_parallel_a_1c	1,0002
timsort_parallel_a_20c	0,9122
timsort_parallel_a_2c	0,4768
timsort_parallel_a_4c	0,4766
timsort_parallel_a_8c	0,9631
timsort_parallel_b_16c	1,2668
timsort_parallel_b_1c	0,9342
timsort_parallel_b_20c	1,1616
timsort_parallel_b_2c	1,1932
timsort_parallel_b_4c	1,3018
timsort_parallel_b_8c	1,2303

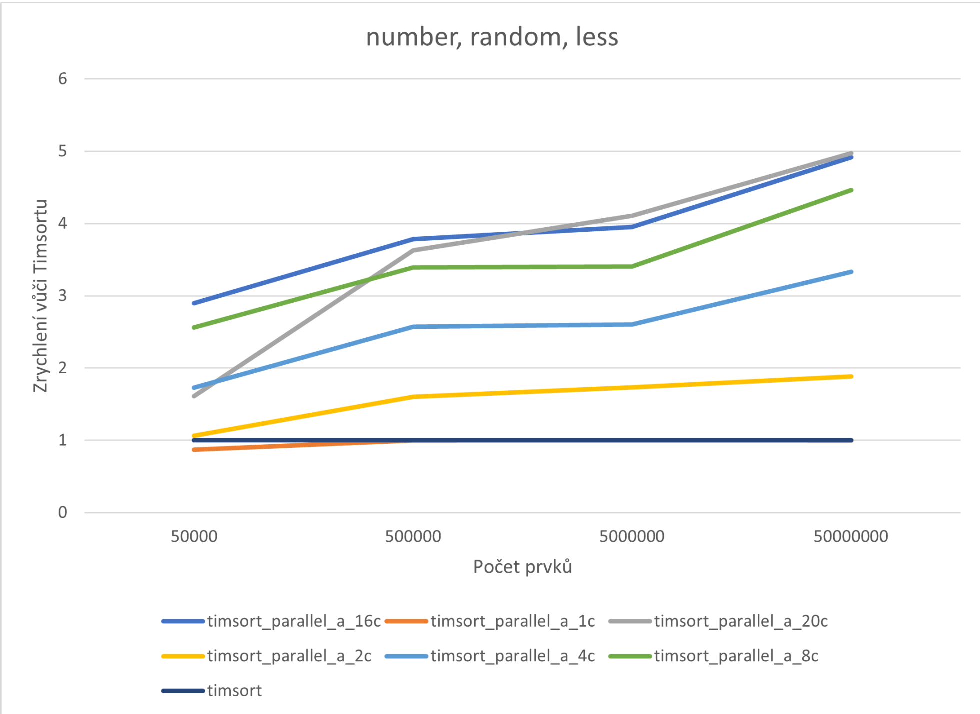
porovnat zrychlení pro různý počet vláken. Doba běhu je opět v logaritmickém měřítku a rozdíl jednoho řádku tedy znamená 10-krát zrychlený algoritmus.



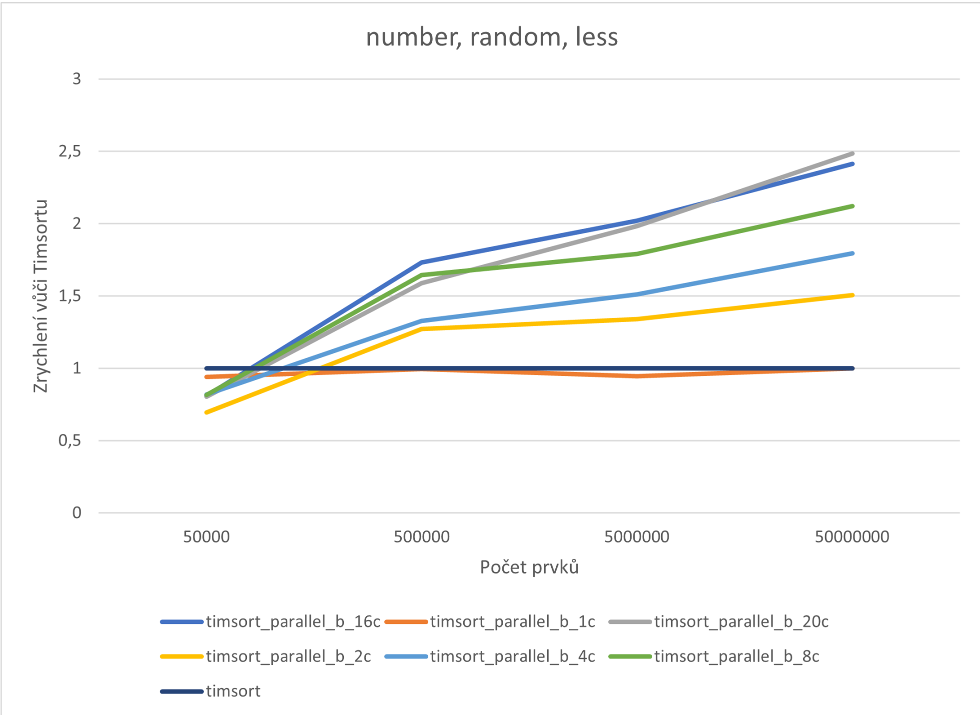
Obrázek 4.20: Zrychlení merge_sort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodná čísla



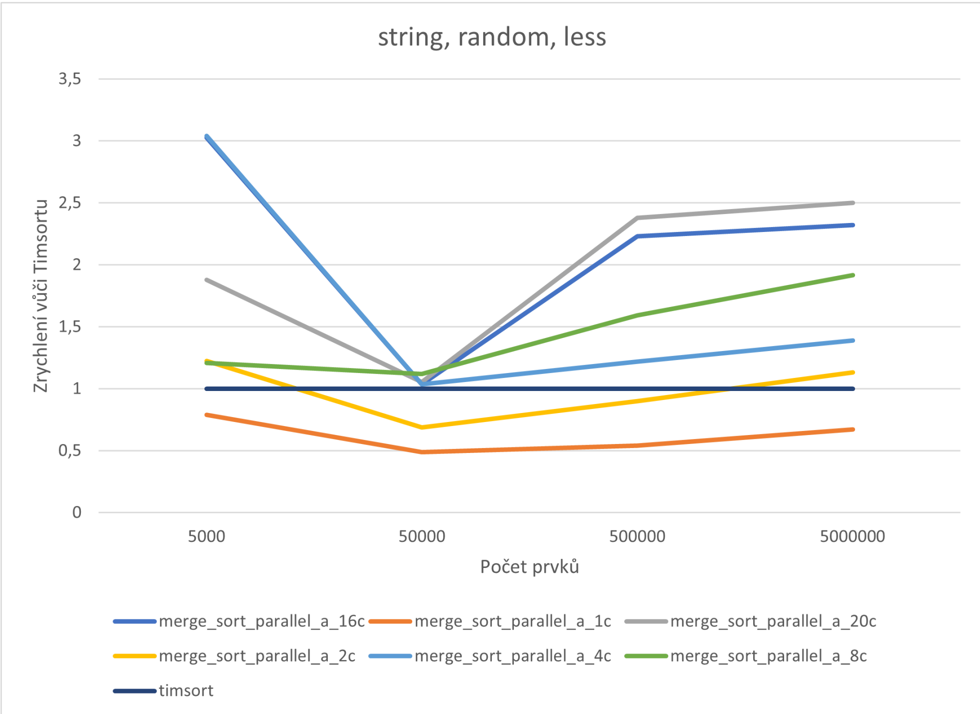
Obrázek 4.21: Zrychlení merge_sort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodná čísla



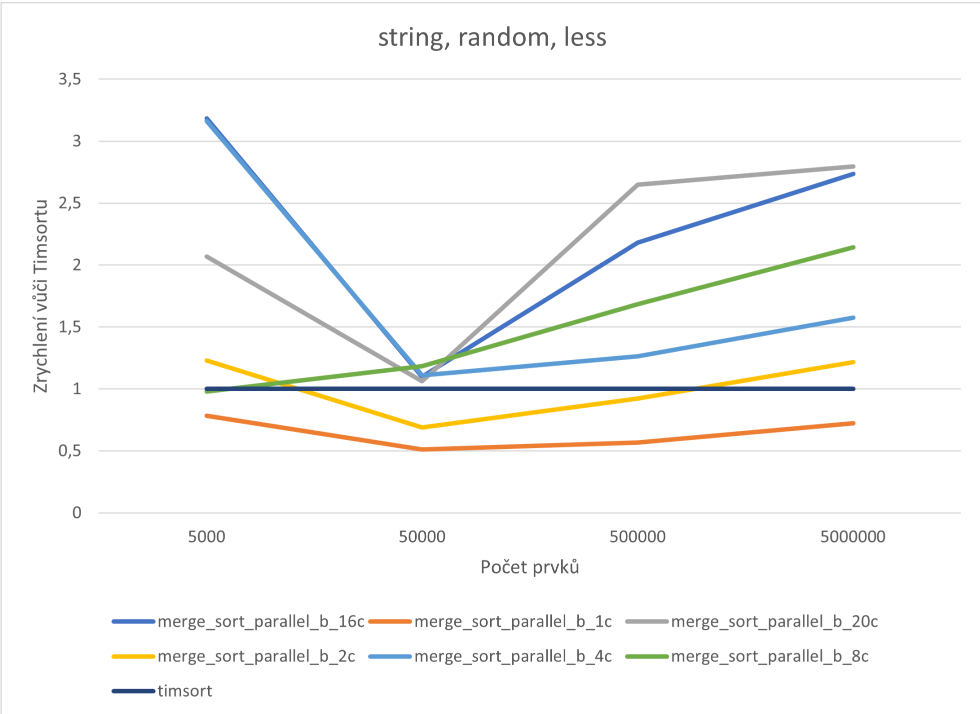
Obrázek 4.22: Zrychlení timsort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodná čísla



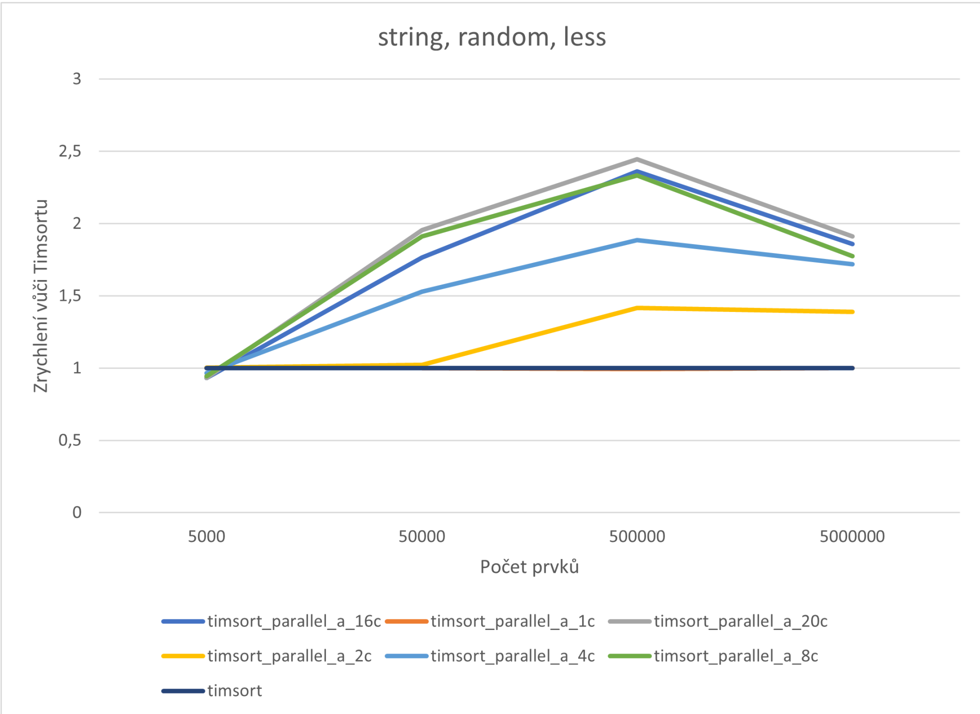
Obrázek 4.23: Zrychlení timsort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodná čísla



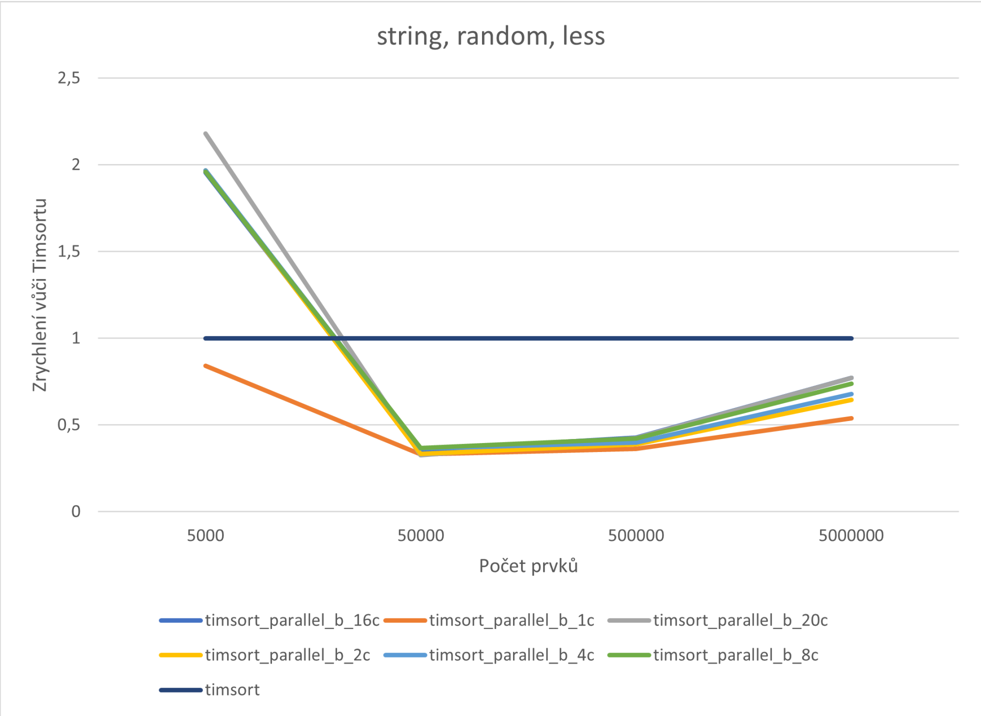
Obrázek 4.24: Zrychlení merge_sort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodné stringy



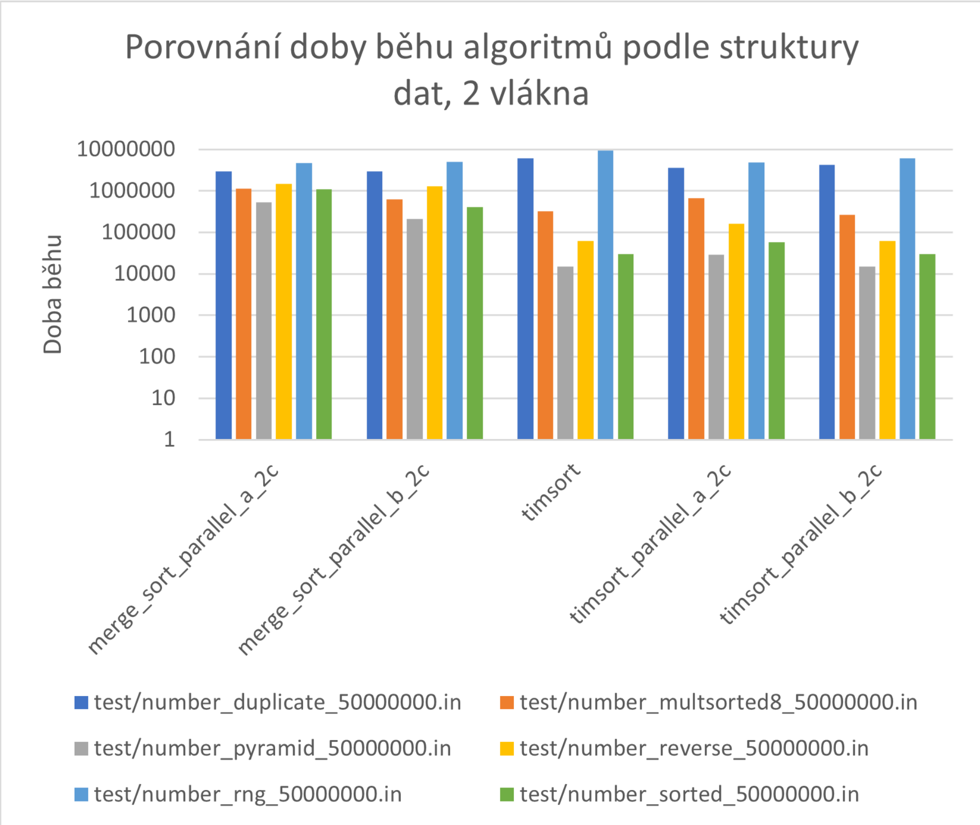
Obrázek 4.25: Zrychlení merge_sort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodné stringy



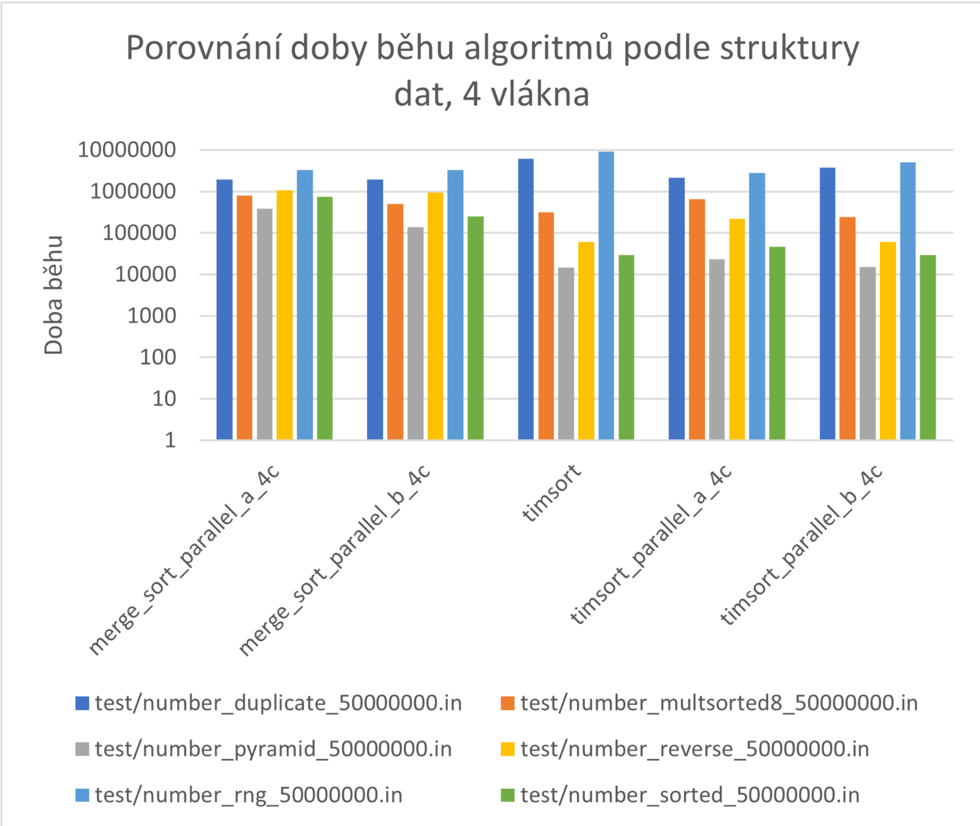
Obrázek 4.26: Zrychlení timsort_parallel_a oproti timsortu v závislosti na počtu vláken, náhodné stringy



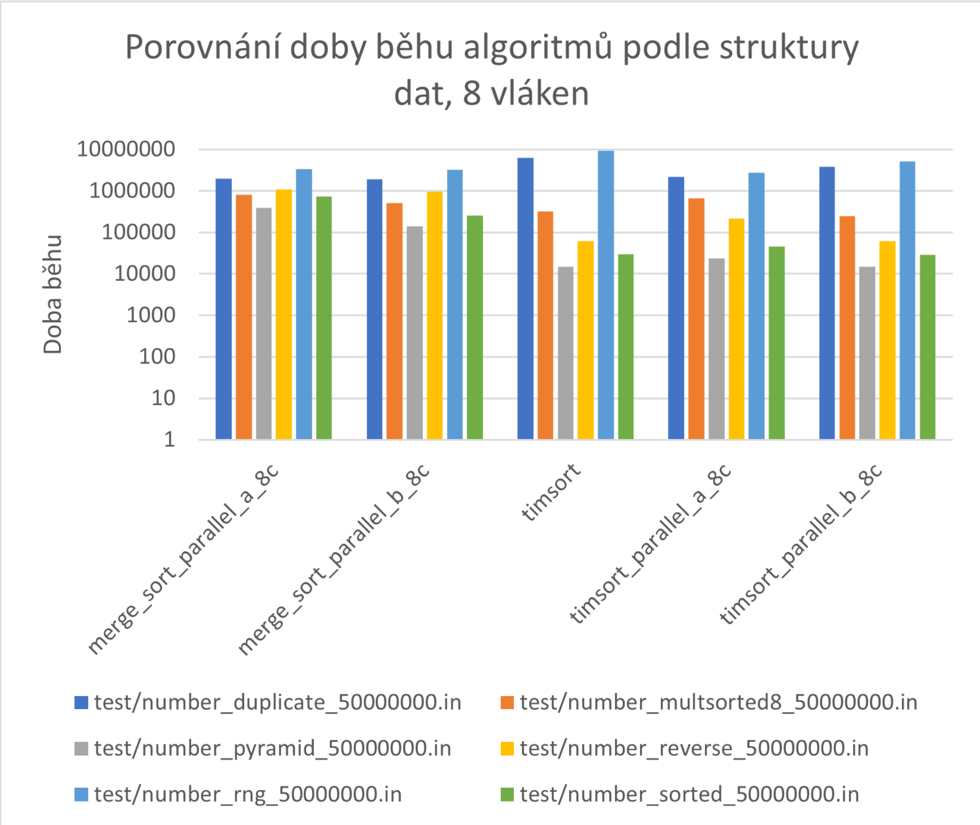
Obrázek 4.27: Zrychlení timsort_parallel_b oproti timsortu v závislosti na počtu vláken, náhodné stringy



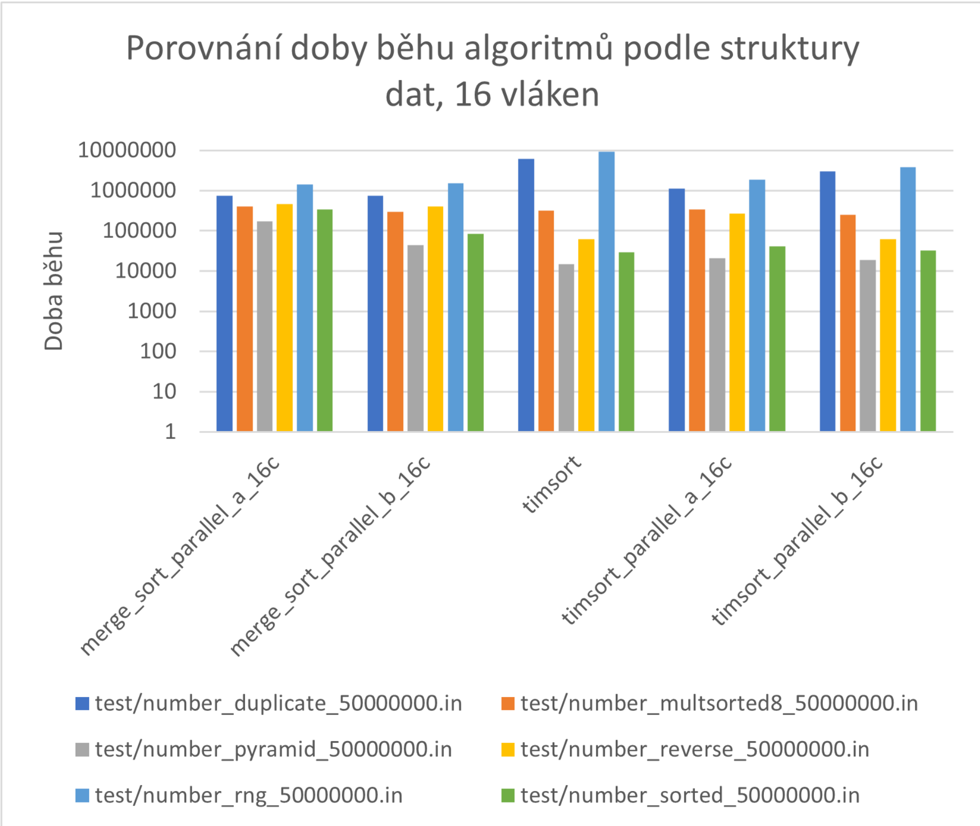
Obrázek 4.28: Porovnání doby běhu algoritmů pro 2 vlákna a různá data o stejné velikosti



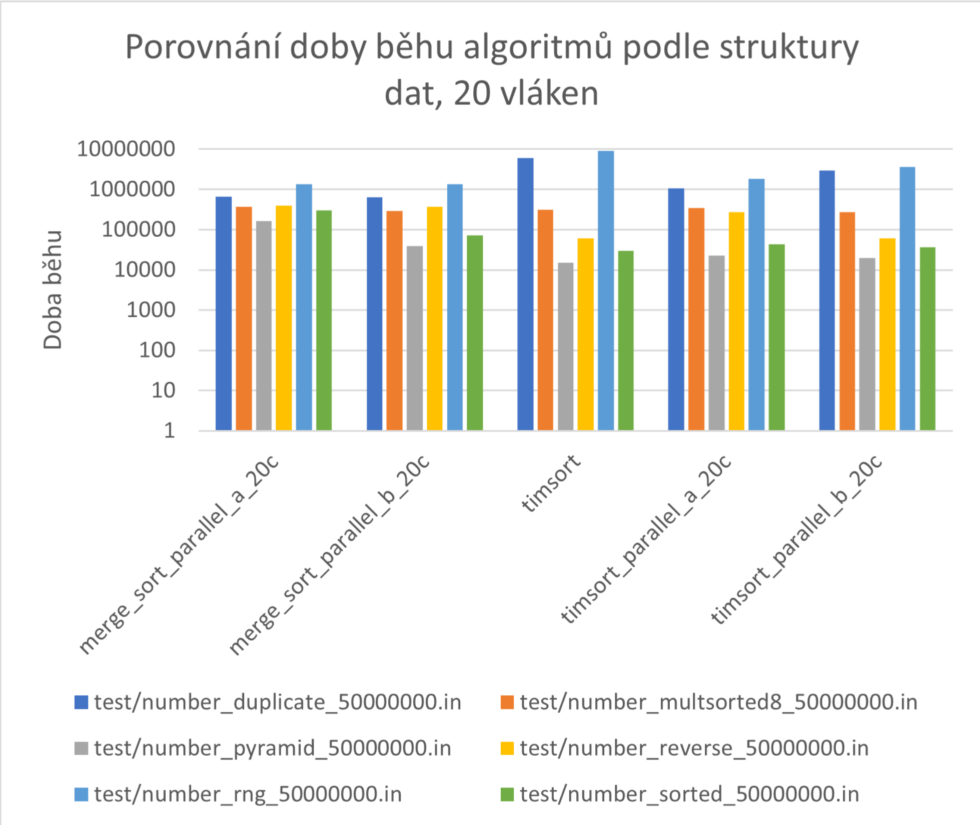
Obrázek 4.29: Porovnání doby běhu algoritmů pro 4 vlákna a různá data o stejné velikosti



Obrázek 4.30: Porovnání doby běhu algoritmů pro 8 vláken a různá data o stejné velikosti



Obrázek 4.31: Porovnání doby běhu algoritmů pro 16 vláken a různá data o stejné velikosti



Obrázek 4.32: Porovnání doby běhu algoritmů pro 20 vláken a různá data o stejné velikosti

Závěr

Prvním cílem této práce bylo studium a implementace algoritmu Timsort. Výstupem této části je funkční řadící algoritmus. Následným testováním jsme zjistili, že jeho chování odpovídá očekávání a chová se stejně jako Timsort z knihovny gfx.

Dalšími body zadání byl návrh optimalizací a paralelizací a jejich následná implementace. Vymyslel jsem proto několik možných sekvenčních optimalizací. Ty zahrnovaly hledání runů od konce pole a slučování více runů najednou. Dále jsem navrhl několik možností paralelizace algoritmu Timsort a vybrané z nich jsem implementoval. Aby byla paralelizace možná, bylo nutné vzdát se některých principů Timsortu a tím jsme přišli o zejména paměťové výhody. Paralelizace ovšem byla úspěšná a zjistili jsme, že každý paralelní algoritmus se hodí na jiný typ dat. To obecně platilo i u sekvenčních algoritmů.

Posledním bodem zadání je porovnání jednotlivých verzí algoritmu na serveru STAR. Ačkoliv testování bylo poměrně obsáhlé, objevil jsem během něj místa, které by bylo vhodné prozkoumat dále do hloubky. Narazil jsem například i na zajímavý bug s řadící funkcí less při řazení stringů. Na všechny detaily testování by nebylo v této práci místo a zmínil jsem proto pouze ty nejdůležitější a největší rozdíly mezi jednotlivými algoritmy. Ověřil jsem, že je skutečně výhodné používat Timsort jako univerzální řadící algoritmus, neboť není o tolik pomalejší při náhodných datech a zároveň je extrémně rychlý při řazení dat obsahující struktury. Zároveň používá méně porovnávacích funkcí a je tedy vhodný, pokud jsou tyto funkce pomalé. Také jsem zjistil, že paralelizovat Timsort má smysl a můžeme tím dosáhnout výrazného zrychlení řazení.

Při diskuzi výsledků jsem navrhoval, s jakými daty šly jednotlivé algoritmy využít a při jakých datech by bylo lepší volit jiný řadící algoritmus. Diskutoval jsem také možnosti využití mnou navržených optimalizací Timsortu. Došel jsem k závěru, že opět záleží na datech, která budeme řadit. Navržené optimalizace mohou algoritmus zrychlit v jednom případě a zpomalit ve druhém. Navíc bychom mohli zase přijít o paměťové úspory Timsortu oproti běžným slučovacím algoritmům.

Na tuto práci lze navázat například pokračováním testování různých dat či porovnávacích funkcí. Dále na ní lze navázat použitím zde zmíněných optimalizací a paralelizací a zkusit je aplikovat na další algoritmy.

Bibliografie

1. BAUERMEISTER, Rylan. *Medium*. Understanding Timsort [online]. 2019. [cit. 2023-05-11]. Dostupné z: https://medium.com/@rylanbauermeister/understanding-timsort-191c758a42f3.
2. cpython/listobject.c. In: *Github* [online]. 2020 [cit. 2023-05-11]. Dostupné z: https://github.com/python/cpython/blob/bcb198385dee469d630a184182df9dc1463e2c47/Objects/listobject.c.
3. rust/src/liballoc/slice.rs. In: *Github* [online]. 2018 [cit. 2023-05-11]. Dostupné z: https://github.com/rust-lang/rust/blob/5f60208ba11171c249284f8fe0ea6b3e9b63383c/src/liballoc/slice.rs#L841-L980.
4. TimSort.java. In: *Git at Google* [online]. 2010 [cit. 2023-05-11]. Dostupné z: https://android.googlesource.com/platform/libcore/+/gingerbread/luni/src/main/java/java/util/TimSort.java.
5. v8/third_party/v8/builtins/array-sort.tq. In: *Github* [online]. 2023 [cit. 2023-05-11]. Dostupné z: https://github.com/v8/v8/blob/main/third_party/v8/builtins/array-sort.tq.
6. swift/stdlib/public/core/Sort.swift. In: *Github* [online]. 2022 [cit. 2023-05-11]. Dostupné z: https://github.com/apple/swift/blob/main/stdlib/public/core/Sort.swift.
7. MERVIN, Eric. *Medium*. What is Tim Sort? [online]. 2021. [cit. 2023-05-11]. Dostupné z: https://ericmervin.medium.com/what-is-timsort-76173b49bd16.
8. PETERS, Tim. [Python-Dev] Sorting. In: *mail.python.org* [online]. 2002 [cit. 2023-05-11]. Dostupné z: https://mail.python.org/pipermail/python-dev/2002-July/026897.html.

9. PETERS, Tim. [Python-Dev] Sorting. In: *mail.python.org* [online]. 2002 [cit. 2023-05-11]. Dostupné z: https://mail.python.org/pipermail/python-dev/2002-July/026853.html.
10. PETERS, Tim. [Python-Dev] Sorting. In: *mail.python.org* [online]. 2002 [cit. 2023-05-11]. Dostupné z: https://mail.python.org/pipermail/python-dev/2002-July/026920.html.
11. cpython/Objects/listsort.txt. In: *Github* [online]. 2020 [cit. 2023-05-11]. Dostupné z: https://github.com/python/cpython/blob/bcb198385dee469d630a184182df9dc1463e2c47/Objects/listsort.txt.
12. *Wikipedia*. Exponential search [online]. 2023. [cit. 2023-05-11]. Dostupné z: https://en.wikipedia.org/wiki/Exponential_search.
13. GILBERT, Scott. [Python-Dev] Sorting. In: *mail.python.org* [online]. 2002 [cit. 2023-05-11]. Dostupné z: https://mail.python.org/pipermail/python-dev/2002-July/026900.html.
14. PETERS, Tim. [Python-Dev] Sorting. In: *mail.python.org* [online]. 2002 [cit. 2023-05-11]. Dostupné z: https://mail.python.org/pipermail/python-dev/2002-July/026902.html.
15. Maximum K for K-way merge. In: *Stack Overflow* [online]. 2017 [cit. 2023-05-11]. Dostupné z: https://stackoverflow.com/questions/41686325/maximum-k-for-k-way-merge.
16. why should we use n-way merge? what are its advantages over 2-way merge? In: *Stack Overflow* [online]. 2013 [cit. 2023-05-11]. Dostupné z: https://stackoverflow.com/questions/14713468/why-should-we-use-n-way-merge-what-are-its-advantages-over-2-way-merge.
17. *Wikipedia*. k-way merge algorithm [online]. 2023. [cit. 2023-05-11]. Dostupné z: https://en.wikipedia.org/wiki/K-way_merge_algorithm.
18. *Wikipedia*. External sorting [online]. 2023. [cit. 2023-05-11]. Dostupné z: https://en.wikipedia.org/wiki/External_sorting.
19. *Wikipedia*. OpenMP [online]. 2021. [cit. 2023-05-11]. Dostupné z: https://cs.wikipedia.org/wiki/OpenMP.
20. SOOD, Saurabh. *Saurabh Sood's Blog*. Parallelizing TimSort [online]. 2017. [cit. 2023-05-11]. Dostupné z: https://saurabhsoodweb.wordpress.com/2017/04/18/parallelizing-timsort/.
21. BUKATA, Libor; DVOŘÁK, Jan. *Advanced programming with OpenMP* [online]. [B.r.]. [cit. 2023-05-11]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/b4m35pag/lab6_slides_advanced_openmp.pdf.
22. timsort/cpp-Timsort: A c++ implementation of timsort. In: *Github* [online]. 2022 [cit. 2023-05-11]. Dostupné z: https://github.com/timsort/cpp-TimSort.

23. *FIT CTU Courses*. Překlad, ladění a spuštění paralelní programu na klastru STAR • NI-PDP • FIT ČVUT Course Pages [online]. 2023. [cit. 2023-05-11]. Dostupné z: `https://courses.fit.cvut.cz/NI-PDP/labs/run-star.html`.

24. *FIT CTU Courses*. Výpočetní klastr Star • NI-PDP • FIT ČVUT Course Pages [online]. 2022. [cit. 2023-05-11]. Dostupné z: `https://courses.fit.cvut.cz/NI-PDP/labs/klastr-star.html`.

25. *cppreference.com*. Date and time utilities [online]. 2022. [cit. 2023-05-11]. Dostupné z: `https://en.cppreference.com/w/cpp/chrono`.

26. *Wikipedia*. Well equidistributed long-period linear [online]. 2022. [cit. 2023-05-11]. Dostupné z: `https://en.wikipedia.org/wiki/Well_equidistributed_long-period_linear`.

Dodatek A

Seznam použitých zkratek

RAM Random Access Memory

SGE Sun Grid Engine

Dodatek B

Obsah přiloženého CD

- readme.txt ... stručný popis obsahu CD
- exe ... adresář se spustitelnou formou implementace
- src
  - impl ... zdrojové kódy implementace
  - thesis ... zdrojová forma práce ve formátu LaTeX
- text ... text práce
  - thesis.pdf ... text práce ve formátu PDF
- results ... výsledky testování použity v této práci