FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Efektivní paralelní Timsort algoritmus
Student:	Jan Píro
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte sekvenční verzi algoritmu Timsort. [1,2,3]
2) Diskutujte možnosti optimalizace a paralelizace tohoto algoritmu [4].
3) Implementujte vybrané optimalizace sekvenční i paralelní verze algoritmu.
4) Porovnejte výkonnost jednotlivých verzí optimalizace a paralelizace (navzájem a i s neoptimalizovanou sekvenční verzí) na školním serveru STAR a diskutujte dosažené výsledky.

Seznam odborné literatury

[1] http://www.drmaciver.com/2010/01/understanding-timsort-1adaptive-mergesort/
[2] https://hal-upec-upem.archives-ouvertes.fr/hal-01212839v2/document
[3] https://mail.python.org/pipermail/python-dev/2002-July/026837.html
[4] http://rabie-ben-atitallah.com/paper/hpcs-2017.pdf

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

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

V Praze dne 7. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Efektivní paralelizace algoritmu Timsort

Jan Píro

Katedra Teoretické Informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

15. května 2018

Poděkování

Rád bych zde poděkoval vedoucímu bakalářské práce doc. Ing. Ivanu Šimečkovi, Ph.D. za jeho rady, čas a trpělivost, kterou mi věnoval.

Prohlášení

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

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

V Praze dne 15. května 2018 .....

Odkaz na tuto práci

Píro, Jan. *Efektivní paralelizace algoritmu Timsort.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato bakalářská práce se zaměřuje na třídící algoritmus Timsort. Cílem bylo paralelizovat algoritmus tak, aby byl efektivnější než jeho sekvenční verze a porovnat jeho rychlost s jinými algoritmy.

Klíčová slova timsort, paralelní timsort, třídící algoritmy, openMP, C++ Thread support library

Abstract

This thesis focuses on sorting algorithm Timsort. The goal is to paralelize the algorithm in a way to make it more efficient than its sequential version and compare its speed with other algorithms.

Keywords timsort, parallel timsort, sorting algorithms, openMP, C++ Thread support library

Obsah

Seznam obrázků

Úvod

S problémem seřadit za sebe nějaké prvky podle určitých kritérií se setkáváme v téměř každém pracovním odvětví. Algoritmy zabývající se těmito problémy se nazývají třídící algoritmy a jsou jedny z nejpoužívanějších algoritmů vůbec. Používají se všude od databází, přes umělou inteligenci až po vojenské technologie. Toto odvětví je tedy stále zkoumáno s cílem zrychlit a zefektivnit stávající algoritmy, případně vymyslet nové.

Jejich složitost posuzujeme především dle počtu operací srovnání dvou prvků (a tedy zjištěním, který z prvků má být ve výsledné řadě dříve). Dále se posuzují podle operační paměti, kterou potřebují alokovat (tedy paměť alokovaná nad rámec zadané řady prvků).

Jedním z takových algoritmů je Timsort, který byl navržen roku 2002 Timem Petersem. Je používán jako standardní třídící algoritmus v Pythonu od verze 2.3 a v Javě SE 7. Jedná se o optimalizovanou kombinaci již známých algoritmů MergeSort a InsertionSort.

V teoretické části je pospsán samotný algoritmus a jsou zde diskutovány možnosti jeho optimalizace a paralelizace. V praktické části jsou vytvořeny verze s různými stupni optimalizace a paralelizace, měřeny jejich výkony a jednotlivé verze porovnány.

Kapitola 1

Cíl práce a základní pojmy

1.1 Cíl a motivace

Timsort má operační složitost v nejhorším případě $\mathcal{O}(n \log n)$, ale v praxi provádí u částečně seřazených polí mnohem méně porovnávání. Jeho nevýhoda je, že podstatná část algoritmu je závislá na vykonávání se v určitém pořadí a tedy je složité ho paralelizovat.

Jeho efektivní paralelizace by mohla vést ke zrychlení standardních algoritmů v různých jazycích a tím i ke zrychlení aplikací, které je využívají.

Hlavním cílem práce je tedy implementace paralelní verze algoritmu Timsort a její porovnání s jinými algoritmy, sekvenční verzí a různými paralelními verzemi. Měření chci provést pro různé typy dat abych měl o efektivitě řešení co nejpřesnější představu.

Pro paralelizaci budu využívat technologii OpenMP a C++ Thread support library, podle konkrétního postupu paralelizace.

1.2 Základní pojmy

V práci budu pracovat s následující termíny:

třídící algoritmy jsou skupina algoritmů zabývající se řazením pole prvků dle určitého kritéria, typicky podle jejich velikosti

vlákno je proud instrukcí, který je zpracováván jádrem CPU [1]

sekvenční algoritmus běží celý pouze na jednom vlákně

paralelní algoritmus využívá v některých svých částech více vláken

run seřazená část tříděného pole v algoritmu Timsort

mutex, condition variable a semaphore jsou synchronizační typy, používané k zajištění výlučného přístupu k datům, případně synchronizace posloupnosti provádění operací na více vláknech [2][3][4]

Merge (slučování) je proces, při kterém spojíme dvě seřazená pole do jednoho většího seřazeného pole

Galloping (cval) je metoda používaná pro rychlejší zjištění délky seřazené posloupnosti menší (resp. větší) než daný prvek

cache je paměť na procesoru, sloužící k rychlejšímu přístupu k datům, se kterými procesor pracuje

1.3 Použité technologie

1.3.1 OpenMP

Pro naivní paralelizaci algoritmu jsem využil technologii OpenMP.

Je to soustava direktiv pro překladač a knihovních procedur pro paralelní programování. Jedná se o standard pro programování počítačů se sdílenou pamětí. OpenMP usnadňuje vytváření vícevláknových programů v programovacích jazycích Fortran, C a C++. [5]

Důvodem ke zvolení této technologie byl fakt, že při použití této technologie programátorovi stačí označit blok kódu, který se má provádět paralelně a překladač to tak naimplementuje. [6]

1.3.2 C++ Thread support library

V komplikovanější druhé verzi paralelního algoritmu bych si s technologií OpenMP nevystačil a musel jsem paralelizovat manuálně. Využívám k tomu C++ Thread support library. Tato knihovna obsahuje mimo jiné třídy a funkce pro správu vláken (vytvoření a zrušení), synchronizaci mezi nimi (uzamykání kritických sekcí) a jejich plánování a spouštění. [7]

Kapitola 2

Algoritmus Timsort

V této práci vycházím z algoritmu tak jak byl pospán v původním Petersově návrhu. Dále jsem čerpal z textů Merge Strategies: from Merge Sort to TimSort a Understanding timsort, Part 1: Adaptive Mergesort. [8], [9], [10]

Timsort je ve své podstatě modifikovaný Merge sort. Algoritmus hledá sekvence již seřazených dat (těmto se říká runy) a průběžně je podle jistých pravidel slučuje.

Další modifikace algoritmu oproti mergesortu spočívá v minimální velikosti těchto runů, tato velikost (*minrun*) se spočítá na základě velikosti celého tříděného pole pomocí jednoduché formule. Pokud je run menší než je velikost *minrun*, pak se uměle zvětší o následující prvky v poli pomocí insertion sortu.

Ve vlastním slučování je implementováno několik dalších heuristik, které zmenšují velikost slučovaných polí (nalezení prvků, které by se při slučování nepohly) a snižují počet porovnávání u částečně seřazených polí.

2.1 Hledání runů

Nejprve algoritmus najde v poli postupným procházením podpousloupnosti, které jsou buď neklesající

$$(a0 <= a1 <= a2 <= ...)$$

nebo ostře klesající

$$(a0 > a1 > a2 > ...)$$

Ostře klesající podposloupnost následně otočíme, postupným prohazováním prvků na začátku a konci ještě nesetříděné podposloupnosti. Tato posloupnost musí být ostře klesající, aby se zabránilo případné destabilizaci algoritmu.

Při práci s náhodnými daty je velmi nepravděpodobné, že narazíme na dlouhé runy. Pokud algoritmus narazí na run kratší než je daná hodnota *minrun* doplní tento run do požadované délky pomocí insertion sortu.

Minrun volíme z intervalu $(32, 64)$ takové, aby $N/minrun$ bylo pokud možno rovné 2^i, kde $i \in \mathbb{N}$. N je zde celkový počet prvků v poli. Pokud toto není možné, chceme aby se $N/minrun$ blížilo některé mocnině dvou zezdola. Tohoto dosáhneme snadno pomocí následujícího algoritmu. Vezměme prvních 6 bitů z N a pokud nejsou všechny zbývající bity rovny nule přičtěme jedničku. Tento výběr pokrývá všechny případy včetně malých N.

V kódu 2.1 vidíme jak algoritmus pracuje. Proměnná r značí zda je některý z bitů po prvních 6 bitech N nastavený na 1. Konstanta *max_merge* je rovná 64 a slouží k zajištění právě 6 prvních bitů čísla N.

Ukázka kódu 2.1: Počítání hodnoty minrun

```
int CountMinRun(int N) {
    int r = 0;
    while (N >= max_merge) {
        r |= (N & 1);
        N >>= 1;
    }
    return N + r;
}
```

2.2 Strategie slučování

Z důvodu využívání částečně setříděných dat můžeme získat runy různých délek. S ohledem na stabilitu můžeme slučovat pouze sousedící runy.

Když nalezneme run, vložíme jeho počátek a délku na zásobník. Poté se kontroluje stav, a rozhoduje se zda budeme runy slučovat, či nikoliv. Slučování nechceme příliš oddalovat, abychom neměli příliš velký zásobník a abychom využili faktu, že právě přidané runy jsou na vyšší pozici v paměťové hiearchii. Na druhou stranu chceme ale slučovat co nejefektivněji a tedy formovat strategii na zakládě informací získaných z více runů.

Merge probíhá rychleji na podobně dlouhých polích. Abychom docílili postupného vyrovnávání délek snažíme se na zásobníku dodržovat dvě nerovnice:

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

$$2.\ B > C$$

Kde A, B, C jsou velikosti posledních tří runů vložených, v tomto pořadí, na zásobník. Druhá nerovnice zajišťuje, že na zásobníku máme klesající posloupnost délek runů a první zajišťuje rychlost růstu délek, počínaje od posledního prvku zásobníku, je minimálně taková, jako rychlost růstu Fibonacciho posloupnosti, a tedy zajištění maximální velikosti zásobníku rovnou $log_\phi N$, kde $\phi \approx 1.618$ je zlatý řez.

Pokud jsou tyto nerovnice porušeny je B sloučeno z kratším z runů A a C. Pokud jsou runy A i C stejně dlouhé, slučujeme B s C, z důvodu čerstvosti v paměti a tedy využíváním cache. Na zásobníku sloučené runy nahradíme jejich výsledným a znovu zkontrolujeme nerovnice.

2.3 Algoritmy slučování

Slučujeme-li dva runy o délkách A a B chceme ušetřit pomocnou paměť kterou využíváme. Jeden z kroků, který proto podnikáme je oříznutí těchto polí o prvky, se kterými již nebudeme hýbat. Pomocí binárního vyhledávání nalezneme pozici $B[0]$ v A a $A[max]$ v B. Prvky v A, před pozicí $B[0]$, resp. prvky v B za pozicí $A[max]$ jsou již setříděné a tedy můžeme pracovat s poli o tyto prvky zkrácenými.

Porovnáme délky zkrácených polí $A1$ a $B1$ a vytvoříme pomocnou paměť o délce kratšího z nich, do které poté kratší pole nahrajeme a zahájíme slučování.

Podle toho, které pole jsme nahráli do pomocné paměti budeme postupovat od nejmenšího nebo největšího prvku a tedy slučovat pole zleva respektive zprava. Obě tyto verze jsou v podstatě analogické, jen zrcadlově obrácené.

Ukázka kódu 2.2: Merge

```
void Merge(Run *a, Run *b) {
    //Hledej a[last] v b
    //Hledej b[0] v a
    //orez slucovana pole

    if (lengthA <= lengthB) {
        MergeLo(startA, startB, lengthA, lengthB);
    } else {
        MergeHi(startA, startB, lengthA, lengthB);
    }

    //Spoj runy na zasobniku
}
```

Ukázka kódu 2.3: MergeLo

```
void MergeLo() {
    NahrajADoTmpArray();
    while (1) {
        if (gallop > 0) {
            //proved galloping
            //zkontroluj zda se galloping vyplatil
        } else {
            //proved linarni pridavani
            if (prekrocen min_gallop)
                //prejdi do gallopingu
        }

        if (dosahl jsem konce jednoho pole) {
            //vypis zbytek druheho pole
            break;
        }
    }
}
```

2.4 Galloping

Klasické slučování používá mezi A a $A + B$ porovnávání. Pokud jsou ovšem data částečně setříděná, můžeme předpokládat že budeme z obou polí vybírat větší celky najednou. Zde můžeme ušetřit operace porovnávání hledáním jak dlouhou sekvenci z jednoho pole budeme vybírat. K tomu použijeme metodu tzv. cvalu (angl. galloping) neboli exponenciálního vyhledávání.

Při hledání pozice $A[0]$ v B, porovnáváme $A[0]$ postupně s $B[0]$, $B[1]$, $B[3]$, $\ldots$, $B[2^i - 1]$, $\ldots$ dokud nenalezneme k takové, že

$$B[2^{k-1} - 1] < A[0] \leq B[2^k - 1]$$

Poté ve vyhledaném úseku nalezneme pozici $A[0]$ pomocí binárního vyhledávání.

Problém s tímto vyhledáváním je ten, že pro $k \leq 6$ provádí lineární vyhledávání stejně, nebo dokonce méně porovnání. Z tohoto důvodu nechceme provádět exponenciální vyhledávání pokud bychom neměli alespoň určité náznaky, že se nám vyplatí. Do módu cvalu se tedy bude algoritmus přepínat pouze tehdy, bude-li brát MIN_GALLOP počet prvků z jednoho pole za sebou.

Ukázka kódu 2.4: Galloping

```
int GallopLeftInTmp() {

    //najdi k takove, ze:
    //B[2^{k-1}-1] < A[0] <= B[2^k-1]

    //pomoci binarniho hledani najdi
    //v danem useku m
    //takove, ze B[m-1]<A[0]<=B[m]

    return m;
}
```

Proměnná MIN_GALLOP bude mít výchozí hodnotu rovnou 7, tedy k takové, pro které cval vítězí nad lineárním vyhledáváním. Tuto proměnnou zvýšíme kdykoli se nám přepnutí na cval nevyplatilo (abychom předešli náhodným přepnutím v polích, kde se zjevně nevyplatí) a naopak sníží pokud se nám vyplatí, abychom se do módu mohli snadněji vrátit, pokud bychom z něj vypadli v poli, kde se nám celkově vyplácí.

Kapitola 3

Analýza a návrh

Při návrhu jsem se snažil o to, aby timsort nabízel stejné využití jako *std* :: *sort*. Pracoval jsem ve dvou fázích.

3.1 Možnosti optimalizace sekvenční verze

Timsort je ve své podstatě velice dobře softwarově optimalizovaný pro sekvenční běh programu. Zde se tedy nabízí pouze využití překladačových optimalazicí.

Z výhod sekvenční verze Timsortu zmíním efektivní práci s cache na procesoru (pracuje se primárně na datech která byla v nedávné době zkoumána a tedy je velká pravděpodobnost, že se stále v cache nachází) a efektivní volení slučování runů stejné délky.

3.2 Možnosti paralelizace

Možností paralelizace se nabízí několik:

3.2.1 Naivní verze

Zjevný a na první pohled nejspíše i nejúspěšnější postup je u velkého pole (u malých by paralelizace naopak spíš zdržovala) rozdělení na X zhruba stejně velkých částí (X je počet CPU které máme k dispozici) a následném spuštění algoritmu paralelně na každé části zvlášť, tyto části nakonec přidám jako runy do klasického Timsortu kde dojde k jejich, teď už sekvenčnímu, sloučení.

3.2.2 Pokročilá verze

Druhou možností je zavrhnout pravidla pro postupné slučování a slučovat runy najednou, jak navrhuje Saurabh Sood [11] Zde sice může nastat situ-

ace neefektivního slučování (dlouhý run s krátkým), nicméně zisk obdržený paralelizací by toto měl více než vykompenzovat.

Pro každý run vytvoříme samostatné vlákno, kterému přidělím pořadové číslo i. Toto vlákno spustí metodu *Check*, které předá i a ukazatel na run, a po jejím provedení uvolní run ke zpracování ostatními vlákny a ukončí se. Metoda *Check* pro lichá i neprovede nic a ukončí se. Pro sudá i vlákno sloučím s runem předchozím (runy jsou ukládány ve spojovém seznamu) a rekurzivně opět zavolám metodu *Check* tentokrát s parametrem $i/2$. n

3.2.3 Další možnosti

Zvažoval jsem i možnost paralelizace algoritmu Timsort tak jak je, tedy nechat běžet hledání runů na jednom vlákně nicméně během analýzy jsem dospěl k tomu, že při řízení se postupným slučováním runů bych paralelizaci pořádně nevyužil. Mohl bych ji využít k prohledávání pole zároveň se slučováním prvních nalezených runů, nicméně u této varianty by mi běželi maximálně dvě vlákna najednou a běh vlákna hledajícího runy je mnohonásobně kratší, než běh vlákna které by nalezené runy slučovalo a tak jsem tuto variantu zavrhl.

Kapitola 4

Realizace

Implementoval jsem tři verze algoritmu:

- Sekvenční verze
- Paralelní verze s OpenMP
- Paralelní verze s C++ Thread

Tyto jsem porovnával s verzí Timsortu podle gfx [12] a funkcí sort z knihovny std. [13]

4.1 Kompilace

Pro dodatečné kompilátorové optimalizace a využití možností procesorů v testovaném prostředí jsem všechny verze kompiloval s přepínači -std=c++11 -O3 -march=corei7-avx.

Dále jsem z důvodů využití technologie OpenMP při kompilaci naivní paralelní verze přidal přepínač -fopenmp.

U pokročilé paralelní verze jsem použil přepínač -pthreads.

4.2 Sekvenční verze

Sekvenční verzi jsem implementoval přesně podle algoritmu popsaného ve třetí kapitole. Pro algoritmus implementuji třídu TimSort, která implementuje funkce sloužící k hledání runů, třídu StackTimSort, které implementuje funkce slučování a udržuje zásobník s runy, a třídu RunTimSort, která reprezentuje jednotlivé runy a udržuje informaci o nich.

Pro dodatečné kompilátorové optimalizace a využití možností procesorů v testovaném prostředí jsem tuto verzi kompiloval s přepínači -std=c++11 -O3 -march=corei7-avx.

4.2.1 Třída TimSort

Třída TimSort má na starosti hledání runů a jejich přidávání na zásobník, implementuje tyto metody:

Sort Hlavní metoda, kde se spouští hledání runů a dále se, po projetí celého seznamu, sloučí ještě nesloučené runy

SearchForRuns Metoda, která vyhledává runy a přidává je na zásobník implementovaný třídou StackTimSort

CountMinRun Pomocná metoda, která vypočítá velikost *minrun* pro danou délku pole

InsertUntilMinRun Metoda, která pomocí insertion sort prodlužuje nalezený run na délku *minrun*

BinarySearch Pomocná metoda, použitá v metodě InsertUntilMinrun, vyhledává pozici přidáváného prvku v již seřazené části runu

Reverse Pokud je nalezený run sestupný, tato metoda ho obrátí

RunLength Hledá sekvenci již seřazených prvků, vrací délku sekvence a informaci o tom, zda je sekvence klesající, či neklesající

a uchovává tyto proměnné:

array ukazatel na začátek tříděného pole

length délka tříděného pole

stack ukazatel na třídu reprezentující zásobník

4.2.2 Třída StackTimSort

Třída StackTimSort reprezentuje zásobník nalezených runů a má na starosti kontrolování pravidel na vrcholu zásobníku a slučování runů. Implementuje tyto metody:

AddRun Metoda, která přidá na zásobník nalezený run s danými parametry a poté zavolá metodu *Check*

Check Meto, která kontroluje dodržování pravidel na vrcholu zásobníku (a tím omezuje jeho maximální velikost), pokud jsou pravidla porušena, zavolá metodu *Merge*

Merge Metoda, která provede přípravné operace před vlastním slučováním, které provádí metody *MergeHi* a *MergeLo*, a následně spojí runy do toho novějšího a starší vymaže.

MergeHi + MergeLo Metody, které provádí samotné slučování, liší se v tom zda postupují zleva či zprava.

ExponentialSearchLeft + ExponentialSearchRight Metody tzv. cvalu, pro nalezení pozice prvku v poli

GallopLeftInTmp + GallopRightInTmp Metody tzv. cvalu, pro nalezení pozice prvku v poli, na rozdíl od *ExponentialSearch* hledá ve vytvořeném pomocném poli

Dále uchovává tyto proměnné:

array ukazatel na začátek tříděného pole

length délka tříděného pole

last ukazatel na poslední run na zásobníku

count počet runů na zásobníku

min_gallop udává po kolika přidání ze stejného pole se přeskočí na mód cvalu

4.2.3 Třída RunTimSort

Třída RunTimSort udržuje informace o jednotlivých runech a skládá se z těchto proměnných:

start pozice začátku setříděného run v poli

length délka setříděného runu

previous ukazatel na předchozí run na zásobníku

next ukazatel na následující run (defaultně *nullptr*)

4.3 Paralelní verze 1

V této verzi jsem využil technologii OpenMP. Vlastní pole jsem při překročení $PARALLEL\_MIN$ prvků, rozdělil na $NUM\_THREADS$ částí.

$PARALLEL\_MIN$ a $NUM\_THREADS$ jsou konstanty, které je potřeba volit podle hardwaru na kterém má algoritmus běžet.

Každá část pracuje se svým vlastním zásobníkem. Tyto seřazené části jsou pak přidávány na hlavní zásobník, který je postupně slučuje podle pravidel Timsortu. Následně se sloučí ještě nesloučené části. Vzhledem k tomu, že počet částí je stejný jako počet vláken, a tedy bude každé vlákno zpracovávat právě jednu iteraci for cyklu, nemusím řešit způsob přidělování práce vláknům.

Vnitřek algoritmu funguje obdobně jako sekvenční verze.

Ukázka kódu 4.1: Parallel Sort

```
#pragma omp parallel for num_threads(NUM_THREADS)
for (int i=0; i < NUM_THREADS ; i++) {
    1. vytvoreni zasobniku pro tento beh Timsortu
    2. provedeni Timsortu na danem useku
    3. smazani zasobiku
}
```

4.4 Paralelní verze 2

V druhé paralelní verzi vlákna implementuji manuálně pomocí C++ Thread support library. Využívám knihovny *thread*, *mutex* a *condition_variable*. Využil jsem také třídu *Semaphore* podle StackOverflow. [14] K implementaci mi přibyla třída TimSortTask, která uchovává informace o zadané práci pro jednotlivá vlákna. Tato informace je ve třídě uložena ve formě ukazatele na run, který bude pro operace slučování výchozí a čísla *num*, které v sobě má uloženou informaci o zda a kolikrát se má run sloučit s předchozím runem, jak bude vysvětleno později. Jedná se v podstatě o implementaci problému producenta a konzumenta. [2]

Ukázka kódu 4.2: TimSortTask

```
class TimSortTask {
public:
    RunTimSort * run;
    int num;
};
```

Metoda *Sort* nejprve vygeneruje $NUM\_THREADS - 1$ vláken na kterých běží metoda *ThreadRun*. Následně se na hlavním vlákně, stejně jako v sekvenční verzi, zavolá metoda *SearchForRuns*, která nalezené runy přidá na zásobník a zároveň přidá do fronty úkolů ukazatel na tento run, společně s číslem *num*, které v tuto chvíli není nic jiného než pořadové číslo nalezeného runu. Tedy $num \geq 1$.

Poté co doběhne metoda *SearchForRuns* se i na hlavním vlákně spustí metoda *ThreadRun*. Tímto způsobem mohu mít neustále zaměstnaných až $NUM\_THREADS$ vláken. Ke konci běhu algoritmu ovšem už není pro vlákna tolik úkolů a nemusí běžet všechna.

Význam čísla *num* je následující. Pro pořadí slučování runů nemám nyní žádná pravidla, a mohl bych tedy slučovat runy postupně jak přicházejí, tento postup ale neumožňuje paralelizaci. Nicméně pokud zde napodobím mergesort, a budu runy slučovat po párech, paralelizace je nejen možná, ale je dokonce i vhodná. Abych tohoto docílil předávám úkolu právě i pořadové číslo

daného runu. Pokud je toto číslo liché, vím, že tento run s jemu předcházejícím runem nemám slučovat, jelikož slučuji právě každý druhý run s runem předchozím. Navíc, po sloučení - a tedy změně předchozího runu - toto číslo vydělím dvěmi a opět zkoumám jeho sudost. Nachází se totiž o jednu hladinu blíže vrcholu pomyslného binárního stromu, kterým se toto slučování dá reprezentovat.

Na následujícím příkladu vidíme jak do sebe struktury uchovávající runy se sudým číslem během slučování pojmou předcházející runy.

$$run1(num = 1) \rightarrow run2(num = 2) \qquad run3(num = 3) \rightarrow run4(num = 4)$$
$$\searrow \qquad\qquad\qquad\qquad \swarrow$$
$$run2(num = 1) \rightarrow run4(num = 2)$$
$$\downarrow$$
$$run4(num = 1)$$

Toto s sebou při paralelizaci nese riziko, že se run bude snažit sloučit s runem, který ještě nedoběhl svou sekvenci slučování. Tento problém řeším pomocí semaforu, který se odemkne až ve chvíli, kdy je tento run označený za uzavřený. Tím upozorní vlákno, které na něj případně čeká, že je již k dispozici a slučování může začít.

4.5 Druhá fáze testování

Před druhou fázi testování jsem algoritmy přebudouval na formát $std :: sort$, tedy aby pro třídění šli použít různé kontejnery, aby algoritmus přijímal srovnávácí funkce apod. Hlavní části algoritmů jsem ovšem ponechal v původní podobě.

Kapitola 5

Metody měření

5.1 server STAR

Server na kterém měřím rychlosti jednotlivých verzí má dva šestijádrové procesory Intel® Xeon® Processor E5-2620 v2. [15] [16] Celkově tedy může současně běžet až 12 vláken, odtud se odvíjí mnou zvolené konstanty pro počet vláken. Velikost RAM je 32 GB a tedy se na ní s velkou rezervou vejdou všechna data, která jsem pro měření výkonů použil.

5.2 Typy dat

Pro testování jsem si vytvořil data několika typů:

Seřazená data Data jsou seřazena vzestupně

Náhodná data Data jsou generována náhodně

Částečně seřazená data Data jsou rozdělena do několika bloků, které jsou seřazené vzestupně

Málo různých hodnot Data jsou generována náhodně z malého počtu různých hodnot

Seřazená data s výjimkami Data jsou řazena vzestupně s 1% pravděpodobností náhodného čísla na každé pozici

Všechna data stejná Všechna data jsou mají stejnou hodnotu

Data jsem ukládal v binárních souborech, kvůli úspoře místa na disku (i tak jsem pracoval s několika GB dat). Na jejich vytvoření jsem si připravil program *data_creator*. Tento program vygeneruje zadané množství dat daného typu.

5.3 Metodika

K měření času jsem využil knihovny *chrono*, *ratio* a *otime*. Řídím se návodem z předmětu BI-EIA. [17]

Testoval jsem ve dvou fázích.

5.3.1 První fáze

V první fázi jsem testoval nezobecněné algoritmy na poli integerů s operátorem $<$. Toto měření bylo důležité pro základní odhad efektivnosti jednotlivých verzí a porovnával jsem ho s cizí implementací sekvenční verze timsortu a *std* :: *sortem*.

Pro testy jsem si vygeneroval data velikostí 1 000 000, 5 000 000, 10 000 000, 50 000 000, 100 000 000 a 500 000 000 prvků.

Nad těmito daty jsem provedl třídění každou metodou 5× pro získání přesnějších výsledků. Tyto výsledné hodnoty jsem následně porovnal s výsledky třídění pomocí *std* :: *sort* a vytvořil grafy poměru rychlosti jednotlivých verzí právě vůči *std* :: *sort*. Tento způsob zobrazování výsledků je v grafu přehlednější, než pouhé časy, neboť čas třídění je velmi odlišný pro jednotlivé velikosti dat i verze algoritmu.

5.3.2 Druhá fáze

V druhé fázi jsem měřil na stejných typech dat, nicméně jsem je pro účely měření reprezentoval jinak. Všechny verze jsem již měl zobecněné pro stejné užití jako *std* :: *sort*. Srovnával jsem *vector* třídy MyClass, která reprezentovala data pomocí pole 32 booleanů. Použitý komparátor nejprve poskládal z pole booleanů původní integer zpět a až poté provedl samotné porovnání. Tento postup měl simulovat řazení složitějších objektů.

Pro testy jsem si vygeneroval data velikostí 1 000 000, 5 000 000, 10 000 000 a 50 000 000 prvků. S více prvky jsem již netestoval, jelikož třídit velká pole s velice pomalou srovnávací funkcí by trvalo příliš dlouho.

Opět jsem provedl měření nad těmito daty 5× a výsledné hodnoty zanesl do grafu jako poměr vůči *std* :: *sort*.

Kapitola 6

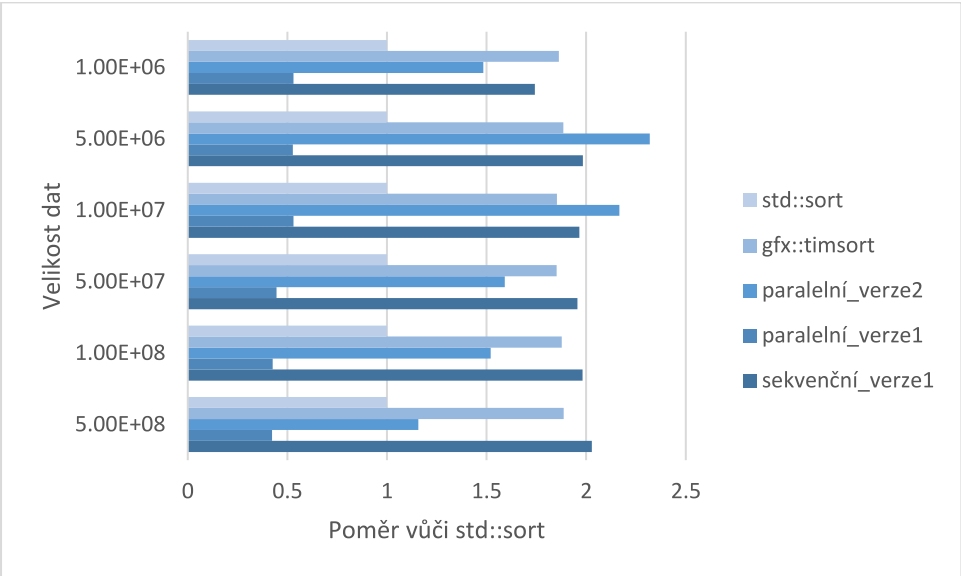
Výsledky měření

6.1 První fáze

6.1.1 Náhodná data

Při pohledu na graf náhodných dat 6.1 si můžeme všimnout, že *std* :: *sort* je kromě naivní paralelní verze nejrychlejší. Toto pozorování odpovídá výsledkům, které naměřil Tim Peters. [18]

Obrázek 6.1: Náhodná data

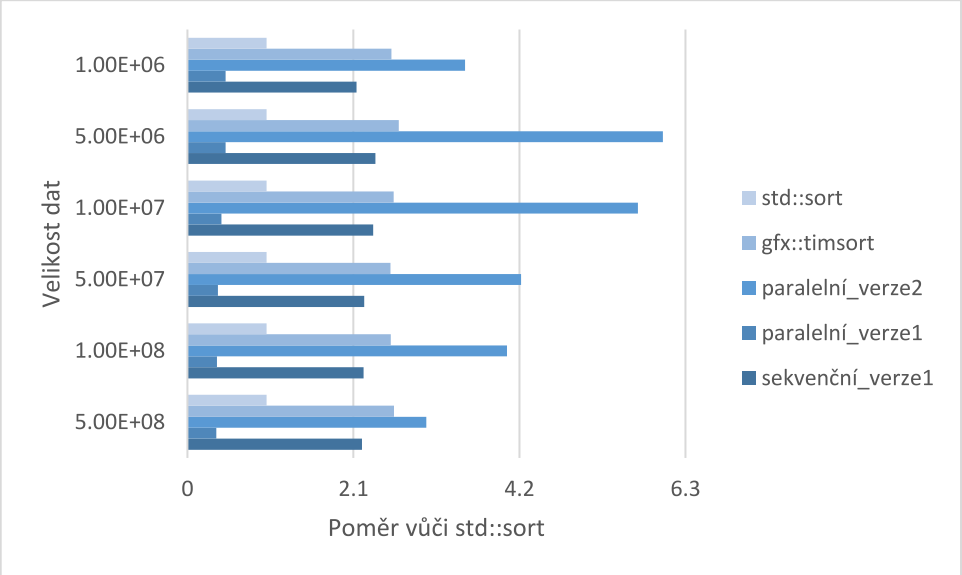


6.1.2 Málo různých hodnot

Na grafu měření dat s málo různými hodnotami 6.2 opět vidíme, že *std* :: *sort* je rychlejší. Opět můžeme stejný trend pozorovat i u Petersových výsledků.

[18]

Obrázek 6.2: Málo ruzných hodnot



6.1.3 Seřazená data s výjimkami

Na grafu měření dat s vyjímkami 6.3 je naivní paralelizace suveréně nejrychlejší, pokročilá paralelizace je zde podobně rychlá jako *std* :: *sort* a tedy několikanásobně pomalejší, než sekvenční verze.

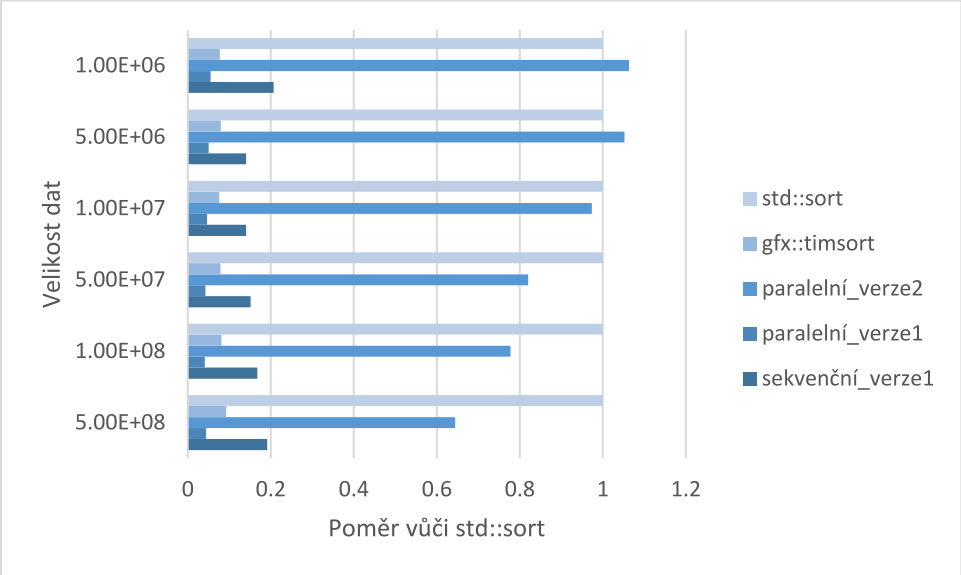
6.1.4 Seřazená a stejná data

Za povšimnutí stojí téměř stejný graf pro seřazená data 6.5 a graf stejných hodnot 6.4. Pro timsort jsou tato data prakticky totožná, pole stejných hodnot je totiž také již seřazené a timsort tedy celé pole projde jen jednou. Zrychlení oproti *std* :: *sort* je téměř desetinásobné u všech variant.

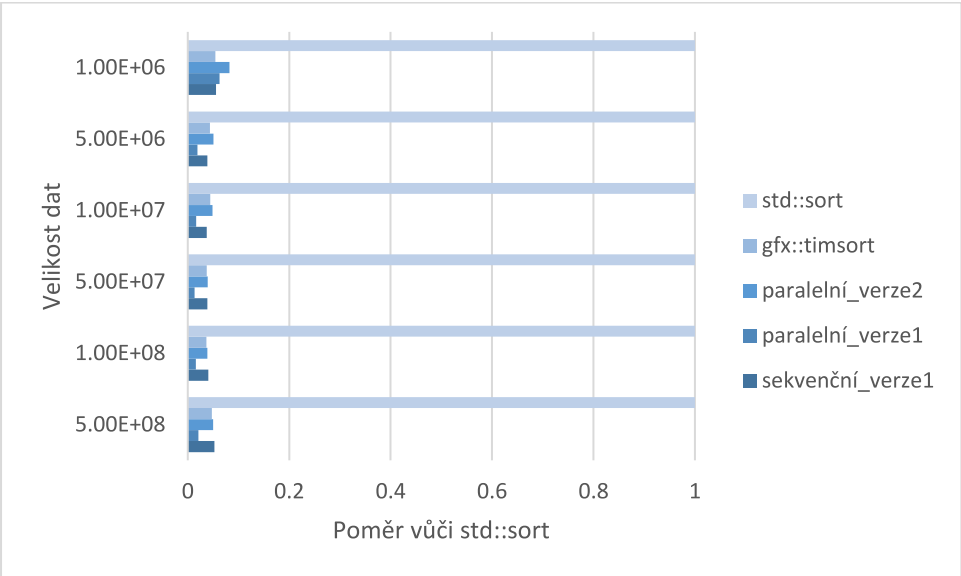
6.1.5 Částečně seřazená

To samé se dá říct i o měření na částečně seřazených datech 6.3 s tím rozdílem, že na těchto datech je pokročilá paralelizace rychlejší, než ta naivní. Tento jev bude způsobený tím, že při naivní paralelizaci dělíme pole na více částí, než kolik máme v datech vnořených seřazených podposloupností. Z tohoto důvodu provádí naivní paralelizace více slučování. Nicméně, jak je vidět, zrychlení vůči sekvenční verzi nenípříliš znatelné.

Obrázek 6.3: Seřazená data s výjimkami



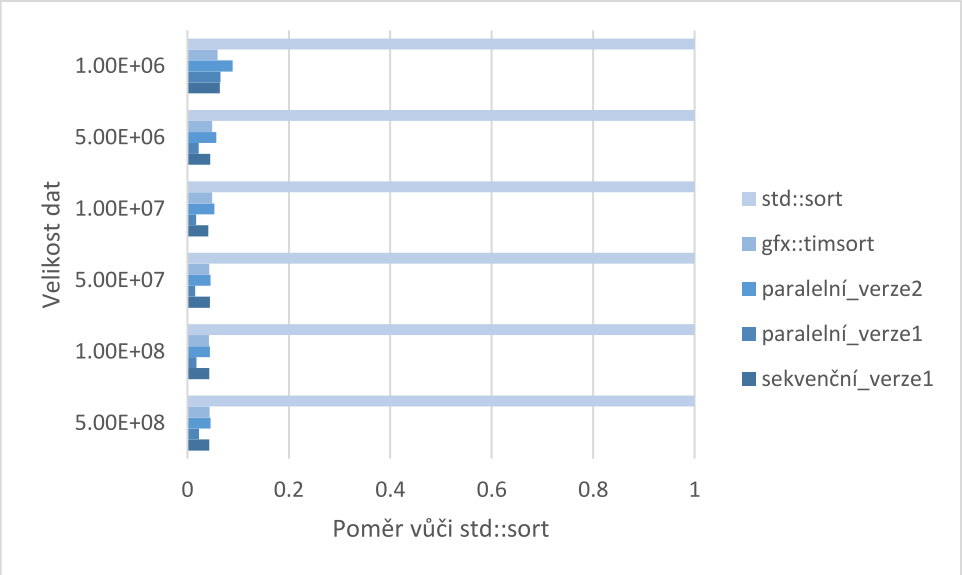
Obrázek 6.4: Všechna data stejná



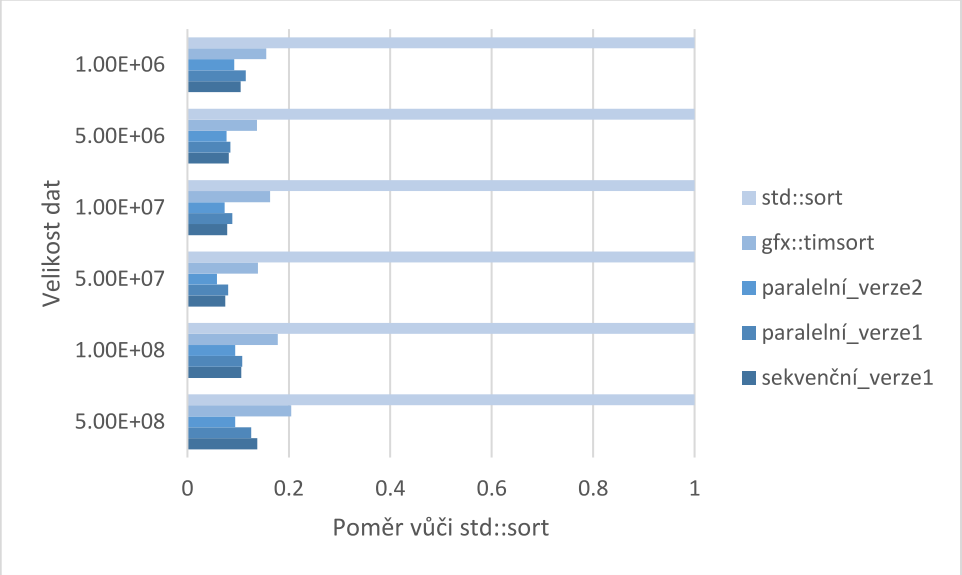
6.1.6 Shrnutí první fáze

Z naměřených dat mi vyplynula jako nejlepší řešení naivní paralelizace. Pokročilá verze se ukázala jako dosti nevhodná. Naivní paralelizace ji předčila na všech typech dat, kromě částečně seřazených, a navíc se u náhodných dat, seřazených s několika výjimkami a na datech s malým počtem různých hodnot ukázala jako naprosto neefektivní.

Obrázek 6.5: Seřazená data



Obrázek 6.6: Částečně seřazená data



Toto může být důsledek spojování různě velkých podposloupností a také plným nevyužitím všech vláken u několika posledních, a tedy i nejnáročnějších, slučování.

Vlákna nejsou plně využita z důvodu, že při slučování několika posledních runů máme méně částí, než vláken a tedy pro vlákna nemáme práci. Práce nad stejnou frontou úkolů také může algoritmus zdržovat, jelikož musíme zabránit

přístupu k frontě více vláknům najednou a tedy roste čas kdy vlákna čekají a nepracují. Ztráty způsobené slučováním runů, které nemusí být v cache a které nemusí být podobně dlouhé algoritmus také zdržují. Zajímavá byla neefektivita tohoto způsobu algoritmu u seřazených dat s vyjímkami, zde jsem očekával od pokročilé paralelizace mnohem lepší vlastnosti. Zdá se ale, že procento náhodných dat přebije množství částečně seřazených posloupností.

6.2 Druhá fáze

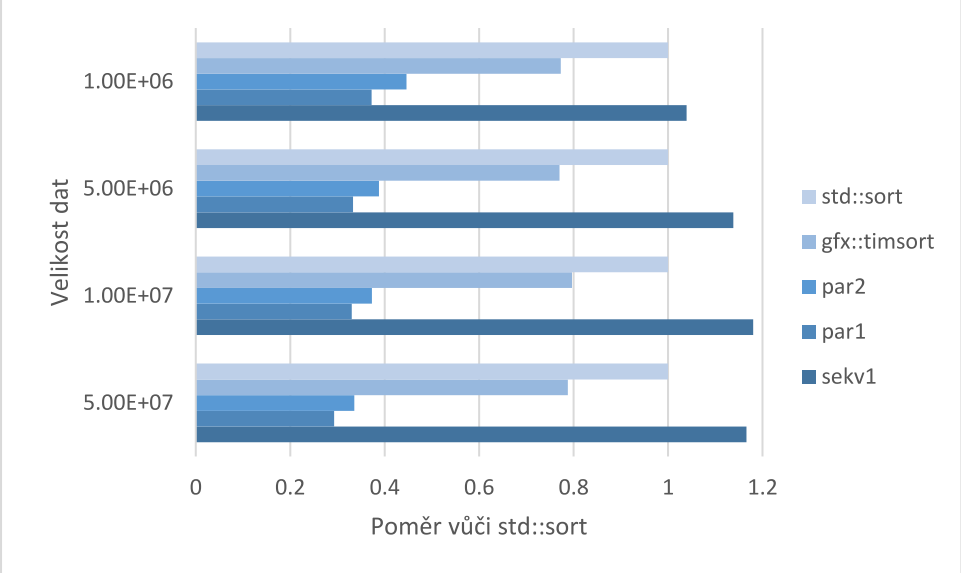
Ve druhé fázi testování byly výsledky mnohem uspokojivější. Naivní paralelní verze se opět ukázala jako lepší, nicméně i pokročilá verze paralelního algoritmu předvedla své kvality.

6.2.1 Náhodná data

Na náhodných datech se opakuje pozorování z první fáze a totiž, že čistě náhodná data nejsou pro timsort to pravé. Obě dvě sekvenční verze poráží *std* :: *sort*.

Oproti první fázi ale vidíme znatelné zlepšení ve výkonu 2. paralelní verze. Vysvětluji si to tím, že na jednoduché porovnávací operace byla náročnost na režii příliš vysoká. Zde trvá srovnání o něco déle a tedy vlákno stráví větší část svého času aktivní prací, namísto čekáním.

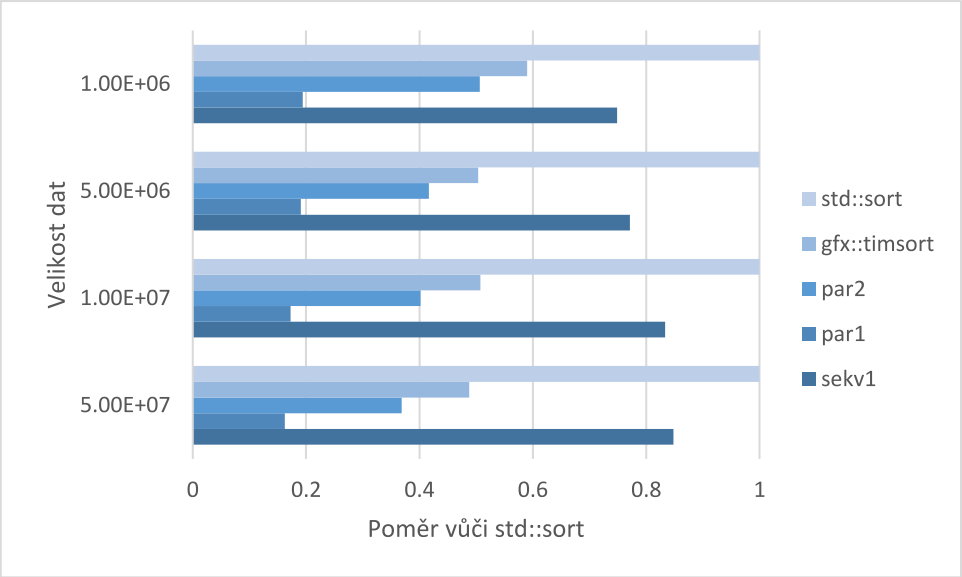
Obrázek 6.7: Náhodná data - Fáze 2



6.2.2 Málo různých hodnot

Opět zde vidíme výrazné zlepšení druhé verze paralelizace, skutečně se projevuje větší poměr aktivního času. Za zmínku stojí výrazné zlepšení obou sekvenčních verzí - projevuje se zde úspornost Timsortu na porovnávacích operacích.

Obrázek 6.8: Málo různých hodnot - Fáze 2



6.2.3 Seřazená data s výjimkami

Naivní verze je pro tyto data až 100× rychlejší, než $std :: sort$. Oproti první fázi se zde jako velmi rychlá verze ukázala i druhá paralelní.

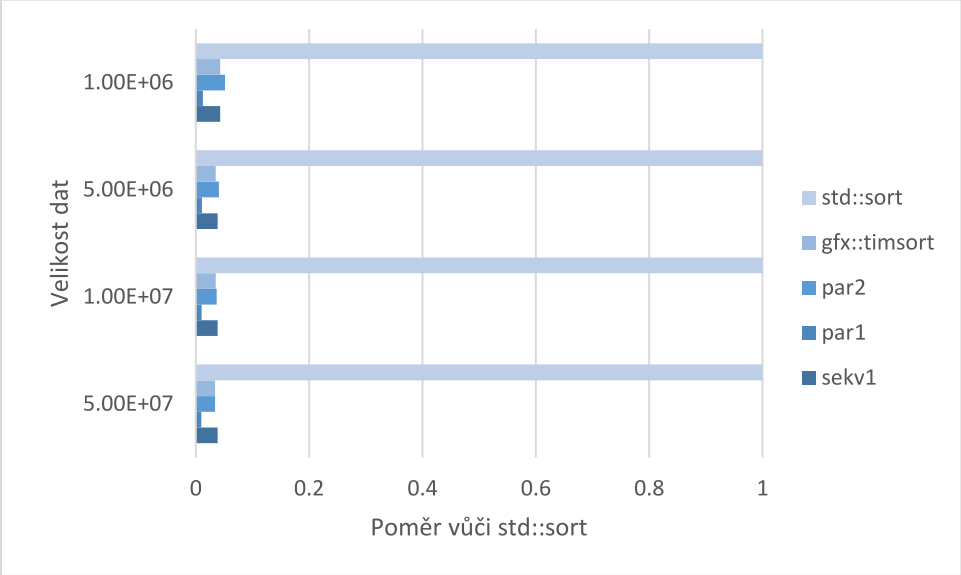
6.2.4 Všechna data stejná a Seřazená data

Podobný efekt pozorujeme i u dalších dvou typů dat. Rozdělení pole na stejné části, které jsou následně zpracovávány timsortem se jednoznačně vyplácí. Zajímavé je ovšem pozorování, že zatímco $gfx :: timsort$ má na obou typech dat zhruba stejné časy, moje implementace je na seřazených datech o 1/4 rychlejší než na datech navzájem rovných. Zatímco tedy na seřazených datech je moje sekvenční verze rychlejší než $gfx :: timsort$, na rovných datech je tomu obráceně.

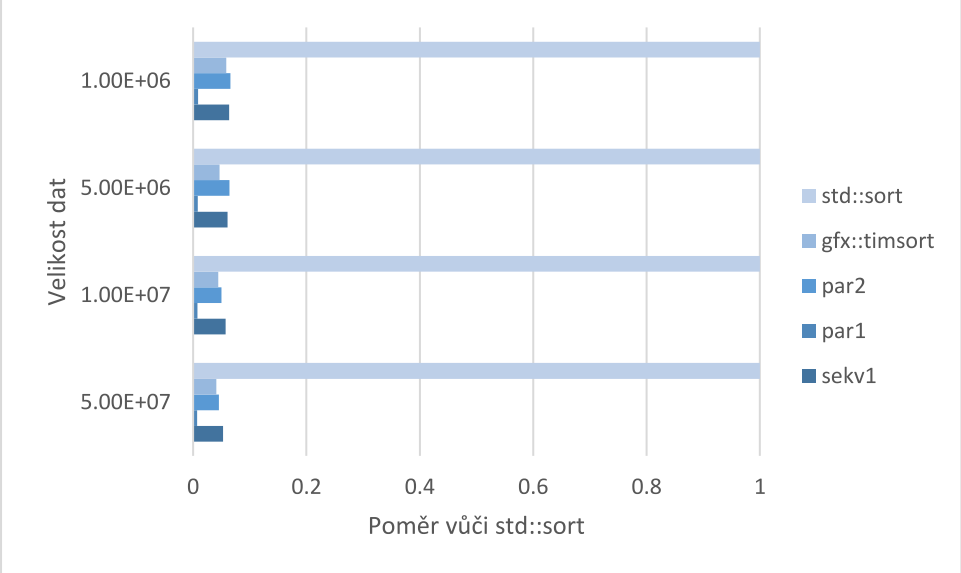
6.2.5 Částečně seřazená

U částečně seřazených dat opět pozorujeme ty samé tendence jako u dat seřazených plně, s tím rozdílem, že pokročilá paralelní verze je zde rychlejší.

Obrázek 6.9: Seřazená data s výjimkami - Fáze 2

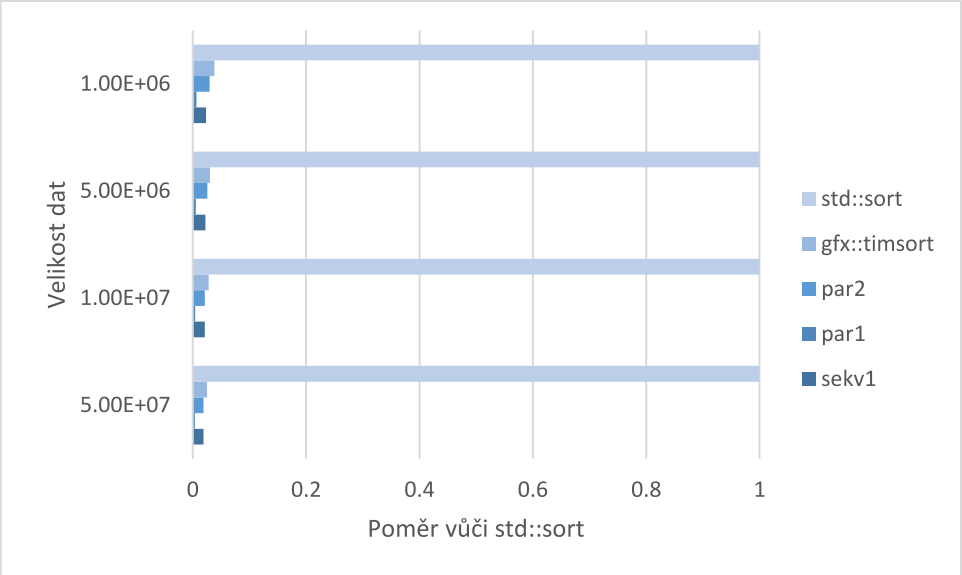


Obrázek 6.10: Všchna data stejná - Fáze 2

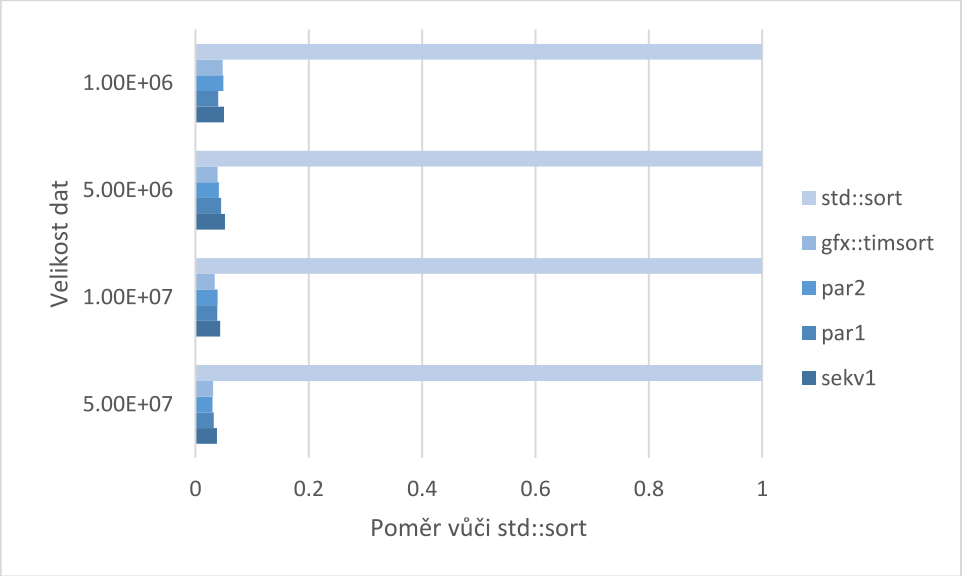


Je to tím, že na seřazených datech pracuje celou dobu vlastně pouze jedno vlákno, zde se dostane ke slovu vláken více.

Obrázek 6.11: Seřazená data - Fáze 2



Obrázek 6.12: Částečně seřazená data - Fáze 2



6.2.6 Shrnutí druhé fáze

Druhá fáze potvrdila pozorování z první a tedy, že naivní verze je funkční a efektivní. Také pokročilá verze se na složitých porovnávacích funkcích ukázala jako efektivní.

Závěr

Během návrhu paralelní verze jsem narazil na řadu slepých uliček, ve výsledku jsem tedy musel přistoupit k paralelní verzi algoritmu, která některé principy Timsortu popírá. Bohužel ani ta se nejprve neukázala jako příliš efektivní. Pro některé typy dat byla dokonce i pomalejší, než verze sekvenční. Ve druhé fázi testování, totiž na datech s drahou porovnávací funkcí se ale pokročilá verze algoritmu o něco lépe a její vývoj tak nebyl úplnou ztrátou času.

Z důvodu přílišného lpění na dodržení všech pravidel Timsortu během počátku návrhu a celkového většího zaměření na algoritmickou stránku problému, oproti stránce implementační mi nezbylo příliš času na podrobné prozkoumání možností implementace pomocí různých technologií.

Za úspěch ale naopak považuji naměření vysoké efektivity naivní paralelní verze, která si na všech typech dat, s jedinou výjimkou, počínala nejlépe.

Navázat na tuto práci se dá tedy ve dvou směrech: zdokonalit a zobecnit naivní verzi pro běžné používání nebo prozkoumat úskalí verze pokročilé, nejspíše příliš drahou režii při obsluze jednotlivých vláken.

Literatura

[1] Trdlička, J.: Procesy a vlákna. [online], [cit. 2018-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-OSY/_media/lectures/02/bi-osy-p02-threads.pdf

[2] Trdlička, J.: Synchronizace procesů/vláken. [online], [cit. 2018-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-OSY/_media/lectures/03/bi-osy-p03-ipc-1.pdf

[3] std::mutex. [online], [cit. 2018-05-10]. Dostupné z: http://en.cppreference.com/w/cpp/thread/mutex

[4] std::condition_variable. [online], [cit. 2018-05-10]. Dostupné z: http://en.cppreference.com/w/cpp/thread/condition_variable

[5] OpenMP. [online], [cit. 2018-05-14]. Dostupné z: https://cs.wikipedia.org/wiki/OpenMP

[6] Šimeček, I.: *Moderní počítačové architektury a optimalizace implementace algoritmů*. Praha: Česká Technika - nakladatelství ČVUT, první vydání, 2015, ISBN 978-80-01-05658-5.

[7] Thread Support Library. [online], [cit. 2018-05-10]. Dostupné z: http://en.cppreference.com/w/cpp/thread

[8] Peters, T.: TimSort. [online], [cit. 2018-04-25]. Dostupné z: https://bugs.python.org/file4451/timsort.txt

[9] Auger, N.; Nicaud, C.; Pivoteau, C.: Merge Strategies: from Merge Sort to TimSort. 2015, [cit. 2018-04-25]. Dostupné z: https://hal-upec-upem.archives-ouvertes.fr/hal-01212839v2/document

[10] MacIver, D. R.: Understanding timsort, Part 1: Adaptive Mergesort. [cit. 2018-04-25]. Dostupné z: https://www.drmaciver.com/2010/01/understanding-timsort-1adaptive-mergesort/

[11] Sood, S.: Parallelizing Timsort. *Saurabh Sood's Blog*, [cit. 2015-10-31]. Dostupné z: https://saurabhsoodweb.wordpress.com/2017/04/18/parallelizing-timsort/

[12] gfx::Timsort. [online], [cit. 2018-04-25]. Dostupné z: https://github.com/gfx/cpp-TimSort

[13] std::sort. [online], [cit. 2018-04-25]. Dostupné z: https://en.cppreference.com/w/cpp/algorithm/sort

[14] Semaphore. [online], [cit. 2018-05-06]. Dostupné z: https://stackoverflow.com/questions/4792449/c0x-has-no-semaphores-how-to-synchronize-threads

[15] Intel® Xeon® Processor E5-2620 v2. [online], [cit. 2018-05-10]. Dostupné z: https://ark.intel.com/products/75789/Intel-Xeon-Processor-E5-2620-v2-15M-Cache-2_10-GHz

[16] Šimeček, I.: Výpočetní prostředky. [online], [cit. 2018-05-10]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-PAP/labs/vypocetni_prostredky

[17] Šimeček, I.: Měření času. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/tutorials/cas

[18] [Python-Dev] Sorting. [online], [cit. 2018-05-11]. Dostupné z: https://mail.python.org/pipermail/python-dev/2002-July/026837.html

Příloha **A**

Obsah přiloženého CD

- `contents.txt` ..... stručný popis obsahu CD
- `readme.txt` ..... návod ke kompilaci a spuštění aplikace
- `Makefile` ..... makefile ke kompilaci a spuštění aplikace
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `Phase1` ..... zdrojové kódy testování první fáze
    - `Phase2` ..... zdrojové kódy testování druhé fáze
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `zadani.pdf` ..... zadání práce ve formátu PDF
- `results` ..... výsledky měření
  - `Phase1` ..... v první fázi
  - `Phase2` ..... ve druhé fázi